TNO-report
FSP-RPT-980017

Parallelization of the linear solvers used in WISH3D

TNO Institute of Applied Physics

Stieltjesweg 1
P.O. Box 155
2600 AD Delft
The Netherlands

Phone (+31) 15 269 20 00
Fax (+31) 15 269 21 11

Date
January 1998

Author(s)
S.M.C.M. van Oudenaarde

Project
Literature study for the Graduation Thesis
Project no. 015.782

To
Graduation committee
Prof.dr.ir. P. Wesseling (TU Delft)
Dr.ir. C. Vuik (TU Delft)
Dr.ir. G.P. Boerstoel (TNO-TPD)
ir. W.J.A. Denissen (TNO-TPD)

No of pages : 35 (excl. RDP & Distr. list)

Contents

1 Introduction

TNO (Netherlands Organization for Applied Scientific Research) works on projects paid by industry or government. TNO activities take place at fifteen institutes spread throughout the Netherlands. One of the institutes is TPD (Institute of Applied Physics) in Delft. Process Physics (PF) is one of the departments of the TPD. The research area of PF is simulation of flows of heat transfer and transport of mass species. Furthermore thermo-physical instrumentation is developed for measuring. One of the projects are furnaces for the glass-industry. PF develops a software package WISH3D to simulate the flows and heat transfer in a furnace. The equations used in WISH3D, e.g. Navier Stokes, enthalpy etc., cannot be solved analytically for the complex geometry of a furnace. WISH3D relies on numerical methods to obtain a solution. The numerical model of WISH3D is implemented with the finite volume method (FVM) in combination with the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) method.

Because the large computation time of computational fluid dynamic (CFD) problems, we will work in this report on reducing the wall-clocktime by parallelization of the WISH3D solvers. With parallelization we partition the computational work of WISH3D in parts, and distribute these over a number of processors or workstations, so they can work simultaneously. There are two paradigms to implement a parallel program, namely the message passing and the data parallel paradigm. We will use the data parallel programming paradigm and use High Performance Fortran (HPF). In HPF we can fully concentrate on the parallelism of the code and we do not program explicitly the communication, which must be programmed explicitly using the message passing paradigm.

In a network, workstations are most of the time idle. If we use this wasted CPU-time we can build an abstract parallel computer. The advantage is a cheap parallel computer with an acceptable performance of our parallel code. The HPF-compiler takes care of the distribution and running of the program on the coupled workstations. We will also test the code on a CRAY T3E. Below we give the outline of the topics in this report.

1.1 Outline of this report

We give the chapters of this report pointwise,

- **Mathematical model used in WISH3D**. Before describing work on parallelization of the solvers of WISH3D, we give first the equations of WISH3D used here, and their discretizations.

- **Solving the discretized equations**. In this chapter will follow some explanation about parallel computation, and this is applied to iterative solvers and preconditioners.
- **HPF syntax and tools for implementation**. Here will specify the syntax and uses of HPF, and tools for measuring implemented parallel code.
- **Suggestions and test problems**. Here we will outline further work to be done in this graduation project.

2 Mathematical model used in WISH3D

We discuss some of the equations used in WISH3D which describe the physical processes in the combustion chamber of high temperature furnaces.
The flow is turbulent and therefore a $(k - \epsilon)$ turbulence model with a Favre decomposition is used for low Mach numbers. The $(k - \epsilon)$ model describes the averaged behaviour of the turbulent flow and the Favre decomposition takes care of avoiding density correlation in the transport equation, for more details see [1] , [2].
Variables '$\tilde{\phi}$' are Favre-averages, and variables '$\bar{\phi}$' are Reynolds-averages of the decomposition. In all equations in this chapter the Einstein summation-convention is used.
Conservation of mass leads to the continuity equation :

$$\frac{\partial \bar{\rho}}{\partial t} + \frac{\partial \bar{\rho} \tilde{u}_j}{\partial x_j} = 0, \tag{2.1}$$

in which $\bar{\rho}$ is the Reynolds averaged density of the fluid and $\tilde{u}_j$ the component of the Favre averaged velocity component in the j-direction.
Conservation of momentum in the i-component, leads to the momentum equations :

