STELLINGEN

behorende bij het proefschrift van F.F. van der Vlugt

1. Een analyse van het tijdgedrag van een algoritme georiënteerde processor samen met enige globale kennis van het te implementeren algoritme kunnen een goed beeld geven van het succes van een implementatie. *Dit proefschrift.*


4. Met behulp van de formules van Maxwell kan aangetoond worden dat het in principe onmogelijk is om in een magneet, met een eindige lengte, met behulp van stukjes ijzer een gebied te creëren met een homogene magnetische fluxdichtheid $B$. Het is echter wel mogelijk om de homogeniteit te verbeteren. *Van der Vlugt, F.F., Shimming of MRI magnets with magnetic materials, afstudeerverslag TUD, 1985.*

5. Bij veel vergelijkende studies naar de kwetsbaarheid van vliegtuigen en helivectors voor projectielen kan volstaan worden met een simpel geometrisch model.

6. De enorme uitbreiding van de wetenschappelijke literatuur heeft tot gevolg dat de individuele wetenschapper een steeds kleiner deel van het wetenschapsgebied kan bijhouden.
7. In het wetenschappelijk onderzoek moet veel routinematig werk verzet worden, terwijl op de hoofdlijnen vaak geïmproviseerd wordt. Het is echter beter om voor de hoofdlijnen een projectmatige aanpak te kiezen zodat zowel naar binnen als naar buiten toe een beter inzicht in de werkzaamheden verkregen wordt.

8. Veel niet-brildragende mensen realiseren zich niet dat een brildrager een beperkt gezichtsvermogen heeft.

9. Het gebruik van tijdcontrole met een prikklok in een research omgeving leidt tot afname van het aantal nuttig besteedde uren.

10. De irritatie die ontstaat bij automobilisten bij het samenvoegen van een aantal rijstroken op een snelweg kan verminderd worden door het langzame verkeer bij het snellere te voegen.

11. Uit de hoeveelheid plezier waarmee veel wetenschappers aan computerprogramma's werken blijkt dat veel wetenschappers geen denkers zijn maar doeners.
THE IMPLEMENTATION AND ANALYSIS
OF AN ALGORITHM ORIENTED PROCESSOR
FOR SOLVING THE NAVIER-STOKES EQUATIONS
THE IMPLEMENTATION AND ANALYSIS
OF AN ALGORITHM ORIENTED PROCESSOR
FOR SOLVING THE NAVIER-STOKES EQUATIONS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus,
Prof. drs P.A. Schenck,
in het openbaar te verdedigen
ten overstaan van
een commissie aangewezen door
het College van Dekanen
op dinsdag 16 april 1991 te 14.00 uur

door

Franciscus Ferdinand van der Vlugt

geboren te Leiden
wiskundig ingenieur
Acknowledgement

This investigation in the program of the Foundation for Fundamental Research on Matter (FOM) has been supported (in part) by the Netherlands Technology Foundation (STW).
PREFACE

The research for this thesis forms a part of the work which has been carried out for a FOM project entitled: Design and implementation of an algorithm oriented processor for solving the Navier-Stokes equations. Two groups of the Faculty of Applied Physics at the Delft University of Technology have participated in the project: the Heat Transfer group in which the author worked and the Computational Physics group. The author would like to thank dr ir Th.H. van der Meer and dr ir A.F. Bakker for setting-up the research project, Prof. ir C.J. Hoogendoorn and Prof. ir B.P.Th. Veltman for supervising the project, ir D.A. van Delft for the good cooperation, many valuable suggestions and support, ir H.P. Wolleswinkel for his contribution to the work, drs V.E. Geerlings for proof-reading, and last those others who have contributed in one way or other to this thesis.
CONTENTS

PREFACE

1. INTRODUCTION
   1.1. The goal of this study 1
   1.2. Layout of this thesis 4

2. NATURAL CONVECTION IN ENCLOSURES
   2.1. Natural convection problems 7
   2.2. The mathematical formulation of the natural convection problem 8

3. A NUMERICAL METHOD FOR SOLVING THE NAVIER-STOKES EQUATIONS
   3.1. The discretisation 13
   3.2. The solution algorithm 15

4. PARALLEL COMPUTERS AND THE DNSP
   4.1. Introduction to parallel computers 21
   4.2. Algorithm oriented processors (AOPs) 24
   4.3. AT&T's/Delft Navier-Stokes Processor (DNSP) 26
       4.3.1. Introduction 26
       4.3.2. The architecture of the DNSP 26
       4.3.3. Programming and timing aspects 30
       4.3.4. The simulator and the assemblers 32

5. ALGORITHMS FOR PARALLEL COMPUTERS
   5.1. Grain levels and parallelisation characteristics of algorithms 35
   5.2. The complexity of algorithms on parallel computers 37
   5.3. Performance models for parallel computers 40

6. BASICS OF THE IMPLEMENTATION
   6.1. Timing considerations of the data transport in the DNSP 45
   6.2. General calculation strategy 48
   6.3. General database structure 50
6.4. Calculation procedure with four processors 52
6.5. Algorithms for solving tridiagonal matrices 58
   6.5.1. Gauss elimination with one processor (Thomas algorithm) 59
   6.5.2. The cyclic reduction and recursive doubling algorithm 60
   6.5.3. The elimination from two sides 60
   6.5.4. The Wang partition method 61
   6.5.5. Conclusions 62

7. THE PROGRAMMING ENVIRONMENT 63
   7.1. Implementation, programming techniques
       (do-loops, do-loops with if-statements) 63
   7.2. Programming environment 66
   7.3. Test-facility, testing, testability 67

8. THE IMPLEMENTATION OF A 2D FLOW ALGORITHM ON THE DNSP 73
   8.1. 2D Domain Decomposition 73
   8.2. The 2D implementation, program possibilities, database 76
   8.3. Timing studies 78
   8.4. Calculations for a square cavity 83

9. THE IMPLEMENTATION OF A 3D FLOW ALGORITHM ON THE DNSP 91
   9.1. 3D Domain Decomposition 91
   9.2. The 3D implementation, program possibilities, database 92
   9.3. Timing studies 95
   9.4. Benchmark results for the computational speed 97
   9.5. Calculations for the turbulent flow in a 3D cavity 105

10. FINAL REMARKS AND CONCLUSIONS 111

REFERENCES 113

APPENDIX 117

SUMMARY 121

SAMENVATTING 123

ABOUT THE AUTHOR 125
1. INTRODUCTION

1.1. The goal of this study
Since the design and building of the first computers, they have always tickled people's imagination. Specially in the beginning of the computer era many people thought that computers could solve every problem. It was then even thought that only a few computers were needed. Soon computer users found out that there were limitations to computers. The more computers were used the more limitations, such as the limited speed and limited memory size appeared. For the application of computers in general not only the possibilities are of importance but also the cost. Although computers become faster every year and solution methods become more sophisticated still many problems cannot be solved. Even though the time per operation on computers decreased with about a factor $10^5$ in 25 year, see Hockney & Jesshope (1981). And the relative performance of numerical methods for flow problems increased with about $10^3$ in 25 year as given by Peterson (1984), depending on the type of flow problem involved.

In the middle of the 1970's one of the first modern supercomputers was introduced by CRAY. In the 1980's other supercomputers were introduced and further developed, giving a strong raise in performance and memory size. For people working in Computational Fluid Dynamics this development gave the opportunity to solve three-dimensional instead of two-dimensional problems or to simulate turbulent flow instead of laminar flow. However, the solution of these problems still requires a CPU-time of at least several hours, which is too much if one wants to perform such calculations routinely. Further one should keep in mind that the use of a high speed computer is a balance between computational speed, core memory and cost.

In other areas of Physics the same problem was present, the performance of computers is improving, but still the performance is not at the required level. In the Computational Physics group of the Faculty of Applied Physics of the Technical University of Delft a Molecular-Dynamics Processor, described by Bakker (1983), and a Monte Carlo Ising Machine, described by Hoogland et al. (1983) were developed, to deal with this problem. The processors (computers) were special-purpose computers which combined a high performance and large memory size with low cost, but were only suitable for one type of problem (i.e. molecular dynamics or monte carlo ising). It was this experience available for building special-purpose hardware that led to the idea of developing a special-purpose computer for flow problems. A project was started together with the Heat Transfer group of the Faculty of Applied Physics, who were to take care of the flow modelling and implementation, because of the experience gained in this field. The Heat Transfer group works on convection-diffusion problems which require the solution of two-dimensional or three-dimensional Navier-Stokes equations. In these cases the Navier-Stokes equations are mostly complemented with one or more convection-diffusion equations for heat transfer, mass transfer and turbulent quantities. In general these partial differential equations are non-linear and coupled. The partial differential equations can be transformed to a set of difference equations with the use of numerical techniques. The set of difference equations can be solved with the aid of
a computer, but due to the special structure of the set of difference equations this usually takes a long time. The newly designed computer should improve the balance between cost and performance.

After developing this idea it was found that in 1977 there had been an initiative of the Rand Corporation to develop a Navier-Stokes Computer, described by Gritton et al. (1977). This study is entitled: Feasibility of a Special-Purpose Computer To Solve the Navier-Stokes Equations. The major goal of the Workshop was to study the feasibility of a special-purpose computer designed to solve the Navier-Stokes equations. The computer should be suited for a fairly general class of fluid mechanics problems. Specially for the Workshop, to get some general starting point for the discussion, a preliminary concept for the Navier-Stokes computer was developed. The concept consisted of an array of 10,000 identical large-scale integrated circuits arranged in a 100 × 100 matrix. To reduce wiring congestion, each processor was limited to communication with its nearest neighbours. Each processor could carry out simple calculations and furthermore each processor had space for storing algorithms. In this way three-dimensional problems could be solved by having each processor represent a point in a two-dimensional plane. The processors which were used in the concept were not expensive, so the architecture concept could lead to a very large performance/cost ratio. It was further noted that to exploit the full power of the array computer, the numerical algorithms should match the machine’s architecture and probably a finite difference method would be well suited for the special-purpose computer. The reason for developing such a computer was to make it possible to perform complex fluid flow calculations routinely. It was suggested that a design program should consist of research on algorithms, microprocessor design, machine architecture, computer software and the design and construction of a small experimental array of processors to test the design concepts. It was further noted during the Workshop that software should have special attention. It was stressed that emphasis should be placed on the development of a high-level language appropriate to the specific machine design and applications. Furthermore, special instructions that are important to performance should be available to the user. Also a language simulator should be available early for program development. An important fact noted was that algorithm development must be concurrent with and must interact with the architectural development. One of the conclusions of the Workshop was that the full concept of the special-purpose computer should be feasible in the early 1980 time period. However, there has been no follow up of the Workshop.

The project which was started in Delft was entitled: Design and implementation of an algorithm oriented processor for solving the Navier-Stokes equations, and has been supported by the Netherlands Technology Foundation (STW). The newly designed computer should be specially suited for solving the Navier-Stokes equations. The basic idea was that there is a general form of the transport equations which, after discretisation, leads to a general difference equation. This general form of the difference equations leads to simple software for the processors. This is close to what was suggested in the Rand Report. The computer which should be designed is an algorithm oriented processor (AOP, or algorithm oriented computer) which means that the computer will be designed for a class of algorithms. This is in an elementary way different from the suggestion of the Rand Report. In the Rand Report it is proposed to
built a special-purpose computer only suitable for solving Navier-Stokes type of problems. This new computer should also be suitable for other problems in the same class, which is much wider in a sense than considered by the Rand Report. Furthermore, the designed computer should be programmable, instead of the special-purpose computer which is not programmable.

The first idea was to work with 200 to 1000 simple (micro)processors working in parallel, which was already substantially lower than the number of processors mentioned in the Rand Report. It should be easy to implement the flow algorithm on the AOP because of its special character. To be sure that the computer is generally applicable it should be suited for all kinds of flow and convection calculations together with turbulence modelling.

Before starting the project a four phase partitioning of the research program was made as follows:

1. Definition of the details of the algorithm; research to make optimal use of parallelism and suited solution methods. Definition of the basic components of the computer; design of a prototype computer.


3. Design of the complete computer; development of a high level programming language. The implementation of turbulence, heat transfer on the prototype computer.

4. Construction of a complete computer with software.

For every phase one year was available.

It should be stressed that an existing flow algorithm was going to be used as a starting point and only novelties concerning the computer architecture should be investigated. In the first phase the concurrent development of the algorithm and the computer is stressed. The details of the algorithm are needed to design the computer and the architecture of the computer is needed to make research into the optimal use of parallelism. Next the prototype computer should be built and used. The implementation and experiments on the computer was estimated to cost one year, leaving enough time for solving fluid flow problems. In the third phase certain problems or inefficiencies of the computer should be solved and the general flow software should be written. In the last phase the work of the first three phases should be put together and a user friendly interphase should be constructed. It was already noted before starting the project that it would be hard to finish all phases in the available time. At the start of the project a staff member of the Computational Physics group, A.F. Bakker, got the opportunity to design a new computer system for Molecular Dynamics calculations at AT&T Bell Laboratories, called ATOMS. ATOMS was specially designed for Molecular Dynamics calculations and for Navier-Stokes type of problems. For the Navier-Stokes application the computer was called DNSP (AT&T's/Delft Navier-Stokes Processor). The designed computer is an algorithm oriented processor, but instead of 200 to 1000 simple processors only a small number of more sophisticated processors (4 to 24) are used.

The main goal of the study presented here was to get a three-dimensional fluid
flow program running on the DNSP and to analyse its performance. For the prototype program a two-dimensional flow algorithm was chosen because it is not so large and more manageable in the first instance. The final program should be the implementation of a three-dimensional flow algorithm. The outcome of the study should give a better idea about the feasibility of such a computer. This means that the programming and programming possibilities should be considered. For the programming there should be an answer to the question whether assembler programming of the Navier-Stokes equations is not to costly compared to programming on a supercomputer. Programme possibilities should give an answer to the structure of problems which are suitable for running on the DNSP. Furthermore, the processing speed and memory size should be analysed. For this analysis the different memories, their access times and the processing of the floating-point chips should be considered. In this thesis these analyses are given. Furthermore, the cost/performance of the DNSP can be compared with nowadays (mini-)supercomputers. Calculations should be made on the DNSP with the developed 3D program to show its capabilities.

On the basis of the analyses a better founded proposal can be given for a future AOP. Also an opinion can be given on the possible use and success of such computers.

1.2. Layout of this thesis
The convection-diffusion problem to be first implemented on the DNSP, a natural convection problem, is used as a test case. A short survey of this mathematical-physical problem is given. The discretisation method of the mathematical equations involved is discussed, followed by a discussion on the SIMPLE solution algorithm. This part should be considered as a very brief introduction to the solution of the natural convection problem.

The next part of the thesis is dedicated to parallel computers and the AOP in particular, and to the software for parallel computers. As an introduction to these subjects some brief remarks are given on the general structure of parallel computers. Next, the definitions, advantages and disadvantages of AOPs are given, followed by a description of the DNSP. After the introduction on the hardware an introduction is given on the algorithms which should run on the parallel computer. The granularity of problems and parallelisation characteristics of algorithms are discussed. The granularity expresses the size of the pieces into which the problem is divided. Furthermore, the complexity of algorithms on parallel computers is discussed. This results in the discussion of a performance model for parallel computers which is introduced by Hockney & Curington (1989). Both discussions give a better understanding of the performance of algorithms on parallel computers.

The introduction on the hardware and software is followed by the discussion of the implementation. First the timing of data transport with user made transport routines is discussed which helps to decide which general calculation strategy should be used on the DNSP. After choosing this calculation strategy the general database structure for the different memories of the DNSP can be defined. This is followed by the discussion of the general calculation procedure on four processors. This is in fact the most innerpart of the program for the DNSP. This general calculation procedure influences the processing speed more than any other part of the program. A thorough
analysis is made of the calculation procedure. In this way several of the variables which are used in the performance model of Hockney & Curington can be obtained. Next, several solution algorithms for tridiagonal matrices are discussed, which are used during the solution process in the Navier-Stokes solver. The solution algorithm for the tridiagonal matrices cannot use the general calculation procedures, therefore they are discussed separately.

Next, the programming environment is discussed. This is started with some general remarks about the implementation and programming techniques, and after that the programming environment is discussed. The positioning of the DNSP in the total computer system is explained. As last the test-facility which is used during the implementation is discussed. It was found that this is a tool which is frequently used during the debugging phase of the source code for the DNSP.

The discussion of the actual implementation is started with the two-dimensional (2D) prototype implementation. First the domain decomposition method for the 2D case is explained, followed by the possibilities of the 2D program (size of the problem, boundary conditions, etc.). Next, several timing studies on the DNSP are discussed and compared with the theoretical model values. After this, several 2D calculations done on the DNSP are shown.

Next, the 3D program is described. The description of the domain decomposition method for 3D problems and program possibilities are followed by the timing of the 3D program on the DNSP. The benchmarks which have been performed on several computers such as the Convex C240 and CRAY-X-MP/2 are analysed and compared to the DNSP. Here one of the main reasons for using the DNSP is found, the low cost/performance. As a last topic several 3D calculations on the DNSP are shown and analysed.
2. NATURAL CONVECTION IN ENCLOSURES

2.1. Natural convection problems
The first application area for which the DNSP is used is the calculation of steady natural convection flows in enclosures (cavities). A natural convection flow is a flow which is induced by a buoyancy force. The buoyancy force arises from density differences in a gravity field, the differences resulting from inhomogeneities in temperature, concentration of chemical species, changes in phase, and many other effects, Gebhart et al. (1988). This study concentrates on temperature induced buoyancy forces. There is a great interest in buoyancy flows because of the large number of applications where buoyancy (natural convection) plays a role.

![Schematic diagram of the experimental set-up.](image)

Figure 2.1. A schematic diagram of the experimental set-up.

In the Heat Transfer group of the Technical University of Delft two experimental set-ups have been built for measuring buoyancy induced flows. The details of the two experimental set-ups and the measurements which have been carried out are given by Lankhorst (1991). A schematic diagram of the 3D configurations (cavity) is given in figure 2.1. In the configuration a coordinate system is placed. The gravity force is working in the negative y-direction. One configuration consists of a box with dimensions 1m \times 1m \times 0.1m and the other box measures 0.45m \times 0.45m \times 0.1m. The small cavity can be filled with water or air, and the larger only with air. Henceforth, in the follow-up we refer to the two cavities as the cavity, because only the dimensions are different.

The left wall of the cavity is heated to a certain constant uniform temperature and the right wall is cooled. The bottom wall, top wall, front wall and back wall are insulated. Of course the boundary conditions are idealised. Due to the finite conductivity of the walls and imperfections of the heating and cooling of the walls, the left and right wall do not have a fully constant uniform temperature. Heat conduction appears in the other four walls, and thus there is not a complete adiabatic boundary condition. The deviations from the ideal situation are generally small. In the cavity a number of
measurements can be done, such as obtaining velocity profiles in the left and right wall boundary layer, velocity profiles in the bottom layer, temperatures and heat flow through the walls, etc., as described by Lankhorst (1991).

Due to the temperature difference of the two opposite walls a flow appears, moving upwards near the left wall and downwards near the right wall. The characteristics of the flow depend on the cavity dimensions, the fluid, the temperature and the temperature difference between the right wall and the left wall. The characteristics of the flow can be expressed by two characteristic numbers:

$$Ra = \frac{g \beta \Delta T H^3}{\nu a}, \quad Pr = \frac{\nu}{\alpha}$$  \hspace{1cm} (2.1)

where $Ra$, $Pr$, $g$, $\beta$, $\Delta T$, $H$, $\nu$, $a$, are the Rayleigh number, Prandtl number, gravitational acceleration, thermal expansion coefficient, temperature difference, height of the cavity, kinematic viscosity, thermal diffusivity, respectively. For small Rayleigh numbers the flow in the cavity is laminar and steady. By increasing the Rayleigh number the velocity maximum in the boundary layers increases and moves towards the wall. When increasing the Rayleigh number above a certain value the flow becomes unsteady and periodic. After a further increase of the Rayleigh number the flow becomes turbulent. For air in the large cavity configuration as given above a turbulent flow appears with a temperature difference of $50^\circ C$ and a Rayleigh number of about $Ra = 5 \cdot 10^9$.

An algorithm for solving a 2D cavity flow has been implemented by Lankhorst (1991). The object is to simulate the flow in the middle $z$ plane of the cavity. In this plane the influence of the front and back wall is small. The algorithm also accounts for radiation effects between the walls of the cavity, which can be important. After gaining enough experience with the 2D algorithm a 3D algorithm has been implemented by Lankhorst. The algorithm is used to simulate the entire cavity. Radiation effects are not taken into account in this case.

The buoyancy flow in the 3D cavity for the laminar situation can be modelled by five partial differential equations (four for the 2D case). For turbulence modelling two extra partial differential equations are added. More details on these will be given in the following paragraphs. These equations are coupled in a number of ways. Due to the strong non-linear character of the partial differential equations and the strong coupling between the equations, solutions cannot be obtained easily. Experience shows that a large computing effort is needed to solve the problems, especially for very high Rayleigh numbers.

2.2. The mathematical formulation of the natural convection problem

In this paragraph the partial differential equations which describe the steady natural convection flow in a cavity with the appropriate boundary conditions are given. More details can be found in standard textbooks such as Bird et al. (1960) or Gebhart et al. (1988).

The continuity equation which describes the conservation of mass in a fluid is given by:

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u_j)}{\partial x_j} = 0$$  \hspace{1cm} (2.2)
where \( \rho \) is the density of the fluid, \( u_j \) the component of the velocity vector in the \( x_j \)-direction and \( t \) the time. The Einstein summation convention is used; when a subscript is repeated in a term a summation of three terms (or two for 2D situations) is implied. The conservation of momentum in the \( x_i \)-direction is described by the momentum equation for the \( x_i \)-direction:

\[
\frac{\partial (\rho u_i)}{\partial t} + \frac{\partial (\rho u_i u_j)}{\partial x_j} = \frac{\partial \tau_{i,j}}{\partial x_i} + \rho g_i \tag{2.3}
\]

where \( \tau_{i,j} \) is a component of the stress tensor and \( g_i \) the component of the gravitational acceleration in the \( x_i \)-direction. For Newtonian fluids the component of the stress tensor can be written as:

\[
\tau_{i,j} = -p \delta_{i,j} + 2\mu (e_{i,j} - \frac{1}{3} e_{k,k} \delta_{i,j}) \tag{2.4}
\]

\[
e_{i,j} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right)
\]

where \( p \) is the pressure, \( \mu \) is the molecular dynamic viscosity and \( \delta_{i,j} \) is the Kronecker delta function. The bulk viscosity \( \kappa \) is neglected. Substitution of equation (2.4) in equation (2.3) gives the Navier-Stokes equations:

\[
\frac{\partial (\rho u_i)}{\partial t} + \frac{\partial (\rho u_i u_j)}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j} \left( \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \right) - \frac{2}{3} \frac{\partial}{\partial x_i} \left( \mu \frac{\partial u_k}{\partial x_k} \right) + \rho g_i \tag{2.5}
\]

For the energy equation, expressing the conservation of energy, one can find that:

\[
\rho \frac{DH}{Dt} = \frac{\partial}{\partial x_j} \left( \lambda \frac{\partial H}{\partial x_j} \right) + \frac{Dp}{Dt} + \mu \Phi \tag{2.6}
\]

where \( H \) is the enthalpy, \( \lambda \) the thermal conductivity, \( c_p \) the specific heat at constant pressure and \( \Phi \) the rate of flow energy dissipation into thermal energy. Furthermore, we have:

\[
\Phi = \frac{\partial u_i}{\partial x_j} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \left( \frac{\partial u_i}{\partial x_i} \right) \left( \frac{\partial u_j}{\partial x_j} \right) \tag{2.7}
\]

The last two terms in equation (2.6) can be neglected for air and water under standard natural convection conditions resulting in:

\[
\frac{\partial \rho H}{\partial t} + \frac{\partial (\rho u_j H)}{\partial x_j} = \frac{\partial}{\partial x_j} \left( \frac{\lambda}{c_p} \frac{\partial H}{\partial x_j} \right) \tag{2.8}
\]

The Boussinesq approximation can be applied in the equations (2.2), (2.5) and (2.8),
which means that the molecular dynamic viscosity \( \mu \) and the density \( \rho \) are taken to be constant and only in the buoyancy term the density is assumed to depend on the temperature.

The equations (2.2), (2.5) and (2.8) combined with the appropriate boundary conditions, describe the fluid flow entirely. However, for turbulent flow the involved time scales and length scales are so small, as described by Tennekes & Lumley (1972), that numerical solutions of the system of equations cannot be obtained with present supercomputers. Therefore, Reynolds decomposition is applied in which the quantity of interest is written as the sum of a mean value and a fluctuating value, as given by equation (2.9).

\[
\phi(x,t) = \overline{\phi}(x,t) + \phi'(x,t)
\]

with

\[
\overline{\phi}(x,t) = \lim_{N \to \infty} \frac{1}{N} \sum_{\alpha=1}^{N} \phi^\alpha(x,t)
\]

where \( \phi^\alpha(x,t) \) are obtained values from experiments. In this way the solution can be obtained for the mean flow. Applying Reynolds decomposition to equation (2.2), (2.5) and (2.8) and assuming that the mean flow is steady results in:

\[
\frac{\partial}{\partial x_j}(\overline{\rho u_j}) + \frac{\partial}{\partial x_j}(\overline{\rho u'_j}) = 0
\]

\[
\frac{\partial}{\partial x_j}(\overline{\rho u_j u_i}) = -\frac{\partial \overline{\rho}}{\partial x_i} - \frac{\partial \overline{\mu}}{\partial x_i} \left[ \frac{2}{3} \mu \left( \frac{\partial \overline{u_k}}{\partial x_k} \right) + \right. + \left. \frac{\partial}{\partial x_j} \left\{ \mu \left[ \frac{\partial \overline{u_i}}{\partial x_j} + \frac{\partial \overline{u_j}}{\partial x_i} \right] \right. \right
\]

\[
\overline{\rho g_i} - \frac{\partial}{\partial x_j}(\overline{\rho u_j u_i}) + \overline{\rho u'_i u_j} + \overline{\rho u'_j u_i} + \overline{\rho u'_j u_i}
\]

\[
\frac{\partial}{\partial x_j}(\overline{\rho u_j H}) = \frac{\partial}{\partial x_j} \left( \frac{\lambda}{c_p} \frac{\partial H}{\partial x_j} \right) - \frac{\partial}{\partial x_j} \left[ \overline{\rho u'_j H'} \right] - \frac{\partial}{\partial x_j} \left( \overline{\rho' H' u_j} + \overline{\rho' u'_j H} + \overline{\rho' u'_j H'} \right)
\]

The system of equations (2.11), (2.12) and (2.13) cannot be solved right away because the terms with the fluctuating quantities are unknown. Two terms, the Reynolds stress \( \overline{u_i u_j} \) and turbulent transport of heat due to velocity fluctuations \( \overline{\rho u_j H'} \), are modelled by using the eddy viscosity concept. It is assumed that there is a linear relation between the Reynolds stresses and turbulent heat transfer due to velocity fluctuations with gradients of mean quantities:
\[-\overline{\rho u_i u_j} = \mu_t \left( \frac{\partial \overline{u_i}}{\partial x_j} + \frac{\partial \overline{u_j}}{\partial x_i} \right) - \frac{2}{3} \delta_{i,j} \left( \overline{\rho' u_k u_k} + \mu_t \frac{\partial \overline{u_k}}{\partial x_k} \right) \tag{2.14} \]

\[-\overline{\rho u_j H'} = \frac{\mu_t}{\sigma_T} \left( \frac{\partial \overline{H'}}{\partial x_j} \right) \tag{2.15} \]

where \(\mu_t\) is the turbulent viscosity and \(\sigma_T\) the turbulent Prandtl number. Now there are still some unknown values: \(\mu_t\) and \(\sigma_T\). \(\sigma_T\) can be found from experiments. There are several ways to model \(\mu_t\), here the two equation model as proposed by Launder & Spalding (1974) is used. The model is the \(k-\varepsilon\) model and it consists of two extra partial differential equations for the turbulent kinetic energy \(k\) and the rate of turbulent energy dissipation \(\varepsilon\), and a relation between \(k\), \(\varepsilon\) and \(\mu_t\). The value of \(\mu_t\) is obtained in the following way:

\[\mu_t = \overline{\rho c_{\mu}} \frac{k^2}{\varepsilon} \tag{2.16} \]

\[\frac{\partial (\overline{\rho u_j k})}{\partial x_j} = \frac{\partial}{\partial x_j} \left( \left[ \frac{\mu + \frac{\mu_t}{\sigma_k}}{\sigma_k} \right] \frac{\partial k}{\partial x_j} \right) + P_k + G_k - \overline{\rho \varepsilon} \tag{2.17} \]

\[\frac{\partial (\overline{\rho u_j \varepsilon})}{\partial x_j} = \frac{\partial}{\partial x_j} \left( \left[ \frac{\mu + \frac{\mu_t}{\sigma_{\varepsilon}}} {\sigma_{\varepsilon}} \right] \frac{\partial \varepsilon}{\partial x_j} \right) + \tag{2.18} \]

\[(c_{1\varepsilon} P_k - c_{2\varepsilon} \overline{\rho \varepsilon} + c_{1\varepsilon} c_{3\varepsilon} G_k) \frac{\varepsilon}{k} \]

where \(\sigma_k\), \(\sigma_{\varepsilon}\) is the turbulent Prandtl number for \(k\) and \(\varepsilon\) respectively. The production term \(P_k\) represents the transfer of kinetic energy from the mean flow to the turbulent motion and the buoyancy term \(G_k\) represents an exchange between the turbulent kinetic energy \(k\) and potential energy. The two terms \(P_k\) and \(G_k\) are modelled by:

\[G_k = -\frac{\mu_t \overline{g_i} \frac{\partial \overline{\rho}}{\partial x_i}}{\sigma_T \overline{\rho}} \tag{2.19} \]

\[P_k = \left[ \mu_t \left( \frac{\partial \overline{u_i}}{\partial x_j} + \frac{\partial \overline{u_j}}{\partial x_i} \right) - \frac{2}{3} \delta_{i,j} \left( \overline{\rho k} + \mu_t \frac{\partial \overline{u_k}}{\partial x_k} \right) \right] \frac{\partial \overline{u_i}}{\partial x_j} \tag{2.20} \]

The constants in the \(k-\varepsilon\) model are chosen according to Launder & Spalding (1974) and are given in table 2.1. The additional buoyancy constant \(c_{3\varepsilon}\) in the \(\varepsilon\) equation and its value is somewhat controversial. In literature many different values are suggested, e.g. Rodi (1980). But in principle there is agreement that the value of \(c_{3\varepsilon}\) should be close to one for a vertical boundary layer (increasing the rate of turbulent energy dissipation in a stably stratified fluid flow) and close to zero for a horizontal boundary layer. For our 3D calculations a function for \(c_{3\varepsilon}\) is used with \(g_i\) in the \(x_2\)-direction,
Table 2.1. The values of the constants in the $k$-$\varepsilon$ model.

<table>
<thead>
<tr>
<th>$c_\mu$</th>
<th>$c_{1\varepsilon}$</th>
<th>$c_{2\varepsilon}$</th>
<th>$\sigma_k$</th>
<th>$\sigma_\varepsilon$</th>
<th>$\sigma_T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.09</td>
<td>1.44</td>
<td>1.92</td>
<td>1.0</td>
<td>1.3</td>
<td>1.0</td>
</tr>
</tbody>
</table>

which is given by:

$$c_{3\varepsilon} = \left[ 1 - 0.7 \frac{\sqrt{u_1^2 + u_3^2}}{\sqrt{u_i u_i}} \right]$$  \hspace{1cm} (2.21)

The values of this function for the horizontal boundary layer and vertical boundary layer is close to what Rodi (1980) suggests.

To complete the system of equations a number of boundary conditions are needed. The boundary conditions for the temperature can be set right away. The boundary conditions for $k$ and $\varepsilon$ at the wall are derived from the logarithmic wall function for forced convection because there is no wall function available for natural convection. The velocity profile close to the solid wall in a forced convection boundary layer with only small pressure gradients is given by the logarithmic wall function (for example for a horizontal flat plate).

$$u^+ = \frac{1}{\kappa} \ln(9y^+)$$  \hspace{1cm} (2.22)

with

$$y^+ = \frac{x_2 u^-}{v} \quad u^+ = \frac{u_1}{u^-} \quad u^- = \left[ v \left( \frac{\partial u_1}{\partial x_2} \right)_w \right]^{\frac{1}{2}}$$  \hspace{1cm} (2.23)

By using equation (2.22) and equation (2.23), assuming $P_k = \varepsilon$ and the fact that the shear stress is approximately equal to the wall shear stress, leads to:

$$k = \frac{u_	au^2}{\sqrt{c_\mu}} \quad \varepsilon = \frac{u_	au^4}{\kappa y^+ v}$$  \hspace{1cm} (2.24)

For the velocity components the no slip condition is prescribed instead of the wall function, as given by equation (2.23).

A closed system of equations with boundary conditions is obtained. Next, it should be tried to obtain solutions for this system. Analytical solutions can only be obtained for strongly simplified versions of the equations. Therefore, it is only possible to find numerical solutions of the system of equations, as discussed in the next chapter.
3. A NUMERICAL METHOD FOR SOLVING THE NAVIER-STOKES EQUATIONS

3.1. The discretisation

Obtaining an analytical solution for the system of partial differential equations, discussed in the preceding chapter, is only possible for strongly simplified versions of the system. Therefore numerical methods should be used to obtain solutions of the system. A number of numerical methods are available such as the finite difference method, the finite volume method, the finite element method, spectral methods, etc. Here the finite volume method is used. There are several reasons to use the finite volume method such as the simple structure of the algorithm which appears, the automatic fulfilling of conservation principles, the ease with which terms can be added to or deleted from the system. The 3D discretisation with the finite volume method is discussed. The 2D discretisation can be performed in the same way.

The partial differential equations for the steady turbulent natural convection flow can be written in a general form as given in equation (3.1).

\[
\frac{\partial (\rho u_j \phi)}{\partial x_j} = \frac{\partial}{\partial x_j} \left[ \Gamma_\phi \frac{\partial \phi}{\partial x_j} \right] + S_\phi
\]  

(3.1)

\[
\frac{\partial \rho u_j}{\partial x_j} = 0
\]

(3.2)

where \( \Gamma_\phi \) is a diffusivity, \( S_\phi \) a source term, and \( \phi \) is one of the variables: \( u_1, u_2, u_3, k, c_\epsilon, c_p T \). The equations (3.1) and (3.2) are discretised. Details of the discretisation of equation (3.2) are discussed in the next paragraph. To apply the finite volume method the calculation domain is divided into a number of 3D cells. For each cell a balance between incoming and outgoing quantities is prescribed.

In every grid cell a grid point is placed in which the variable is defined. For the placement of the grid points the staggered grid distribution is used. The velocities are defined in a grid point which is shifted half cell in their coordinate direction, thus the \( u_1 \) velocity is shifted in the \( x_1 \)-direction, etc. The other basic variables are evaluated in the grid points that are not shifted. When a non-staggered grid is used, i.e. the basic variables \( \phi \) are all discretised in the same grid points, it is found that the pressure field is decoupled, as described by Patankar (1980). The pressure in neighbouring points have no influence on each other. In fact the pressure field is split into two fields for a one-dimensional calculation, four fields for a two-dimensional calculation and nine fields for a three-dimensional calculation. For the grid point distribution a large number of possibilities are available. For example an equidistant grid or a non-equidistant grid where the grid points are found with a sinus function can be chosen. Of course the locations of the grid points is of importance, it affects the discretisation error. In principle grid points should be clustered in the areas with large gradients of the major variables.

The discretisation of equation (3.1) is found by integrating it over the grid cell.
By applying the Gauss divergence theorem this integral over the grid cell is replaced by an integral over the boundary of the grid cell. Equation (3.2) is also integrated over the grid cell (Gauss theorem) and subtracted from the integrated form of equation (3.1). By making several assumptions with respect to the integrand of the integrals and using central differences, the difference equation of equation (3.1) is obtained. The general form of this difference equation is given by equation (3.3).

\[
\begin{align*}
ap_{i,j,k} & = a_{i,j,k} \phi_{i+1,j,k} + aw_{i,j,k} \phi_{i-1,j,k} + \\\n& + an_{i,j,k} \phi_{i,j+1,k} + as_{i,j,k} \phi_{i,j-1,k} + \\\n& + af_{i,j,k} \phi_{i,j,k+1} + ab_{i,j,k} \phi_{i,j,k-1} + s_{i,j,k}
\end{align*}
\]

(3.3)

where \(a_{i,j,k}, aw_{i,j,k}, \) etc. are the coefficients of the difference equations, \(s_{i,j,k}\) the source term, \(\phi_{i,j,k}\) one of the variables and \(N_i\) the number of grid points in the \(x_i\)-direction. The expressions for the coefficients are not given here, for they can be found in standard textbooks such as Patankar (1980). The coefficients are composed of convection through the grid cell boundary and diffusion through the grid cell boundary. The convection and diffusion are combined to find the total transport of the variable \(\phi\) through the control volume face. The coefficients depend on the velocities.

Further, several of the source terms contain one or more of the other variables. Because of the appearance of the velocities in the coefficients, the equation (3.3) is non-linear for the velocities. The equation (3.3) for \(k \in \{1, 2\}\) is non-linear because \(\mu_t (= \frac{c_1}{k^2})\) appears in the coefficients. The coefficients of each of the variables are different because of the use of the staggered grid. Neighbouring grid points are coupled by equation (3.3). The coefficients of the scalar variables \(T, k\) and \(\varepsilon\) differ only because a different value for the Prandtl number is chosen. The source terms \(s_{i,j,k}\) of the difference equations are also different.

When convection becomes large in comparison with diffusion, wiggles (point to point oscillations) appear in the solution of the difference equations. To account for this effect modified difference schemes have been introduced. Then wiggles appear later (or not at all), but the discretisation error of the schemes will be larger and has another asymptotic behaviour. Several other schemes can be used such as the upwind scheme or hybrid scheme. When the upwind scheme is used a grid point downstream cannot affect the value of a variable in the grid point under consideration. Only a point upstream can have an effect by convection. The order of the error in the differences is lower (first order) then in the central difference scheme (second order), therefore for sufficient small grid-sizes the discretisation error is greater. But on the other hand the upwind difference scheme leads to a more stable iteration process for the matrix. The hybrid scheme is a combination of the central difference scheme and the upwind scheme. For large Peclet numbers \(|P| \geq 2\), the Peclet number is equal to the ratio of the convection through a cell boundary and the diffusion through a cell boundary (the hybrid scheme behaves like the upwind scheme in which diffusion is set to zero, and for small Peclet numbers \(|P| \leq 2\) it behaves like a central difference scheme. Due to the use of the upwind or hybrid difference scheme the coefficients are all greater or equal to zero. For the diagonal of the system one can find that:

\[
a_p = \sum_{nb} a_{nb} - S_p \Delta V
\]

(3.4)
where \(-S_p \Delta V\) is positive and \(a_{nb}\) are the coefficients of the surrounding points. Because of the positiveness of the coefficients the matrix is diagonal dominant. Furthermore, the term \(-S_p \Delta V\) increases the diagonal \(a_p\). The adding of \(-S_p \Delta V\) is a form of relaxation discussed in paragraph 3.2.

To complete the discretisation the strategy to incorporate the boundary conditions into the difference equations should be chosen. There are several possible boundary conditions such as Dirichlet, Neumann, Robin. Dirichlet is simple, the variables on the boundary are set to the prescribed value and never changed. For Neumann, after solving the difference equations with a fixed value at the boundary, the fixed value is changed to fulfill the Neumann condition. The Robin boundary condition can be handled in the same way.

3.2. The solution algorithm
In the preceding paragraph the discretisation of equation (3.1) was discussed, resulting in a difference equation for each of the variables except for the pressure \(p\). The continuity equation (3.2) can be used to obtain the pressure. However, this equation does not contain the pressure itself. There is a wide variety of methods which deal with this problem. Here one of these methods called SIMPLE (Semi Implicit Method for Pressure Linked Equations), which is described by Patankar (1980), is used. The SIMPLE method couples the momentum equations and the continuity equation. For the SIMPLE method the velocity and pressure are assumed to be composed of a known part \((u_i^*, p^*)\) and a correction part \((u_i', p')\). An equation for the pressure correction is obtained in the following way. A relation between the velocity corrections and the pressure correction is obtained from the discretised momentum equations. These relations are simplified and substituted in the discretisation of the continuity equation. The result is an equation for the pressure correction with a source term which contains the discretisation of the continuity equation for \(u_i^*\). The pressure correction is zero when the continuity equation for \(u_i^*\) is fulfilled. When the pressure correction is unequal to zero after solving the pressure correction equation, the pressure correction is added to the pressure. Further, the velocity correction can be calculated by using the relation between the pressure correction and velocity correction, and it can be added to the velocity. The following schematic description of the SIMPLE method can be given:

1. Estimate the pressure field and velocity field.
2. The momentum equation for \(u_1\), \(u_2\) and \(u_3\) are solved one after the other by using the pressure and velocities from the preceding step.
3. The pressure correction is calculated by using the pressure correction equation.
4. The velocity corrections are calculated from the pressure correction.
5. Field quantities such as \(H\), \(k\), \(\epsilon\), etc., are calculated one after the other. If convergence is not reached execution continues at step 2.

The calculation of fluid properties such as \(\rho\) and \(\mu\) are combined with the calculation of the other field quantities. Steps 1 to 5 are repeated until the pressure correction is small enough and the other quantities are converged.

In the method described above there are still some choices possible. The difference equations for the field quantities can be solved simultaneously but we solve them
one after the other. One difference equation can be solved by inverting a band matrix with the size $N_1^2N_2^2N_3^2$. With some special storage strategies the storage requirements for the matrix can be reduced to $8 \times N_1N_2N_3$ (seven for the coefficients and one for the source). However, the storage requirements for the matrix with this strategy are still large and are of the same order of magnitude as the storage requirements of the major variables. Direct or iterative solvers can be used to solve the band matrix. However, it should be noted that the system of difference equations is non-linear and thus the band matrix should be solved a number of times. Another strategy is to solve the difference equations on a restricted area of the 3D domain, e.g. only planes or lines can be solved. We have chosen for a method in which only one line is solved at a time. The variables on the neighbouring lines are assumed to be known and fixed, from the preceding line iteration. What results is a tridiagonal matrix which has the form as given by equation (3.5)

$$
I_j \phi_{j-1} + d_j \phi_j + r_j \phi_{j+1} = ll_j, \quad j = 1, \ldots, N_2
$$

$$
l_1 = 0, \quad r_{N_2} = 0
$$

The tridiagonal matrix (difference equations) can be solved with Gauss elimination.

So iterations are made line by line in the calculation domain. Of course the iteration should be performed on each of the lines in the 3D calculation domain. The following strategy is adopted for the iterations on the 3D calculation domain. First for the direction of the lines one of the coordinate directions is chosen. Next a coordinate direction is added to get a 2D plane of parallel lines. Now an iteration in the 3D calculation domain is made by iteration on the first 2D plane onto the last 2D plane, and back by iteration on the last 2D plane to the first 2D plane. An iteration on a 2D plane consists of line iterations on the first line to the last line and back to the first line. So for one iteration on the 3D domain, on each line four line iterations have been performed. In the follow up one iteration shall be counted as four sweeps. So one sweep consists of one iteration on each line in the 3D domain. Of course the described procedure is arbitrary, but there is not a criteria to decide which procedure should be chosen. The sweeps on the 3D domain can be repeated until the iteration process is converged.

In structure diagram 3.1 the structure of the Navier-Stokes program is given. This structure diagram is useful in understanding the structure which has been chosen for the calculation on the DNSP. The algorithm which is used is slightly different from the one discussed above. For the calculation of the variables $H, k$ and $\epsilon$ the old velocities are used. So when the velocities $u_i$ are solved the values are moved to the temporary variables $u_i^*$. The pressure-correction is calculated at the end of a line iteration.

The inside of the first do-loop (the first Do in the structure diagram) is what we call the inner-loop. It is the part of the program which uses most of the CPU time. The inner-loop uses initial field values calculated outside the first do-loop and passes through intermediate fields. The second and third do-loop orchestrate the sweep process. In the fourth do-loop the solution for all the variables on one line is obtained. First the stress terms on the boundaries are calculated (calculate tau). In the fifth do-loop the coefficients of the finite difference equations are calculated for one line and
boundary conditions are incorporated in the finite difference equations. Next the system of difference equations is solved. After the solution of $u_i$, $k$, $\varepsilon$ and $H$, the fluid properties $\rho$ and $\mu$ are updated. After updating $u_i$ (substitute $u_i^*$ into $u_i$) the pressure correction equation is solved. Finally, the boundary conditions related to Neumann boundary condition are fulfilled.

After completing the discussion of the discretisation some overview of the algorithm can now be obtained. What should be noted is that the structure of the algorithm is rather simple. Only basic operations (i.e. addition, subtraction, multiplication and division) are used except for the maximum and the minimum in the calculation of the hybrid difference scheme. Furthermore, the square root and log function are used for the calculation of the boundary values of $k$ and $\varepsilon$. Operations are primary performed on vectors, only with stride one. It is this simple structure which will show to be valuable for the implementation of the algorithm on the DNSP.

Above the entire algorithm for solving the system of difference equations is
discussed. In practice some special techniques are added to the iteration procedure to assure a better iteration behaviour. In a number of situations the iteration process diverges or convergences very slowly. Of course a good initial field is of major importance for the time needed to obtain a converged solution. Two of the techniques to improve the convergence are relaxation and the use of false-time steps. Relaxation is simple, only a part of the change of the variable is added to the variable. Under-relaxation is used to reduce divergence of the iteration process. For the false-time step technique a quasi-time step is added to the difference equations. Furthermore, this false-time step may depend on the position in the domain and differs for each of the difference equations. The false-time step works as an under-relaxation. Of course the character of the difference equations, which is influenced by the value of the Rayleigh number, has a large influence on the iteration behaviour. Furthermore, the number of grid points has a large influence on the convergence of the iteration process. When the number of grid points increases the number of operations increases more than proportional.

In the preceding subsection the convergence of the iteration procedure was discussed, but how to measure convergence was not defined. The basic interest is of course to obtain a solution which satisfies the difference equations. A first step is to calculate the residual \( r_{i,j,k} \) for each point in the domain as given by equation (3.6), which is obtained from equation (3.3).

\[
    r_{i,j,k} = \sum_{nb} a_{nb}^{\phi} \phi_{nb} - a_{i,j,k}^{\phi} \phi_{i,j,k} + s_{i,j,k} \tag{3.6}
\]

where \( a_{nb}^{\phi} \) are coefficients of (six) variables in neighbouring grid points and \( \phi_{nb} \) are variables in neighbouring grid points. The solution at some stage of the iteration procedure is substituted into equation (3.6). The residual field which is obtained is a measure for the error in the solution. However, it is difficult to give a bound for the residual, because one does not know how to relate the residual to the error in the solution. So in practice residuals are only checked to see whether they are small.

Another option is to look at some monitoring points and follow their history. When they change with very small amounts the solution is obtained. One should be careful with this method, because monitoring values can change a little at some stage of the iteration procedure, while at a later point in the iteration procedure they vary more. The equations are non-linear therefore such behaviour can be expected. The rate of change which is allowed in practice depends largely on experience. For the calculations of a 2D turbulent natural convection flow in a cavity, discussed in chapter 8, the rate of change is demanded to be below \( \pm 1/10000 \) sweeps. Further the energy balance over the boundaries of the domain can be checked with the help of equation (3.7).

\[
    \text{energy error} = \text{entot}/\text{absen} \times 100\% \tag{3.7}
\]

where

\[
    \text{entot} = \sum_{\text{walls}} \text{heat flux} \tag{3.8}
\]

\[
    \text{absen} = \sum_{\text{walls}} |\text{heat flux}| \tag{3.9}
\]
In fact the Gauss divergence theorem is applied to the summation over the domain of the residual values as given by equation (3.6) resulting in equation (3.8). The expression is normalised with equation (3.9). For the 2D turbulent natural convection flow the energy error is demanded to be below $\pm 10^{-4}$. Also the continuity equation can be checked by using an equation of the form as given by equation (3.10) and (3.11) (for 2D calculations).

\[
s_{flowx} = \sum_{i=2}^{n-1} \left[ \delta x_i \sum_{j=2}^{n-1} \rho_{i,j} \delta y_j u_{i,j} \right]
\]

\[
s_{flowy} = \sum_{j=2}^{n-1} \left[ \delta y_j \sum_{i=2}^{n-1} \rho_{i,j} \delta x_i v_{i,j} \right]
\]

(3.10)

(3.11)

It is assumed that the boundaries are solid. It should be noted that from $s_{flowx} = s_{flowy} = 0$ it does not follow that the continuity equation is fulfilled. For the 2D flow $s_{flowx}$ and $s_{flowy}$ are demanded to be smaller than $\pm 10^{-4}$. In practice all the above described criteria are used. Furthermore, as noted before, the exact moment to stop the iteration procedure also depends on experience.

In the chapters on the implementation of the algorithm on the DNSP we want to calculate the processing speed of the DNSP and compare it with other computers. The processing speed is obtained by dividing the number of floating-point operations by the CPU-time needed for the execution. The processing speed is in most instances given in Mflops, which is $10^6$ floating-point operations/sec. Some reference should be chosen for a floating-point operation. Here the reference is chosen to be the multiplication. It is assumed that the time for addition and subtraction is equal to the time for multiplication. The evaluation of the maximum of two floating-point numbers is counted for one floating-point operation. The division is chosen to be equal to five multiplications, because it measures like that on the DNSP.

<table>
<thead>
<tr>
<th>Table 3.1. The number of floating-point operations for the 2D program.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>tridiagonal (6)</td>
</tr>
<tr>
<td>coeff. source</td>
</tr>
<tr>
<td>total</td>
</tr>
</tbody>
</table>

To obtain the processing speed the number of floating-point operations per sweep per grid point should be found. The results of the counting of the floating-point operations of the 2D program are given in table 3.1. In this table for each of the basic operations the count is given. The cost of the six tridiagonal matrix calculations are set apart, because they are needed later. The total number of floating-point operations is: $275 + 168 + 5 \times 13 + 30 = 538$. 
Table 3.2. The number of floating-point operations for the 3D program.

<table>
<thead>
<tr>
<th></th>
<th>multiplication</th>
<th>addition subtraction</th>
<th>division</th>
<th>maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>tridiagonal (7)</td>
<td>35</td>
<td>21</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>coeff. source</td>
<td>548</td>
<td>396</td>
<td>11</td>
<td>60</td>
</tr>
<tr>
<td>total</td>
<td>583</td>
<td>417</td>
<td>18</td>
<td>60</td>
</tr>
</tbody>
</table>

For the 3D program the results as given in table 3.2 are found. The total number of floating-point operations for the 3D program is: $583 + 417 + 5 \times 18 + 60 = 1150$. As one can see, only a small fraction of the cost is for the tridiagonal matrix: 7.9%. The main computational effort is on calculating the coefficients. The number of divisions clearly is small.
4. PARALLEL COMPUTERS AND THE DNSP

4.1. Introduction to parallel computers
It is not the intention to present a thorough discussion on parallel computers and parallel processing. Only those issues are discussed or summarised that are of importance for understanding and analysing the architecture and the global working of the DNSP. For a thorough discussion on parallel computers and parallel processing, the reader is referred to Hockney & Jesshope (1981) and Hwang & Briggs (1984).

The first electronic digital computers were based on the stored-program concept of Von Neumann. These computers contain an input and output (I/O) device, a single memory for storing both data and instructions, a single control unit for interpreting the instructions and a single arithmetic and logical unit for processing the data. The control unit, the arithmetic unit and the logical unit are referred to as the central processing unit or CPU. The basic concept of the Von Neumann architecture is that operations are performed sequentially. Several bottlenecks appeared with the use of computers based on the Von Neumann architecture. For instance, there was no balance in the use of units, which often resulted in having units that were idle most of the time. New concepts were introduced to improve the performance of the Von Neumann architecture. The balance between the units was improved by the addition of functional units. In the CPU several resources could operate simultaneously, which is referred to as parallelism. Pipelining was introduced, which is related to parallelism in that the resources are lined-up like an assembly-line and data passes through each of the devices. Operations were overlapped such as the operations of the CPU and operations of the I/O unit. Multiprogramming was introduced, in which the system is working on several programs at the same time. A further improvement came from the time-sharing concept, in which large operations for the units are cut into pieces to improve the work load balance between the units. Of coarse this also resulted in a better balanced throughput of the systems. Programs with little CPU-time could be processed parallel with time consuming programs.

Another logical way to enhance the efficiency of the system units, and thus improving the performance of the computer system, is the introduction of parallel processing. The imbalance between units is removed or reduced by adding several specific units to the system, such as control units, memories, execution units and/or I/O devices. The task which is performed by one unit can be divided into several sub-tasks. The functionality of the execution units can be changed. The memory can be split into parts which are connected with the other units in some special way. A choice has to be made for the way in which all the units are interconnected. In this way a whole new range of architectures appeared, which were distinct from the Von Neumann architecture and from each other. By choosing for a certain architecture several special characteristics are obtained for the computer system. It should be noted that not all the tasks which have to be performed by the computer system have the same structure. Therefore, both the Von Neumann architecture and the new architectures are suitable for tasks with a specific structure. Of course a task can consist of several sub-tasks which are distinct in structure, so one sub-task can be suitable for the
architecture involved and another maybe not. Therefore, bottlenecks can still appear and the performance of several units can decrease. This effect has given rise to new architectures which are more balanced.

Because of the great variety of architectures, taxonomies had to be introduced to keep some overview of parallel computers. The taxonomy proposed by Flynn (1966, 1972) is often used and also applicable to the DNSP. The taxonomy of Flynn states that the computing process, in its essential form, is the performance of a sequence of instructions on a set (sequence) of data. Before giving the taxonomy, the concept of instruction stream and data stream should be defined. Instruction stream is the sequence of instructions as performed by the machine. Data stream is the sequence of data called for by the instruction stream (including input and partial or temporary results). With the aid of the definitions given above the organisational classes of Flynn are given by:

SISD
Single Instruction Stream - Single Data Stream; this is the conventional serial Von Neumann computer, it is irrelevant whether there is pipelining.

SIMD
Single Instruction Stream - Multiple Data Stream; this is a computer that retains a single stream of instructions but has vector instructions that initiate many operations.

MISD
Multiple Instruction Stream - Single Data Stream; According to Hockney & Jesshope (1981) this class is void, but Flynn (1966, 1972) gives an example of such a system.

MIMD
Multiple Instruction Stream - Multiple Data Stream; Multiple instruction streams imply the existence of several instruction processing units and thus necessarily implies several data streams. This class therefore includes all forms of multiprocessor configurations, from linked main-frame computers to large arrays of microprocessors.

It should be noted that the taxonomy of Flynn characterises only the organisations by the multiplicity of the hardware; the interaction of the streams is not taken into account. According to Hockney & Jesshope (1981), Flynn does not base his macroscopic classification of parallel architecture on the structure of the machines but rather on how the machine relates its instructions to the data being processed. So it can be concluded that the taxonomy of Flynn is too simple to give a good taxonomy of computer architectures.

For the discussion on the DNSP it is of importance to know more about processor arrays. A processor array consists of the following components; a number of processors, a number of memory banks, a communication network, a local control and a global control. The global control is directly connected with the processors. Each processor is "independent" i.e., it has its own registers and storage. The processors can have a local control which operates on command from the global control. The global control initiates, terminates and synchronises the processors. It is also possible that the processors operate without local control, then only the instructions from the
global control are performed on each of the processors. There are several reasons for choosing processor arrays. An economic reason is that by adding processors to the system only the computing power (theoretical linearly) and local storage increases, other components of the system are not multiplied. The size of the processor array can be chosen such that it is efficient and large enough for the computing problems involved, it is in principle not limited. A disadvantage of processor arrays is that it must be possible to divide the problem over several processors. In fact the problem set which can be solved with a processor array is restricted.

![Diagram of processor array organisation](image)

Figure 4.1. A typical processor array organisation.

Given the building blocks for the processor array, many different architectures can be chosen. An example of a processor array organisation is given in figure 4.1. Every processor has its own memory and communication between the processors takes place via the switching network. How the memory is connected to the processors and how it is addressed are two important features of the processor array organisation. The processors can be connected in various ways, e.g. linear array, ring, star, tree, near-neighbour mesh, completely connected, cube, etc.

One of the variables for the processor array architecture is the number of processors. The required complexity of one processor influences the number of processors. If the processors are to be complex, they are expensive and only a relatively small number of processors can be used. The amount of memory on one processor can also be important. For many applications data should be kept local as to obtain a reasonable processing speed. Otherwise data transfer becomes a severe bottleneck. Further, the number of processors allowed depends on the chosen interconnection between the processors. Some interconnection methods allow only a small number of processors, because of the amount of communication between the processors which affects the efficiency. The application plays in this respect also an important role. Some applications (algorithms) require much communication between the processors, e.g. phenomena with effects on large distance (electric/magnetic fields). There are also applications with only a small amount of parallelism, and thus only a small number of processors can be used efficiently, e.g. the simulation of the behaviour of electronic
devices or the placement of small sub-systems on an electronic device. Other applications allow a large number of processors \((10^3 \text{ to } 10^4)\), e.g. the simulation of the behaviour of atoms. But then the processors should be simple and thus cheap. The iPSC computer (Intel, Portland, OR) and the Connection Machine (Thinking Machines, Cambridge, MA) are representative of the class of parallel computers where tens of thousands to millions of small memory/computational units are interconnected to form a parallel computer with significant speed and memory.

4.2. Algorithm oriented processors (AOPs)

AOPs are computer systems which are specially designed for efficient use of an algorithm or class of algorithms. The DNSP which is discussed in the next paragraph is an AOP because it is built-up in such a way that it is very suitable for a certain class of algorithms (algorithms for near neighbour problems). The class of algorithms is not restricted to one application field. For instance, the DNSP is used for Molecular Dynamics calculations (for which it is designed) and for solving the Navier-Stokes equations (with several convection-diffusion equations added to it). Because an AOP is not restricted to one application field it is certainly not a special-purpose computer. An AOP is application directed programmable while a special-purpose computer is designed for only one application. However, the programming of an AOP is more specialised than the minimal programming tools available for a general purpose computer. An AOP has a number of special features. It is therefore difficult to design a high-level language compiler for it, although the use of a high-level language is desirable in view of the usability and maintainability of the software. Of course, a high-level language compiler can be designed but then an undesirable decrease in performance is to be expected. So instead of using for example the fortran or C programming language, an assembler language is used which is more suitable for exploiting the special features of the architecture of the computer system. However, programming in this assembler language is more difficult and thus more time consuming. Due to this fact the problem set is usually reduced to problems which require a long execution time on the computer system. When the execution time is not long, it is not worth to make the extra software effort and the problem can better be solved on a general purpose computer.

One of the special features of an AOP is the tailored use of vector operations and parallel execution. Furthermore, pipelining can be used in various stages of the operations. The devices are configured in such a way that they are used efficiently and the result is an optimal performance level. The cost of the system can be kept low because only those devices which are really used are added, no exotic technology is used and there is no large overhead to make the AOP a universal one. So the cost/performance ratio can be very small due to this the strong coupling between the architecture of the AOP and the algorithm. Restricted use of devices leads to a computer system with a large MTBF (mean time between failure). During the design of the AOP the designer has a specific class of problems in mind which should be solved. In practice these problems are very large, because that are the problems where the AOP should be used for. In this way systems are built that can be used for solving problems which are exorbitantly expensive and time consuming when using general purpose computers. When the system is built-up of a number of processor boards it is in principle possible to add (or delete) processor boards. A choice can be made
whether the system should be extremely fast or that a lower speed is acceptable along with a proportional decrease in cost. This property and the fixed algorithm makes it possible to use the AOP as a single user computer, eliminating time sharing with other users and bringing the response time equal to the CPU-time. For real-time simulations this is very important.

The advantages and disadvantages of AOPs over ordinary general purpose computers have been pointed out above. These were global differences, but it should be noted that working with AOPs requires a totally new strategy of designing and using (new) software. An example of the different strategy is that algorithms should not be tested for their numerical behaviour on the AOP. It takes too much time for implementation, while it is not certain that the algorithm is really going to be used. Therefore the algorithm should first be implemented on a general purpose computer with some high-level programming language. After testing the numerical behaviour of the algorithm an analysis can be made of the efficiency of the algorithm on the AOP. However, it might be even better to perform tests for parallelisation on the general purpose computer. Other, often simple looking, modifications cannot be tried out by a small effort. It might be so that the modification does not fit the structure of the AOP, the chosen data structures or the software structure. The general conclusion is that implementation on the AOP should not be started before it is clearly decided what software has to be used. Furthermore, it is important that the software is sufficiently general and flexible.

An example of what we call an AOP is the Navier-Stokes Computer (NSC) described by Nosencnuck et al. (1986). The NSC is a parallel-processing supercomputer targeted initially for direct numerical simulations of fluid flow problems. However, it is not a hard-wired machine and may be programmed to address a wide range of problems. The NSC is specially suited to address problems that meet the following criteria:

- the problem is numerically intensive.
- the code makes use of long vectors (>>10).
- the code is computationally local in nature.

The NSC is comprised of memory (computational-units called Nodes), where each Node has a vector-oriented architecture. The Nodes are connected to a front-end computer and are interconnected with each other through the global topology of a hypercube. The primary design feature of the Node is a reconfigurable arithmetic/logic pipeline interconnected to multiple independent memory planes. There are three Node versions; the µNode 60 Mflops 1 Mbyte, the mNode 180 Mflops 16 Mbyte and the Node 480 Mflops 256 to 2048 Mbyte. The pipelines are programmed by the user.

Many of the other specially designed computers are more related to general purpose computers then to AOPs. For example, fluid flow calculations can be performed on systems of transputers. A description of an implementation of a fluid flow algorithm on a transputer system is given by Cox et al. (1990), they obtained an efficiency of 62% for a system of nine transputers. Such a transputer system is not an AOP, because its design is based on general requirements. This is shown by the fact that an efficiency of 62% is obtained, due to data transport there is too much overhead. However, a system of transputers can be configured in a mesh like connection
system, giving a good performance.

4.3. AT&T's/Delft Navier-Stokes Processor (DNSP)

4.3.1. Introduction

In this paragraph a specific AOP for solving the Navier-Stokes equations (and several convection-diffusion equations) is discussed. The used solution methods for the 2D and 3D Navier-Stokes equations are given in chapter 3. The solution methods have in common that for the calculation in one location in the calculation domain only variables are needed from the direct surroundings of this location. This is called nearest neighbour interaction (local interaction) which is typical for finite volume methods, for finite difference methods and for finite element methods. Due to the structure of the solution method many calculations can be done in parallel or in vector mode which enables an enhancement of the processing speed. It should be noted that the behaviour of solutions of the Navier-Stokes equations, e.g. in a natural convection flow in a cavity, is certainly not local. Heating in one location effects in most situations the fluid flow in the entire cavity. It is expected that a global solution algorithm performs better than a local algorithm for solving the system of difference equations. Of course a local solution algorithm can be chosen, e.g. a point or line iteration technique, but it can have a negative effect on the convergence speed of the iteration process. There are methods to solve matrices in a local way with only a relatively small data transport over a long distance, for example multigrid methods or parallel methods for inverting a band matrix. For the underlying project a line iteration technique is used but other algorithms can also be implemented. Local interaction is one of the main features of the DNSP. Another property of finite volume methods is that the same variables are used for the calculation of surrounding grid points. This means that if the data remains locally available the ratio between transport of data and the calculations with these data is small. Another application field for which the computer system is suited is Molecular Dynamics calculations. This is because solving Navier-Stokes problems and Molecular Dynamics calculations have several things in common such as the localness and repeated use of local data.

The idea of localness and short transport due to repeated use of data was introduced at AT&T Bell Laboratories by A.F. Bakker to design a new computer system, called ATOMS. This computer is a multi-processor system consisting of a number of processor boards. To be able to use ATOMS for solving Navier-Stokes equations the processor boards were connected as a closely coupled linear processor array to permit data transport between neighbouring processor boards. Molecular Dynamics calculations were implemented on the multi-processor system, described by Bakker et al. (1990). For the Navier-Stokes application this computer system is called DNSP.

4.3.2. The architecture of the DNSP

In this paragraph the architecture of the DNSP is discussed. Only a functional description is given, for a detailed hardware description the reader is referred to Van Delft (1991). Because the DNSP is designed specifically for the problem involved it is simply structured. The DNSP is programmed in a specially developed assembler language. This has been done for optimal use of the possibilities of the DNSP. The instructions that are used most often will be discussed in order to understand their
timming.

In figure 4.2 a global layout of the computer system is depicted. A host computer (e.g. a Microdutch-400 or a MicroVAX running under the Unix operating system) acts as central point and is connected to several devices such as terminals, disk drives, gratic workstations, printers, networks, etc. The DNSP is attached to the host computer via an interface board which separates the bus of the host computer from the bus of the DNSP. The DNSP consists of a control board, a number of memory boards (containing the large common memory) and a number of processor boards. All boards are connected with the host computer via a common bus, called the WME-bus. The processor boards are connected with each other as a linear processor array. On the processor boards the actual floating-point calculations are performed. The boards operate as a MIMD (multiple-instruction multiple-data) computer. The host computer can send instructions to the control board and will receive status signals from the control board. Numerical data can be transported between the host computer and the large memory. In the DNSP the control board supervises the processor boards and the large memory.
The large memory is built up of commercially available dynamic RAM (Random Access Memory) boards. It contains 1M words (32 bits) of memory in the prototype computer system, but can be expanded to fit the problem size by adding memory boards. The large memory has two different functions: it is the shared memory of the processor boards and it is the pass-through area for data to be stored locally on the processor boards. The host computer can write (or read) numerical data only to (from) the large memory. In this mode addresses are supplied by the host computer. Data can be transported between the large memory and the other boards via the common bus. In this situation the large memory is addressed by the control board.