$$\frac{\partial \bar{\rho} \tilde{u}_i}{\partial t} + \frac{\partial \bar{\rho} \tilde{u}_j \tilde{u}_i}{\partial x_j} = -\frac{\partial \bar{p}}{\partial x_i} - \frac{\partial \tilde{\tau}_{ij}}{\partial x_j} + \bar{\rho} g_i, \tag{2.2}$$

in which $\bar{p}$ is the pressure, g_i the i-component of gravitation and $\tilde{\tau}_{ij}$ the stress tensor as given by:

$$\tilde{\tau}_{ij} = (\mu + \mu_t) \left(\frac{\partial \tilde{u}_i}{\partial x_j} + \frac{\partial \tilde{u}_j}{\partial x_i} \right) + \frac{2}{3} (\mu + \mu_t) \frac{\partial \tilde{u}_k}{\partial x_k} \delta_{ij} + \frac{2}{3} \bar{\rho} \tilde{k} \delta_{ij}. \tag{2.3}$$

Here μ is the molecular dynamic viscosity, δ_{ij} is the Kronecker delta, μ_t is the turbulent viscosity, and k the turbulent kinetic energy, which belong to the turbulence model.
Conservation of energy rewritten in the form of the enthalpy equation gives,

$$\frac{\partial (\bar{\rho} \tilde{h} - \tilde{p})}{\partial t} + \frac{\partial (\bar{\rho} \tilde{u}_j \tilde{h})}{\partial x_j} = \frac{\partial \left((\frac{\lambda}{C_p} + \frac{\mu_t}{\sigma_h}) \frac{\partial \tilde{h}}{\partial x_j} \right)}{\partial x_j} + \tilde{S}_{rad}. \tag{2.4}$$

Here $\tilde{h}$ is the total enthalpy, λ is the thermal conductivity, C_p is the specific heat of the gas mixture at constant pressure, σ_h is the turbulent Prandtl number, and $\tilde{S}_{rad}$ is the radiation source term. For a description of all equations used in WISH3D, we refer to [1], [2].

2.1 Discretization

We can rewrite equations (2.1), (2.2), and (2.4) in the following general form,

$$\frac{\partial \rho\phi}{\partial t} + \frac{\partial J_j}{\partial x_j} = \rho S_\phi,$$
$$J_j = \rho u_j \phi - \Gamma_\phi \frac{\partial \phi}{\partial x_j} \quad , j = 1,2,3. \tag{2.5}$$

The Favre / Reynolds signs above the variables are omitted for shorter notation. For ϕ we substitute the variables of (2.1), (2.2) or (2.4). The flux J_j constitutes of the sum of the convective and diffusive flux ϕ in the x_j-direction.
In WISH3D the discretization is implemented with the finite volume method (FVM), see [4] for more details. The combustion chamber of the furnace is discretized by subdivision of the continuum into a finite number of non-overlapping volume elements (control volumes), which surround a grid node in which all scalar quantities are defined. The grid nodes of the velocity components are defined on the faces of the control volume. Therefore we get a staggered grid, see figure 2.1. In this way the stability of the pressure field is automatically guaranteed.

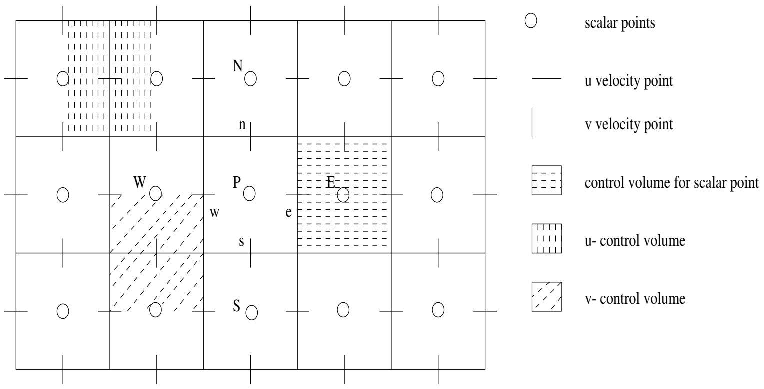

Figure 2.1: 2D staggered grid

The basic idea of FVM is that the differential equation (2.5) is integrated over a volume. With the Gauss theorem we can write the volume integral equation into a surface integral. Thereafter the surface integrals are approximated with the (scalar) values in the grid points. Below we give a more detailed description.
Before applying the FVM, we first need a time integration-scheme for the instationary term. We use the implicit scheme, the advantage of the implicit scheme is that it is stable, also for large time steps, for details see [1]. In the time interval $[t, t + \Delta t]$:

$$\int_t^{t+\Delta t} \frac{\partial \rho\phi}{\partial \tau} d\tau + \int_t^{t+\Delta t} \frac{\partial J_j}{\partial x_j} d\tau = \int_t^{t+\Delta \tau} \rho S_\phi d\tau, \tag{2.6}$$

if $\frac{\partial J_j}{\partial x_j}$, and ρS_ϕ are approximated to be constant in the interval $[t, t+\Delta t]$, we get :

$$([\rho\phi]_P - [\rho\phi]_{P,0}) - \frac{\partial J_j}{\partial x_j}\Delta t = \rho S_\phi \Delta t. \tag{2.7}$$

All variables with a subscript "0" are the old values on time t. Now we apply FVM to (2.7),

$$\int_V ([\rho\phi]_P - [\rho\phi]_{P,0})dV - \int_V \frac{\partial J_j}{\partial x_j}\Delta t dV = \int_V \rho S_\phi \Delta t dV. \tag{2.8}$$

We evaluate this equation per term:
The source term with S_ϕ is approximated by:

$$\int_V \rho S_\phi \Delta t dV \approx \rho S_\phi \Delta t \Delta V. \tag{2.9}$$

The first term of the left-hand side is approximated by:

$$\int_V ([\rho\phi]_P - [\rho\phi]_{P,0})dV \approx ([\rho\phi]_P - [\rho\phi]_{P,0})\Delta V. \tag{2.10}$$

Finally for the second term on the left-hand side, we use the Gauss theorem and get:

$$\int_V \frac{\partial J_j}{\partial x_j}\Delta t dV = \int_A J_j n_j \Delta t dA = \sum_{i=1}^{6} \int_{A_i} J_j n_j \Delta t dA_i \approx \sum_{i=1}^{6} (J_j n_j)_i A_i \Delta t, \tag{2.11}$$

with $(n_1, n_2, n_3)^T$ the outward pointing normal vector at the surface. As finite volume we take a cube, see figure 2.2, therefore the surface integral (2.11) splits into six parts corresponding to the surfaces A_i for $i = 1, ..., 6$ (or denoted with s,n,e,w,t,b resp.). We use Cartesian-coordinates and the cube is oriented along the coordinate axes. For each face of (2.11) in the last term, we have three contributions (Einstein summation convention). Since normal vectors are parallel to the coordinate axes two of these contributions are equal to zero.

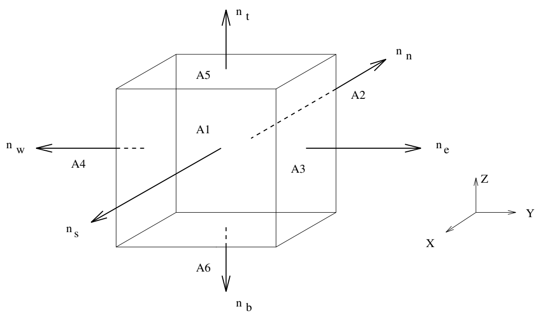

Figure 2.2: 3D volume element

Hence (2.11) becomes:

$$\sum_{i=1}^{6}(J_j n_j)_i A_i \Delta t = \left((J_e A_e - J_w A_w) + (J_n A_n - J_s A_s) + (J_t A_t - J_b A_b)\right)\Delta t. \tag{2.12}$$

All evaluated terms combined give the discretization of a scalar variable in point P,

$$\frac{([\rho\phi]_P - [\rho\phi]_{P,0})\,\Delta V_P}{\Delta t} + J_e A_e - J_w A_w + J_n A_n - J_s A_s + J_t A_t - J_b A_b = \rho S_\phi \Delta V_P. \tag{2.13}$$