![Diagram of processor board](image)

Figure 4.3. The layout of a processor board.

The prototype computer system was limited to four processor boards, but it can be expanded to a processor system with up to 16 processor boards. The layout of a processor board is given in figure 4.3. The processor board is divided into two functionally different sides; the data management side or X-side and the floating-point side or F-side. Each side has its own micro-programmable control. The cycle time of the X-side is 100 ns, thus giving a cycle frequency of $10^7$ X-cycles/s. The cycle time of the F-side is half the cycle time of the X-side resulting in a cycle frequency $2 \times 10^7$ F-cycles/s. Data is transported within a processor board via the internal data bus which is 32 bits wide. The internal data bus is connected to the WME-bus via the two latches wbusil (W-bus input latch) and wbusol (W-bus output latch). Data communication with the left neighbour takes place via the latch NL and with the right neighbour via the latch NR.

At the X-side communication with the control board and supervision of the F-side takes place, and the data storage and transport on the processor board are controlled. The micro-program for controlling the X-side is stored in a 16k × 64 bits writable control store (WCS). Fixed-point operations are performed by the 16 bits processor XP. From the internal data bus a 32 bits number is fed into the two latches of the 32 bits
input register xpir (XP input register). The two 16 bits numbers can be read into XP once per X-cycle. Output of the XP is routed to the internal address bus which is 16 bits wide. The output of XP is stored in xmac, lmcl, lmcm, xpor or xorm. Two 16 bits numbers are merged in xpor into a 32 bits number (the least significant part and the most significant part), that is transported to the internal data bus.

Data can be stored on the data management side in two memories, LM which is a 1M words (32 bits) dynamic RAM memory and XM which is a 64k words static RAM memory. The LM memory can be enlarged to 4M words.

Figure 4.4. The layout of the four processors.

The actual floating-point calculations are performed at the floating-point side. The control of the F-side can access a writable control store (WCS) of 16k instructions of 32 bits. The floating-point operations are performed by four processors, each attached to a small memory (called SM, 2 × 32 words). The layout of the four processors and the SM memories is given in figure 4.4. The four processors work SIMD (Single-Instruction Multiple-Data), i.e. they all perform the same instructions on different data. The SM memories are divided into two parts; SMA and SMB. The data management side and the four processors can access SMA and SMB in parallel. The floating-point operations of one processor are performed parallel by two floating-point chips, one for multiplications and the other for ALU operations such as additions, subtractions, divisions, etc. The floating-point operations can also be performed in double precision numbers (64 bits), this halves the speed.

The control board is almost equal to the X-side of a processor board. It does not contain an F-side, because floating-point calculations are only performed on the processor boards. The difference with the X-side of the processor boards is that the control board has some extra control facilities for the communication with the processor boards and does not contain an LM memory. Instructions can be initiated and supervised by the control board such as transport of data between the large memory and the processor boards, transport of data between a processor board and its neighbours via
the special interconnection or starting the execution on one or more processor boards.

4.3.3. Programming and timing aspects
First the timing and programming of the four processors depicted in figure 4.4, will be discussed. The two floating-point chips can perform many operations on single (32 bits) or double (64 bits) precision numbers. Only the double precision operations are discussed here. A double precision number consists of a mantissa of 16 decimals and an exponent between $10^{-306}$ and $10^{306}$. One of the floating-point chips (Weitek WTL 1165) is an arithmetic logic unit (ALU) which performs additions, subtractions, divisions, conversion from floating-point to fixed-point data and vice versa, conversion from double to single precision and vice versa, etc. The second floating-point chip (Weitek WTL 1164) is a multiplier. Several flags can be set as result of a floating-point operation. Conditional storage can be achieved depending on the value of a flag. A floating-point routine is built up of a label followed by a number of instructions and is ended with an end statement. Obtaining the timing of this floating-point routine is difficult without using the compiler for the F-side because the timing depends on the optimisation techniques which are used by the compiler. For a multiplication, subtraction or addition one finds approximately 16 F-cycles per operation per chip, in the situation where two floating-point chips are in operation. For more details on the timing of the floating-point chips the reader is referred to Van Delft (1991). The approximate timing for the division operation is: $5 \times 16 = 80$ F-cycles. It can be concluded that every two F-cycles a floating-point operation (multiplication, addition or subtraction) is performed by the four processors resulting in a processing speed of 10 Mflops for one processor board. This processing speed is used as the global peak rate of one processor board.

The SM memory is divided into two parts (SMA and SMB) as depicted in figure 4.4. Memory positions are addressed from the X-side or the F-side in two ways; direct referencing to SMA or SMB, or relative addressing of SMA or SMB based on the value of a pointer which is set on the X-side. When the addressing depends on the pointer its value can be changed and then the other part of SM is selected. The positions in one part of SM (SMA or SMB) are numbered from 0 to 31. Getting a double precision number means reading two positions (two words of 32 bits). In one F-cycle a word is moved from SM to one of the floating-point chips or vice versa. Via the latches små and smol the X-side has access to the SM memories. One of the SM memories has to be selected and then transport can take place. Transport between SM and the internal data bus is pipelined, the address is given two X-cycles before the data is transported. This slows down data transport when only a small number of floating-point numbers is involved. To start execution on the F-side, first a label is given by the X-side and then a flag is sent. After finishing the floating-point routine the F-side sends flags to the X-side.

Before writing or reading XM the address counter/register xmac is set. In the next X-cycle the data can be transported between XM and the internal data bus. It is also possible to increment or decrement xmac with one address position. For LM an address counter/register lmac which contains two 16 bits words (lmac1 and lmacm) has to be set, this takes at least two X-cycles. The address counter/register can be incremented or decremented with one address position within one X-cycle. It takes three
X-cycles to transport a word between LM and the internal data bus.

The 16 bits processor XP performs the fixed-point calculations on the X-side. XP contains 32 registers and an accumulator. Per X-cycle one fixed-point operation is performed. Binary integer operations are available such as addition, subtraction, etc. There are a number of unary operations available such as bit-setting, bit-shifting, negative, incrementing, etc. As the result of some fixed-point operations a carry-bit is set. This carry-bit can be used for the higher precision integer arithmetic. 16 bits integers can be used in operations, but the instruction takes two X-cycles because the integer has to be decoded which takes one X-cycle.

Words can be put in the latches NL and NR (figure 4.3), next they can be moved to the left neighbour board or the right neighbour board respectively. It takes one X-cycle to bring the word into the latch, one to transport it to the neighbour and one to get it out of the neighbours latch into one of the memories or registers. Two of these three operations can be combined resulting in two X-cycles for transport between the processor boards.

For the X-side a program has to be written. After compilation the micro-instructions can be down loaded into the WCS of the X-side. The program consists of a number of (program) lines with instructions. A program line is referred to by calling a label. During the execution the program counter of the sequencer points to the micro-instruction which is to be executed. The program counter is incremented after the execution of a micro-instruction, unless some other instruction has to be performed such as jumping to another position in the WCS. Several values of the program counter can be pushed on the stack of the control processor. The program counters on the stack can be used as jump addresses by popping them back. By changing the program counter some structure can be brought into the micro-program. Loops and if-statements can be performed (do-loops, while-loops, until-loops, etc.). Furthermore, the control processor contains a register/counter which can contain the position of a micro-instruction or an integer which can be used for loop counting. Jump and loop operations are performed depending on a status which can be set.

The control board sends instructions to the processor board. In fact, an instruction is only the address in WCS where execution should be started. The processor board sends flags to the control board to confirm that it has received an instruction or that it is ready.

The calculation of a number of floating-point routines proceeds in the following way. The two micro-programs are down loaded into the two writable control stores. Then data is transported from the large memory to XM. Before starting up the floating-point routine on the F-side, data is transported from XM to each of the SM memories. After the first data transport the pointer (pp) which selects SMA or SMB is changed. This means that for the next data transport between XM and the SM memories the other part of SM is used. Then the label of the floating-point routine is send to the F-side followed in the next X-cycle by a flag. After receiving this flag the F-side starts execution at the label position and uses the data which is just put in the SM memories. At the same moment the X-side starts loading the other part of the SM memories. When the F-side is ready, which can be detected by the X-side, the pointer pp is changed and the F-side is started again. At this point the X-side can read the
results of the first calculation from the SM memories into XM and next new data can be placed in the SM memories for the next calculation. When both the X-side and the F-side are ready the pointer pp is changed and the procedure is repeated until all routines have been carried out. After finishing the last floating-point calculation the X-side reads the data from the SM memories while the F-side is idle.

4.3.4. The simulator and the assemblers
Several compilers are used to decode the assembler instructions to machine level instructions. The compilers were developed at AT&T Bell Laboratories for ATOMS. The compiler for the F-side optimises the code of the floating-point routines by interleaving multiplication and alu operations. Several options for the compiler are available to obtain for example the timing of the instructions. The compiler named "fas" generates a file with the machine code and a file with label names and the label positions in the machine code file. The labels and their positions are used by the compiler for the program of the X-side, called "mas". The compiler for the program of the X-side generates a file with machine code for the X-side and a file with labels and their positions in the machine code file. These can be used for the compiler of the program of the control board. The compiler for the program of the control board again generates a file with machine code and a file with labels and their positions in the machine code file. These labels are used to start an instruction on the DSNP from the host computer. The three compilers give diagnostics when there is a syntax error in one of the programs. For more details on the compilers the reader is referred to Van Delft (1991).

Because the DSNP was not available at the start of the project, it was decided to build a simulator. However, later during the research project it was found that the simulator was very useful for program development and after the development of the first program and running on the DSNP we found that it was not only useful but in fact absolutely necessary. The simulator and interpreter for the simulator are designed and implemented by Van Delft, a detailed description of which is given in Van Delft (1991). The simulator is called "sim" and it runs as a background process and is non-interactive. Sim runs on its own until interrupted by a host process (which is a simulated process). With sim a functional simulation of the DSNP is performed. For example the floating-point operations are performed in host format and then converted to DSNP format. Control of the simulator is identical to that of the DSNP except for a software interface. All requests to the simulator are handled by one routine, with which a read or write request with user supplied data, a user supplied address and the number of bytes to transfer can be specified. There are several debugging options available e.g.: display program counter values of the X-side and F-side, run-time errors such as stack overflow in the control processor of the X-side, etc. A desired configuration can be chosen by specifying the number of processor boards, the memory sizes of XM, LM and the large memory. Naturally, the simulator is running slower than the DSNP (it is around 10000 times slower), so only parts of the flow program can be simulated.

The interpreter for the simulator and the DSNP is called "control". It takes its input from "stdin" (the standard input, UNIX) if no file is specified. The main task of control is to simplify the communication with the simulator, by replacing the I/O
routines control also works with the DNSP. To simplify debugging, the XM, LM and SM memories may be down or uploaded. A number of commands can be given, such as general commands concerning the control of the processor, execution of shell commands, displaying of expressions or variables, etc.
5. ALGORITHMS FOR PARALLEL COMPUTERS

5.1. Grain levels and parallelisation characteristics of algorithms
The title of this chapter "algorithms for parallel computers" is used to stress the fact that extra demands are imposed on algorithms when they are used with parallel computers. The extra demands depend on the architecture of the parallel computer that is used. The result is that more than one algorithm is optimal for one problem, depending on the parallel computer. This differs from the experience with the sequential Von Neumann computer where (in most cases) there is only one algorithm that performs better than all others. The extension to parallel computers gives rise to new classifications for algorithms. The usefulness of the algorithm now depends also on the parallelisation characteristics of the computer involved. For sequential Von Neumann computers the architecture has little influence on the way in which the algorithm should be designed. Of course algorithms can be tuned somewhat to the computer but the improvements are small. It can be shown that the architecture of a parallel computer has enormous influence on the performance of the algorithm. Therefore it is in general not possible to obtain a high efficiency for a certain algorithm on a parallel computer right away. The algorithm has to be structured at the right levels in the right way. There are strategies to make an algorithm more suitable for the parallel computer involved. These strategies depend on the architecture of the parallel computer and on the structure of the algorithm. In general the algorithm has to be decomposed into a number of independent or weakly related tasks. In an algorithm one or more levels of parallelism can be distinguished. A commonly used term in this respect is the granularity of the problem. The granularity expresses the size of the pieces into which the problem is divided. This can be expressed in numbers of equivalent instructions or statements. Almasi (1985) gives an example of the possible grain levels as depicted in figure 5.1. As can be seen there is a wide variety in grain levels possible. It depends on the algorithm and on the parallel computer at which level tasks should be created.

There can be large parts in the algorithm that can be decomposed right away. For instance, non-recursive do-loops appear in the calculation of the coefficients of a difference equation. The grain size of these tasks is rather small, a size of 2 to 10 operations. But for a higher grain level it can be difficult to find these tasks. Further there should be some kind of balance between the tasks, otherwise several of the processors could be idle for a certain time. When there are dependent tasks one can try to change the algorithm in such a way that tasks appear that are only weakly coupled. In general a number of techniques for creating algorithms for parallel computers can be distinguished. The methods for creating parallel algorithms are divided into several groups.

1. By examining the algorithm and the restriction of the algorithm, an algorithm can be obtained which is mathematically equivalent to the original algorithm and has a greater degree of parallelism. This is what is called explicit parallelism. For example, for SIMD machines a serial algorithm can be converted into an algorithm operating on vectors. At a higher grain level the problem can be split
up into a number of subproblems that are solved in parallel. The solution of the whole problem is obtained by combining the solutions of the subproblems.

2. The recursive doubling techniques; a task of size \( n \) is split into two independent tasks of identical complexity of size \( n/2 \), and this splitting can be continued recursively. For \( N = 2^n \) elements, this method requires \( \log_2 N \) parallel steps. The serial implementation requires \( N-1 \) steps (implicit parallelism).

3. The "divide and conquer" techniques; the problem is split into a number of weakly coupled subproblems. The subproblems are solved in parallel. The solution can then be obtained by simple manipulation of the solutions of the subproblems.

4. The method of vector iteration: a direct serial algorithm is converted into an iterative method that rapidly converges to the solution under certain conditions. The speedup factor of the parallel version depends on the ratio of the steps needed in the direct version to those required by the iteration.

Most of the algorithms for parallel computers can be related to one of the four groups mentioned above. For the methods of groups 2, 3 and 4 the order of the calculations is changed. Due to this change numerical characteristics of the algorithm can be altered. For example, rounding-off errors can be different, the propagation of errors and the stability can be changed. These factors have to be considered when using a different algorithm that is more suitable for a parallel computer.

In practice many methods are used that are based on a combination of the four methods that are described above. For example, Lin (1989) suggests several
techniques to divide a Navier-Stokes problem into several tasks which can be solved alone or in combination. The techniques are based on the fact that there are basically two different elements that mathematically describe the problem, the domain $\Omega$ and the set of partial differential equations. It is suggested that there are two independent directions along which parallelism can be explored. The first technique is called the domain decomposition technique, in which the domain is decomposed in a number of subdomains which are connected via a number of boundaries. The solving of the problem on the domain can be transformed to solving a problem on the boundaries. The transformation on each of the subdomains can be performed parallel. When the solution on the boundaries is obtained then the solution can be obtained on each of the subdomains. For this technique a direct approach can be adopted (e.g. the Wang method, Wang (1981)). This method is a representative of the third parallelisation method. It is also possible to apply an iterative domain decomposition technique.

This more heuristic method, to solve the Navier-Stokes equations on several subdomains, is described by Meakin and Street (1987). This method is a representative of the fourth parallelisation method. The iterative technique can certainly be used when the overall computation is also iterative. For the iterative technique first the boundaries are computed and next the interior points (e.g. Block Jacobi). The second technique discussed by Lin (1989) is called the operator splitting technique in which the set of partial differential equations is split into several sub-systems in such a way that an appropriate parallel combination of their solutions will give the final result. Of course the iterative procedure can still be adopted for this technique. An example of an operator splitting method is to solve the Navier-Stokes equations on one processor and the energy equation on another processor. These techniques can be used on several grain sizes, giving the opportunity to use a large number of processors.

5.2. The complexity of algorithms on parallel computers

In this paragraph fundamental aspects in the complexity of algorithms are discussed. This is needed because one wants to know how much time is needed for a given algorithm and how many resources are involved. For sequential computers one searches for the algorithm which minimises these two parameters. The complexity analysis for sequential computers is easy to perform and depends only a little on the computer architecture. Several definitions related to complexity are given by Aho et al. (1974). The time complexity of an algorithm is defined as the time needed by an algorithm expressed as a function of the size of a problem. The limiting behaviour of the time complexity as size increases is called the asymptotic time complexity. Analogous definitions can be made for space complexity and asymptotic space complexity. If an algorithm processes inputs of size $n$ in time $cn^2$ for some constant $c$, then the time complexity of that algorithm is said to be $O(n^2)$. By allowing parallel execution new parameters have to be introduced. The timing of the algorithm on a parallel computer depends on the number of processors. In practice the parallel computer has a fixed number of processors, independent of the application. For theoretical considerations often unlimited parallelism is allowed i.e. a sufficient number of processors, where the number of processors may vary with the application. In this respect the term parallel complexity is of importance. Parallel complexity is the time needed for the algorithm when a sufficient number of processors is available. To compare the parallel complexity of computer programs the speedup ratio is introduced, as defined in equation (5.1).
\[ S_p = \frac{T_1}{T_p} \quad (5.1) \]

where \( T_1 \) is the computing time on a computer with one processor and \( T_p \) the time with \( p \) processors. Jordan (1982) and Stone (1973) categorise the operations of interest into a number of levels of parallel complexity.

1. \( S = kp \), constant-time operations (\( k \) is a constant).
   There are a large number of fundamental operations or calculations that can be done concurrently whenever there are many to be done, e.g. \( z_i = x_i + y_i \), for \( i = 1, 2, \ldots, n \). Thus in an ideal computer the time required to process an arbitrary number of such calculations is the same as for one operation, independent of the number \( n \).

2. \( S = kp/\log_2 p \), \( \log_2 \)-time operations.
   Perhaps the most frequently occurring operations that may reduce the anticipated speedup in a multi-processor system are those for which the parallel complexity is \( O(\log_2 t) \), where \( t \) is the time required if performed sequentially. These include the following operations:
   - Reduction operations on vectors, sums, norms, etc.
   - Applications requiring low-order linear recursions, such as first-order linear recursion, etc.
   The \( \log_2 \)-time operations are optimally performed through recursive doubling strategies best represented by an associative fan-in algorithm, described by Heller (1978).

3. \( S = k \log_2 p \), e.g. for searching algorithms.

4. \( S = k \), e.g. for certain nonlinear recurrences.

The categories 1 and 2 are the most interesting for our application. As noted before, in practice the number of processors is fixed. In that case the interest is in the timing behaviour of algorithms, but one is also interested in the speed improvement and efficiency of the processors. The speedup is defined as in equation (5.1). The efficiency of the processors can be defined as:

\[ E_p = \frac{S_p}{p} \quad (\leq 1) \quad (5.2) \]

Schendel (1984) further defines the measure of both speedup and efficiency of the algorithm \( F_p \) as:

\[ F_p = \frac{S_p}{pT_p} \quad (5.3) \]

An algorithm is effective when it maximises \( F_p \). According to Heller (1978) a simple simulation argument shows that \( S_p \leq p \) and \( E_p \leq 1 \). Note \( E_1 = 1 \), so one doesn’t necessarily want to maximise this function.

According to Heller (1978) the goal is to construct algorithms exhibiting linear (in \( p \)) speedup and hence utilising the processors efficiently. That is, for problems of size \( n \) one wants an asymptotic speedup of the form \( S_p(n) = kp - g(p,n) \), where \( 0 < k \leq 1, 0 \leq g(p,n) = O(1) \) as \( n \to \infty \); \( k \) should be independent of \( p \) and close to
1. Think of $g(\ldots)$ as a penalty for the use of parallelism on small problems: for large problems the rewards should outweigh the penalties.

Linear speedup is not always possible. There are certain computations for which the maximal speedup is $S_p(N) < d$ with $d$ a constant. Such a computation clearly makes poor use of parallelism. For many problems in linear algebra the best speedup is $S_p(n) = kp/\log_2 p - g(p,n)$, which is acceptable though less than linear.

Until now only very simple algorithms for several identical processors have been discussed. When there is a scalar and vector unit and when the software contains scalar and vector operations, another effect is observed. This effect is due to what is called Amdahl’s law, described by Amdahl (1967). This law gives a very simple model for the gain $G$ which can be obtained by adding a vector unit (with a vector speed $V$) to a system with only a scalar unit (with a scalar speed $S$). The gain depends in principle on the ratio of the vector to scalar speed (given by $R = V/S$) and on the fraction $P$ of the program which is vectorised. The gain $G$ is given by:

$$G = \frac{V}{(1-P)\ V + PS}$$

Figure 5.2 displays the characteristic curve for ratios $R = 10$ and $R = \infty$ and table 5.1 gives some values, from Gentzsch & Neves (1988). According to Gentzsch & Neves the former ratio ($R = 10$) is typical of current vector/scalar technology. The latter ratio, represented by the dotted line in figure 5.2 is the impossible case of an infinitely fast vector unit. The horizontal axis represents the percentage of the computational work moved to the vector unit. Another look can be taken at the same phenomenon, at 75% vectorisation. At this percentage of vectorisation, the ability to infinitely increase the vector unit’s performance would improve overall performance from 3.1 times scalar speed to 4 times scalar speed. However, the striking result is
Table 5.1. Some points on the characteristic curves of the gain \( G \) as given by equation (5.4).

<table>
<thead>
<tr>
<th>( P )</th>
<th>0.25</th>
<th>0.50</th>
<th>0.65</th>
<th>0.75</th>
<th>0.85</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R = 10 )</td>
<td>1.3</td>
<td>1.8</td>
<td>2.4</td>
<td>3.1</td>
<td>4.3</td>
<td>10.0</td>
</tr>
<tr>
<td>( R = \infty )</td>
<td>1.3</td>
<td>2.0</td>
<td>2.9</td>
<td>4.0</td>
<td>6.7</td>
<td>\infty</td>
</tr>
</tbody>
</table>

that by increasing the percentage of vectorisation from 75% to 90% the performance increases from 3.1 times scalar to 5.26 scalar speed. This latter improvement is entirely an algorithm design issue. Nevertheless, the pay-off is better than that attainable by the suggested hardware improvement. This illustrates that architectural improvements are really only opportunities for performance improvements and that the opportunity is only realised with software that can take advantage of the hardware design. Amdahl’s law as stated above is, in some sense, only an ideal case. When one assumes the vector unit speed is, e.g. ten times the scalar speed, the implicit assumption is that the asymptotic performance is ten times faster than the scalar speed. As will be pointed out later the performance for short vectors can be very different.

5.3. Performance models for parallel computers

In this paragraph several fundamental aspects of performance of algorithms and performance evaluation of algorithms on parallel computers are discussed. The performance is evaluated by analysing the flow of data and control information in the computer system. On sequential computers performance evaluation is rather straightforward. The performance of the algorithms will be close to the theoretical peak performance. On parallel computers this changes drastically. The interaction between the architecture of the parallel computer and the algorithm involved has strong influence on the performance. Performance characteristics, such as utilisation, throughput and response time, give information about the efficiency of the hardware/software structure and allow the detection and elimination of bottlenecks. For example, the performance can be less than 10% of the peak performance and in most cases the performance is significantly smaller than the peak performance.

To analyse the performance of algorithms on parallel computers the two parameter characterisation \((n_{1/2}, r_{\infty})\) was introduced by Hockney, as described by Hockney & Jesshope (1981). These parameters are hardware parameters because they can be related to characteristics of the hardware. But in the evaluation of the performance of algorithms they also play an important role. The timing model is specially suited for describing the timing behaviour of SIMD computers.

A timing description which can also be applied to MIMD computers is given by Hockney (1984). This timing description is not discussed here, because the interaction between the processor boards is very weak. Therefore the performance is not effected by the processor board interaction. A generalised timing description is given by Hockney & Curington (1989). In this description also the effect of the bandwidth (the bandwidth of a system is defined as the number of operations performed per unit time, Hwang & Briggs (1984)) between the main memory and the vector registers (local memory) on the performance is taken into account. This improvement on the timing model is necessary because the memory bandwidth between the main memory and the
vector registers (local memory) often reduces the performance of the vector unit. Here a short summary of the general timing model is given. The model uses a number of parameters. The models can be used to evaluate basic vector operations such as vector add, vector multiply or SAXPY (evaluation of $aX + Y$). But the timing model can also be applied for the evaluation of less elementary operations such as the calculation of the coefficients of a finite difference equation. For more elementary and basic vector operations the parameters can be obtained by analysing the hardware architecture and the system software. When the timing model is applied to entire algorithms this can become very difficult. Therefore the parameters can be found by running several tests with varying parameters.

The following terms are used by Hockney and Curington (1989) in the analysis of the performance of a computer system.

\[ n \] Vector length.

\[ r_{\infty} \] Asymptotic performance of a pipeline that is approached as the vector length goes to infinity. Measured in floating-point operations per second in an arithmetic pipeline or in words transferred per second in a memory access pipeline.

\[ n_{5/6} \] Vector length required to achieve half the asymptotic performance, called the half-performance length.

\[ f \] Average number of floating-point operations performed per main memory reference, called the computational intensity.

\[ \bar{r}_{\infty}(f) \] Asymptotic performance of a combined memory and arithmetic pipeline, for a given value of \( f \).

\[ \bar{n}_{5/6}(f) \] Half-performance length of the combined memory and arithmetic pipeline, for a given value of \( f \).

\[ \hat{r}_{\infty} \] Peak performance, i.e. the limit of the asymptotic performance as \( f \) goes to infinity.

\[ f_{5/6} \] Value of \( f \) required to raise the asymptotic performance of the combined memory and arithmetic pipelines to half the peak value, called the half-performance intensity.

\[ r_{\infty}^{(m)}, r_{\infty}^{(a)} \] Asymptotic performance of the memory pipeline and the arithmetic pipeline, respectively.

\[ n_{5/6}^{(m)}, n_{5/6}^{(a)} \] Half-performance length of the memory pipeline and the arithmetic pipeline, respectively.

\[ t_m \] Time to transfer a block of \( n \) data between main and local memories.

\[ t_a \] Time for one vector operation of length \( n \) on data stored in local memory.

\[ \bar{t}(f) \] Average time for one vector operation of length \( n \), when there are \( f \) floating-point operations for every memory reference.
The following timing relations can be obtained:

\[ t_m = (n + n_Y^{(m)})/r_\infty^{(m)} \]  
(5.5)

\[ t_a = (n + n_Y^{(a)})/r_\infty^{(a)} \]  
(5.6)

\[ \bar{t} = (n + \bar{n}_Y)/\bar{r}_\infty \]  
(5.7)

The first timing model described by Hockney & Jesshope (1981) assumes that \( t_m << t_a \) (or \( r_\infty^{(m)} >> r_\infty^{(a)} \)). For a particular vector operation or for a number of operations the asymptotic speed \( \bar{r}_\infty (f) \) and the half-performance vector length \( \bar{n}_Y (f) \) can be found. These can be plotted in the two-parameter domain as given in figure 5.3.

![Figure 5.3. The two parameter domain.](image)

The performance point, illustrated in figure 5.3, has the following meaning: given an operation or algorithm that depends on "length" \( n \), the illustrated point indicates that the performance of the computer system for very large \( n \) approaches 30 Mflops, and for length \( n = 50 \), the performance would be 15 Mflops.

The timing model can be applied to many computer systems, and several two-parameter domains are given in Hockney & Jesshope (1981) and Gentzsch & Neves (1988). The assumption that \( t_m << t_a \) is not valid for most of the computer systems. The memory bandwidth between the main memory and the local memory influences the performance. Due to this effect \( r_\infty \) is smaller than \( \bar{r}_\infty \). Two cases are described by Hockney & Curington (1989), respectively sequential I/O and overlapped I/O. Here only overlapped I/O is discussed, because sequential I/O is of little importance for the DSNP.

For overlapped I/O the following timing formula is found:

\[ \bar{t} = \text{max}[t_m/f, t_a] \]  
(5.8)
Now two basic situations can be distinguished: either the I/O time can be dominant or the arithmetic time dominates. If the I/O time dominates the overall timing is:

\[ \bar{t} = \frac{t_m}{f} = \frac{(n + n_{i/2}^{(m)})}{(fr_{\infty}^{(m)})} \]  

(5.9)

If equation (5.9) is compared to equation (5.7) it is found that \( \bar{r}_{\infty} = fr_{\infty}^{(m)} \), \( \bar{n}_{i/2} = n_{i/2}^{(m)} \). The other case is that the arithmetic time dominates. There the timing is:

\[ \bar{t} = t_a = \frac{(n + n_{i/2}^{(a)})}{r_{\infty}^{(a)}} \]  

(5.10)

When this is compared with equation (5.7) one finds that: \( \bar{r}_{\infty} = r_{\infty}^{(a)} \), \( \bar{n}_{i/2} = n_{i/2}^{(a)} \). In order to find out whether the I/O or the arithmetic dominates, one must examine the break-even vector length, \( n_{br} \), at which I/O and arithmetic take equal times. This occurs when \( t_m/f = t_a \), with equation (5.5) and equation (5.6) it is found that:

\[ n_{br} = \frac{(n_{i/2}^{(m)} - z n_{i/2}^{(a)})}{(z - 1)} \] \(, z = \frac{fr_{\infty}^{(m)}}{r_{\infty}^{(a)}} \)  

(5.11)

There are three situations possible: I/O dominates, arithmetic dominates and I/O or arithmetic dominates depending on the vector length. In the following chapter the performance model which is described above is applied to the DNSP. In the chapter on the 3D implementation a two parameter domain for several tested supercomputers is given.
6. BASICS OF THE IMPLEMENTATION

6.1. Timing considerations of the data transport in the DNSSP
In this paragraph several of the standard data transport routines and their timing are analysed. First the asymptotic behaviour is discussed which is prescribed by the hardware and next several specific (software) routines which have been designed are analysed. Data can be transported between a number of memories: host <-> MM, MM <-> LM, MM <-> XM, LM <-> XM, XM <-> XM, LM <-> LM, LM <-> SM, XM <-> SM, LM <-> LM between processor boards and XM <-> XM between processor boards. Timing of the data transport routines, which have been designed for this purpose, is discussed below.

Data can be transported in several forms and quantities. It is of course possible to send only one value of a variable where a variable is one 32 bits word for single precision, or two 32 bits words for double precision. Further routines are designed which transport a number of variables between two memories or inside one memory. The address (the source address) of the first variable is given together with a fixed increment (the source increment), which is used to find the next variable. Furthermore, the first address of the new locations (the destination address) is given together with a fixed increment (the destination increment), which is used to find the next new location. Finally, the number of floating-point numbers (n) which are transported is given. The floating-point numbers are always 64 bits which means that always two 32 bits words are transported.

Table 6.1. Data transport between the large memory and the host computer.

<table>
<thead>
<tr>
<th>the number of 32 bits words</th>
<th>without disk access (sec)</th>
<th>with disk access (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^1$</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>$10^2$</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>$10^3$</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>$5 \cdot 10^3$</td>
<td>0.3</td>
<td>0.5</td>
</tr>
<tr>
<td>$10^4$</td>
<td>0.3</td>
<td>0.6</td>
</tr>
<tr>
<td>$5 \cdot 10^4$</td>
<td>0.5</td>
<td>7.8</td>
</tr>
<tr>
<td>$10^5$</td>
<td>0.7</td>
<td>18.4</td>
</tr>
<tr>
<td>$5 \cdot 10^5$</td>
<td>2.3</td>
<td>86.1</td>
</tr>
<tr>
<td>$10^6$</td>
<td>3.9</td>
<td>176.0</td>
</tr>
</tbody>
</table>

For data transport between the host and the large memory in all cases the control program (Van Delft (1991)) is used. Loading of the data costs more time than reading because the words are loaded into the large memory and next read back to check the loading. Some tests have been done with the reading of the data. Two cases were considered; with disk access and without disk access. In table 6.1 the results are
Table 6.2. The timing of data transport between the LM memory and the XM memory.

<table>
<thead>
<tr>
<th>routine</th>
<th>incr</th>
<th>n = 0</th>
<th>n = 1</th>
<th>n ≥ 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lm -&gt; xm and</td>
<td>0</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td></td>
<td>2 + 6n</td>
</tr>
<tr>
<td>xm -&gt; lm</td>
<td>i</td>
<td>4</td>
<td></td>
<td>8 + 7n</td>
</tr>
</tbody>
</table>

Table 6.3. The timing of data transport in the XM memory.

<table>
<thead>
<tr>
<th>routine</th>
<th>incr</th>
<th>n = 0</th>
<th>n = 1</th>
<th>n ≥ 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>xm -&gt; xm</td>
<td>0</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td>4 + 5n</td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>6</td>
<td></td>
<td>8 + 5n</td>
</tr>
</tbody>
</table>

given. One finds that the data transfer time is influenced enormously by the disk access time. The factor between the two is about 45 for a large number of words transferred.

Transport between the large memory and the LM or XM memory is always carried into effect by the general data transport routines. In most situations a large number of 32 bits words are transported. Some tests have been done to find the timing behaviour of this transport routine. For transport cost (in X-cycles) between the large memory and LM it is found that:

\[
cost = 276 + 44 \, n
\]  

(6.1)

where \( n \) is the number of 64 bits words. For transport cost (in X-cycles) between the large memory and XM it is found that:

\[
cost = 277 + 26 \, n
\]  

(6.2)

The transport with XM is just 1.7 times faster than LM transport because synchronisation plays an important role. It is possible to decrease the data transport cost by using more specific routines, in which several general possibilities are deleted. These specific routines have not been designed because data transport between the large memory and XM or LM is significantly smaller than other data transport.

In table 6.2 the data transport cost between LM and XM are shown. It can be seen that data transport with an increment of one in XM and LM is faster than data transport with a fixed increment \( i \) which is higher than one. The transport cost for the increment one is exactly as expected from the hardware timing. There is no overhead due to operations in XP.

The timing of data transport between XM and XM is given in table 6.3. Transport in XM is sometimes needed when data has to be rearranged but it is not of great importance for the application discussed here. The cost for data transport are higher than expected from the hardware (five instead of four) because there is overhead due to calculation of new addresses and counting of the number of words which have been
transported.

For the transport cost between XM and SM a general relation can be obtained. This relation is based on a subroutine which uses a special database structure. It is assumed that a number of arrays (given by \( N_{\text{var}} \), with a stride one) have to be transported to the SM memories. A list of pointers is constructed which contains pointers to the current position in each of the arrays. This list of pointers is used during the data transport to keep track of the locations in the array. After data transport the pointers are updated. Because one side of the SM memories can only contain sixteen 64 bits words, a restricted number of values (given by \( N_{\text{val}} \)) of each array can be transported. Therefore there is the constraint:

\[
N_{\text{var}} N_{\text{val}} \leq 16 \tag{6.3}
\]

Further, it should be decided how many SM memories (processors) are used (given by \( N_{\text{sm}} \)). For the cost (in X-cycles) for transporting \( N_{\text{var}} \times N_{\text{val}} \times N_{\text{sm}} \) 64 bits words from the XM memory to the SM memories (or vice versa) it can be found that:

\[
\text{cost} = 3 + N_{\text{var}} (2 N_{\text{val}} N_{\text{sm}} + 3) \tag{6.4}
\]

The cost is built up in the following way. To every SM memory \( N_{\text{val}} \) values of one variable are transported after each other, taking in total \( 2 \times N_{\text{val}} \times N_{\text{sm}} \) X-cycles. After this data transport the pointer in the pointer-list is updated, taking three X-cycles. This procedure is repeated for each variable pointed by the list of pointers (multiplying by \( N_{\text{var}} \)). Before data transport the pointer-list should be found and the first pointer loaded into XP, which costs three X-cycles. For transport cost (in X-cycles) between LM and SM, with the same database structure, it can be found that:

\[
\text{cost} = 6 + 6 N_{\text{var}} N_{\text{val}} N_{\text{sm}} \tag{6.5}
\]

The overhead of updating the pointers in the pointer-list is absent. It can be done during the data transport.

The last data transport routines to be discussed are those for data transport between neighbouring boards. Three specific routines for data transport between neighbours are described here. It is assumed that the data is stored concatenated. Data is transported from every processor board to one of the neighbouring boards. For data transport (in X-cycles) between LM memories it can be found that:

\[
\text{cost} = 3 + 60 n \tag{6.6}
\]

Rather a large time is needed due to the fact that between every transport of a 32 bits word the processor boards are synchronised. This is necessary because of the refreshing of the LM memory. During refreshing, the clock of the LM memory is kept fixed. This happens asynchronous. A way to reduce the overhead due to synchronisation is to move the data from the LM memory on one processor board to the LM memory on another processor board via the XM memories. First the data is moved from the LM memory to the XM memory. Next the data is moved to the XM memory on the neighbouring board and finally the data is moved from the XM memory to the LM memory. Synchronisation only has to be done once before the data transport between the processor boards. When large numbers of data are transported the data is partitioned into several blocks. For the data transport between LM memories via XM
memories it can be found that:

\[
\text{cost} = 38 + 18 \, n \quad (6.7)
\]

The synchronisation overhead is moved to the constant part. The cost is built up of six X-cycles \(LM \leftrightarrow XM\), six X-cycles \(XM \leftrightarrow XM\) (this could be reduced to four X-cycles, but this has not been done) and six X-cycles \(XM \leftrightarrow LM\). For the transport cost (in X-cycles) between XM memories it can be found that:

\[
\text{cost} = 30 + 6 \, n \quad (6.8)
\]

In this case synchronisation has to be done once and next data can be transported without overhead.

The timing for the data transport routines given above will be used for the analysis of the calculation strategy and the analysis of the calculation with four processors.

6.2. General calculation strategy

In the preceding paragraphs some basics have been discussed of the DNSP timing and the timing behaviour of several specific data transport routines. These timing figures are used in this paragraph and the next paragraph to make clear why we have chosen for the particular calculation strategy and database structure. Data transport cost have a large influence on the calculation strategy. One of the first questions to be asked apart from the database structure is on which level data should be stored. Should data be kept on the host computer and simple (vector) operations distributed on the DNSP, should data be kept in the large memory on the DNSP, or should data be distributed over the processor boards. This basic choice has a large influence on the structure of the calculation process, the efficiency of the calculations and the data transport. The data transport of the 3D algorithm is taken as example. In chapter 3 it was shown that about 1150 floating-point operations are performed to solve the difference equations of the variables for one grid point in the calculation domain (a division is counted as five multiplications). A number of variable fields are needed for this calculation. It was shown that there are twelve; i.e. \(u, v, w, p, T, k, \varepsilon, \mu_1, \mu_2, p, du, dv\). For data transport only the fields of order \(O(nx \, ny \, nz)\) are counted. Assume case 1: data is stored on the host computer and only simple calculations are performed on the DNSP. For instance a 3D vector is added to another 3D vector. Transport time between the host computer and the large memory is (assuming sufficiently large fields, \(10^5\) 32 bits words):

\[
\text{transport time} = N\text{var} \, nx \, ny \, nz \times 2 \times 1.84 \cdot 10^{-4} \, \text{sec.}
\]

Next, data should be distributed over the four processor boards (assume straight away to XM), with equation (6.2) it can be found that:

\[
\text{transport time} = 4 \, N\text{var} \, (277 + 26 \, nx \, ny \, nz / 4) \times 10^{-7} \, \text{sec.}
\]

For data transport between XM and SM, \(Nsm = 4, Nval = 1, N\text{var} \leq 16\), four processor boards parallel, with equation (6.4) it can be found that:

\[
\text{transport time} = \left(\frac{nx \, ny \, nz}{4 \times 4}\right) (3 + N\text{var} \, (2 \times 4 + 3)) \times 10^{-7} \, \text{sec.}
\]
The cost in equation (6.4) is multiplied by \((nx \, ny \, nz)/16\) because data transport is done on the four processor boards and equation (6.4) is for four values of one variable. For the total time for data transport, between the host computer and SM via the large memory, LM and XM, it is found that:

\[
\text{transport total} = N\text{var \, nx \, ny \, nz} \times 3.68 \cdot 10^{-4} + \\
N\text{var \, nx \, ny \, nz} \times 26 \cdot 10^{-7} + \\
N\text{var \, nx \, ny \, nz} \times 11/16 \times 10^{-7} = \\
N\text{var \, nx \, ny \, nz} \times 3.68 \cdot 10^{-4} \text{ sec.}
\]

Of course data should also be transported back to the host:

\[
\text{transport total} = (N\text{var}_w + N\text{var}_r) \, nx \, ny \, nz \times 3.68 \cdot 10^{-4} \text{ sec.}
\]

Suppose every \(N\text{var}_w\) leads to an operation; 50% efficiency for the operations:

\[
\text{calc time} = \frac{N\text{var}_w \, nx \, ny \, nz}{16} 4 \times 2 \times 10^{-7} = \\
N\text{var}_w \, nx \, ny \, nz \times 0.5 \cdot 10^{-7} \text{ sec.}
\]

In the calculation above the number of floating-point operations \((N\text{var}_w \times nx \times ny \times nz/16)\) is multiplied with the number of X-cycles per floating-point operation (4), the efficiency (1/0.5) and the number of seconds per X-cycle. The ratio between the data transport and the floating-point calculations is given by \((N\text{var}_r = 0)\):

\[
\text{ratio} = \frac{(N\text{var}_w + N\text{var}_r) \, nx \, ny \, nz \times 3.68 \cdot 10^{-4}}{N\text{var}_w \, nx \, ny \, nz \times 0.5 \cdot 10^{-7}} \approx 7.32 \cdot 10^3
\]

The ratio between the data transport and the floating-point calculations shows that the processing speed for floating-point operations and data transport is out of order. It can be concluded that the first case is not applicable to the DNBP.

For the second case only data transport between the large memory and XM, XM and SM is of importance:

\[
\text{transport total} = N\text{var \, nx \, ny \, nz} (26 \cdot 10^{-7} + 11/16 \times 10^{-7}) \approx \\
N\text{var \, nx \, ny \, nz} \times 2.7 \cdot 10^{-6} \text{ sec.}
\]

For the ratio between the data transport and the floating-point calculations can be found that \((N\text{var}_r = 0)\):

\[
\text{ratio} = \frac{(N\text{var}_w + N\text{var}_r) \, nx \, ny \, nz \times 2.7 \cdot 10^{-6}}{N\text{var}_w \, nx \, ny \, nz \times 0.5 \cdot 10^{-7}} \approx 54
\]

Still data transport is much slower than the speed of the floating-point operations.

For the third case the following ratio between the data transport and the floating-point calculations is obtained:

\[
\text{ratio} = \frac{(N\text{var}_w + N\text{var}_r) \, nx \, ny \, nz \times 11/16 \times 10^{-7}}{N\text{var}_w \, nx \, ny \, nz \times 0.5 \cdot 10^{-7}} \approx 11/8
\]
This is close to the required situation. In fact the analysis given here is not really correct anymore because of the small difference between the floating-point calculations and data transport. But it is clear that data should be distributed over the processor boards. It is not clear from the analysis above which value should be chosen for Nvar, low or high. This follows later when a more detailed analysis of the data transport between XM and SM and the floating-point calculations is given. Of course the results found above follow from the architecture and compiler used, so in fact the designer of those has deliberately chosen for a distribution of the data over the processor boards of the DNSP.

6.3. General database structure
As discussed in the preceding paragraphs during the execution of the program on the DNSP all the variable fields, constants and other floating-point data, have to be stored in the XM and the LM memory of the processor boards. Further, some distribution method should be used to distribute the calculation process over the four processor boards. As discussed later a method is used in which the calculation domain is divided in a number of sub-domains. All data for one sub-domain is stored on one processor board.

The data in the XM memory and the LM memory can be stored in very different ways, which greatly influences the speed of data transport and the complexity of the processor software which has to be written. The access time difference between the XM memory and the LM memory suggests storing frequently used data in the XM memory and bulk data in the LM memory. So in fact all data for the calculations is temporarily stored in the XM memory. During and between the calculations data is transported between the XM memory and the LM memory. The storage of data in the SM memory is also of importance but rather straightforward. We have decided to use a line structure for the database in the XM memory and the LM memory. There are several reasons why the line structure should be chosen. First it should be noted that the database in the XM memory is not coupled with the database of the SM memory because the storage in the SM memory is influenced by the structure of the calculations and the programmability of the floating-point routines. One of the basic reasons to use the line structure is that the algorithm for solving the Navier-Stokes equations has a line structure. Because of this 1-1 relation data transport on several levels (both in the machine and in the software) can be fast and easily implemented. For example during data transport between the XM memory and the SM memory the source and destination locations in the XM memory can be incremented with one. This can be done in the address counter for the XM memory (xmac) without using the 16 bits processor XP. Because it is expected that the data transport is the bottleneck, efficient data transport can lead to efficient calculations. Furthermore, the line structure is an efficient way of storage. So in the line structured database a line of one variable on one position is considered to be a unity. In the LM memory the lines of one variable are stored one after the other. Data transport of the line can be done fast because increments can be done in the address counters of the XM memory and the LM memory (xmac and lmac). Further, in the LM memory some grid information and some boundary information can be stored. In the XM memory the database information is stored. In figure 6.1 a layout of the database structure is given (part of the structure has been suggested by Van Delft (1991)).
Figure 6.1. The database structure of XM.

There are three levels in the database structure, as depicted in figure 6.1. The first level is farthest to the left containing initial (basic) pointers. The second level contains pointers to data (tables of pointers and pointer-lists) and the third only data (data areas, the column farthest to the right). The first positions in the XM memory are reserved for pointers to the tables with pointers and pointer-lists. A distinction is made between tables and pointer-lists although they both contain pointers. The difference is that the tables are used for storage purpose while the pointer-lists are used for data transport purpose. Furthermore, two pointers on the first locations point to the data area of the integer constants (DATA_I_CONST) and the data area of the floating-point constants (DATA_R_CONST). Two of the tables (TABLE_I_LM and TABLE_R_LM) contain pointers to locations in the fields in the LM memory. In principle one pointer is enough for a field of one variable in the LM memory, because the lines are stored concatenated. But for convenience several other pointers are added to the table such as a pointer to the line which is calculated and pointers to overlap areas with other processor boards. Pointers in the tables TABLE_I_1... point to data fields in the data area DATA_I_TYPE1..., and pointers in the tables TABLE_R_1... point to the data fields in the data areas DATA_R_TYPE1... . For instance, a pointer
in TABLE_R_2 points to a line containing the \( u \) velocity in the data area DATA_R_TYPE4. The data area DATA_R_TYPE4 contains the \( u \) velocity on the lines \((ix-1, iz),(ix, iz),(ix+1, iz)\), where \( ix \) and \( iz \) are relative and are defined by the contents of two registers in XP. Three pointers in one of the tables point to the first position of each line. By rearranging the pointers in a table, the precedence of the lines can be changed. So by means of the pointer structure relative addressing is created in the XM memory. At the second level there is an area for pointers (LIST_POINTERS) to pointer-lists (in DATA_POINTERS). The pointers in LIST_POINTERS are stored on the least part and on the most part a fixed-point number is stored which gives the number of pointers in the pointer-list (in DATA_POINTERS). Pointers in these pointer-lists (in DATA_POINTERS) can point to TABLE areas or DATA areas. The pointer-lists are used in the data transport routines for data transport between the XM memory and the SM memory.

![Figure 6.2. The structure of the SM memory.](image)

In figure 6.2 the storage of the variables in the SM memory is shown. The constants in the SM memory which are used a number of times are stored at the end of the SM memories. These constants are distributed over the 'a' part and 'b' part of one SM memory. So for example two constants are placed on 'a31' and 'b31' and they use only one of the thirty-two (or sixteen for double precision) locations in the ping-pong SM memories. The transport of the constants to the four SM memories takes place simultaneously. Other input in the SM memory is placed in the first positions. During the execution of the floating-point routine the output is placed at the beginning of the SM memories. This special construction is chosen because the addresses in one part of the SM memory cannot be assigned absolutely.

### 6.4. Calculation procedure with four processors

Given the database structure in XM a calculation procedure should be found which is most efficient. A calculation procedure is constructed for a number of vector operations. It is assumed that these calculations do not contain recurrences. The calculation procedure with the four processors has been described in paragraph 6.1. The transport routines are based on the routines described in paragraph 6.1 and the database structure has been defined in paragraph 6.3. The number of X-cycles which are needed for the general do-loop structure are counted and are given by equation (6.9).
\[ nr\_cycles = 12 + setup\_w + trans\_w + \]
\[ \text{max}[\text{float/2}, 5 + setup\_r + trans\_w] + \]
\[ \left(\frac{ny}{Nsm\_Nval}\right) - 2 \times \]
\[ 3 + \text{max}[\text{float/2}, 5 + trans\_w + trans\_r] \]
\[ \text{max}[\text{float/2}, 2 + trans\_r] + trans\_r \]

with

\[ setup\_w = 6 + 6 Nvar\_w \]
\[ setup\_r = 6 + 6 Nvar\_r \]
\[ trans\_w = 3 + Nvar\_w (2 Nval Nsm + 3) \]
\[ trans\_r = 3 + Nvar\_r (2 Nval Nsm + 3) \]

where

\[ \lceil a \rceil \quad : \quad \text{the smallest integer greater than or equal to } a \]

\[ Nvar\_w \quad : \quad \text{the number of different variables which are written to SM} \]

\[ Nvar\_r \quad : \quad \text{the number of different variables which are read from SM} \]

\[ Nval \quad : \quad \text{the number of loops which are calculated with one processor} \]

\[ Nsm \quad : \quad \text{the number of SM memories (processors) which are actually used} \]

\[ \text{float} \quad : \quad \text{the number of F-cycles required to perform the floating-point routine} \]

\[ ny \quad : \quad \text{the loop length} \]

Equation (6.9) is analysed briefly. \textit{setup\_w} is the number of X-cycles which are needed to transform a static pointer-list with pointers pointing to table entries into a dynamic pointer-list with pointers pointing to lines in the data fields. The dynamic pointer-list has to be ready before data transport. Next the first four groups of data are transported to the SM memories (\textit{trans\_w}). At this stage the floating-point calculation is started. At the same time the dynamic pointer-list for reading is made (\textit{setup\_r}), followed by the transport of four groups of data to the SM memories. When both actions are finished (\text{max}), a new floating-point calculation is started, results are transported from SM to XM and new data is transported from XM to SM. This is repeated a number of times until four or less groups of data, which have to be calculated, are left. When the floating-point calculation of the last groups of data is started only results are transported from the SM memories to XM. After finishing on both sides of the processor board the last results are transported from SM to XM. It should be noted that it is possible that the last \textit{trans\_w} and \textit{trans\_r} cost less time than the others because only the remaining values have to be transported. Here it is assumed that the cost of the last transport are equal to the others. Substitution of equation (6.10) in equation (6.9) gives:
\[ nr\_cycles = 24 + 6\ Nvar\_w + (Nvar\_w + Nvar\_r)(2\ Nval\ Nsm + 3) + \\
max[\text{float}/2, 14 + 6\ Nvar\_r + Nvar\_w (2\ Nval\ Nsm + 3)] + \\
max[\text{float}/2, 5 + Nvar\_r (2\ Nval\ Nsm + 3)] + \\
\left[\left\lceil{\frac{ny}{Nsm\ Nval}}\right\rceil - 2\right] \times \\
\left[3 + \max[\text{float}/2, 11 + (Nvar\_w + Nvar\_r)(2\ Nval\ Nsm + 3)]\right] \]

(6.11)

More than one loop is calculated by one floating-point routine by setting \( Nval \) to 2, 3, etc. For \( Nval, Nvar\_w \) and \( Nvar\_r \) there are two constraints, as given by equation (6.12), resulting from the fact that one side of the SM memory has only sixteen 64 bits positions. It is assumed that there are no constants needed in the SM memory.

\[ Nvar\_w\ Nval \leq 16 \]

\[ Nvar\_r\ Nval \leq 16 \]  

(6.12)

So when \( Nvar\_w \) and \( Nvar\_r \) have a small value, \( Nval \) can be set to a value greater than one and data transport between XM and SM becomes faster (due to the smaller overhead). The number of F-cycles which are used by the floating-point routine is found by compilation of the floating-point program which gives the number of F-cycles. For large \( ny \) for the number of cycles per loop one finds that:

\[ \frac{nr\_cycles/\text{loop}}{Nsm\ Nval} = \frac{1}{Nsm\ Nval} \times \\
\left[3 + \max[\text{float}/2, 11 + (Nvar\_w + Nvar\_r)(2\ Nval\ Nsm + 3)]\right] \]

(6.13)

With equation (6.13) the timing of particular do-loops can be investigated.

For example the performance of a SAXPY on one processor board can be obtained. The SAXPY is defined by:

for \( i=1 \) to \( ny \) step 1 do

\[ z(i) = a \times x(i) + y(i) \]

enddo

where \( a \) is constant and \( x, y \) and \( z \) are vectors of length \( ny \). For the data transport and floating-point routine it can be found that; \( Nvar\_w = 2, Nvar\_r = 1, Nval = 7, Nsm = 4, \text{float} = 218 \). The number of X-cycles per loop for large \( ny \) can be found by applying equation (6.13):

\[ nr\_cycles/\text{loop} = \frac{1}{4 \times 7} \left[3 + \max[218/2, 11 + (2 + 1)(2 \times 7 \times 4 + 3)]\right] = 6.8 \]

The speed of the four processors can be obtained by dividing the number of operations per loop by the time needed for one loop: \( \bar{\tau}_\infty = 2 / (6.8 \times 10^{-7}) = 2.94 \) Mflops. For \( Nval = 1 \) for the asymptotic speed it is found that: \( \bar{\tau}_\infty = 1.7 \) Mflops. It follows from these results that for \( Nval = 7 \) the speed is 70% higher than for \( Nval = 1 \). Data transport cost are dominant so the speedup of the data transport is used to enhance the performance. For the vector operation \( z = x + y \) it is found that \( \bar{\tau}_\infty = 1.49 \) Mflops for
\( Nval = 8 \) and \( r_{\infty} = 0.85 \) Mflops for \( Nval = 1 \).

The preceding two examples show that the ratio of the number of operations performed and the number of variables that are transported is very important for the performance. Here some relations are given for the influence of \( Nvar_w, Nvar_r \) and \( Nval \) on the performance. Suppose that data transport is the bottleneck for the calculations. In that situation it can be found with equation (6.13) that:

\[
\frac{nr\_cycles}{loop} = \frac{14}{Nsm\ Nval} + \left(\frac{Nvar\_w + Nvar\_r}{Nsm\ Nval}\right) \left(2 + \frac{3}{Nsm\ Nval}\right) \tag{6.14}
\]

We are interested in finding the best way in which data transport can be performed. With \( Nsm = 4 \) and equation (6.14) the \( nr\_cycles/IO \) can be obtained:

\[
\frac{nr\_cycles}{IO} = \frac{3.5}{Nval\ (Nvar\_w + Nvar\_r)} + 2 + \frac{0.75}{Nval} \tag{6.15}
\]

This equation is analyzed subject to the constraints:

\( Nval\ Nvar\_w \leq 16 \) and \( Nval\ Nvar\_r \leq 16 \)

\((Nval + 1)\ Nvar\_w > 16 \) or \((Nval + 1)\ Nvar\_r > 16 \)

\( Nvar\_w \geq Nvar\_r, \ Nvar\_w \geq 2, \ Nvar\_r \geq 1, \ Nval \geq 1 \)

The minimum of the \( nr\_cycles/IO \) is 2.203 for \( Nval = 8, Nvar\_w = 2, Nvar\_r = 2 \) and the maximum is 3.10 for \( Nval = 1, Nvar\_w = 9, Nvar\_r = 1 \). In table 6.4 the results are summarized and in the squares the number of times that the situation occurs is given. Taking \( Nval \) greater than one reduces the overhead. This can be seen clearly in the table; when \( Nval \) is larger, the \( nr\_cycles/IO \) is smaller. There is a factor 1.41 between the minimum and the maximum. So when only basic operations are performed, \( Nval \) should be as large as possible. But in practice in many cases calculations can be combined giving a higher operation per I/O ratio and \( Nvar\_r \) is much lower. Suppose that four large vectors should be added. This can be done in two ways: add three times two vectors (IO: 9) or add the four vectors right away (IO: 5). For the first method the \( nr\_cycles/IO \) is the smallest but the second method as a whole is 1.7 times faster. The results of several other experiments with the 3D software for the DNSP has shown also that it is better to combine calculations than to choose a large value of \( Nval \).

There are a large number of do-loops in the Navier-Stokes program. We want to obtain a general relation for the cost of these do-loops because then the processing speed of the program can be estimated. To find this general relation equation (6.13) has to be applied to a sequence of do-loops:

\[
\frac{nr\_cycles}{point} = \sum_{i=1}^{nr\_loops} \frac{1}{Nsm\_i\ Nval\_i} \times \left[3 + \max\{float\_i/2, 11 + (Nvar\_w\_i + Nvar\_r\_i)(2\ Nval\_i\ Nsm\_i + 3)\}\right] \tag{6.16}
\]
Table 6.4. A summary of the equation (6.15).

<table>
<thead>
<tr>
<th>Nval</th>
<th>8</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{nr\_cycles/IO} = 2 \times 4 + 3 = 11
\]

For a large number of calculations the following is valid; \( Nsm_i = 4, Nval_i = 1, \)
\( float_i/2 \leq 11 + (Nvar\_w_i + Nvar\_r_i)(2 \times Nval_i Nsm_i + 3) \). Therefore, equation
(6.16) reduces to:

\[
\text{nr\_cycles/point} = \sum_{i=1}^{nr\_loops} \frac{1}{4} \left[ 3 + 11 + (Nvar\_w_i + Nvar\_r_i)(2 \times 4 + 3) \right] = \frac{14}{4} \text{nr\_loops} + \frac{11}{4} \sum_{i=1}^{nr\_loops} (Nvar\_w_i + Nvar\_r_i) \quad (6.17)
\]

Now several of the parameters which are defined by Hockney & Curington
(1989), following from paragraph 5.3, can be discussed. These parameters give a
better insight in the data transport and calculation characteristics of the processors
boards. First the arithmetic pipeline is discussed. It is assumed that the SM memories
and the floating-point chips are one unit. Thus when we refer to data transport we
mean transport between the XM memory and the SM memories. As discussed in the
paragraph on the software for the DNSP, the asymptotic speed of the arithmetic pipeline is:
\( r_{\infty}^{(a)} = 4 \times 2 /16 \text{ op/F-cycle} = 1 \text{ op/X-cycle} = 10 \text{ Mflops} \). To find \( n_2^{(a)} \) some
tests should be done because its value depends on the compiler of the floating-point
side. For the SAXPY operation, which is a combination of a multiplication and addition,
it is found that \( n_2^{(a)} \) is smaller than 1.

Next the memory pipeline is investigated. For data transport between XM and
SM equation (6.4) was obtained for transport of \( Nvar \times Nval \times Nsm \) words:

\[
\text{transport} = 3 + Nvar (2 \times Nval \times Nsm + 3)
\]

For the asymptotic speed of the memory pipeline \( (r_{\infty}^{(m)}) \) it can be found that:

\[
r_{\infty}^{(m)} = \frac{Nvar Nval Nsm}{2 \times Nvar Nval Nsm \times 10^{-7}} = 5 \cdot 10^6 \text{ words/sec}
\]
The half-performance length of the memory pipeline \( n_{\frac{m}{2}}^{(m)} \) is small due to the small overhead. It can be found that: \( n_{\frac{m}{2}}^{(m)} = 3 \).

On the processor board the combination of memory (the XM memory) and processors (the SM memories and the floating-point chips) is used. To give some idea of the interaction of the two, the asymptotic performance of the combined memory and arithmetic pipeline \( \bar{r}_\infty (f) \) and the half-performance length of the combined memory and arithmetic pipeline \( \bar{n}_{\frac{m}{2}} (f) \) should be obtained. For the computational intensity it can be found that:

\[
f = \frac{op}{mem~ref} = \frac{\text{float}}{8 \cdot (Nvar_w + Nvar_r)} \tag{6.18}
\]

This is based on the asymptotic speed of the floating-point processors:

\[op \approx \text{float}/8\tag{6.19}\]

Combining equation (6.13) and equation (6.18) gives:

\[
nr_{cycles/loop} = \frac{\max[3 + \text{float}/2,14 + (2 \cdot Nval \cdot Nsm + 3) \cdot \text{float}/8f]}{Nsm \cdot Nval} \tag{6.20}
\]

Assume that \( Nval = 1, Nsm = 4 \):

\[
nr_{cycles/loop} = \frac{1}{4} \max[3 + 4 \cdot op, 14 + 11 \cdot op/f] \tag{6.21}
\]

For the asymptotic speed it can be found that:

\[
\bar{r}_\infty (f) = \frac{4 \cdot 10^7}{\max[3/\text{op} + 4, 14/\text{op} + 11/f]} \tag{6.22}
\]

For \( op = 10 \); \( \bar{r}_\infty (1) = 3.22 \) Mflops. The peak performance can be found (assuming in equation (6.22) that 3/\text{op} + 4 \approx 14/\text{op} and 3/\text{op} \ll 4):

\[
\hat{r}_\infty = \lim_{f \to \infty} \bar{r}_\infty (f) = 8 - 10 \text{ Mflops}
\]

For \( \hat{r}_\infty \) only the startup of the processors is of importance. The answer is of course what was expected.

Next the half-performance length of the combined memory and arithmetic pipeline \( \bar{n}_{\frac{m}{2}} (f) \) should be found. In principle \( \bar{n}_{\frac{m}{2}} \) is a function of \( f \), but it can be shown that there is an upper-bound for \( \bar{n}_{\frac{m}{2}} \) independent of \( f \). An analysis and assuming that \( Nsm = 4 \) gives:

\[
\bar{n}_{\frac{m}{2}} (f) \leq 36/7 \cdot Nval
\]

In most situations we choose \( Nval = 1 \), which gives \( \bar{n}_{\frac{m}{2}} (f) \leq 36/7 \). This is a relative small value.

To finish the analysis the half-performance intensity \( f_{\frac{m}{2}} \) should be obtained. For the \( nr_{cycles/loop} \) with \( f \to \infty \) (peak rate) and equation (6.13) it can be found that:

\[
nr_{cycles/loop} = \frac{1}{Nsm \cdot Nval} \cdot \left[ 3 + \text{float}/2 \right]
\]

To find the half-performance intensity \( f_{\frac{m}{2}} \), solve:
\[ \begin{align*}
\frac{2}{\text{Nsm \ Nval}} &= \frac{\text{max}[3 + \text{float}/2, 14 + (N\text{var}_w + N\text{var}_r)(2 \text{ Nval \ Nsm} + 3)]}{\text{Nsm \ Nval}} \\
\end{align*} \]

By using equation (6.18) and the equation above it can be found that:

\[ f_{\text{vis}} = \frac{1}{\text{Nvar}_w + \text{Nvar}_r} + \frac{2 \text{ Nval \ Nsm} + 3}{8} \quad \text{(6.23)} \]

Suppose that \( \text{Nval} = 1, \text{Nsm} = 4 \) then \( f_{\text{vis}} = 11/8 + 1/(\text{Nvar}_w + \text{Nvar}_r) \leq 12/8 \). This is a relative small value. Hockney & Curington (1989) found \( f_{\text{vis}} = 2.2 \) for overlapped IO on the FPS 5000. The value for the DNSP is significantly smaller. Assuming that data transport is faster than the floating-point calculations, the peak rate \( (\hat{f}_{\text{peak}}) \) is obtained (with \( \text{Nsm} = 4 \)) if:

\[ f = \frac{11}{4(\text{Nvar}_w + \text{Nvar}_r)} + \frac{8 \text{ Nval} + 3}{4} \quad \text{(6.24)} \]

When \( \text{Nval} = 1 \): \( f \leq 11/4 + 11/32 = 99/32 \). It should be noted that in practice the value of \( f \) will be close to one, for example for the calculation of the coefficients of a finite difference equation. When variables in SM can be used more than once the value of \( f \) increases. It should be noted that constants can be loaded into the SM memories before the calculation is started. They can be used for every loop, so the number of operations increases while the memory references are constant.

6.5. Algorithms for solving tridiagonal matrices

In paragraph 3.2 the solution algorithm for the Navier-Stokes equations with a convection-diffusion equation added to it is discussed. The result is a system of difference equations which should be solved. The difference equations have the general form given by equation (3.5).

\[ \begin{align*}
\ell_j \phi_{j-1} + d_j \phi_j + r_j \phi_{j+1} &= \text{lhs}_j \quad j = 1, \ldots, N_2 \\
\ell_1 &= 0, \quad r_{N_2} = 0
\end{align*} \quad \text{(3.5)} \]

The matrix which represents equation (3.5) has a tridiagonal form. The equation (3.5) should be solved as fast as possible. If possible four processors should be used.

In literature several algorithms are proposed which can be used when there is more than one processor available. In this paragraph it is our object to find out which algorithm performs the best. Therefore, for each algorithm the timing should be obtained. For this purpose the two timing relations which were obtained in the preceding paragraph are used. The general timing behaviour of a do-loop is given by equation (6.11). It should be noted that the timing given by equation (6.11) consists of initialisation of the pointer-lists and the calculation procedure. Further the asymptotic behaviour of equation (6.11) was obtained given by equation (6.13).