The lowercase letters e,w,n,s,t,b denote the six faces of the control volume with area A, and the uppercase letters E,W,N,S,T,B denote the six neighbouring gridpoints of point P, see figure 2.1.
To evaluate the fluxes J_i through the control volume faces we have used the hybrid difference scheme, for more details see [3]. The hybrid scheme is a combination of the central and upwind difference schemes, and dependent on the grid Péclet number (Pe), which is defined as the local ratio of convective flux to diffusive flux. Equation (2.13) can be rewritten to obtain:

$$a_P \phi_P = \sum_{nb} a_{nb}\phi_{nb} + b_P, \tag{2.14}$$

where the summation is over the neighbouring gridpoints E,W,N,S,T,B. The coefficients a_P and b_P are given by:

$$a_P = \sum_{nb} a_{nb} + (\rho_P \Delta t - S_P)\Delta V_P, \tag{2.15}$$

$$b_P = (\rho_{P,0}\phi_{P,0}\Delta t + S_C)\Delta V_P. \tag{2.16}$$

Where the source term is linearized by $\rho S_\phi = S_C + S_P \phi_P$. The neighbour coefficients, with the use of the hybrid scheme, are given as:

$$a_E = D_e max(0; 1 - \frac{1}{2}Pe_e; -Pe_e), \tag{2.17}$$

where Pe_e and D_e are defined as:

$$Pe_e = \frac{\rho_e u_e \Delta x_e}{\Gamma_{\phi,e}} \;\; ; \;\; D_e = \frac{\Gamma_{\phi,e} A_e}{\Delta x_e}. \tag{2.18}$$

Here Δx_e is the internode distance between point P and E, and A_e is the area of the east face of the control volume. The velocity u_e lies on the volume face, due to the staggering. A similar approach is used for the other neighbour gridpoints. Since the values of ρ and Γ are defined in the scalar gridpoints, we need to interpolate them to obtain the value at the volume face. Combining all gridpoints we can rewrite equation (2.14) into a matrix vector form, which gives the following algebraic system of equations:

$$Ax = b, \tag{2.19}$$

in which x contains the unknown values in all gridpoints.

2.2 The SIMPLE method

In the former section the coupled system of non-linear equations is described. The solution of this system gives problems if the pressure field is not given beforehand because there is no explicit equation for solving the pressure. Therefore we use the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) method as described in Patankar [3]. In the SIMPLE method the continuity equation is rewritten into a pressure correction equation.
We begin with an estimated pressure field in the momentum equations to obtain the velocity components. These components are approximations, and the velocity components do not satisfy the continuity equation. The formulated pressure correction is used to correct the velocity field, and also the pressure field. Other flow dependent equations are solved with these corrected velocity and pressure fields. These quantities also influence the velocity and pressure. We begin this process again and start with the improved pressure field. Hence SIMPLE is nothing more than an iterative process, and the velocity field will converge to the one that satisfies the continuity equation, and the pressure field then also converges to the correct pressure field.

The SIMPLE method splits the coupled equations into a number of linear algebraic systems, which must be solved in each (SIMPLE) iteration step. This is a very time consuming process, and therefore we try to parallelize the solvers for reducing the wall-clocktime.

3 Solving the discretized equations

3.1 Basics about parallel computation

The linear systems derived by SIMPLE will be solved by numerical methods in a parallel way. Before we construct the solvers, some knowledge of parallel computing is needed. In a parallel computation the problem is partitioned and divided over the processors available. There are two programming paradigms to construct parallel programs, i.e. message passing and data parallel. In this section we will discuss both, however in the remainder of this report we primarily concentrate on the data parallel paradigm.

Message Passing

A part of WISH3D was implemented as a test-case with PVM, a message passing software package. There were difficulties with the traditional solvers to parallelize. In next paragraph we come back to this issue. For more details about the PVM parallelized version of WISH3D see [11].