In the following paragraphs it is assumed that the number of columns of the matrix can be divided by for instance 2 or 3, which is allowed by adding several rows and columns to the matrix. The enlargement of the matrix only slightly effects the efficiency because for the original matrix several processors would be idle during the last calculations of a do-loop. Furthermore, it is clear that adding several equations to
Table 6.5. The timing of the algorithms for the tridiagonal matrix on the DNP.

<table>
<thead>
<tr>
<th>method</th>
<th>Nsm</th>
<th>Nval</th>
<th>Nvar_(w)</th>
<th>Nvar_(r)</th>
<th>float</th>
<th>cost</th>
<th>bottleneck</th>
<th>const</th>
<th>op</th>
<th>(n_{50})</th>
</tr>
</thead>
<tbody>
<tr>
<td>thof1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>128/2</td>
<td>67</td>
<td>float</td>
<td>111</td>
<td>11</td>
<td>1.6</td>
</tr>
<tr>
<td>thof3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>330/2</td>
<td>56</td>
<td>float</td>
<td>133</td>
<td>11</td>
<td>2.4</td>
</tr>
<tr>
<td>thob1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>32/2</td>
<td>29</td>
<td>fixed</td>
<td>61</td>
<td>2</td>
<td>2.1</td>
</tr>
<tr>
<td>thob6</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>112/2</td>
<td>9.8</td>
<td>-</td>
<td>104</td>
<td>2</td>
<td>10.2</td>
</tr>
<tr>
<td>2sif1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>166/2</td>
<td>43</td>
<td>float</td>
<td>150</td>
<td>11</td>
<td>3.5</td>
</tr>
<tr>
<td>2sif3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>424/2</td>
<td>35.8</td>
<td>float</td>
<td>196</td>
<td>11</td>
<td>5.5</td>
</tr>
<tr>
<td>2sib3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>64/2</td>
<td>9.8</td>
<td>fixed</td>
<td>69</td>
<td>2</td>
<td>7.0</td>
</tr>
<tr>
<td>wanf1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>140/2</td>
<td>22.75</td>
<td>fixed</td>
<td>127</td>
<td>13</td>
<td>5.6</td>
</tr>
<tr>
<td>wanf3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>388/2</td>
<td>16.9</td>
<td>fixed</td>
<td>259</td>
<td>13</td>
<td>15.3</td>
</tr>
<tr>
<td>wanb1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>136/2</td>
<td>20.0</td>
<td>fixed</td>
<td>102</td>
<td>5</td>
<td>5.1</td>
</tr>
<tr>
<td>wanb3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>220/2</td>
<td>14.66</td>
<td>fixed</td>
<td>99</td>
<td>5</td>
<td>6.7</td>
</tr>
<tr>
<td>want1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>wans1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>40/2</td>
<td>14.5</td>
<td>fixed</td>
<td>16</td>
<td>4</td>
<td>1.1</td>
</tr>
<tr>
<td>wans3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>96/2</td>
<td>10.16</td>
<td>fixed</td>
<td>25</td>
<td>4</td>
<td>2.5</td>
</tr>
<tr>
<td>cycf1</td>
<td>4</td>
<td>1</td>
<td>12</td>
<td>4</td>
<td>-</td>
<td>47.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cyctb2</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>-</td>
<td>18.4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.6. The timing for the different methods.

<table>
<thead>
<tr>
<th></th>
<th>(1/\tilde{F})</th>
<th>(n_{1/2})</th>
<th>(n_{0.9})</th>
<th>(n=100)</th>
<th>(n=50)</th>
<th>(S_4)</th>
<th>op</th>
<th>Mflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thomas</td>
<td>65.8</td>
<td>3.6</td>
<td>32.4</td>
<td>6817</td>
<td>3527</td>
<td>1.0</td>
<td>13</td>
<td>2.0</td>
</tr>
<tr>
<td>2sides</td>
<td>45.6</td>
<td>5.8</td>
<td>52.2</td>
<td>4825</td>
<td>2545</td>
<td>1.44</td>
<td>13</td>
<td>2.9</td>
</tr>
<tr>
<td>cyclic red.</td>
<td>65.9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.0</td>
<td>20</td>
<td>2.0</td>
</tr>
<tr>
<td>Wang</td>
<td>41.8</td>
<td>16.8</td>
<td>151.2</td>
<td>4882</td>
<td>2792</td>
<td>1.57</td>
<td>22</td>
<td>3.3</td>
</tr>
</tbody>
</table>

the system is easier than writing a general routine for several matrix sizes.

6.5.1. Gauss elimination with one processor (Thomas algorithm)

Here the sequential reference algorithm, based on Gauss elimination, is discussed. It is the fastest possible algorithm on one processor. The Gauss elimination algorithm on one processor is given by algorithm 6.1 in the Appendix. The Gauss elimination algorithm is implemented on the DNP. As mentioned above only one processor is used because the algorithm contains several recurrences. The equation (6.11) is not used because pointers are only initialised once for the two do-loops, reducing the overhead considerably. Therefore, the analysis has been done somewhat different, with a modification of equation (6.11). Equation (6.13) can be used because it is an asymptotic relation which is still valid. The results of the two analyses are given in table 6.5 and table 6.6. The algorithm is split into two loops (forward and backward); two cases are considered as well for the forward loop as for the backward loop. For the operation count for the forward loop is found: 11 (the division is counted for five) and for the backward loop: 2. For the forward loop one should note that the floating-point calculations are the bottleneck. This is due to the fact that only one processor is used, therefore data transport is fast enough. The number of X-cycles for the forward loop
for \( Nval = 1 \) and \( Nval = 3 \) are counted. The situation for \( Nval = 3 \) has a larger asymptotic speed but also a larger half-performance length (\( \bar{n}_{\frac{1}{2}} \)). However, the value of \( \bar{n}_{\frac{1}{2}} \) is small enough to have no negative influence. For the back substitution \( Nval = 1 \) and \( Nval = 6 \) is examined, \( Nval = 6 \) has a higher asymptotic speed. As shown in table 6.6 in total 65.8 X-cycles are needed, giving a speed of 2.0 Mflops. The 90% performance vector length is 32.4 which shows that in practical situations the speed is close to the asymptotic speed.

6.5.2. The cyclic reduction and recursive doubling algorithm

An algorithm for cyclic reduction is described by Heller (1976). The recursive doubling algorithm described by Stone (1973) is not discussed here because it has the same characteristics on the DNSP as the cyclic reduction algorithm. It should be noted that for the recursive doubling algorithm \( O(N\log_2 N) \) operations are needed compared to \( O(N) \) for the cyclic reduction, as given by Heller (1978). For the cyclic reduction algorithm rewrite equation (3.5) as:

\[
\begin{align*}
& l_{j,0} \phi_{j-1} + d_{j,0} \phi_j + r_{j,0} \phi_{j+1} = l h s_{j,0} & j = 1, ..., N_2 \\
& l_{1,0} = 0, r_{N_2,0} = 0
\end{align*}
\]

The cyclic reduction algorithm is given by algorithm 6.2 in the Appendix. Now the timing should be obtained for the cyclic reduction algorithm. It is assumed that the reduction step and back substitution can be done in one do-loop. This can probably be done because of the special database structure which can be used on the DNSP. Furthermore, it is assumed that data transport is the bottleneck, which is reasonable in cases where four processors are used. The number of operations which have to be performed is given by:

\[
\begin{align*}
& \sum_{l=1}^{\log_2 N} \left[ \frac{N}{2^l} \text{ divide } + 4 \text{ plus } + 8 \text{ multiply} \right] + \\
& \sum_{l=0}^{\log_2 N + 2^k} \left[ \frac{N+2^k}{2^{k+1}} \text{ 2plus } + 3 \text{ multiply} \right] \approx N \left[ \text{ divide } + 6 \text{ plus } + 11 \text{ multiply} \right]
\end{align*}
\]

With equation (6.13) the asymptotic speed is obtained, the results are given in table 6.5 and table 6.6. The total cost is 65.9 X-cycles which is almost equal to the Gauss elimination with one processor. For the processing speed 2.0 Mflops is found. The half-performance vector length is not obtained because the asymptotic equation (6.13) is used. Because the cyclic reduction algorithm performs not well, it is not used.

6.5.3. The elimination from two sides

An algorithm for Gauss elimination from the bottom and the top of the tridiagonal matrix (called twisted factorisation) is described by Van der Vorst (1987A). It has the same computational complexity as the standard Gauss elimination with one processor. One of the processors starts the Gauss elimination at the top and the other at the bottom of the matrix. During the eliminations the first and second processor arrive at two neighbouring points in the middle of the matrix, next a \( 2 \times 2 \) matrix is solved and the elimination can be finished. The algorithm for the elimination from two sides is given by algorithm 6.3 in the Appendix. According to Van der Vorst (1987A) such
factorisation exists when the matrix is symmetric positive definite, an M matrix or when it is diagonally dominant. As shown in paragraph 3.1 the matrix is diagonally dominant. The timing for the elimination from two sides is given in table 6.5 and table 6.6. No use is made of equation (6.11) but the operations are merely counted. A few things should be noted. The pointer-lists for data transport are only made once. The data transport routines which are used are special because the lines in XM are scanned from the first position and the last position. However, the data transport speed is equal to the speed for the general routines. The solution of the $2 \times 2$ system is computed by both processors, only four floating-point numbers should be moved from one SM to another SM. As with the standard Gauss elimination algorithm the floating-point calculations are the bottleneck for the first loop. For the second loop data transport is the bottleneck. For both loops $Nval = 3$ is taken. For the second loop $Nval = 6$ could also be chosen but this gives only a slight improvement in asymptotic speed (7.9 X-cycles instead of 9.8 X-cycles) but $\bar{n}_{\nu}$ would be much larger. For the loops together we find 45.6 X-cycles/loop giving a speed of 2.9 Mflops. The 90% performance vector length is 52.2 which is still acceptable.

6.5.4. The Wang partition method
An algorithm for solving tridiagonal matrices parallel with more than two processors is described by Wang (1981). The tridiagonal matrix is split into four parts (for the processor configuration discussed here) which are calculated parallel on the four processors. For the implementation a modification of the Wang algorithm described by Michielse & Van der Vorst (1988) is used. This modification ensures a better distribution of the calculation process over the four processors. The modified Wang algorithm is given by algorithm 6.4 in the Appendix. A better understanding of the Wang algorithm can be obtained in the following way, according to Lin (1989). The matrix describes the relation between the grid points on a line and between the two neighbouring lines. Suppose there are $N_2$ relations, the line is divided into four sub-domains. The four boundary points are chosen at $N_2/4$, $N_2/2$, $3N_2/4$, $N_2$. Now the difference equation (3.5) can be written as:

$$A_I \phi^I + A_{IB} \phi^B = f^I$$
$$A_B \phi^I + A_B \phi^B = f^B$$

(6.26)

Where $\phi^I$ is the solution on the interior points and $\phi^B$ on the boundary points. $A_I^{-1}$ can easily be computed because of the tridiagonal structure. By applying block Gauss elimination the following system for the boundary points is obtained:

$$C \phi^B = g$$

(6.27)

where

$$g = f^B - A_{BI} A_I^{-1} f^I$$
$$C = A_B - A_{BI} A_I^{-1} A_{IB}$$

(6.28)

Once equation (6.27) is solved, the original problem is decoupled and the solution $\phi^I$ at the sub-domains can be computed by solving four independent subproblems of the type:
\[ A_{i}, \phi_i^I = f_i - A_{iB}, \phi_i^B \quad i = 1, 2, 3, 4 \]  

This is nothing more than solving \( \phi_i^I \) on each sub-domain where the values \( \phi_i^B \), which were already computed, are the appropriate boundary conditions.

The cost of the Wang algorithm on the four processors are analysed. The results are given in table 6.5 and table 6.6. For data transport special routines are designed. Data should be transported between four positions in the line and the four SM memories. For the first loop (wanf3) \( Nval = 3 \) is chosen, which means that the matrix size should be a multiple of 12. After the first loop the pointers are set back two positions and the back substitution can be performed (wanb3). The second loop and the third loop are combined. In the third part (want1) the \( 4 \times 4 \) matrix is solved and values are put back to the other SM memories. Then the pointers are initiated for the last loop calculation. In the fourth part (wans3, last do-loop) the solution is obtained. As can be seen the first, the second and the fourth loop can be performed parallel on the four processors. The total number of X-cycles is 41.8 but the Wang method has a large constant part resulting in a large half performance vector length. For the Wang method a speed of 3.3 MFlops (13/41.8 \( \times 10 \)) is found. For the Wang method about 1.7 times as much operations are performed, so in fact the speed is 5.3 MFlops which is relatively large (about 53% of the peak performance). In Wang (1981) a too high operation count is given: 21 operations for Wang instead of our 18 and 20 for cyclic reduction instead of our 18. Also too many divisions are used, which for most of the computing systems cost more than a multiplication or addition (subtraction).

6.5.5. Conclusions
In the preceding paragraphs the timing of several algorithms is analysed and the results are given in table 6.6. Two of the methods should be preferred, namely the elimination from two sides or the Wang partition method. For the timing of the Wang algorithm it is found that:

\[ cost = 705 + 41.7 \, ny \]

For the timing of the elimination from two sides it is found that:

\[ cost = 265 + 45.6 \, ny \]

The two methods have the same cost for \( ny = 113 \). For \( ny < 113 \), the elimination from two sides is used, because the start-up is smaller than the start-up of the Wang method. For \( ny \geq 113 \) the Wang method is used.

The stability of the two algorithms is now briefly discussed. Both algorithms do not include pivoting. Therefore, for a general tridiagonal system the methods might fail. However, according to Wang (1981) the partition method is stable for diagonally dominant systems. As noted in paragraph 3.1 the system is diagonal dominant. Van der Vorst (1987B) analysed the stability of the elimination from two sides for a symmetric tridiagonal matrix. For a special case stability is proved. It is assumed that diagonal dominancy is sufficient for the elimination from two sides to be stable.
7. THE PROGRAMMING ENVIRONMENT

7.1. Implementation, programming techniques (do-loops, do-loops with if-statements)
In this paragraph the fundamental aspects of the implementation of the 2D and 3D Navier-Stokes algorithm are discussed. This deals with the construction of the database, the implementation of the do-loops and other calculations.

During the implementation mnemonic names are used for registers and integer constants. For the mnemonic names the following arrangement is made:

- **Cname** a fixed-point number used for addressing in the XM memory, the LM memory and the large memory.
- **CCnumber_name** a fixed-point number (e.g. CCONE value 1).
- **Wregister_name** a register (e.g. WCOUNT value w10).
- **WP_pointer_name** a register which contains a pointer (e.g. WP_POINTER value w9).

The use of mnemonic names improves the surveyability of the assembler program and therefore reduces errors during the implementation. Before starting the implementation, the database for the XM, LM and the large memory is constructed. The database has the general structure as given in paragraph 6.3. First the different types that are used have to be inventoried. Next, a definition file is created which defines the relations between the TABLE areas, DATA areas and POINTER-LISTS within the database. Furthermore, a file is made which contains a list of specifications of all the variables. For every variable its type has to be given and the length (for arrays) if needed. By combining the definition file and the file with the specification of the variables using the m4 preprocessor, Kernighan & Ritchie (1978), a new definition file is created with the definitions of the mnemonic names. In the definitions of the mnemonic names the structure of the whole database is contained. There are mnemonic names for the start positions of the TABLE areas, the offsets for variables in TABLE areas, the start locations of DATA areas for all the variables, the start location of the area LIST_POINTERS, the offsets for pointers in the LIST POINTER area, the start positions of pointer-lists in the data area DATA_POINTERS, the number of pointers in the TABLE areas, the number of variables of the different types, the size of the DATA areas, etc.

As noted before the assembler language that is used is rather primitive. Due to this fact the implementation is difficult and errors are easily made. As a result the implementation is time consuming. The use of mnemonic names is a first step to ease the implementation. A further step is made by noting that groups of assembler code are used more than once. The first suggestion is then of course that a subroutine can be used. In many cases this is possible, reducing the programming effort considerably.
But when in the assembler lines one or more mnemonic names are used that are different every time, a subroutine cannot be used. In these situations, which often occur, a so-called macro is used. The macro is a pre-defined mnemonic name, which in this case is defined to be a number of assembler lines. A number of variables can be used in the macro definition, the arguments of the macro are substituted at the appropriate place. In this way large structures with slightly different code, that are often used, are implemented easily. Furthermore, errors are avoided because the macros only have to be written once and tested once. There exist a large number of macros for general loop structures, for various kinds of do-loops, for obtaining addresses of various variables, for obtaining values of integers and pointers (into the 16 bits processor XP), for transport of data between the XM memory and the LM memory, for transport between processor boards, for rotating pointers in TABLE areas, for pre-definition of pointers in TABLE areas, for transport of constants from the XM memory to the SM memories, etc. During the implementation the macros are used in the assembler code file. By processing the assembler code file and the macro definition file with the m4 preprocessor, the macro names in the assembler code file are replaced by the assembler lines.

In figure 7.1 the overall structure of the generation of the machine code is shown. A memory definition file and memory specification file are used to create a file with the definitions of the mnemonic names. The source file which contains assembler code, subroutine calls, macro names and documentation is processed with the m4 preprocessor together with the file with macro definitions for source code and the file with the definitions of the mnemonic names. The result is a file which only contains assembler code and which can be compiled to machine code.

Before the calculation is started the database is initialised in the XM memory. The table values and pointer values are initialised outside the DNSP and read-in before starting the processing. The values are obtained by combining a special definition file with the file of specifications of all the variables. When the program is started the registers w0 to w31 in XP are initialised. A number of these registers contain the starting positions of the TABLE areas. At the start of the program the pointer-lists which contain the I/O variables for the floating-point calculations are initialised.

In paragraph 6.4 the computation of a small non-recursive do-loop with the four processors is discussed. Groups of four loops (or a multiple of four) are computed until the do-loop is finished. In practice there are also do-loops which contain if-statements. In cases where the if-statements have small branches it is preferred to use the do-loop source code, which is described in paragraph 6.4, and to use some conditional storage statements in the floating-point routine. If the branches are large the special source code for do-loops with if-statements should be used. For this source code an array has to be supplied which specifies whether the branch has to be computed or not. Loops which do not have to be computed are skipped before data transport. For the other loops the variables are moved to the SM memories. In this way the routine is efficient, but its performance strongly depends on the division of the calculations over the two (or one) branches. Also general loop structures have been designed for do-loops with square root calculations.

For the implementation the various programming techniques and database structures should be combined. There are basically two strategies for designing the
Figure 7.1. The generation of machine code with the m4 preprocessor.

software available, designing bottom-up and designing top-down. When designing bottom-up, the most inner parts of the program are built first and tested. When some equal level modules are ready they are integrated to form a new module. After integrating all the modules the program is ready. When designing top-down, first the overall routines are written. Later sub-modules are added, and more and more detail comes into the program. For the implementation of the assembler program on the DNSP we have chosen for the bottom-up strategy. The most important reason to choose this strategy is that for the DNSP special variables are created. These variables, together with other variables, have to be transported at various levels in the
program. Therefore, the variables have to be known. The program becomes well structured when it is designed by integrating small routines into large modules. This is of importance for the maintenance of the software. The bottom-up designing process starts with the routines for computing the coefficients and source terms of the difference equations. For the do-loops the calculation procedure discussed in paragraph 6.4 is used. First small do-loops are created with the number of I/O variables smaller than 16 for input and output. This should be done in such a way that the total I/O is minimised. Next the floating-point routine is designed followed by the I/O routine. The floating-point routine can be optimised by interchanging operations, but in most cases the time of the floating-point calculations is smaller than data transport so optimisation is not needed. For the solution of the tridiagonal matrix the algorithms discussed in paragraph 6.5 are used. Then the subroutines are combined into modules for solving one of the variables followed by the integration of these modules into one module for the iteration on one line. As discussed before, the data for the line and its neighbours is stored in the XM memory. When the iteration on one line is performed a shift to the new line is made. Before the shift is made the variables of the line that is solved are transported from the XM memory to the LM memory. The shift is made in the XM memory by moving some pointers and updating ix. After the shift in the XM memory the variables of a new line are transported to the XM memory. Then the iteration on the new line can be started.

When the implementation is ready there are a number of general subroutines e.g. for calculating a part of the domain, for starting up a sweep, etc. These subroutines can be started interactively or they can be combined in an iteration procedure for a large number of sweeps.

7.2. Programming environment
In paragraph 6.2 some reasons were given for choosing a particular calculation strategy. It followed from these analyses that the data should be distributed over the processor boards. Therefore, the iteration procedure is performed entirely on the DNSP, without interference of the host computer. Around the iteration procedure there are of course a number of other calculations, such as obtaining the energy balance over the boundaries that are not involved in the iteration steps itself but are needed to check e.g. convergence. These calculations can in principle be done on the DNSP, but we have chosen to perform them on the host computer. The main reason for this is that the time consuming DNSP programming effort should be reduced as much as possible. Furthermore, these calculations require communication between processor boards; that makes them more difficult. Another reason is that the calculations have to be performed in between a large number of iterations. Also from the viewpoint of testing and being sure to have the right solution it is better to make use of the tested software on the host computer. Therefore only what is called the inner-loop of the Navier-Stokes program is calculated on the DNSP. The basic strategy for the computation on the DNSP is to start with the Navier-Stokes program on the host computer. After one iteration is made data is written to a file in a standard way. This data is loaded into the DNSP and the calculation is started for a specified number of iterations (e.g. 1000). After finishing the calculations, data is transported from the large memory to the host computer. Now the balances can be checked with the Navier-Stokes program on the host computer.
In figure 7.2 the communication between the various programs is given. In the
centre the host computer with the Navier-Stokes fortran program is depicted. The host
is connected with a (mini-)supercomputer (Convex C240) that can be used as pre- or
postprocessor. Grids are sometimes so large that it is not possible to run the program
on the host (not enough core memory) or running would take a large amount of CPU-
time (5 to 10 minutes). Before loading data into the DNSP and after reading data, it
has to be converted. This is not necessary when the host format and DNSP format are
equal. Output of the DNSP can be used again for the processor or it can be read-in
into the Navier-Stokes program on the host computer.

7.3. Test-facility, testing, testability
In this paragraph the test-facility for line oriented programs on the DNSP is discussed.
It is obvious that errors are made during programming, specially when the program-
ning language is rather primitive. The errors are not all of the same sort. Two kinds
of programming errors can be distinguished: syntactical and structural. Syntactical
errors are detected by the compiler when some mistyping has occurred or when a used
string is not recognised. Other syntactical errors are those based on the underlying
hardware. The compiler gives an error message telling that the construction is not
allowed. Structural errors deal with errors in the structure of the program and are hard
to find. The compiler does not detect these errors and therefore testing and debugging
is needed.

There is a difference between testing and debugging, but they are closely related.
Testing is the process of establishing the existence of program errors, while debugging
is the process of locating the incorrect part in the code. It is important to notice that
tests can never show that a program is correct. Theoretically every possible input
range of variables can be tried and checked, but one should realise that this is impossi-
ble for our purposes. For every other testing method it is possible that undetected
errors still exist after the most comprehensive testing. So there are no 'always work-
ing' test methods.

For testing programs there are two basic strategies; testing bottom-up, testing
top-down. These two strategies are closely related to the way of designing software.
When testing bottom-up, the most inner parts of the program are tested first. When
some equal level modules are ready they are integrated and this integration process is
also tested. When testing top-down, during the fill in of the subroutines the program
can be tested. The implementation of the DNSP is a special case. The algorithm is
already implemented in fortran on the host computer. It is assumed that there are no
errors made during this implementation, the fortran program is the reference program.
The inner-loop of the fortran program is translated into assembler code for the DNSP.
When making this translation it is found that there are a number of differences
between the fortran program on the host computer and the assembler program for the
DNSP.

- The comparison between fortran and assembler code variables is difficult because
different names are used and many assembler code names do not have fortran
equivalents and conversely.

- The data structure in fortran is array oriented while the data structure of the
assembler code is line oriented.
Figure 7.2. The programming environment.
- The data structure in the XM memory mainly uses tables and pointers which the fortran program does not use.

- Data on the DSNP is stored in two memories, the LM memory for the fields and the XM memory for several lines in the fields.

Data types of both programs should be changeable without making a new test program. The above mentioned facts have resulted in designing the so-called test-facility, which stands in between the assembler program on the simulator of the DSNP and the fortran program, providing a flexible way of testing and debugging. Furthermore, the test-facility provides the interface between the data structured according to the fortran program data structure and the DSNP data structure.

In what follows the fundamental aspects of the test-facility are described. The test-facility is a tool for testing and debugging the assembler programs for the DSNP. One of the important aspects in the design of the test-facility was the need for flexibility. Changes in the fortran code as well as in the assembler code for the DSNP or in the data structure for the XM memory or the LM memory may never result in changes for the test-facility. Several requirements for the test-facility program are given below. A first requirement is that the communication between the program and the user must be interactive, by answering questions, or command oriented. It is obvious that the commands must be clear and the operation simple. Further, the program must be able to control also the simulator of the DSNP with a small set of commands. The basic purpose of the program is to be an interface for data between the fortran program on the host computer and the assembler programs on the DSNP. For example, it must be possible to compare variables of the fortran program with variables of the assembler program. Further it must be possible to compare data of the DSNP memories with older copies from the same memories. The program must be able to compute the variables for the DSNP by using the values of variables of the fortran program. This is used for generating dump-files which must be downloaded into the DSNP when the calculation process is started. Serving the same purpose, the program must generate a list of pointers out of the data structure definition for initiation of the tables pointing to the variables. The program must give the ability to search certain data in the DSNP as well as pointers to the data fields and the program should have a facility to search fortran variables and give their values at the time the dump was made.

The interface function of the test-facility between the fortran program and the assembler program for the DSNP (running on the simulator) is depicted in figure 7.3. All three modules have their own specific related files. The files related to the fortran program are source code files, object files, common blocks and different files with information about setting up the fortran program as well as data fields containing calculated flow fields. The files related to the DSNP are the translated program files for the control board and the processor boards. The files related to the test-facility are dump-files from fortran, conversion-files for assembler variables, name-files of assembler variables, the definition-file for the data structure of the XM memory and the LM memory and many data files containing the results of the calculations or conversions. Also dump-files made for the DSNP are related to the test-facility. The main data streams around the test-facility are depicted in figure 7.4. The test-facility program has a UNIX-shell based structure giving the ability to use it interactively. The common blocks of the fortran program are used as definition and specification of the
global variables. The common analyser creates a file Var_cham with all the variables and their dimensions. The dump, which is made of the global variables, is based on the common blocks. The dump-files (dump_int for the integers and dump_real for the reals) are indexed with a number of special index-files. In this way every global variable on every location in the dump-file can be found. In var_DUMP the dump location and conditions are stored. The file view_file contains all the definitions of the mnemonic names and is used to create the database structure. The dump-files contain the variables stored according to the DSNP database but are still separate for each variable.

The testing process of the assembler program proceeds step by step. The bottom-up strategy is used because first the errors in the smallest routines should be found. Checkpoints are built-in in the assembler program and the fortran program. One checkpoint is chosen as starting point, the values of the variables at this point (for the fortran program) are saved and used as starting data on the DSNP. Next on both the host and DSNP the calculation is continued until the next checkpoint. The values of the variables at this checkpoint can be compared to evaluate the working of the assembler code. In the test-facility the block structure has been made, giving the possibility to copy the data available in the memories of the DSNP to a block on the host computer. The data from the dumping process of fortran on the host computer is not
Figure 7.4. Main data streams around the test-facility.

directly moved to the DNSP memories but to a block on the host computer. The contents of one block can be examined very quickly and the contents of two blocks can be compared. For example, one block can contain data from the DNSP and the other data from the fortran program. The DNSP variable name is given to the test-facility and it returns the values, which is a line or just one value, integer or real. Furthermore, pointer values can be compared, as well as addresses of variables.
The dump-files contain the data of the fortran program on one of the checkpoints. The checkpoint is defined by a number of parameters (stored in var_DUMP) such as the variable to be solved, the sweep number, the x-coordinate, etc. The main purpose of the test-facility is to link the variables from the fortran program to the variables in the memories of the DNSP. The question arises how to define the assembler variables as a function of fortran variables without having to make compiler like structures. The following considerations were made when developing a conversion system. Each variable in the assembler program must be described in an easy way as a function of fortran variables. This definition can exist of some basic converting computations which must be carried out by the conversion program. The relation between the fortran variables and assembler variables is made with a stack structure (based on LIFO, Last in First out). With the conversion files, index files and the data obtained from the fortran program, the values of all the variables in the DNSP can be obtained. Furthermore, a pointer structure can be constructed. The result is that all the data needed for the calculation of a line is available and can be downloaded into the DNSP.

After finishing the test stage it is assumed that the assembler program works well. However, errors still can be present. Therefore, an extra test after each calculation is built-in. The normal calculation procedure with the DNSP proceeds in the following way. First the fortran program is executed with the appropriate input files. After one sweep a dump is made. This dump is not the common block based dump, but a dump which makes a few files (dump-files) which can be read into the DNSP right away. To make this dump, a number of fortran routines are generated out of the conversion files of the test-facility (dump_code). These subroutines are linked to the fortran program. So during normal execution the test-facility is not used. After finishing a number of sweeps on the DNSP the global variables are read into the fortran program and balances and convergence can be checked. Assembler program errors can be found when convergence is reached while the balances are still not fulfilled.
8. THE IMPLEMENTATION OF A 2D FLOW ALGORITHM ON THE DNSP

8.1. 2D Domain Decomposition
To divide the operations for the solution algorithm over the processor boards a domain decomposition method is used. Some of the features of this method have already been discussed in paragraph 5.1. Before applying this method to the 2D algorithm some tests were done with the fortran program, in order to investigate a possible reduction in convergence speed. A reduction of the convergence speed will reduce the efficiency of the solution algorithm. For the 2D case the domain is divided into a number of blocks of vertical lines. The computations for one vertical block are performed by one processor board. For the blocks of vertical lines the four processors on each processor board operate parallel on one vertical line. The iteration procedure on one processor board is depicted in figure 8.1. Sweep 1 is started on a single line in the centre of the vertical block and performed onto lines to the right boundary and then back to the line in the centre of the vertical block. The second sweep is performed to the left boundary of the vertical block. The total procedure is shown in figure 8.2. Four processor boards are shown with their neighbour connections. The first sweep is performed on the right side of the vertical blocks. After this sweep boundary values on the right side of the vertical blocks are moved to the right neighbour board. Next the second sweep is done and afterwards data on the left side is moved to the left neighbour board. The whole procedure described above can be repeated. In fact this procedure is a red-black ordering algorithm with four red parts and four black parts.

Figure 8.1. An iteration method for one processor-board.
In several other tests two or four lines were calculated parallel. Between two calculated lines there was one line situated on which no calculation was performed. The idea was to use two (or four) processors of a processor board to iterate on two (or four) lines. One processor would perform the calculations of one line, so the two (or four) lines are solved in parallel. Later it was found that it is better to use a parallel tridiagonal matrix solver and iterate on one line, because only a small time is spent on the tridiagonal matrices and the applied tridiagonal matrix solver already has a significant speedup.

Furthermore, for the test cases the number of line iterations was varied. Several tests have been done with the standard cavity configuration for $Ra = 10^6$ and $Pr = 0.71$, as described in paragraph 2.1. For this configuration the flow is laminar, however the $k$-$\varepsilon$ model is turned on. A $30 \times 30$ non-equidistant grid was used. To compare the various cases a method is needed which gives the differences in the convergence speed. It was decided to first calculate the solution of the reference problem, called $x^{sol}$. Many more iterations than usual were applied in order to be sure that the solution was accurate enough. Next, for every case a number of iterations were done, in which a rather deviating initial field was used. Intermediate fields, which are called $x^{cal}$, were saved during the iteration process. The absolute error of the solution $x^{cal}$: $\|x^{sol} - x^{cal}\|_1$ and the deviation of the absolute error: $\|x^{sol} - x^{cal}\|_2$, which are normalised by dividing by the maximum minus the minimum of the solution on the
Table 8.1. The ranking of the convergence speed of four methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple method</td>
<td>11</td>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Two lines parallel on board</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Four lines parallel on board</td>
<td>9</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Four processor boards parallel</td>
<td>10</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2. The reduction of the absolute error of the v velocity in $10^4$ sweeps.