The first step by a message passing system is to partition the grid, see [15] for more details. Each part is assigned to a processor which takes care of its own part. On the edges of the partitions communication is needed. The programmer has to implement this communication between the processors. On every processor a separated program is loaded and running, in which the communication is done with routines of the communication library of the message passing package.
The aim is to choose a suitable problem decomposition (partition) to provide an evenly balanced load, with every processor being kept busy, and also to minimise the inter-processor communication.

In summary, the message passing paradigm gives the programmer much freedom in program control but the programmer is responsible for making the program to work and keeping the processors busy. (including a correct and efficient implementation of the communication)

Data Parallel

By a data parallel paradigm the parallelism is based on array operations. The array operations are executed in parallel. Such (operation) routines, are the most efficient implementations often realized by the manufacturer. It will increase the productivity of the programmer by avoiding ”re-inventing the wheel” and even to make simple syntax errors.
Parts of WISH3D are also implemented as test-case in HPF, see [6] for more details.

All communication and data mapping is done by the compiler. The programmer can control the parallelism behaviour. Data parallel programming is concerned with

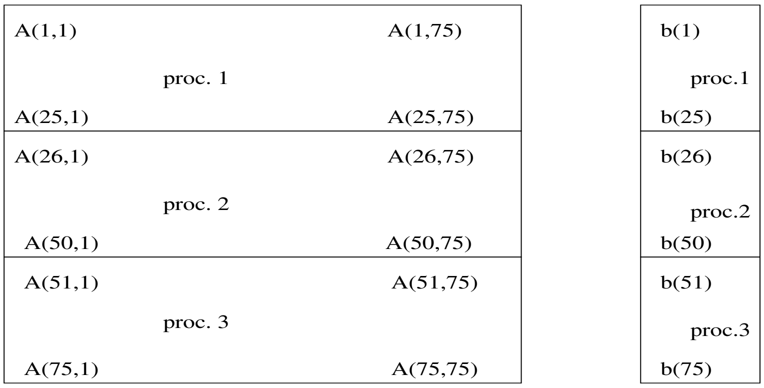

Figure 3.1: partitioning of a matrix A and a vector b

defining collective operations on array or sets of array elements, with these arrays distributed over a number of processors. If the solver for the linear systems can be expressed in terms of such array operations then it is likely that a data parallel implementation will be efficient.

We illustrate this by the following example (example 1):
In this example we take 3 processors, and suppose a matrix A with dimensions (75×75) and a vector b (75×1) are given, see also figure 3.1. Consider the matrix-vector multiplication:

$$c = Ab \tag{3.1}$$

Here A is row-wise distributed over the processors. So the rows $1, ..., 25$ are on processor 1 etc. The vectors have a similar distribution. On each processor a part of the multiplication is done. Consider the first processor, the first 25 columns can be multiplied locally with the local part of the vector b. When this is done processor 1 has to communicate with processors 2 and 3, for the other parts of the vector b. Because with the second part of vector b we can multiply columns 26 till 50, and with the third we finish the multiplication on processor 1. The same approach is used for processors 2 and 3, and this completes the multiplication $c = Ab$. Remember that the answer is also spread over the processors. More details about the structure and how to control HPF are given in the last chapter.

In summary, the programmer can fully concentrate on an application because the low level parallelism and communication is done by the compiler.

3.2 Parallel Iterative methods

The parallel iterative methods considered in this section are (parallelized) traditional iterative methods. Thus no pure parallel iterative methods are described here. To solve the equations of (2.19) iterative methods are used in WISH3D. Iterative methods are used instead of direct methods. In direct methods the computation time is of order n^2 for banded matrices, and in general n is large. We try to construct the iterative methods so that they approximate the solution in a fraction of n steps. Per iteration the amount of work is proportional with n. This will cost less computation time.

We will start here with the Generalized Minimal Residual method (GMRES). This iterative method is difficult to parallelize. The thought behind this is, if we can construct a good parallel implementation of this method, we can construct it for many other iterative methods as well. But first we mention the robust TDMA-solvers.

3.2.1 TDMA Solvers

In the sequential version of WISH3D the TDMA solvers are used. In [1], the TDMA solvers are specified in detail. TDMA solvers look like Block Gauss Seidel solvers in 2D (plane-TDMA) and in 3D as an ILU-solver (space-TDMA). There also exists a 1D version (line-TDMA), but it is scarcely used. We explain the behaviour and the difficulties to parallelize the plane-TDMA solver in this paragraph. We give here the algorithm of the plane-TDMA:

x_0 start value
for $m = 1, ..., Max$ **do**
  for $k = 1, ..., NK$ **do**
    $A_k x_k^m = B_k x_{k-1}^m + T_k x_{k+1}^{m-1} + b_k$
  end for
end for

A_k operates on a plane of points (of the 3D grid), B_k operates on the previous plane-points and T_k operates on the next plane-points if we walk through the (physical) grid in one direction; this is called a sweep. In the above algorithm we walk in the z-direction, thus (a forward) sweep in z-direction. In plane-TDMA the blocks are solved with an ILU-method.

Most solvers have problems with vortices from the turbulent behaviour of the flow, this circulation is mostly damped by solvers, because they solve per point (mostly). This results in a slow convergence, but the TDMA solver shall convergence the same or even better by using blocks to keep information together and sweep in all directions to get a robust solver. (independent of the physical behaviour, e.g. vortices).

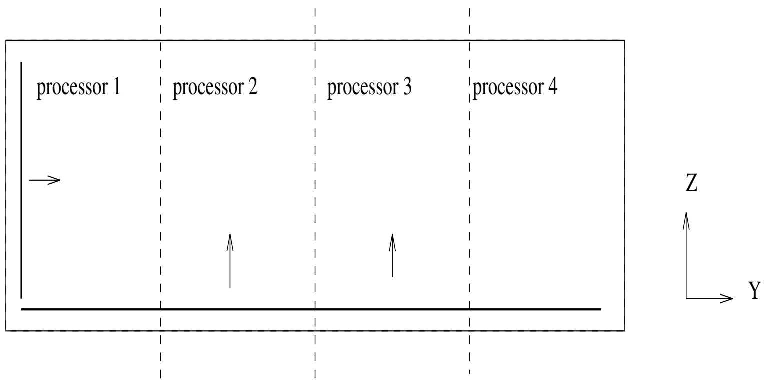

Figure 3.2: plane-TDMA solver with y- and z-sweep directions

Parallelization will lead to a partitioning of the data objects. Suppose that parallelization will lead in the following grid partitioning, as figure 3.2. We have to change the TDMA-solver such that it can work in parallel. For all sweep directions we get a different approach. In figure 3.2 we see that the y-sweep is not parallel, we have to change the structure of the TDMA-solver to obtain a parallel implementation, but this will result in slower convergence speed. For example we can do local y-sweeps, by using old information (if needed) of the neighbour partition, just to start in parallel. And if we have done the local job we communicate the updates to the neighbour partition, which costs some communication time.

In figure 3.2 we see for the z-sweep in parallel, that we have to communicate one neighbouring value of the partitions at each sweep-step. This communication will slow down the process of solving, maybe convergence speed will be the same, due communication we do not get any (not enough to do it in parallel) speedup of our parallel solver. We have showed here some of the difficulties by parallelization of the plane-TDMA. We have to experiment with HPF to come up with the right conclusions for parallelization of the TDMA solvers. Maybe that avoiding some of the sweep directions leads to acceptable results.

3.2.2 GMRES

GMRES is an iterative method for solving large linear systems of equations, like (2.19), in which $A \in \mathbb{R}^{n \times n}$ is non-symmetric. If A is symmetric then the Conjugate Gradient (CG) method is used. More about iterative methods see [7], [8].

GMRES starts with an initial approximate solution x_0 and initial residual $r_0 = b - Ax_0$. At the m-th iteration, a correction z_m is determined in the Krylov subspace. The Krylov subspace is defined by:

$$K_m(A; v_1) = span\{v_1, Av_1, ..., A^{m-1}v_1\}. \tag{3.2}$$

The Krylov subspace is built with Arnoldi's method, which constructs an orthonormal basis for $K_m(A; v_1)$.