<table>
<thead>
<tr>
<th>Method</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple method</td>
<td>1.8768</td>
<td>1.9408</td>
<td>2.0049</td>
<td>2.0455</td>
</tr>
<tr>
<td>Two lines parallel on board</td>
<td>1.9409</td>
<td>1.9410</td>
<td>2.0454</td>
<td></td>
</tr>
<tr>
<td>Four lines parallel on board</td>
<td>1.9240</td>
<td>1.9673</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Four processor boards parallel</td>
<td>1.8920</td>
<td>2.0350</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

whole field, can be calculated at several points in the iteration process by using:

$$ \| x_{sol} - x_{cal} \|_1 = \sum_{i,j} | x_{i,j}^{sol} - x_{i,j}^{cal} | $$

$$ \| x_{sol} - x_{cal} \|_2 = \left( \sum_{i,j} (x_{i,j}^{sol} - x_{i,j}^{cal})^2 \right)^{\frac{1}{2}} $$

Tests have been performed for the four methods which are shown in table 8.1. The number of iterations on one line which are performed before iteration on the next line (NTDMA) was varied. For each of the variables $u, v, H, p, k, \varepsilon$ the absolute error and deviation of the absolute error was obtained for a wide variety of iterations. Comparison of the results of the test showed that the two norms gave the same ranking for the four methods. An example of the reduction of the absolute error for the $v$ velocity is given in table 8.2. In this table the reduction in $10^4$ sweeps is given. The convergence of the variables was almost equal. The ranking in convergence speed for the four methods is given in table 8.1 (the best method has the lowest number).

For all the cases performing more calculations on one line seems to improve the convergence speed. For NTDMA = 1 two lines parallel seems to be the best, followed
by four lines parallel, four processor boards and the standard SIMPLE procedure. It is of course remarkable that changes in the algorithm have such a small influence. There are several explanations for this fact. First of all, iterations are carried out because of the non-linearity of the equations; so the solution is not obtained at once. Furthermore, under-relaxation is used which diminishes the differences. The general conclusion of the discussion of the results is that under these circumstances it does not matter which method is used for the iterations. Therefore, it was decided to use the domain decomposition method and that on each processor board every line is solved by four processors in parallel.

8.2. The 2D implementation, program possibilities, database
For the 2D problems in a rectangular calculation domain a new fortran program has been designed based on a program of A.M. Lankhorst. The difference between these two programs is the ability to use different boundary conditions on one wall. For the new program boundary conditions are generated outside the Navier-Stokes program with a pre-processing program. The pre-processing program generates an input file for the Navier-Stokes program. A number of boundary conditions can be prescribed by the program. First it should be noted that every boundary is divided into a number of cells. There are two kinds of cells; one for the scalar variables \( (T, k, \epsilon, \text{ etc.}) \) and the non-staggered (in the direction of the boundary) velocity, and one for the staggered (normal) velocity. For every variable a separate boundary condition can be given in each cell. There are three possible boundary conditions; "Dirichlet with a wall", "Dirichlet without a wall" and "Neumann". "Dirichlet with a wall" means that there is a solid wall at that boundary and therefore, when \( y^+ > 11.5 \), for the velocity wall functions are used along the wall. The values of \( k \) and \( \epsilon \) are obtained from the wall functions. For the temperature at the wall a value can be given. When the boundary condition "Dirichlet without a wall" is used for the variable, the appropriate boundary value should be given. The third possible boundary condition is the "Neumann" boundary condition. Then the value of the partial derivative perpendicular to the wall is given. With this boundary condition either a heat flux or the variable at an outflow boundary can be prescribed. The boundary choice and the value for each variable are stored in two arrays. Of course this structure is also applied to the assembler program for the DNSP. With the possible boundary conditions a wide range of problems in rectangular geometries can be solved on the DNSP.

The 2D program has been implemented on the DNSP as described in paragraph 7.1. Several timing statistics which can be obtained are discussed below. Also some statistics of the program can be obtained. First the floating-point routines are considered. The floating-point routines without comment and end statements contain 683 program lines (with 33 divisions). These lines are translated into 7484 lines of machine code, from which follows \( \text{float/\text{op}} = 9.2 \). In paragraph 6.4 it was derived in equation (6.19) that \( \text{float/\text{op}} = 8 \). The X-side routine without comment contains 5014 lines of code. Furthermore, 807 labels are used and 781 jump subroutine calls. The file contains 1490 macros. After filling in the macros the file has a size of 6396 lines. After translation into machine code the size is 8397 lines. The X-side program for the control board is small. It contains 298 lines and after compilation 323 lines.

After the implementation a detailed analysis can be done on the possible grid-
sizes. The variables of the different types are counted and listed below. A general memory count can be given for the LM memory and the XM memory. For the LM memory it is found that:

\[ LM = 2 \sum_{\text{fields}} nr_{y_f} (nr_x + 2) \]  

(8.1)

where \( nr_x \) is the number of points in the x-direction stored on one processor board and \( nr_{y_f} \) the number of points on one line for field \( f \). The number of points in the x-direction is enlarged with two because of overlap with other processor boards. For the 2D program there are ten fields with a length \( ny \) and one with a length 29 (for storing variables that are constant in the y-direction), giving for LM:

\[ LM = 2 \times [10 \; ny \; (nr_x + 2) + 29 \; (nr_x + 2)] = (20 \; ny + 58)(nr_x + 2) \]  

(8.2)

The general equation for the XM memory is given by:

\[ XM = 10 + 4 \; nr_{lm\; r\; xy} + \]

\[ nr_{r\; xy} \; (2 \; ny + 1) + nr_{i\; xy} \; (ny + 1) + \]

\[ nr_{r\; y} \; (2 \; ny + 1) + nr_{i\; y} \; (ny + 1) + \]

\[ nr_{r\; x} \; (2 \; nr_{r\; x\; y} + 1) + nr_{i\; x} \; (nr_{i\; x\; y} + 1) + \]

\[ nr_{list\; p} \; (l_p + 1) + nr_{i\; const} + 2 \; nr_{r\; const} \]

(8.3)

where

- \( nr_{r\; xy} \) and \( nr_{i\; xy} \) the number of fields of type \( r_{xy} \) and \( i_{xy} \)
- \( nr_{r\; y} \) and \( nr_{i\; y} \) the number of fields of type \( r_{y} \) and \( i_{y} \)
- \( nr_{r\; x} \) and \( nr_{i\; x} \) the number of fields of type \( r_{x} \) and \( i_{x} \)
- \( nr_{r\; x\; y} \) the number of points in the line of type \( r_{x} \)
- \( nr_{i\; x\; y} \) the number of points in the line of type \( i_{x} \)
- \( nr_{list\; p} \) the number of pointer-lists
- \( l_p \) the number of pointers in the pointer-list
- \( nr_{r\; const} \) the number of floating-point constants
- \( nr_{i\; const} \) the number of integer constants

For most of the fields, lines and pointer-lists, a pointer is needed (adding one position). Furthermore, for the fields in the LM memory four pointers are needed (for the first, the last, the current and for a working point). The values of the variables for the 2D program are:

\[ nr_{r\; xy} = 69 \quad nr_{i\; xy} = 0 \]
\[ nr_{r\; y} = 19 \quad nr_{i\; y} = 16 \]
\[ nr_{r\; x} = 3 \quad nr_{i\; x} = 3 \]
\[ nr_{r\; x\; y} = 29 \quad nr_{i\; x\; y} = 15 \]
\[ nr_{list\; p} = 110 \quad l_p = 16 \]
\[ nr_{r\; const} = 120 \quad nr_{i\; const} = 120 \]
\[ nr_{lm\; r\; xy} = 10 \]
Table 8.3. The data transport cost for the 2D program.

<table>
<thead>
<tr>
<th>transport</th>
<th>speed X-cycles/word (64 bits)</th>
<th>number of variables</th>
<th>number of X-cycles/point</th>
</tr>
</thead>
<tbody>
<tr>
<td>XM ←→ LM</td>
<td>6</td>
<td>20</td>
<td>120</td>
</tr>
<tr>
<td>proc. board ←→ proc. board</td>
<td>18</td>
<td>11</td>
<td>198</td>
</tr>
<tr>
<td>large mem. ←→ LM</td>
<td>44</td>
<td>10</td>
<td>440</td>
</tr>
</tbody>
</table>

By substituting these values into equation (8.3) the following equation is found:

\[ XM = 2613 + 192 \ ny \]  \hspace{1cm} (8.4)

Based on the two equations (8.2) and (8.4) the maximum grid-sizes can be obtained. For the XM memory it is found that:

\[ 2613 + 192 \ ny \leq 65536 \Rightarrow ny \leq 327 \]

So there is an upper-bound \( ny = 327 \) for the vector length of the lines on our processor. For the LM memory it is found that:

\[ (20 ny + 58)(nr_x + 2) \leq 2^{20} \Rightarrow nr_x \leq 52428 \]

This gives a combined upper-bound for \( ny \) and \( nr_x \). Of course the number of grid points in the x-direction can be enlarged by using more processor boards. For example four processor boards can be used, giving:

\[ ny \times x \leq 4 \times 52428 = 209711 \]

With the upper-bound for the XM memory in mind we find as limits \( nx = 641 \) and \( ny = 327 \). An enlargement of LM from 1M words to 4M words gives a four times larger limit for \( nx \).

8.3. Timing studies

In this paragraph more details are given on the timing of the DSNSP in relation to the processing speed. First the processing speed is analysed with the asymptotic equation (6.17). Next several timing experiments which are performed on the DSNSP are discussed. A comparison with other computers is only done for the 3D case.

For the calculation of the coefficients and source it was found that 223 variables are transported from the XM memory to the SM memories and 81 variables from the SM memories to the XM memory. The number of separate do-loops is 25, the number of floating-point operations (equivalents of a multiplication) which are performed is 460. Next the speed of one processor board for the coefficient and source calculation can be estimated. Substitution in equation (6.17) gives: \( nr\_cycles/point = 924 \) X-cycles. For the processing speed of the coefficient and source term calculation on four processors it is found that: speed = \( 460/(924 \times 10^{-7}) = 4.98 \) Mflops.

Next the calculation of the tridiagonal matrices has to be added. In paragraph 6.5 several algorithms for the tridiagonal matrices are discussed. Of course an attempt to improve the tridiagonal matrix solver should only be made when the time used for solving the tridiagonal matrix is not small compared to the total calculation time. In
the flow algorithm 924 X-cycles are needed to perform 460 floating-point operations for the coefficients and source terms. Six tridiagonal matrices are solved for every line so the number of operations per point for the tridiagonal matrices is $6 \times 13 = 78$ (for Gauss elimination five multiplications, three subtractions and one division are needed, a division is counted as five operations, giving a total of thirteen). For Gauss elimination by using one processor $65.8 \times 6 = 395$ X-cycles/point are needed. In paragraph 6.5 it was found that for Gauss elimination $65.8$ X-cycles/point are needed. This is $395/(924 + 395) \times 100\% = 29.9\%$ of the total execution time, but the number of operations is $78/538 \times 100\% = 14.5\%$ ($460 + 78 = 538$) of the total number of operations per point. This shows that the performance of the Gauss elimination should be improved. For the speed of the program with Gauss elimination, using one processor it is found that:

$$speed = \frac{538}{(924 + 395) \times 10^{-7}} = 4.08 \text{ Mflops}$$

This speed is 18% less than the speed for only the coefficients and source terms. For the speed of the calculation of one vertical line, when the "elimination from two sides" is used, it is found that:

$$speed = \frac{538}{(924 + 6 \times 45.6) \times 10^{-7}} = 4.49 \text{ Mflops}$$

The processing speed is still significantly smaller than the processing speed of the source and coefficient calculation, but 10% better than the standard Gauss elimination algorithm.

Above it was found that 1198 X-cycles per point are required for the calculation of one vertical line. Next the transport of data between the XM memory and the LM memory, between the processor boards and between the processor boards and the large memory are examined. The performance decreases because the floating-point calculations are stopped during these data transports. In table 8.3 a summary of the transport cost is given.

Before the calculation of a line, ten sets of variables on the line have to be transported from the LM memory to the XM memory and after finishing the calculation of a line ten updated sets of variables on the line have to be transported back. The data transport between the XM memory and the LM memory costs $120/(1198 + 120) \times 100\% = 9.1\%$ of the calculation time of one point. This is a permitted percentage. The data transport cost between the processor boards, and the large memory and the LM memory can be divided among the lines on one processor board. In addition to the sets of variables which have to be transported between the processor boards data on one extra line for the horizontal velocity has to be transported because of the staggered grid. If there are ten lines on a processor board the cost are; for processor board $\leftrightarrow$ processor board: $198/(10 \times (1198 + 120)) \times 100\% = 1.5\%$ and for the large memory $\leftrightarrow$ the LM memory (with four processor boards): $4 \times 440/(10 \times (1198 + 120)) \times 100\% = 13.4\%$. The cost for data transport between processor boards can be neglected because in most calculations there are more than ten lines on the processor board. The transport cost between the large memory and the LM memory can be neglected because not every sweep data is transported to the large memory. For the
Table 8.4. The value of the half-performance intensity $f_{\gamma_5}$, that is obtained with equation (6.23).

<table>
<thead>
<tr>
<th>$N_{var , w} + N_{var , r}$</th>
<th>$f_{\gamma_5}$</th>
<th>$op$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.87</td>
<td>3.75</td>
</tr>
<tr>
<td>4</td>
<td>1.63</td>
<td>6.50</td>
</tr>
<tr>
<td>6</td>
<td>1.54</td>
<td>9.25</td>
</tr>
<tr>
<td>8</td>
<td>1.50</td>
<td>12.00</td>
</tr>
<tr>
<td>10</td>
<td>1.48</td>
<td>14.75</td>
</tr>
<tr>
<td>12</td>
<td>1.46</td>
<td>17.50</td>
</tr>
<tr>
<td>14</td>
<td>1.45</td>
<td>20.25</td>
</tr>
<tr>
<td>16</td>
<td>1.44</td>
<td>23.00</td>
</tr>
<tr>
<td>18</td>
<td>1.43</td>
<td>25.75</td>
</tr>
<tr>
<td>20</td>
<td>1.425</td>
<td>28.50</td>
</tr>
<tr>
<td>$\infty$</td>
<td>1.375</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

The speed of one processor board one finds approximately:

$$speed = \frac{538}{(1198 + 120) \times 10^{-7}} = 4.08 \text{ Mflops}$$

The speed of the four processor board system is 16.32 Mflops. It can be concluded that by the analysis given above it is found that an efficiency of about 40% can be obtained for the DNSP.

The 2D calculations can further be analysed. With equation (6.18) the computational intensity $f$ can be found:

$$f = \frac{op}{transport} = \frac{460}{223 + 81} = 1.51$$

With this value the break-even vector length, $n_{br}$, as given in equation (5.11) can be found. With $r_{\infty}^{(m)} = 5 \cdot 10^6$ words/sec., $r_{\infty}^{(d)} = 10^7$ operations/sec., $n_{\infty}^{(m)} \approx 3$, $n_{\infty}^{(d)} < 1$ and $f \approx 1.5$, it is found that $n_{br} \approx 2.5$. To give a better understanding of the value of the computational intensity, the half-performance intensity $f_{\gamma_5}$ should be obtained with equation (6.23). For the average I/O per do-loop it is found that: $(N_{var \, w} + N_{var \, r})/hr \, loops = (223 + 81)/25 = 12.16$. The equation (6.23) is given in table 8.4 for $Nval = 1$ and $Nsm = 4$. It should be noted that the half performance intensity only varies a little over the I/O range. Substitution of $N_{var \, w} + N_{var \, r} = 12.16$ in equation (6.23) gives for the half-performance intensity $f_{\gamma_5} = 1.457$. So it is found that the computational intensity $f$ for the 2D program is close to the half-performance intensity. For the source and coefficient calculation it was found that the processing speed is 4.98 Mflops, which is 50% of the peak rate. The computational intensity for which the peak rate is obtained is found with equation (6.24). The equation is given in table 8.5 for $Nsm = 4$ and $Nval = 1$. For the source and coefficient calculation of the 2D program one finds that: $f = 2.976$.

Several experiments have been done for the 2D program on the DNSP. The standard cavity case has been taken and twenty grid points are used in the x-direction and a variable number of grid points in the y-direction. The number of grid points in the
Table 8.5. The value of the computational intensity $f$, that is obtained with equation (6.24).

<table>
<thead>
<tr>
<th>$N_{var_w} + N_{var_r}$</th>
<th>$f$</th>
<th>$op$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4.125</td>
<td>8.25</td>
</tr>
<tr>
<td>4</td>
<td>3.438</td>
<td>13.75</td>
</tr>
<tr>
<td>6</td>
<td>3.208</td>
<td>19.25</td>
</tr>
<tr>
<td>8</td>
<td>3.094</td>
<td>24.75</td>
</tr>
<tr>
<td>10</td>
<td>3.025</td>
<td>30.25</td>
</tr>
<tr>
<td>12</td>
<td>2.979</td>
<td>35.75</td>
</tr>
<tr>
<td>14</td>
<td>2.946</td>
<td>41.25</td>
</tr>
<tr>
<td>16</td>
<td>2.922</td>
<td>46.75</td>
</tr>
<tr>
<td>18</td>
<td>2.903</td>
<td>52.25</td>
</tr>
<tr>
<td>20</td>
<td>2.888</td>
<td>57.75</td>
</tr>
<tr>
<td>$\infty$</td>
<td>2.75</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Table 8.6. The speed of the DNSP as function of the grid size.

<table>
<thead>
<tr>
<th>grid</th>
<th>speed (Mflops)</th>
<th>speed based on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DNSP</td>
<td>$\bar{n}<em>{\Delta} = 10.8$ and $r</em>\infty = 4.03$</td>
</tr>
<tr>
<td></td>
<td>1 proc. board</td>
<td></td>
</tr>
<tr>
<td>$20 \times 20$</td>
<td>2.52</td>
<td>2.62</td>
</tr>
<tr>
<td>$20 \times 40$</td>
<td>3.14</td>
<td>3.17</td>
</tr>
<tr>
<td>$20 \times 60$</td>
<td>3.40</td>
<td>3.42</td>
</tr>
<tr>
<td>$20 \times 80$</td>
<td>3.54</td>
<td>3.55</td>
</tr>
<tr>
<td>$20 \times 100$</td>
<td>3.63</td>
<td>3.64</td>
</tr>
<tr>
<td>$20 \times 120$</td>
<td>3.69</td>
<td>3.70</td>
</tr>
</tbody>
</table>

The $x$-direction is kept fixed because it only influences the calculation speed a little. The number of grid points in the $y$-direction strongly influences the calculation speed. To measure the time $10^4$ sweeps are performed. In this way overhead of starting and finishing the calculations is reduced to a neglectable level. After downloading the DNSP and initialising the pointers and the XM memory the time counting and calculation is started. After the $10^4$ sweeps the time counter is stopped. The results of the timing are given in table 8.6. In this table the grid-size and the processing speed (in Mflops) is given. This processing speed is obtained in the following way. First the number of floating-point operations for one sweep is counted with:

$$nr\_op = 538 \times (nx - 2)(ny - 2)$$  \hspace{1cm} (8.5)

In equation (8.5) 538 is the number of floating-point operations for one grid point as found in paragraph 3.2. $nx$ and $ny$ are reduced with two because the first and last point on the line are boundary points for which the balance equations are not prescribed. Other floating-point operations on the boundary for the boundary conditions are neglected. The obtained values as depicted in table 8.6 range from 2.52
Table 8.7. The efficiency \( E_p \) and a measure of both efficiency and speedup \( F_p \).

<table>
<thead>
<tr>
<th>number of processors</th>
<th>( E_p ) ( (/V) )</th>
<th>( F_p ) (* ( T_1 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.73</td>
<td>1.07</td>
</tr>
<tr>
<td>3</td>
<td>0.56</td>
<td>0.94</td>
</tr>
<tr>
<td>4</td>
<td>0.45</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Table 8.8. The modified efficiency \( \hat{E}_p \) and a modified measure of both efficiency and speedup \( \hat{F}_p \).

<table>
<thead>
<tr>
<th>number of processors</th>
<th>( \hat{E}_p ) ( (/V) ) ( C/I = 3.7 )</th>
<th>( \hat{F}_p ) ( (\times T_1) /V ) ( C/I = 3.7 )</th>
<th>( \hat{E}_p ) ( (/V) ) ( C/I = 4.37 )</th>
<th>( \hat{F}_p ) ( (\times T_1) /V ) ( C/I = 4.37 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.21</td>
<td>0.21</td>
<td>0.19</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>0.26</td>
<td>0.37</td>
<td>0.23</td>
<td>0.33</td>
</tr>
<tr>
<td>3</td>
<td>0.25</td>
<td>0.42</td>
<td>0.23</td>
<td>0.38</td>
</tr>
<tr>
<td>4</td>
<td>0.23</td>
<td>0.42</td>
<td>0.22</td>
<td>0.39</td>
</tr>
</tbody>
</table>

Mflops for the coarsest grid to 3.69 Mflops for the finest grid. An analysis of the results in table 8.6 by assuming \( T = (n + \bar{n}_N)/\bar{T}_m \) (applying the least squares method, see paragraph 9.4) gives that the asymptotic performance is about 4.03 Mflops and \( \bar{n}_{0.9} \approx 97.2 \) (\( \bar{n}_{N} = 10.8 \)). It is found that the equation for the time is good for the vector lengths between 20 and 120. It can be concluded that the asymptotic speed is about 4.03 Mflops. This is close to the value of 4.08 Mflops for the speed which is found above by theoretical means. In paragraph 6.4 an upper-bound was found for the half-performance length of the combined memory and arithmetic pipeline: \( \bar{n}_{N}(f) \leq 36/7 \times Nval \approx 5.14 \times Nval \). However, it should be noted that calculations for the boundary conditions at the bottom and top of the line are not dependent on the vector length so \( \bar{n}_{N} \) is enlarged.

From the discussion above one gets the idea that instead of four processors two can be used without a great loss of computational speed. This is based on the fact that the computational intensity \( f \) is equal to 1.51 while the computational intensity for which the peak rate is obtained is \( f = 2.976 \). This idea shall be examined in more detail. Equation (6.16) is used to obtain the number of X-cycles which are needed per point (for the source and coefficient calculation). It is further assumed that data transport is slower than the floating-point calculations for all the do-loops (an equation similar to equation (6.17) is obtained). Some evidence for this fact is given later. For two processors it is found that, with \( nr\_loops = 25 \), \( \sum_i Nvar\_w_i = 223 \), \( \sum_i Nvar\_r_i = 81 \):

\[
\frac{nr\_cycles/point = \frac{14 \times 25}{2} + \frac{7}{2} (223 + 81) = 1239 \times cycles}{2}
\]
Including the tridiagonal matrix calculations (elimination from two sides) and data transport between the XM memory and the LM memory, this gives a processing speed of:

\[
\text{speed} = \frac{538}{(1239 + 6 \times 45.6 + 120) \times 10^{-7}} = 3.3 \text{ Mflops}
\]

Next the computational intensity for which the peak rate is obtained should be found. This is the intensity at the point where data transport and floating-point operations are equal in time. For this purpose an equation such as equation (6.24) for \(Nsm = 2\) is used. It is found that \(f = 1.98\). This value is still larger than the computational intensity of the program, so in most do-loops data transport is slower than the floating-point calculations. One sees that the speed with two processors is only 19% smaller than the speed with four processors. This means that the efficiency with two processors is larger. To find the efficiency one should also analyse the case with one processor. The parameters as given above for \(Nsm = 2\) have also been obtained for \(Nsm = 1\). It is found that: \(\text{nr\_cycles/point} = 1870\) X-cycles, speed = \(538/(1870 + 395 + 120) \times 10^{-7} = 2.26\) Mflops, \(f = 1.48\). For the tridiagonal matrix calculations also one processor is used. The computational intensity for which the peak rate is obtained is now a bit smaller than the value of \(f\) for the program. It is assumed that the floating-point calculations cost are lower or just a bit higher than the data transport cost. The efficiency \(E_p\) as defined by equation (5.2) and the measure of both speedup and efficiency \(F_p\) as given by equation (5.3) are given in table 8.7. One can see that the efficiency decreases very quickly. On the other hand, \(F_p\) has a maximum for two processors. However, \(F_p\) is not a good measure in this case because the cost of a processor board not only consists of the cost of the processors. The cost of a processor board is given by:

\[
\text{cost} = C + Vp
\]

(8.6)

So a new function should be introduced for the efficiency and the measure of speedup and efficiency:

\[
\hat{F}_p = \frac{V}{T_1} \frac{S_p^2}{C/V + p}
\]

(8.7)

\[
\hat{E}_p = V \frac{S_p}{C/V + p}
\]

(8.8)

For a processor board is found that \(C = 4700\) dollar and \(V = 1075\) dollar. The values of the parameters \(\hat{E}_p\) and \(\hat{F}_p\) are given in table 8.8. As can be seen for \(C/V > 3.7\) the parameter \(\hat{F}_p\) is larger than the other three. For the DNSP \(C/V\) is about 4.37. Values for \(\hat{E}_p\) and \(\hat{F}_p\) for this case are given in table 8.8. It can be concluded that four processors on a processor board is most suited in this case, because the measure of efficiency and speedup is maximised.

### 8.4. Calculations for a square cavity

To illustrate the capabilities of the DNSP, calculations have been done for air in a standard 2D cavity. These calculations have also been reported in Van der Vlugt et al. (1989). The standard 2D cavity is the model of the mid plane of a 3D cavity with a
a third dimension which is large compared with the other two dimensions. The left wall of the 2D cavity is heated to a constant uniform temperature and the right wall is cooled. The bottom and top wall are adiabatic. Due to the temperature difference of the two opposite walls a flow appears, moving upwards near the left (hot) wall and downwards near the right (cold) wall. The characteristics of the flow can be expressed with two characteristic numbers (paragraph 2.1):

\[ Ra = \frac{g \beta \Delta T H^3}{\nu a}, \quad Pr = \frac{v}{a} \]

For the laminar situation calculations have been done in the square cavity by Henkes et al. (1988) for \( Ra = 10^3 \) up to \( 10^{10} \). It was found that the wall heat transfer scales as \( Ra^{1/4} \) and the vertical velocity \( v/|g \beta \Delta TV|^{1/3} \) scales as \( Ra^{1/6} \). For \( Ra \to \infty \) the averaged Nusselt number is 0.309 \( Ra^{-1/4} \). De Vahl Davis (1983) gives accurate benchmark results for \( Ra = 10^6 \).

Calculations on the DNSP were started for the laminar situation. Three cavity situations, with \( Ra = 10^6, 10^7 \) and \( 10^8 \), have been investigated. Around \( Ra = 10^9 \) the flow becomes turbulent, for the \( k-\varepsilon \) model. To decide whether the obtained solution is accurate enough one should examine the convergence and the discretisation accuracy. For the convergence the criteria given in paragraph 3.2 are used. The discretisation accuracy can be examined by applying grid refinement.

The grid which has been used is given by:

\[ \frac{x_i(k)}{H} = \left( k - 1 \right) + \frac{1}{N_i - 1} \sin \left( \frac{2\pi(k - 1)}{N_i - 1} \right) \quad k = 1, ..., N_i \quad (8.9) \]

Here \( x_i(k) \) is the position in the \( i \) direction of the pressure point of grid cell \( k \) and \( N_i \) is the number of grid points in the \( i \) direction. By using this grid equation the grid is refined near the walls where the largest gradients occur. The number of grid points was varied from 45 x 45 to 90 x 90 to investigate the discretisation accuracy.

Supposing \( x_{sol} \) is the solution of a quantity at a mesh point for \( h_i \to 0 \), and the computed value using a mesh characterised with \( h_i \) is given by \( x_i \), one can assume that:

\[ x_{sol} = x_i + C h_i^n \quad i = 1, 2, 3 \quad (8.10) \]

where \( n \) is the order of the truncation error and \( C \) is assumed to be independent of \( h_i \). The order of the truncation error \( n \) can be found by solving:

\[ \frac{x_1 - x_2}{x_2 - x_3} = \frac{h_1^n - h_2^n}{h_2^n - h_3^n} \quad (8.11) \]

After finding \( n \), the constant \( C \) can be found at each mesh point. If there are only two grids available, the order of the truncation error \( n \) cannot be obtained. For the central difference scheme the truncation error \( n \) is equal to two, for the upwind difference scheme it equals one. Because the used difference scheme is a combination of both the truncation error \( n \) lies between one and two. To find the maximum error one should use for the truncation error \( n = 1 \). Assuming that \( h_1/h_2 = 2 \) for the truncation
error it is found that:

\[ x_{sol} - x_2 = \frac{x_2 - x_1}{2^n - 1} \]  

The results for the three situations on the 45 × 45 and 90 × 90 grid differ less than one percent. It can be concluded from the above that the 45 × 45 grid, with the grid distribution as given by equation (8.9), is fine enough for this case.

In figure 8.3 the streamlines, isotherms and vector plot of the velocity are depicted for \( Ra_b = 10^8 \). As can be seen the core is thermally stratified and almost stagnant. The temperature of the bottom and top wall is almost constant. There are thin boundary layers along the vertical sides of the cavity. The Nusselt number on the hot wall, given by: \(-[\partial(T - T_0) / \partial x]_w / \Delta T\), is shown in figure 8.4. At the bottom the Nusselt number is large due to the large temperature difference between the wall and the fluid and it decreases to almost zero at the top of the wall. Several characteristic quantities are given in table 8.9, where \( Nu_{min} \), \( Nu_{max} \), \( \bar{Nu} \) and \( v_{max} \) are the minimum and the maximum of the Nusselt number on the hot wall, the averaged Nusselt number of the hot wall and the maximum of the vertical velocity at half the cavity height. The maximum velocity at half the cavity height (\( v_{max} \)) is scaled with the laminar velocity scale \( u_b = (g\beta \Delta T d)^{1/2} \). The Nusselt number at half the cavity height scales very well with \(Ra^{1/4}\) and the velocity maximum scales well with the laminar velocity scale \( u_b \), which is in agreement with Henkes et al. (1988). The averaged Nusselt number scales less well with \(Ra^{1/4}\) because of corner effects.

The flow field for \( Ra = 10^6 \) on a 90 × 90 grid is compared with the benchmark results of De Vahl Davis (1983), which are given in table 8.10. The typical variables used in the benchmark for our 90 × 90 grid results and the benchmark results of De Vahl Davis differ less than 1%. De Vahl Davis used an equidistant 41 × 41 and 81 × 81 grid and extrapolated the solutions of these two grids to get a solution with a higher accuracy. It can be concluded that the results of the calculations on the DNSP are in very good agreement with the benchmark solution.

For the turbulent situation calculations have been performed for \( Ra = 10^{10}, 10^{11} \) and \( 10^{12} \). Three grid-sizes have been used: 45 × 45, 90 × 90 and 180 × 180, with the grid distribution of equation (8.9). In figure 8.5 the Nusselt number on the hot wall is shown for \( Ra = 10^{12} \) on three grids. As can be seen the Nusselt number near the bottom of the wall still varies between the 90 × 90 and 180 × 180 grid. This is probably due to the fact that the grid is not fine enough between the location of the velocity maximum in the vertical boundary layer and the wall. Furthermore, the value of \( \epsilon \) increases on the wall when the grid-size decreases. On the rest of the wall the values of the 90 × 90 and 180 × 180 grid differ less than 3%. Several characteristic quantities for the turbulent situation are given in table 8.11. From table 8.11 it follows that the results for a 90 × 90 grid and a 180 × 180 grid differ less than 2.5%. For \( Ra = 10^{10} \) and \( 10^{11} \) the results on the two grids differ less than 2%. The averaged Nusselt number on the 45 × 45 grid and 90 × 90 for \( Ra = 10^{12} \) differs 18%, so at least a 90 × 90 should be used.

The Nusselt number for \( Ra = 10^{10}, 10^{11} \) and \( 10^{12} \) on a 180 × 180 grid is given in figure 8.6 and several characteristic quantities are summarised in table 8.12. The Nusselt number at half the cavity height scales well with \( Ra^{1/3} \). The 1/3 power in the
Figure 8.3a. The stream function for the laminar flow with $Ra = 10^8$.

Figure 8.3b. The dimensionless temperature for the laminar flow with $Ra = 10^8$. 
Figure 8.3c. A vector plot of the velocity field for the laminar flow with $Ra = 10^8$.

Table 8.9. Several characteristic quantities for the laminar situation, $90 \times 90$ grid.

<table>
<thead>
<tr>
<th>$Ra$</th>
<th>$10^6$</th>
<th>$10^7$</th>
<th>$10^8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Nu_{\min}/Ra^{1/4}$</td>
<td>0.0316</td>
<td>0.0258</td>
<td>0.0226</td>
</tr>
<tr>
<td>$Nu_{\max}/Ra^{1/4}$</td>
<td>0.554</td>
<td>0.698</td>
<td>0.877</td>
</tr>
<tr>
<td>$Nu/Ra^{1/4}$</td>
<td>0.279</td>
<td>0.293</td>
<td>0.301</td>
</tr>
<tr>
<td>$Nu/Ra^{1/4}$ at half the height</td>
<td>0.264</td>
<td>0.264</td>
<td>0.264</td>
</tr>
<tr>
<td>$v_{\max}/u_b$</td>
<td>0.262</td>
<td>0.263</td>
<td>0.266</td>
</tr>
<tr>
<td>the coordinate of $v_{\max} (x_1/d)$</td>
<td>0.0359</td>
<td>0.0196</td>
<td>0.0120</td>
</tr>
</tbody>
</table>

Table 8.10. Several characteristic quantities for $Ra = 10^6$ by the DNSP and De Vahl Davis.