ARNOLDI METHOD:

for $i = 1, 2, 3, ...$ **do**
  $v_{i+1} = Av_i$
  for $j = 1, 2, ..., i$ **do**
    $h_{j,i} = v_j^T v_{i+1}$
  end for
  for $j = 1, 2, ..., i$ **do**
    $v_{i+1} = v_{i+1} - h_{j,i} v_j$
  end for
  $h_{i+1,i} = \|v_{i+1}\|_2$
  $v_{i+1} = \frac{v_{i+1}}{h_{i+1,i}}$
end for

We see that Gram-Schmidt (GS) is used in Arnoldi's method for the orthonormalisation. Because of the presence of round-off, GS is not a reliable method for the orthonormalisation; better is to use the Modified GS (MGS). For this we have only to combine the two for-loops, to get:

for $j = 1, 2, ..., i$ **do**
  $h_{j,i} = v_j^T v_{i+1}$
  $v_{i+1} = v_{i+1} - h_{j,i} v_j$
end for

If $h_{i+1,i} = 0$ then we stop; we cannot construct a base any further. The vectors $v_1, ..., v_i$ will then span an invariant subspace for Krylov subspaces and the exact solution is computed.

We can summarize the Arnoldi process as follows: let V_i be the $n \times i$ matrix with vectors $v_1, v_2, ..., v_i$ as its columns. The Hessenberg matrix $H_i \in \mathbb{R}^{(i+1)\times i}$ contains the coefficients $h_{j,i}$ of the Arnoldi process:

$$H_i = \begin{pmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,i} \\ h_{2,1} & h_{2,2} & & \\ & h_{3,2} & \ddots & \vdots \\ & & \ddots & h_{i,i} \\ & & & h_{i+1,i} \end{pmatrix}. \tag{3.3}$$

The Arnoldi method is equivalent to:

$$AV_i = V_{i+1} H_i \text{ and } V_{i+1}^T V_{i+1} = I. \tag{3.4}$$

As we mentioned earlier, the correction, $x_m = x_0 + z_m$ is determined as follows. Take z_m as a linear combination of $v_1, ..., v_m$ thus $z_m = V_m y$ for $y \in \mathbb{R}^m$. One has to solve the least-squares problem for the y:

$$\min_{z\in K_m(v_1)} \|b - A(x_0 + z)\|_2 \;=\; \min_{y\in\mathbb{R}^m} \|r_o - AV_m y\|_2 \;=\; \min_{y\in\mathbb{R}^m} \|\|r_0\|e_1 - H_m y\|_2. \tag{3.5}$$

Because in every iteration a new basis vector v_i has to be computed and stored, the computational complexity and the memory requirements grow with the number of iterations. Therefore we restart the GMRES process after a fixed number, say m iterations. GMRES with restart is known as GMRES(m), we shall give a total overview of the algorithm,

GMRES(m) algorithm

x_0 given :$r_0 = b - Ax_0$;$v_1 = \frac{r_0}{\|r_0\|}$
begin
for $i = 1, 2, ..., m$ **do**
  $v_{i+1} = Av_i$
  for $j = 1, 2, ..., i$ **do**
    $h_{j,i} = (v_{i+1}, v_j)$
    $v_{i+1} = v_{i+1} - h_{j,i}v_j$
  end for
  $h_{i+1,j} = \|v_{i+1}\|_2$
  if $h_{i+1,j} = 0$ **then**
    set m=j goto solve least-square problem
  else
    $v_{i+1} = v_{i+1}/h_{i+1,j}$
  end if
end for
least-square problem
$\min_{y\in\mathbb{R}^m} \|\|r_0\|e + 1 - H_m y\|_2$ now compute x_m
if $statified$ **then**
  stop
else
  set $x_0 = x_m$ and start over again
end if

3.2.3 Principal parts of GMRES

In the GMRES algorithm we can discover the following building blocks:

- $y = y + a.x$ vector update,
- $\|y\|$ the norm, but basically (x, y) the inner product,
- $A * v$ matrix vector product.

3.2.4 Parallel aspects of GMRES

In example 1 is a matrix vector multiplication distributed on a number of processors. We use this distribution to give an idea of the parallelization of the principal parts of the GMRES method, although there are complete implemented operations in HPF, e.g. the for inner product. In [15], a similar approach is given about parallel aspects.