<table>
<thead>
<tr>
<th>$Ra$</th>
<th>DNSP</th>
<th>De Vahl Davis</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Nu_{\min}/Ra^{1/4}$</td>
<td>0.0316</td>
<td>0.0313</td>
</tr>
<tr>
<td>$Nu_{\max}/Ra^{1/4}$</td>
<td>0.554</td>
<td>0.558</td>
</tr>
<tr>
<td>$\bar{Nu}/Ra^{1/4}$</td>
<td>0.279</td>
<td>0.279</td>
</tr>
<tr>
<td>$Nu/Ra^{1/4}$ at half the height</td>
<td>0.264</td>
<td>-</td>
</tr>
<tr>
<td>$v_{\max}/u_b$</td>
<td>0.262</td>
<td>0.260</td>
</tr>
<tr>
<td>the coordinate of $v_{\max} (x_1/d)$</td>
<td>0.0359</td>
<td>0.0379</td>
</tr>
</tbody>
</table>
Figure 8.4. The wall-heat transfer for the laminar situation.

Figure 8.5. The wall-heat transfer for the turbulent situation on three grids for $Ra = 10^{12}$. 
Figure 8.6. The wall-heat transfer for the turbulent situation for three Rayleigh numbers on the $180 \times 180$ grid.

Table 8.11. Several characteristic quantities for $Ra = 10^{12}$ on three grids.

<table>
<thead>
<tr>
<th>Grid</th>
<th>45 $\times$ 45</th>
<th>90 $\times$ 90</th>
<th>180 $\times$ 180</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Nu/Ra^{1/3}$</td>
<td>0.0570</td>
<td>0.0694</td>
<td>0.0719</td>
</tr>
<tr>
<td>$Nu/Ra^{1/3}$ at half the height</td>
<td>0.0597</td>
<td>0.0733</td>
<td>0.0751</td>
</tr>
<tr>
<td>$v_{\text{max}}/(u_b \ Ra^{-1/18})$</td>
<td>0.676</td>
<td>0.659</td>
<td>0.662</td>
</tr>
<tr>
<td>the coordinate of $v_{\text{max}} \ (x_1/d)$</td>
<td>0.0021</td>
<td>0.00199</td>
<td>0.00196</td>
</tr>
</tbody>
</table>

Table 8.12. Several characteristic quantities for the turbulent situation, $180 \times 180$ grid.

<table>
<thead>
<tr>
<th>$Ra$</th>
<th>$10^{10}$</th>
<th>$10^{11}$</th>
<th>$10^{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Nu/Ra^{1/3}$</td>
<td>0.0657</td>
<td>0.0683</td>
<td>0.0719</td>
</tr>
<tr>
<td>$Nu/Ra^{1/3}$ at half the height</td>
<td>0.0689</td>
<td>0.0717</td>
<td>0.0751</td>
</tr>
<tr>
<td>$v_{\text{max}}/(u_b \ Ra^{-1/18})$</td>
<td>0.642</td>
<td>0.643</td>
<td>0.662</td>
</tr>
<tr>
<td>the coordinate of $v_{\text{max}} \ (x_1/d)$</td>
<td>0.0046</td>
<td>0.00249</td>
<td>0.00196</td>
</tr>
</tbody>
</table>
Nusselt number at half the cavity height is well known. However the scaling is less good when compared to the laminar case. Near the bottom wall the Nusselt number scales with $Ra^{1/4}$, which is the laminar scaling. This is not strange because in that area the flow is still laminar. The transition point is moving to the bottom wall for an increasing Rayleigh number. Further the velocity maximum scales with $u_b \ Ra^{-1/18}$, this agrees with Henkes & Hoogendoorn (1989).

To obtain the solution for the laminar and turbulent situations a large number of iterations were needed. The number of iterations is a function of the Rayleigh number and a function of the grid-size. When an arbitrary initial field is used the number of required iterations increases very fast as function of the Rayleigh number. When a converged solution on the $45 \times 45$ grid is used as initial field for the calculation on the $90 \times 90$ grid then the required number of iterations is significantly smaller. However, still a large number of iterations is required. For example, for $Ra = 10^{12}$ and a $90 \times 90$ grid about $2,3 \times 10^6$ iterations on the field are needed when starting with the converged $45 \times 45$ grid. This is equivalent with about 368 hours CPU on the DNSP. During these calculations it became clear that the algorithm is far from optimal for these problems at very high Rayleigh numbers. The required number of iterations should be reduced with an improved algorithm. Still the performance of the DNSP even with this far from optimal algorithm has been very good. Converged solutions on fine grids have been obtained. As mentioned before, the comparison between the converged solution of the laminar case at $Ra = 10^6$ and the generally accepted De Vahl Davis benchmark result shows an excellent agreement.
9. THE IMPLEMENTATION OF A 3D FLOW ALGORITHM ON THE DNSP

9.1. 3D Domain Decomposition
In paragraph 8.1 a heuristic domain decomposition method was discussed which is used for the 2D case. In a similar way a domain decomposition method has been used for the 3D case. For the 3D case no tests have been done to investigate the change in convergence speed. It is assumed that the change is small, as was the case for the 2D case.

For the 3D domain decomposition method the domain is divided into a number of slices. One slice is a 3D box containing a number of x-y planes of the 3D calculation domain. For the four processor board configuration for example, four slices can be used. The size of the slices on each processor board will be chosen as close to each other as possible, to reduce the synchronisation time between the processor boards after data transport. For the iteration procedure the same strategy as for the 2D case is adopted. On each processor board iterations are performed on a number of x-y planes (which is called a sweep) and between the iterations data is exchanged between the processor boards. A sweep on one processor board is started on the x-y plane in the middle of the slice, as depicted in figure 9.1. After the iteration on the right side of the slice, the data on the right boundary of the slice is transported to the right neighbour board and next the sweep on the left side of the slice is done.

In the prototype DNSP the large memory has the same size as the LM memory on one processor board. In that case the above described procedure is followed. When the large memory is larger another calculation procedure can be adopted. The extra memory on the large memory can be used to store several slices. In this way several slices are stored on the processor boards and several on the large memory. The size of the 3D calculation domain can be larger for this case, because the maximum size of the slices is fixed. For example, the 3D domain can be divided into 12 slices. Three slices are assigned to one processor board, but they are stored on the large memory. After iteration on a slice the data of the slice is transported to the large memory and the data of the next slice is transported to the processor board.

The extra data transport between the processor boards and the large memory gives overhead. The order of magnitude of this overhead can be compared to the time needed for the calculations. By using equation (6.1) it is found that 1056 X-cycles per point are needed for transport between the large memory and the LM memory (taking \( n = 2 \times 12 \)). In paragraph 3.2 it was found that per point 1150 floating-point operations are performed. By assuming 40\% efficiency this would take 2875 X-cycles to be performed. So the data transport cost take 26.8\% of the sum of the calculation time and the data transport time. We think that this leads, together with other overhead, to a too large reduction of the processing speed of the DNSP. Therefore, it is preferred to store the data local on the processor boards. In case this is not possible the 26.8\% reduction of the processing speed should be taken for granted.
9.2. The 3D implementation, program possibilities, database

For the 3D implementation on the DNSP a program has been used which has been designed by A.M. Lankhorst. A large variety of cavity flows can be calculated with this program. The velocities on the boundaries are set to zero because of the solid walls. The boundary conditions for $k$ and $\varepsilon$ are obtained by using the wall functions for forced convection, discussed in paragraph 2.2. For the temperature several boundary conditions have been implemented. A few standard boundary conditions are: a constant temperature on the wall, a temperature which is changing linearly with one of the three coordinates or the heat transfer through the wall can be set to zero (adiabatic wall). For these boundary conditions it is assumed that the thickness of the cavity walls is zero. For the comparison of the numerical results with the experimental results the wall can be given a certain thickness. The heat transfer in the walls and heat losses through the walls to the surrounding fluid (Robin boundary condition) can be calculated.

The 3D program has been implemented on the DNSP as described in paragraph 7.1. Several timing statistics are discussed in the next paragraph. We can also obtain
some statistics for the program itself. First the floating-point routines are considered. The floating-point routines without comment and end statements contain 890 lines (with 25 multiplications). These lines are translated into 9094 lines of machine code, from which follows float/op = 9.2. This value is equal to the value in the 2D program. In paragraph 6.4 we defined in equation (6.19) float/op = 8. These two values are reasonably close to each other. The fixed-side routine without comment contains 8015 lines of code. Furthermore, 1122 labels are used and 1429 jump subroutine calls. The file contains 3043 macros. After filling in the macros the file has a size of 10679 lines. After translation into machine code the size is 14799 lines. The fixed-side program for the control board is small and contains 298 lines and after compilation 323 lines. When counting the routines for the boundary conditions we find that there are 1082 lines of code which contain 540 macros. Furthermore, the implementation of the boundary conditions is difficult because of the large number of exceptions. On each of the six boundaries different variables should be used. The fortran code which is translated to assembler code is also counted. It was found that 218 lines of code are used for controlling the calculations, 1058 lines are used for the source and coefficient calculation and 1196 lines of code are used for the boundary conditions. The total fortran code is 6172 lines. It should be noted that for the boundary conditions more program lines are used than for the coefficient and source calculation. It can be concluded that about 4.7 times ((10679 + 890)/(218 + 1058 + 1196)) as much program lines are needed for the assembler program as for the fortran program. During the implementation the assembler code was tested as described in paragraph 7.3. Considering the facts stated above and obtained experience, it can be concluded that the assembler programming on the DNSP is very costly compared to fortran programming on a (super)computer.

After the implementation a detailed analysis can be done on the possible grid-sizes. The variables of the different types are counted and listed below. First the LM memory is analysed. Only the 3D variables are counted. The others are neglected, such as variables which only vary in the z-direction. There are 12 fields in the LM memory giving for the LM memory size:

\[ LM = 2 \times 12 \left( nx \ ny \left( \frac{nz}{nr_b} + 2 \right) \right) \]  \hspace{1cm} (9.1)

where \( nx, ny, nz \) and \( nr_b \) are the number of grid points in the x-direction, y-direction, z-direction and the number of processor boards which are used (the number of slices is chosen to be equal to the number of processor boards). The 2 is added to \( nz/nr_b \) because of overlap of data between the processor boards. Next, the size of the XM memory should be discussed, which is more complicated for the 3D case than for the 2D case. For the 3D case not only the variables on the line under consideration are stored in the XM memory but also the variables on a large number of neighbour lines. For the line under consideration 18 line variables are stored (12 for the variables and 6 for the temporaries). For the variables on the surrounding lines 84 line variables are stored giving a total of 102. The total count for the size of the XM memory is given in equation (9.2).
\[ XM = 10 + 5 \, nr_{lm\_r\_xyz} + \]
\[ nr_{r\_xyz} \, (2 \, ny + 1) + nr_{r\_y} \, (2 \, ny + 1) + \]
\[ nr_{r\_x} \, (2 \, nr_{r\_x\_y} + 1) + nr_{r\_z} \, (2 \, nr_{r\_z\_y} + 1) + \]
\[ nr_{list\_p} \, (l_p + 1) + nr_{i\_const} + 2 \, nr_{r\_const} + 2 \, nr_{table} \]  \hspace{1cm} (9.2)

\[ \text{number of fields in LM} \]
\[ \text{the number of fields of type } r_{xyz} \text{ and } r_y \]
\[ \text{the number of fields of type } r_x \text{ and } r_z \]
\[ \text{the number of points in the line of type } r_x \text{ and } r_z \]
\[ \text{the number of pointer-lists} \]
\[ \text{the number of pointers in the pointer-list} \]
\[ \text{the number of integer constants} \]
\[ \text{the number of floating-point constants} \]
\[ \text{the number of table values} \]

For the fields in the LM memory five pointers are needed. The values of these variables for the 3D program are:

\[ nr_{lm\_r\_xyz} = 12 \]
\[ nr_{r\_xyz} = 102 \]
\[ nr_{r\_x} = 3 \]
\[ nr_{r\_z} = 3 \]
\[ nr_{list\_p} = 178 \]
\[ n_{r\_const} = 97 \]
\[ nr_{table} = 300 \]
\[ nr_{r\_y} = 87 + 9 \]
\[ nr_{r\_x\_y} = 10 \]
\[ nr_{r\_z\_y} = 10 \]
\[ l_p = 16 \]
\[ nr_{i\_const} = 68 \]

By substituting these values in equation (9.2), it is found that:

\[ XM = 4282 + 396 \, ny \]  \hspace{1cm} (9.3)

Equation (9.3) gives an upper-bound for the vector length (\( ny \)):

\[ 4282 + 396 \, ny \leq 65536 \Rightarrow ny \leq 154 \]  \hspace{1cm} (9.4)

The size of the required LM memory is given by equation (9.1). The grid-size is not only determined by the size of the LM memory, but also by the number of processor boards. Assuming that there are four processor boards (\( nr\_b = 4 \)), then an upper-bound for the grid-size is obtained, given by:

\[ nx \, ny \left( \frac{1}{4} \, nz + 2 \right) \leq \frac{2^{20}}{24} = 43690 \]  \hspace{1cm} (9.5)

The constraints given by equation (9.4) and (9.5) give a range of grid-sizes which can be used. For example by assuming \( nx = ny = 2nz \) it is found that: \( 64 \times 64 \times 32 \), which is 131072 grid points (and when the LM memory has a size of 4M words (32 bits): \( 105 \times 105 \times 52 \), which is 573300 grid points). When assuming \( nx = ny = nz \) it is found \( 52 \times 52 \times 52 \), which is 140608 grid-points (for LM = 4M: \( 84 \times 84 \times 84 \), which is 592704 grid points). As can be checked, the constraint given by equation (9.4) is never used. Assuming that \( ny = 154 \) and \( nx = nz \) gives a grid with size 28 \times
154 × 28.

9.3. Timing studies
In this paragraph more details are given on the timing of the DSNP for the 3D program. The processing speed is analysed with the asymptotic equation (6.17). Each processor board is performing iterations on a number of 2D planes. For the analysis first one line in one 2D plane is examined. The speed on the 2D planes follows directly from this analysis by adding the data transport cost.

For the calculation of the coefficients and source it is found that 492 variables are transported from the XM memory to the SM memory and 175 variables from the SM memory to the XM memory. The number of separate do-loops is 53, the number of floating-point operations which are performed is 1059. With these parameters the speed of one processor board for the coefficient and source calculation can be estimated. Substitution in equation (6.17) gives: \( \text{nr\_cycles/point} = 2020 \times \text{X-cycles} \). And for the processing speed of the coefficient and source term calculation on four processors it is found that: \( \text{speed} = \frac{1059}{(2020 \times 10^{-7})} = 5.24 \text{ Mflops} \).

Next the effect of the tridiagonal matrix solver on the processing speed is examined. In the flow algorithm 2020 X-cycles are needed to perform 1059 operations for the coefficients and source terms. Seven tridiagonal matrices are solved for every line so the number of operations per point for the tridiagonal matrices is \( 7 \times 13 = 91 \) (for Gauss elimination five multiplications, three subtractions and one division are needed, a division is counted as five operations, giving a total of thirteen). For Gauss elimination by using one processor \( 65.8 \times 7 = 460 \times \text{X-cycles/point} \) are needed, as shown in paragraph 6.5. This is \( \frac{460}{2020 + 460} \times 100\% = 18.5\% \) of the total execution time, but the number of operations is \( 91/1150 \times 100\% = 7.9\% \) of the total number of operations per point. For the speed of the program with Gauss elimination by applying one processor is found that:

\[
\text{speed} = \frac{1150}{(2020 + 460) \times 10^{-7}} = 4.64 \text{ Mflops}
\]

This speed is 11.5% less than the speed for only the coefficients and source terms.

From the discussion in paragraph 6.5 it is clear that the elimination from two sides is used. For the speed of the calculation of one vertical line it is found that:

\[
\text{speed} = \frac{1150}{(2020 + 7 \times 45.6) \times 10^{-7}} = 4.92 \text{ Mflops}
\]

The use of the elimination from two sides algorithm gives an improvement of about 6%.

Above it was found that 2340 X-cycles per point are required for the calculation of one vertical line. Next the transport of data between the XM memory and the LM memory, between the processor boards and between the processor boards and the large memory is examined. The performance decreases because the floating-point calculations are stopped during these data transports. In table 9.1 a summary of the transport cost is given.

The transport between the XM memory and the LM memory is \( \frac{342}{2340} \times 100\% = 14.6\% \) of the calculation time of one point. The transport cost between the
Table 9.1. The data transport cost for the 3D program.

<table>
<thead>
<tr>
<th>transport</th>
<th>speed X-cycles/word (64 bits)</th>
<th>number of variables</th>
<th>number of X-cycles/point</th>
</tr>
</thead>
<tbody>
<tr>
<td>XM $\rightarrow$ LM</td>
<td>6</td>
<td>43 + 14</td>
<td>342</td>
</tr>
<tr>
<td>proc. board $\rightarrow$ proc. board</td>
<td>18</td>
<td>14</td>
<td>252</td>
</tr>
<tr>
<td>large mem. $\rightarrow$ LM</td>
<td>44</td>
<td>14</td>
<td>616</td>
</tr>
</tbody>
</table>

Processor boards and between the large memory and the LM memory can be divided among the number of points in the z-direction on one processor board. If there is one 2D plane on a processor board the cost are (always two iterations are performed on a 2D plane); for proc $\leftrightarrow$ proc: $252/(2 \times 2340) \times 100\% = 5.4\%$ and large memory $\leftrightarrow$ LM: $616/4680 \times 100\% = 13.2\%$. The cost for transport between processor boards can be neglected because in most calculations there are 5 to 10 2D planes on the processor board reducing the cost to less than 1%. The transport cost between the large memory and the LM memory can be neglected because not every sweep data is transported to the large memory, so the 26.8% of data transport cost of the total time is reduced to a very small amount. For the speed of one processor board it can be found approximately that:

\[
\text{speed} = \frac{1150}{(2340 + 342) \times 10^{-7}} = 4.29 \text{ Mflops}
\]

The speed of the four processor board system is 17.15 Mflops.

The 3D calculations can further be analysed. With equation (6.18) the computational intensity $f$ can be found:

\[
f = \frac{op}{\text{transport}} = \frac{1059}{667} = 1.59
\]

Further the value of the half-performance intensity $f_{1/2}$ can be obtained with equation (6.23). For the average I/O per do-loop it is found: $(Nvar_w + Nvar_r)/nr\_loops = (492 + 175)/53 = 12.6$. Substitution of $Nvar_w + Nvar_r = 12.6$ in equation (6.23) gives for the half-performance intensity $f_{1/2}: 1.45$. For the source and coefficient calculation it was found that the processing speed is 5.24 Mflops which is 52.4% of the peak rate and indeed $f$ of the program is larger than $f_{1/2}$. The computational intensity for which the peak rate is obtained is $f = 2.968$.

As noted above the value of the computational intensity has a dominant influence on the efficiency of the processor system. This fact can be applied to other solution algorithms. It is then found that when the computational intensity of these algorithms is close to the algorithm described in chapter 3 the efficiency of the algorithms is also almost equal. This fact is expressed by equation (9.6) which is obtained from equation (6.17) by neglecting the start-up.

\[
speed = \frac{\sum_{i=1}^{nr\_loops} op_i}{\frac{11}{4} \sum_{i=1}^{nr\_loops} (Nvar\_w_i + Nvar\_r_i) \times 10^{-7}} = \frac{40}{11} \times f \times 10^6 \tag{9.6}
\]
where \( f \leq 2.75 \). It should be noted that this is a very global relation, but it gives a good idea about the relation between the speed and the computational intensity. Further the algorithms should of course be based on local interaction. Based on simple theoretical reasons it can be concluded that certainly \( f \geq 0.5 \). This is based on the fact that at least one floating-point operation is performed on the floating-point number, which has to be transported from the XM memory to the SM memory and back. With the computational intensity \( f = 0.5 \) and equation (9.6) it is found that the processing speed is 1.8 Mflops. This value for the speed is 18% of the peak rate. The type of operation which is used for the finite volume method, finite difference methods and finite element methods assures that \( f \geq 1 \). For these methods coefficients are calculated with a large number of variables. For this type of operation a large number of floating-point numbers are transported from the XM memory to the SM memory and only a few variables are transported back. Furthermore, for every transport at least one floating-point operation is performed. Also, it should be noted that when constants are used in an expression there is no transport cost for the constant. For \( f = 1 \) with equation (9.6) it is found that: speed = 3.64 Mflops, which is 36% of the peak rate. This efficiency is already considerably high. In practice, as shown above, the computational intensity will be higher because of a multiple use of variables or the use of constants and the use of divisions (which are counted for five multiplications). It can be concluded that some simple analyses, using the local behaviour, shows that a large efficiency can be obtained on the DNSP for finite volume and finite difference based computations.

9.4. Benchmark results for the computational speed

In this paragraph the results of a benchmark which has been performed on several computer systems is discussed. All the calculations have been performed in double precision. The benchmark has been performed to find out how the DNSP compares with other computer systems. Before discussing the results first some general remarks should be given on benchmarking.

Gentzsch & Neves (1988) state the following on the art of benchmarking supercomputers: the relation between subtle architectural features of modern supercomputers and the computational structures of application programs has been shown to be very critical to performance. This relation is also very sensitive. A small architectural variation can lead to a large performance variation. Using mathematical terminology, the benchmark process can be termed as an unstable process. That is, small variations in the input parameters (such as machine characteristics, compiler performance, library kernel software and even application program input parameters) can result in very large changes in performance. Consequently, even a simple matter of obtaining a performance benchmark of a single application program can be very complex.

Benchmarking is often used to compare (super)computers and to give some kind of ranking. It should be noted from the above statement by Gentzsch & Neves that an absolute ranking of (super)computers does not exist. Only for one specific case this ranking can be obtained. One should keep in mind that a more global ranking depends heavily on statistics. When ranking (super)computers a proper measure has to be defined. For instance, several (super)computers can be compared for one program with one set of input parameters, the result being a list of CPU times in seconds.
These CPU times contain some information but they have no global meaning. To give them global meaning more information should be given on the structure of the program, on the number of various operations, etc. However, in practice giving more information in one specific case can still lead to ambiguity. The question in most cases is: what should be counted? For instance one can count the floating-point operations (Mflops), the instructions (Mips), etc. And for floating-point operations the question is: should every operation on floating-point numbers be counted as one floating-point operation? For instance, should the operation \( a = -a \) be counted as a floating-point operation if \( a \) is a floating-point number? In paragraph 3.2 we already discussed some of our choices. The multiplication is taken as reference and the time for addition and subtraction equal to the time for multiplication. The division is chosen to be equal to five multiplications.

The timing experiments have been done with the 3D program. On each computer system a number of tests were done with varying grid-sizes. The number of grid points in the x-direction and z-direction is fixed, because they have a small influence on the performance. The number of grid points in the y-direction is varied from 10 up to 160 because the performance depends strongly on this number. The input parameters of the program have only small influence on the performance, they only influence the boundary conditions. It is here of course not important to know which problem actually has been solved. The input data has also little influence on the performance. For most of the experiments 20 to 200 sweeps have been run. This has been done to be able to neglect the startup and finishing of the program. For various computer systems more than one processor can be used. In general the efficiency of a processor decreases when more than one processor is used. The calculation with one processor is taken to be the reference for the computer systems. The number of operations that are used for the 3D program is given in paragraph 3.2. It was found that for one grid point 1150 floating-point operations were needed. The processing speed (in Mflops) is obtained in the following way. First the number of floating-point operations for one sweep is counted with:

\[
 nr_{\text{op}} = 1150 \times (nx - 2)(ny - 2)(nz - 2) \tag{9.7}
\]

In equation (9.7) \( nx, ny \) and \( nz \) are reduced with two because on boundary points the balances are not prescribed. Other floating-point operations for the boundary conditions are neglected. We have not used a special algorithm for the tridiagonal matrices on the computer system, except for the DNSP. If we had done so probably an improvement of 10% to 20% could have been obtained.

For every computer system a graph with a number of points is obtained which represents the performance. From this graph the half-performance vector length \( \bar{n}_{\bar{y}} \) and the asymptotic speed \( \bar{F}_{\infty} \) can be obtained. With equation (5.4) the following equation can be obtained:

\[
 \bar{F} = \frac{\bar{F}_{\infty}}{1 + \frac{\bar{n}_{\bar{y}}}{n}} \tag{9.8}
\]

where \( \bar{F} \) and \( n \) are respectively the performance (in Mflops) of the combined memory
and arithmetic pipeline and the vector length. For the 90% performance vector length it can be derived that: $\tilde{n}_{0.9} = 9 \tilde{n}_{\frac{1}{3}}$. From each point on the graph a linear equation is obtained for $\tilde{r}_\infty$ and $\tilde{n}_{\frac{1}{3}}$. Of course more than two linear equations are obtained. There are two possibilities for the solution of this system of equations. The first possibility is that equation (9.8) is the right performance model for the computer system. In that case the system of equations resulting from the timing result give an exact value for $\tilde{r}_\infty$ and $\tilde{n}_{\frac{1}{3}}$. However, in the other case it is not possible to find a solution of the system of linear equations. But still a solution can be obtained which represents the data as well as possible. Therefore the least squares method is used in which the linear equations are squared and added. Next the minimum of the sum can be obtained giving the value for $\tilde{r}_\infty$ and $\tilde{n}_{\frac{1}{3}}$. When the graph points deviate too much from a curve as given by equation (9.8) then the values for $\tilde{r}_\infty$ and $\tilde{n}_{\frac{1}{3}}$ deviate too much from the real value.

The equation (9.8) is given in figure 9.2 for several values of $\tilde{n}_{\frac{1}{3}}$. Because the graph of $\tilde{r}$ and $n$ can deviate too much from a graph as given by equation (9.8), a second method is chosen to obtain the values for $\tilde{r}_\infty$ and $\tilde{n}_{\frac{1}{3}}$. For this method $\tilde{r}_\infty$ is taken to be the largest value for $\tilde{r}$ in the set of graph points. And next $\tilde{n}_{\frac{1}{3}}$ is obtained by finding the value for $n$ for which $\tilde{r}$ is half the value for $\tilde{r}_\infty$ (using linear interpolation between the graph points).

In figure 9.3 the results for the DNSP, Convex C240, CRAY-X-MP/2, Stellar GS1000 and Alliant FX/4 are given. The processing speed ($\tilde{r}$ normalised with $\tilde{r}_\infty$ which is obtained from the graph points with the least squares method) is shown as a function of the vector length $n$. It should be noted that the scaling of the graphs is changed when another $\tilde{r}_\infty$ is used. In figure 9.4 the results for the Convex C240 are given, and in figure 9.5 the results of the Alliant FX/4. The results as depicted in figure 9.3 are summarised in table 9.2 and table 9.3. The asymptotic performance, the half-performance vector length (calculated with the least squares method and the heuristic method) and the 90% performance vector length are given in table 9.2. The efficiency and performance/cost are given in table 9.3. In the sequel the values obtained with the least squares method are considered.

The first system to be discussed is the DNSP. For the timing experiment one processor board is used. Timing is started right after starting the calculations and stopped right afterwards. For these experiments $10^3$ sweeps are performed because control (which controls the I/O between the host computer and the DNSP) may cause some delay. The vector length could not be chosen larger than 154 as found in the preceding paragraph. The processing speed for a four board system can be found by multiplying the processing speed of one processor board with four (giving 16.36 Mflops).

Several timing experiments have been done on the CRAY-X-MP/2 in Bracknell, U.K (August 1989). The operating system used is UNICOS, version 4.0.6 and the fortran compiler used is CFT77 version 3.0.2. For the experiment one processor is used. Most of the code vectorised well except for the coefficient routines and solve routines. It was found that loops which contain vector operations but also recurrences are not partially vectorised, although this is possible. Further the use of temporary scalar variables which are assigned at the end of the loop and used at the beginning of the next
Figure 9.2. The performance $\bar{r}$ as a function of the vector length $n$ for a number of half-performance vector lengths ($\bar{n}_{h} = 5, 10, 15, 20, 25, 30, 35$), as given by equation (9.8).

Figure 9.3. The performance $\bar{r}$ as a function of the vector length $n$ for several computer systems.
Figure 9.4. The performance $\bar{r}$ as a function of the vector length $n$ for the Convex C240 for several compiler options.

Figure 9.5. The performance $\bar{r}$ as a function of the vector length $n$ for the Alliant FX/4 for several compiler options.
Table 9.2. The asymptotic performance and half-performance vector length for several computer systems.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Least squares</th>
<th>Maximum perf.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\bar{r}_\infty$ (Mflops)</td>
<td>$n_{1/2}$</td>
</tr>
<tr>
<td>DSNP</td>
<td>4.33</td>
<td>8.4</td>
</tr>
<tr>
<td>CRAY-X-MP/2</td>
<td>54.56</td>
<td>20.6</td>
</tr>
<tr>
<td>Convex C240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td>10.33</td>
<td>11.0</td>
</tr>
<tr>
<td>-O3 -ep2</td>
<td>21.52</td>
<td>32.9</td>
</tr>
<tr>
<td>-O3 -ep3</td>
<td>27.89</td>
<td>33.9</td>
</tr>
<tr>
<td>-O3 -ep4</td>
<td>32.11</td>
<td>29.3</td>
</tr>
<tr>
<td>Alliant FX/4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O -DAS</td>
<td>3.11</td>
<td>21.2</td>
</tr>
<tr>
<td>-c1</td>
<td>1.22</td>
<td>7.1</td>
</tr>
<tr>
<td>-c2</td>
<td>2.03</td>
<td>11.9</td>
</tr>
<tr>
<td>-c3</td>
<td>2.70</td>
<td>19.3</td>
</tr>
<tr>
<td>-c4</td>
<td>3.13</td>
<td>22.0</td>
</tr>
<tr>
<td>Stellar GS1000</td>
<td>4.23</td>
<td>9.0</td>
</tr>
</tbody>
</table>

Table 9.3. The efficiency and performance/cost for several computer systems.

<table>
<thead>
<tr>
<th>Computer</th>
<th>peak rate (Mflops)</th>
<th>$\bar{r}_\infty$ (Mflops)</th>
<th>%</th>
<th>cost (dollars)</th>
<th>perf/cost flops/dollar</th>
<th>$\bar{r}_\infty$ versus DSNP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSNP</td>
<td>40</td>
<td>16.36</td>
<td>41%</td>
<td>80</td>
<td>204</td>
<td>1</td>
</tr>
<tr>
<td>CRAY-X-MP/2</td>
<td>235</td>
<td>48.32</td>
<td>20.6%</td>
<td>8000/2</td>
<td>12.08</td>
<td>2.95</td>
</tr>
<tr>
<td>Convex C240 (-O2)</td>
<td>50</td>
<td>9.66</td>
<td>19.3%</td>
<td>3000/4</td>
<td>12.88</td>
<td>0.59</td>
</tr>
<tr>
<td>Alliant FX/4 (-c1)</td>
<td>11.8</td>
<td>1.17</td>
<td>9.9%</td>
<td>250/4</td>
<td>18.72</td>
<td>0.07</td>
</tr>
<tr>
<td>Stellar GS1000</td>
<td>40</td>
<td>4.06</td>
<td>10.2%</td>
<td>175</td>
<td>23.2</td>
<td>0.25</td>
</tr>
<tr>
<td>HP9000-835</td>
<td>--</td>
<td>1.32</td>
<td>--</td>
<td>60</td>
<td>22.0</td>
<td>0.08</td>
</tr>
</tbody>
</table>

loop gave some problems (carry-around scalar). After rearranging the do-loops they were vectorised well. For the timing of each case 20 sweeps were done and further the timing routine SECOND() was used. Due to the use of the timing routine overhead can be neglected.

Next we discuss the experiments with the Convex C240 (September 1989) of the TU Delft. The fortran compiler FC version V5.0 (release 1 August 1989) was used. Several experiments have been done with one processor (-O2, only vectorisation) and with more than one processor (-O3 -ep i, vectorisation and parallelisation, with i processors). All tests were done while there were no other users on the computer system. It was found that all the subroutines vectorised well, even those that didn't vectorise on the CRAY-X-MP/2. For the timing of each case 20 sweeps were done and further the timing routine cputime() was used. For the cases with more than one processor
the processing speed was obtained by dividing the total CPU-time by the number of processors. This time is smaller than the turn-around time. In fact we are only interested in the total use of the CPU’s because it gives the potential of the system.

For the experiments on the Alliant FX/4 (September 1989) a system (with four processors) of the faculty of Mathematics and Informatics of the TU Delft was used. The Alliant FX/Fortran Compiler V4.1.35 (2.24.D30) was used. Several compiler options were used such as: -O -DAS which is vectorisation, parallelisation and optimisation for associative transformations (sum, dot product). Further the option -ci was used where i is the number of processors, which means that for i=1 only vectorisation is done and for i > 1 also parallelisation. For each case 20 sweeps were done. For the timing the UNIX utility time was used. Of course in this way there is some overhead, but we found that it could be neglected.

The last vector machine to be discussed is the Stellar GS1000 (December 1989). We did use the operating system Stellix Rel. 2.0 with the fortran compiler Version 21.2. We did 200 sweeps for each case and did use the UNIX utility time. There are several compiler options for optimisation, we did use -O2 which uses one processor. Not all of the fortran code was vectorised right away. Some problems appeared for the coefficient routines as did with the CRAY-X-MP/2. After rearranging the do-loops the code vectorised well.

Further, a test was done on a HP9000-835 system (June 1989). The operating system HP-UX Release A.B3.10 and its fortran compiler was used. For optimising the assembly code, the -O compiler option was used, which inhibits of course not vectorisation. A peak rate for the HP9000-835 was not found.

Now that the benchmark experiments have been discussed, the results and some conclusions can be given. The graph points for the DNSP in figure 9.3 lie on a smooth graph, which is due to the fact that measurements of the speed on the DNSP cannot be disturbed by other users. The graph points follow equation (9.8) rather well. For the CRAY-X-MP/2 the graph is not a smooth line, therefore equation (9.8) does not fit well. The graph of the CRAY-X-MP/2 lies a bit lower because the asymptotic performance which is found with the least squares method is rather high. The Stellar, Convex C240 and Alliant FX/4 have almost the same shape for the graph. In figure 9.4 several results of the Convex C240 with different compiler options are listed. The option -O2 is the best followed by the option -O3 with 2, 3 and 4 processors. The figure further shows that for option -O2 the asymptote is reached around a vector length of 128, while with more than one processor it is reached later. Further the shapes of the -O3 graphs are almost the same. In figure 9.5 the results of the Alliant FX/4 are given for several compiler options. The option -O -DAS and -c4 are almost equal, option -DAS has no influence. Furthermore, the order of the graphs is what we expected, the one processor case being the highest, etc.

In table 9.2 the timing results are summarised. The results of the two methods for finding $\bar{r}_s$ and $\bar{r}_b$ are listed. As shown, the Alliant FX/4 with the compiler option -c1 has the smallest half-performance vector length, followed by the DNSP and the Stellar GS1000. The half-performance vector length of the CRAY-X-MP/2 is 2.5 times as large as for the DNSP. Furthermore, we see that the half performance vector length ($\bar{r}_b$) for the Alliant FX/4 is increased when the number of processors is
enlarged. For the Convex C240 using four instead of one processor gives an improvement of the asymptotic speed of about 3.1 (which is 78% efficiency), the half-performance vector length for 2, 3 and 4 processors is almost equal. For the Alliant FX/4 the improvement is 2.6 (efficiency 64%). In table 9.3 one sees that the DNSP runs at 41% of the peak rate. For the asymptotic performance the values of the heuristic method have been taken. The efficiency of the CRAY-X-MP/2 and the Convex C240 is around 20%, for the Alliant FX/4 it is 10%. In the fifth column of the table the cost for the systems as estimated by us are listed. The real cost might deviate from these values because they change very quickly and buyers can get different reductions. Furthermore, the supercomputer manufacturers want to make profit, so the real cost of the systems will be lower than the list price. In the table we have listed the list price (December 1989). Of course the cost might also deviate because the systems have for example not an equal number of disks or other features. We have tried to reduce this effect as much as possible. Further the percentage of the cost which is spent for these features may be close together. And in fact we are only interested in the order of magnitude, because we want to compare the performance/cost of the systems. The performance/cost of the CRAY-X-MP/2 and the Convex C240 are close together just as for the Stellar GS1000 and the HP9000-835, which is almost twice as good as for the CRAY-X-MP/2 or Convex C240. It should be noted that the efficiency of the Alliant FX/4 is rather low while its performance/cost is rather high compared with the Convex C240 and the CRAY-X-MP/2. In general the table shows that the performance/cost of the DNSP is an order of magnitude higher than the performance/cost of the general purpose computers. However, it should be noted that there is a difference between the manufacturing price and the commercial price, which is said to be around a factor three. Then the performance/cost is about 70, which still is much better than the other (mini-)supercomputers. The difference between the general purpose computers is small.

For the DNSP more conclusions can be drawn. First it should be noted that the asymptotic performance found with the least squares method (4.33 Mflops) is very close to the theoretical asymptotic speed found in paragraph 9.3 (4.29 Mflops). For the 2D program it was found that the asymptotic speed $\tilde{r}_\infty = 4.03$ Mflops and that the half-performance vector length $\bar{n}_{xy} = 10.8$, while the theoretical value of the processing speed was 4.08 Mflops. The asymptotic speed of the 3D program is 7.4% higher than the asymptotic performance of the 2D program which can be accounted for by the fact that the computational intensity of the 3D program ($f = 1.588$) is 5% higher than for the 2D program ($f = 1.513$). Furthermore, the I/O between the XM memory and the SM memory for the 3D program is more efficient because the I/O per loop is 12.6 for the 3D program (as found in paragraph 9.3.1) and 12.16 for the 2D program (as found in paragraph 8.3.1), which is 3.6% higher. With equation (6.15) it can be found that this gives an improvement of 0.3%, which is rather small. The half-performance vector length of the 3D program is smaller than for the 2D program. This is probably due to the fact that the boundary calculations for the 3D program are of the same size as for the 2D program. While the calculations per point for the 3D program are larger. Furthermore, the I/O per loop is larger for 3D.

The difference between the theoretically obtained processing speed and the measured speed is very small. This is due to two facts. First equation (5.7) which is used
describes the behaviour of the processors very well. Secondly, equation (6.13) together with the transport equations (6.1) and (6.7) describe the behaviour of the processor boards with enough detail. It can be concluded that with some basic knowledge of the algorithm, such as operation count and I/O, timing on the DNSP can be obtained.

9.5. Calculations for the turbulent flow in a 3D cavity

As illustration of the capabilities of the DNSP, calculations have been done on one specific cavity configuration. Results of these calculations have also been reported in Van der Vlugt et al. (1990). For this specific cavity configuration also several experiments have been done, as described by Lankhorst (1991). In this paragraph attention will be paid to two subjects. The first subject is the accuracy of the obtained solutions. The DNSP with its large memory and high speed give the opportunity to perform grid refinement routinely. Secondly, it is interesting to know more about the 3D structure of the flow. Several vectorplots will be shown to clarify the flow structure.

The cavity discussed here is filled with air and has the dimensions: 1m × 1m × 0.1m. The left wall has a temperature of 45.8°C and the right wall 5.4°C. For the thermal boundary conditions of the bottom and top walls experimentally obtained temperature distributions have been used. The front and back wall have a thickness 6·10⁻³m and a conductivity of 0.9 W/mK, thus heat is conducted through the walls. The walls are built-up of one layer of grid cells. It is assumed that gradients of the temperature are small enough to give accurate solutions. The heat transfer to the ambient is set to zero. The Rayleigh number for the above described situation is \( Ra = 3.88 \cdot 10^9 \). The flow in the experiments and calculations is turbulent.

For the calculations following the grid distribution has been used for the \( x_1 \) direction:

\[
\frac{x_1(k)}{H} = \frac{1}{2} \left[ 1 + \frac{\tanh[\alpha_1 ((k-1)/(N_i-1)-\frac{1}{2})]}{\tanh(\alpha_1/2)} \right] \quad k = 1, \ldots, N_i \tag{9.9}
\]

where \( \alpha_1 \) follows from \( \alpha_2 = \alpha_1/sinh(\alpha_1) \). For the \( x_2 \)- and \( x_3 \)-direction the grid distribution given by equation (8.9) is used. For the grid in the \( x_1 \)-direction there are more grid points close to the walls than for the \( x_2 \)- and \( x_3 \)-direction. This has been done because there are thin boundary layers along the left and right wall. Solutions of the problem have been obtained on the following grids: 20 × 20 × 10, 20 × 20 × 20, 40 × 40 × 20, 40 × 40 × 40, 60 × 60 × 30 and 45 × 45 × 20.

To get some idea of the structure of the 3D flow several figures of the flow for a 45 × 45 × 20 grid will be shown and discussed. A vectorplot of the velocities and isotherms on the mid-plane of the cavity (\( x_3 = \text{const} \)) is given in figure 9.6. The flow concentrates in boundary layers along the walls and the core is thermally stratified. The order of magnitude of the velocities is 0.2 m/s. In figure 9.7 the velocities and isotherms on the mid-plane with \( x_1 = \text{constant} \) are given. The horizontal coordinate is enlarged by a factor ten. A vortex appears because the fluid near the front and back wall is heated less than the fluid in the centre of the left wall. This is due to the fact that heat is conducted through the front and back wall, and the flow velocities are reduced because of the extra stress effects of the front and back wall. The flow
Figure 9.6. The flow field on the mid-plane for $x_3 = \text{constant}$ for $Ra = 3.88 \cdot 10^9$; (a) the velocities (order 0.2 m/s), (b) the isotherms.
Figure 9.7. The flow field on the mid-plane for $x_1 = \text{constant}$ for $Ra = 3.88 \cdot 10^9$; (a) the velocities (order 0.01 m/s), (b) the isotherms.
Figure 9.8a. The Nusselt number along the vertical line in the middle of the left wall for $Ra = 3.88 \cdot 10^9$.

Figure 9.8b. The $v$ velocity on the line from the middle of the left wall to the center of the cavity for $Ra = 3.88 \cdot 10^9$. 
concentrates in the corners and is about 20% of the main velocities. So there are 3D effects, however, they are an order of magnitude smaller than the main flow quantities.

Several characteristics of the flow for the grids are given in table 9.4. \(Nu_l\) is the Nusselt number (dimensionless heat transfer) averaged over a vertical line in the center of the left wall, \(Nu_f\) is the mean Nusselt number on the left wall, \(v_{\text{max}}\) is the maximum \(v\) velocity on a line from the center of the left wall to the center of the cavity and \(u_{\text{max}}\) is the maximum \(u\) velocity on a line from the center of the bottom wall to the center of the cavity. As can be seen in the table the solutions on the three finest
grids deviate just a little for the characteristic numbers. The maximum deviation is about 2%. Of course this does not prove that the solution on the finest grid \((60 \times 60 \times 30)\) is close to the solution of the difference equations for \(h \to 0\), where \(h\) is some measure for the grid spacing. We try to give more evidence by examining three profiles. In figure 9.8a the Nusselt number along the vertical line in the center of the left wall is given for the three grids. In figure 9.8b the \(v\) velocity on the line from the middle of the left wall to the center of the cavity is given, and in figure 9.8c the \(u\) velocity on the line from the middle of the bottom wall to the middle of the top wall is given. These three profiles represent the characteristics of the flow and they are used to check the accuracy of the solution. One should realise that in fact the seven (variable) fields should be examined on the entire domain, but the three profiles are considered to be a good representation. The \(20 \times 20 \times 10, 40 \times 40 \times 20\) and \(60 \times 60 \times 30\) grids have been compared. To find the accuracy of the solution (profiles) some error estimation should be done. It is expected that the error in the solution in every point behaves as \(O(h^p)\), where \(p\) is constant. We tried to find \(p\) but found that the solution on the \(20 \times 20 \times 10\) grid is not in the asymptotic region. In fact a solution on a \(80 \times 80 \times 40\) grid is needed, assuming that the solution of the \(40 \times 40 \times 20\) grid lies already in the asymptotic region. However, a solution on a \(80 \times 80 \times 40\) grid cannot be obtained with the DNSP at this moment, as shown in paragraph 9.2. So the grid cannot be made fine enough to find the accuracy of the solution. On the other hand we think that the three profiles give enough evidence to conclude that the solution on the \(60 \times 60 \times 30\) grid is close to the solution of the difference equations for \(h \to 0\).

It can be concluded that large 3D calculations, as shown above, can be done routinely on the DNSP. The calculation for the \(60 \times 60 \times 30\) calculation cost about 8 days, requiring about 60,000 field iterations. Comparison with experimental results is given by Lankhorst (1991). Lankhorst shows that the 3D solution compares better to the experimental values for the \(v\) velocity profiles and turbulence intensities in the vertical boundary layer than the 2D solutions.
10. FINAL REMARKS AND CONCLUSIONS

The implementation of a flow algorithm (2D and 3D) on the DNSP (AT&T's/Delft Navier-Stokes Processor) has been discussed. It was found that the implementation of the flow algorithms on the DNSP was time consuming. Special macros and subroutines were designed to reduce errors and implementation time and to find errors in the programs a special test-facility had to be designed.

The 2D flow algorithm has been used to solve several natural convection problems, i.e. the square cavity differentially heated over the vertical walls and bottom and top wall adiabatic. For $Ra = 10^6$ the results compared very well with the benchmark results of De Vahl Davis (1983). Results for higher Rayleigh numbers showed agreement with results of Henkes & Hoogendoorn (1989).

With the 3D flow algorithm several calculations were done for the cavity. The left and right wall are differentially heated, the bottom and top wall have an experimentally obtained temperature and the front and back wall conduct heat but heat transfer to the ambient is set to zero. The accuracy of the obtained solutions is investigated and it was found that the flow is mainly 2D with some smaller but still important 3D effects. Comparison with experimental results is given by Lankhorst (1991).

For the 2D flow algorithm an asymptotic speed is obtained of 4.03 Mflops, which is approached for small vector lengths because of the small half-performance vector length (10.8). For the 3D flow algorithm a 10% higher asymptotic speed (4.33 Mflops) is obtained, with a half-performance vector length of 8.4. It was found that the DNSP performs well for local interaction problems with a high computational intensity ($f \approx 3$) The theoretical timing model of the processor boards describes its behaviour very well. With the timing model the performance of the other algorithms on the DNSP can be analysed.

Comparison of the DNSP with other (mini-)supercomputers shows that the cost/performance of the DNSP is at least an order of magnitude better. The DNSP is more efficient, it reaches about 40% of its peak rate while this is 10% to 20% for other (mini-)supercomputers. Furthermore, due to its specialised character it is cheaper than other (mini-)supercomputers. For the DNSP a longer implementation time for a code is needed compared with the (mini-)supercomputers, but in cases where only one simple algorithm is used for a large number of time consuming calculations, this extra effort is worth doing.

For future use of algorithm oriented processors there are strong demands on the cost, size of memories, processing speed, programming and architecture. A next algorithm oriented processor should have a more user friendly compiler, which should take care of the database organisation and data transport. This will also improve the testability of the software and reduce errors in programs. The fixed specialised architecture of the algorithm oriented processor can appear to be inflexible in several cases, e.g. the implementation of matrix inversion algorithms can be very time consuming or almost impossible. Processing speed, cost and memory size are closely related to other (mini-)supercomputers and the involved problems. We think that this could lead
to successful use of algorithm oriented processors in future. However, commercialisation still requires a long way to go.
REFERENCES


Almasi, G.S., Overview of parallel processing, Parallel Computing 2, 1985, 191-203.


Hockney, R.W. & Jesshope, C.R., *Parallel Computers (Architecture Programming and
Hockney, R.W. & Curington, I.J., $f_{\frac{1}{2}}$ : A parameter to characterize memory and communication bottlenecks, Parallel Computing 10, 1989, 277-286.
Eight GAMM-Conf. on Num. Meth. in Fluid Mechanics, Vol. 29, Vieweg, 1990, 544-553.
for $i = 1$ to $N/2$ do
  term = $1.0 / [d(i) - l(i) * r(i-1)]$
  $r(i) = r(i) * \text{term}$
  $lhs(i) = [lhs(i) - l(i) * lhs(i-1)] * \text{term}$
od

for $i = N/2$ to $1$ step $-1$ do
  $\phi(i) = lhs(i) - r(i) * \phi(i+1)$
od

Algorithm 6.1. The Gauss elimination with one processor.

for $k = 1$ to $\log_2(N)$ do
  for $i = 2^k$ to $N$ step $2^k$ do
    $p = - l(i, k-1) / d(i-2^k-1, k-1)$
    $q = - r(i, k-1) / d(i+2^k-1, k-1)$
    $l(i, k) = p * l(i-2^k-1, k-1)$
    $r(i, k) = q * r(i+2^k-1, k-1)$
    $d(i, k) = d(i, k-1) + p * r(i-2^k-1, k-1) + q * l(i+2^k-1, k-1)$
    $lhs(i, k) = lhs(i, k-1) + p * lhs(i-2^k-1, k-1) + q * lhs(i+2^k-1, k-1)$
od
  od

for $k = \log_2 N$ to $0$ step $-1$ do
  for $i = 2^k$ to $N$ step $2^{k+1}$ do
    $\phi(i+1) = [lhs(i, k) - l(i-2^k, k) * \phi(i-2^k) + r(i+2^k, k) * \phi(i+2^k)] / d(i, k)$
od
  od

Algorithm 6.2. The cyclic reduction algorithm.
\[ k = N/2 \]
for \( j = 1 \) to \( k \) step 1 do
\begin{align*}
\text{case P1:} \\
& h = j \\
& d(h) = 1.0 / [d(h) - l(h) \times r(h-1)] \\
& r(h) = d(h) \times r(h) \\
& \text{lhs}(h) = d(h) \times [\text{lhs}(h) - l(h) \times \text{lhs}(h-1)] \\
\text{case P2:} \\
& h = N + 1 - j \\
& d(h) = 1.0 / [d(h) - r(h) \times r(h+1)] \\
& r(h) = d(h) \times l(h) \\
& \text{lhs}(h) = d(h) \times [\text{lhs}(h) - r(h) \times \text{lhs}(h+1)]
\end{align*}
\text{od}

\[ \phi(k) = [\text{lhs}(k) - r(k) \times \text{lhs}(k+1)] / [1 - r(k) \times r(k+1)] \]
\[ \phi(k+1) = [\text{lhs}(k+1) - r(k+1) \times \text{lhs}(k)] / [1 - r(k) \times r(k+1)] \]

for \( j = 2 \) to \( k \) step 1 do
\begin{align*}
\text{case P1:} \\
& h = k + 1 - j \\
& \phi(h) = \text{lhs}(h) - r(h) \times \phi(h+1)
\end{align*}
\text{case P2:} \\
\begin{align*}
& h = k + j \\
& \phi(h) = \text{lhs}(h) - r(h) \times \phi(h-1)
\end{align*}
\text{od}

Algorithm 6.3. The elimination from two sides.
\( k = \frac{N}{4} \)
for \( i = 0 \) to \( 3 \) do
\[
\begin{align*}
\text{r}(i*k) &= 0; \text{lhs}(i*k) = 0; \quad \text{l}(i*k) = 1 \\
\text{od}
\end{align*}
\]
for \( i = 0 \) to \( 3 \) do concurrent
\[
\begin{align*}
\text{for } j = 1 \text{ to } k \text{ do} \\
\text{h} &= i*k+j \\
\text{d}(h) &= 1.0 / [\text{d}(h) - \text{l}(h) * \text{r}(h-1)] \\
\text{l}(h) &= -\text{l}(h) * \text{l}(h-1) * \text{d}(h) \\
\text{r}(h) &= \text{r}(h) * \text{d}(h) \\
\text{lhs}(h) &= \text{lhs}(h) - \text{lhs}(h-1) * \text{l}(h) * \text{d}(h) \\
\text{od}
\end{align*}
\]
\text{od}
\]
for \( i = 0 \) to \( 3 \) do concurrent
\[
\begin{align*}
\text{for } j = k-2 \text{ to } 1 \text{ step } -1 \text{ do} \\
\text{h} &= i*k+j \\
\text{l}(h) &= \text{l}(h) - \text{l}(h+1) * \text{r}(h) \\
\text{lhs}(h) &= \text{lhs}(h) - \text{lhs}(h+1) * \text{r}(h) \\
\text{r}(h) &= -\text{r}(h+1) * \text{r}(h) \\
\text{od}
\end{align*}
\]
\text{od}
\]
for \( i = 0 \) to \( 3 \) do concurrent
\[
\begin{align*}
\text{j} &= 0; \quad \text{h} = i*k+j \\
\text{lhs}(h) &= \text{lhs}(h) - \text{lhs}(h+1) * \text{r}(h) \\
\text{d}(h) &= 1 - \text{l}(h+1) * \text{r}(h) \\
\text{r}(h) &= - \text{r}(h+1) * \text{r}(h) \\
\text{r}(h) &= \text{r}(h) / \text{d}(h) \\
\text{lhs}(h) &= \text{lhs}(h) / \text{d}(h) \\
\text{l}(h) &= \text{l}(h) / \text{d}(h) \\
\text{od}
\end{align*}
\]
solve the system:
\[
\begin{align*}
\phi(k) + l(k)\phi(2k) &= \text{lhs}(k) - \text{r}(k)\phi(0) \\
\text{r}(2k)\phi(k) + \phi(2k) + l(2k)\phi(3k) &= \text{lhs}(2k) \\
\text{r}(3k)\phi(2k) + \phi(3k) + l(3k)\phi(4k) &= \text{lhs}(3k) \\
\text{r}(4k)\phi(3k) + \phi(4k) &= \text{lhs}(4k) - l(4k)\phi(4k+1)
\end{align*}
\]
for \( i = 0 \) to \( 3 \) do concurrent
\[
\begin{align*}
\text{for } j = 1 \text{ to } k-1 \text{ do} \\
\text{h} &= i*k+j; \quad \text{l} = (i+1)*k; \quad \text{m} = i*k \\
\phi(h) &= \text{lhs}(h) - \text{l}(h) * \phi(m) - \text{r}(h) * \phi(l) \\
\text{od}
\end{align*}
\]
Algorithm 6.4. The modified Wang algorithm.
SUMMARY

This thesis describes the implementation and analysis of a flow algorithm on the DNSP (AT&T's/Delft Navier-Stokes Processor). The flow algorithm is used for the solution of natural convection problems in 2D and 3D cavities. The flow in the cavity is described by the Navier-Stokes equations, the continuity equation and the energy equation. Turbulence is modelled with the k-ε model for which two extra convection-diffusion equations are added. The partial differential equations are discretised by using a finite volume method. For the pressure-velocity coupling the SIMPLE method is used. The equations are discretised on one vertical line in the calculation domain, after solution of the difference equations one of its neighbouring lines is to be solved.

Some basics of parallel computers are discussed briefly such as the taxonomy of Flynn. The definition of an algorithm oriented processor is given and the advantages and disadvantages are discussed. Algorithm oriented processors are processor systems which are specially designed for efficient use of an algorithm or class of algorithms. The DNSP is very suitable for near neighbour algorithms such as a class of algorithms used for Navier-Stokes problems. One of the special features of an algorithm oriented processor is the tailored use of vector operations and parallel execution. Furthermore, pipelining can be used in various stages of the operations. Because the architecture and algorithm match very well a high processing speed can be obtained.

The DNSP, which is attached to a host computer, consists of a control board, a number of memory boards, and a number of processor boards. The processor boards are connected with each other as a linear processor array. On the processor boards the actual floating-point calculations are performed. A processor board contains 1M words (32 bits) of main memory and has a maximum processing speed of 10 Mflops double precision (64 bits). The DNSP is programmed in assembly. Test runs can be done on a simulator of the DNSP which performs a functional simulation of the DNSP.

Timing of data transport and operations are analysed. Several timing equations are obtained. On the basis of these timing equations an optimal calculation strategy could be found. All data is distributed over the processor boards before starting the calculations and at the end of the calculations the results are moved to the host computer. The calculations on the processor boards with four processors are analysed with the timing model of Hockney & C Birington. Furthermore, several algorithms for solving tridiagonal matrices are analysed for the DNSP. The elimination from two sides with two processors is the fastest for vector lengths smaller than 113. The implementation of the flow algorithms and several programming techniques, such as for do-loops and do-loops with if-statements, are discussed in more detail. The use of a test-facility, which is necessary for debugging purpose, is discussed.

The 2D domain decomposition method is discussed. The calculation domain is divided into a number of blocks of vertical lines, every processor board calculates the solution on one block. During the calculations, data on the boundary of the blocks is exchanged between the processor boards. The results of several tests showed that the use of the domain decomposition method has no significant influence on the
convergence speed. For the 2D program it is found that solutions can be obtained on grids up to $641 \times 327$ for a four processor board system. The timing model of the processor boards is used to obtain a theoretical asymptotic speed. This asymptotic speed is 4.08 Mflops for one processor board which is very close to the experimentally obtained speed of 4.03 Mflops.

Calculations have been done for the standard square cavity problem. The side walls are differentially heated and the bottom and top wall are adiabatic. For $Ra = 10^6$ the solutions are in very good agreement with the benchmark solution of De Vahl Davis. Furthermore, solutions are obtained for $Ra = 10^{10}$, $10^{11}$ and $10^{12}$ on three grids $45 \times 45$, $90 \times 90$ and $180 \times 180$. The $180 \times 180$ grid is fine enough and it is found that the Nusselt number at half the cavity height scales with $Ra^{1/3}$ and the velocity maximum scales with $u_b Ra^{-1/18}$ as found by Henkes & Hoogendoorn.

For 3D a domain decomposition method is used in which the calculation domain is divided into a number of slices. It is found that on four processor boards solutions can be obtained on grids with sizes $52 \times 52 \times 52$. With the timing model the asymptotic speed of one processor board is obtained: 4.29 Mflops.

With the 3D program calculations have been done for the flow in a cavity with dimensions $1m \times 1m \times 0.1m$. The left and right wall are differentially heated, for the bottom and top wall experimentally obtained temperatures are used and the front and back wall conduct heat, but heat transfer to the ambient is set to zero. Calculations have been done on several grids to find the accuracy of the solution. Lankhorst showed that 3D results are closer to experimental results than 2D results.

Benchmark calculations have been performed on several (mini-)supercomputers systems such as the CRAY-X-MP/2, Convex C240, Alliant FX/4, Stellar GS1000, HP9000-835. For the DNSP an asymptotic speed on one processor board of 4.33 Mflops is found and a half performance vector length of 8.4. Only the CRAY-X-MP/2 is faster, with an asymptotic speed of 48.32 Mflops, than a four processor board DNSP (17.32 Mflops). In cost/performance the DNSP is more than an order of magnitude better than all other systems.
SAMENVATTING (summary in Dutch)

Dit proefschrift beschrijft de implementatie en analyse van een stromingsalgorithmie op de DNSP (AT&T's/Delft Navier-Stokes Processor). Het stromingsalgorithmie wordt gebruikt voor het oplossen van natuurlijke convectie problemen in 2D en 3D gesloten ruimten. De stroming in de gesloten ruimte wordt beschreven met behulp van de Navier-Stokes vergelijkingen, de continuiteitsvergelijking en de energievergelijking. Turbulentie wordt gemodelleerd met het $k$-$\varepsilon$ model, waarvoor twee extra convectie-diffusie vergelijkingen toegevoegd moeten worden. De partiële differentiaalvergelijkingen worden gediscretiseerd met behulp van een eindige volume methode. Voor de snelheids-druk koppeling wordt de SIMPLE methode gebruikt. De vergelijkingen worden gediscretiseerd op een verticale lijn in het rekendomein, na het oplossen van de differentievergelijkingen wordt één van de buurlijnen aangepakt.

Enkele basisbegrippen voor parallelle computers worden besproken, zoals de taxonomy van Flynn. De definitie van een algoritme georiënteerde processor wordt gegeven en enkele voordelen en nadeelen worden besproken. Een algoritme georiënteerde processor is een processor systeem dat speciaal ontworpen is voor het efficiënt oplossen van een algoritme of een klasse van algoritmen. De DNSP is zeer geschikt voor naastebuur algoritmen zoals een klasse van algoritmen die vaak gebruikt worden voor Navier-Stokes problemen. Een van de speciale kenmerken van een algoritme georiënteerde processor is het gebruik van vector operaties en parallelle verwerking. Verder kan pipelining toegepast worden op verschillende niveaus in de operaties. Omdat de architectuur en het algoritme zeer goed op elkaar aansluiten kan een hoge rekensnelheid gehaald worden.

De DNSP, die verbonden is met een host computer, is opgebouwd uit een controlekaart, een aantal geheugenkaarten en een aantal rekenkaarten. De rekenkaarten zijn met elkaar verbonden als een lineaire processor array. Op de rekenkaarten vinden de floating-point berekeningen plaats. Een rekenkaart bevat 1M woorden (32 bits) main memory en heeft een maximale rekensnelheid van 10 Mflops dubbele precisie (64 bits). De DNSP wordt in assembler geprogrammeerd. Testberekeningen kunnen op een simulator van de DNSP gedaan worden, deze simulator voert een functionele simulatie uit van de DNSP.

Het tijdgedrag van datatransport en berekeningen zijn geanalyseerd. Verscheidene vergelijkingen voor het tijdgedrag zijn bepaald. Met behulp van deze vergelijkingen kon een optimale rekenstrategie bepaald worden. Alle data wordt over de rekenkaarten verdeeld voordat de berekeningen gestart worden en aan het eind van de berekeningen worden de resultaten naar de host computer gehaald. De berekeningen op een rekenkaart met vier processoren zijn geanalyseerd met behulp van het tijdmoment van Hockney & Cumington. Verder zijn verschillende algoritmen voor het oplossen van een tridiagonale matrix geanalyseerd voor gebruik op de DNSP. Eliminatie van twee zijden met twee processoren bleek het snelst te zijn voor vectoren kleiner dan 113. De implementatie van het stromingsalgorithmie en verschillende programmeertechnieken, zoals voor do-loops en do-loops met if-statements, zijn besproken. Het gebruik van een testfaciliteit, dat nodig is voor het debuggen van het assembler programma, is
besproken.

De 2D domein decompositie methode is besproken. Het rekengebied is opgedeeld in een aantal blokken met verticale lijnen, elke rekenkaart berekent de oplossing in een blok. Tijdens de berekeningen wordt data op de randen van de blokken uitgewisseld tussen rekenkaarten. De resultaten van enkele testberekeningen lieten zien dat de convergentiesnelheid niet significant veranderde als de domein decompositie methode gebruikt werd. Voor het 2D programma kunnen oplossingen verkregen worden op roosters tot 641 \times 327, op een systeem bestaande uit vier rekenkaarten. Het tijdmodel van de rekenkaarten is gebruikt om een theoretische maximum rekensnelheid te bepalen. Deze maximum rekentijd is 4.08 Mflops voor een rekenkaart wat zeer dicht ligt bij de experimenteel verkregen rekensnelheid van 4.03 Mflops.

Berekeningen zijn gedaan voor het standaard probleem van een gesloten vierkante ruimte, de zijwanden hebben een verschillende temperatuur en de bodem en het plafond zijn adiabatisch. Voor \( Ra = 10^6 \) blijkt de oplossing zeer goed overeen te komen met de benchmark oplossing van De Vahl Davis. Verder zijn oplossingen verkregen voor \( Ra = 10^{10}, 10^{11} \) en \( 10^{12} \) op drie rekenroosters 45 \times 45, 90 \times 90 en 180 \times 180. Het 180 \times 180 rooster is fijn genoeg en het bleek dat het Nusselt getal op de halve hoogte goed schaalt met \( Ra^{1/3} \) en het snelheidsmaximum schaalt met \( u_b Ra^{-1/18} \), zoals ook gevonden is door Henkes & Hoogendoorn.

In het 3D geval is een domein decompositie methode gebruikt waarbij het rekendomein is opgedeeld in een aantal plakken. Met vier rekenkaarten kan er op de roosters gerekend worden met een afmeting tot 52 \times 52 \times 52. Met het tijdmodel kon de maximum rekensnelheid op een rekenkaart bepaald worden: 4.29 Mflops.

Met het 3D programma zijn berekeningen gedaan voor de stroming in een gesloten ruimte met dimensies 1m \times 1m \times 0.1m. De linkerk- en rechterwand hebben een verschillende temperatuur, voor de bodem en het plafond worden experimenteel verkregen waarden gebruikt en de voor- en achterwand geleiden warmte, maar er is geen warmteoverdracht naar de omgeving. Er zijn berekeningen gedaan op verschillende roosters om de nauwkeurigheid van de oplossing te kunnen bepalen. Lankhorst liet zien dat de 3D resultaten dichter bij de experimenten liggen dan de 2D resultaten.

Benchmark berekeningen zijn uitgevoerd op een aantal (mini-)supercomputer systemen zoals de CRAY-X-MP/2, Convex C240, Alliant FX/4, Stellar GS1000, HP9000-835. Voor de DNSP is een maximum rekensnelheid gevonden, op één rekenkaart, van 4.33 Mflops en de half-performance vectorlengte is 8.4. Alleen de CRAY-X-MP/2 is sneller dan de DNSP met vier rekenkaarten (17.32 Mflops), met een maximum rekensnelheid van 48.32 Mflops. In cost/performance is de DNSP meer dan een orde grootte beter dan alle andere systemen.
ABOUT THE AUTHOR


From Oktober 1985 he was employed by the Stichting FOM and did his Ph.D. research in the Heat Transfer group of the Faculty of Applied Physics at the Delft University of Technology for four years.

On 15 January 1990 he entered military service, where he is working as research assistant in the group Weapon Effectiveness of the Prins Maurits Laboratory/TNO in Rijswijk.