1. **vector update**,
   Because of the distribution we can do the vector update without communication. Consider figure 3.3, locally on the ith processor we do the part of the vector update with $x((i-1)\frac{n}{p}+1)...x(i\frac{n}{p})$ and $y((i-1)\frac{n}{p}+1)...y(i\frac{n}{p})$. This is done for all processors in parallel.
2. **inner product**,
   each part of the vectors is assigned to each processor. The inner product of these parts can be computed locally, see figure 3.4. The sub-answer c_i must be exchanged and summed up between all processors (communication) for the complete inner product $c = \sum c_i$.

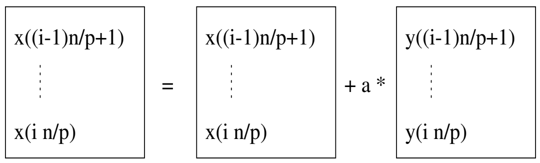

Figure 3.3: principal of vector update on a processor

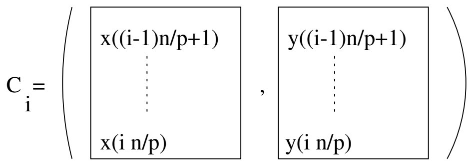

Figure 3.4: inner product part on a processor

variant of the inner product
The most inner products in the GMRES method are in the Arnoldi step. The Arnoldi step is implemented with MGS. Suppose we use GS, and if we have p processors. If we choose $m = p$ then the GMRES(p) build an p dimensional subspace with the Arnoldi step, which consist of p inner products. Instead of partitioning the vectors and assigning those part to the processor we assign the whole vectors such that each processor can compute an inner product locally. The advantage is no communication in the inner products, which speeds up the Arnoldi step. Disadvantages of GS are less stability with respect to rounding errors, and secondly we must use enough processors for building a large enough subspace because if the subspace is too small we must do a lot of GMRES-steps.

3. **matrix-vector**,
This is already discussed in example 1, we have seen that some communication is required.

The system of equations constructed with the SIMPLE method are sparse. The diagonals of sparse matrices are stored in vectors for an efficient use of the memory. This storage differs from the one used in example 1.

3.3 Parallel Preconditioners

For the same reason as for parallel iterative methods, we investigate the parallel behaviour of traditional (sequential) preconditioners. Preconditioners are necessary to speed up the iterative processes. This section is based on the ideas for preconditioners as given in [7], and [10]. The preconditioning transforms the system of equations such that it easier to solve for the iterative method. We discuss preconditioners of the type,

$$M^{-1}Ax \;=\; M^{-1}b, \tag{3.6}$$

in which M is the preconditioner matrix. Note that the solution x is not changed by the preconditioning.

There is an equivalence between an iterative process and a preconditioner. We can write an iterative process as a preconditioner. To solve a linear system (2.19) a classical iterative process is used,

$$x_{i+1} = x_i + M^{-1}(b - Ax_i), \tag{3.7}$$

where M is chosen such that it is cheap to calculate (3.7). Rewrite this as

$$x_{i+1} = (I - M^{-1}A)x_i + M^{-1}b = Gx_i + f. \tag{3.8}$$

Expression (3.8) can be viewed as a technique for solving the system

$$(I - G)x = f. \tag{3.9}$$

Substitute the definition of G, and f into (3.9):

$$M^{-1}Ax = M^{-1}b, \tag{3.10}$$

and we see our preconditioned linear system. This illustrates the relationship between a preconditioner and an iterative process.

We describe a number of parallel preconditioners. A disadvantage is the fact that these preconditioners are not optimal with respect to convergence. Our parallel machine consist of coupled workstations, with a relatively large communication time. So we choose preconditioners with no or a small amount of communication. First we will give the traditional preconditioners and after that we show some parallel aspects of the preconditioners used in the GMRES method.

3.3.1 Diagonal preconditioning (Jacobi)

By (point) Jacobi we take $M = diag(A)$ as preconditioner. This preconditioner will give a better condition-number for solving (2.19), for more details see [7]. The inverse of M is easy to obtain, e.g. $M^{-1} = 1/diag(A)$. Because this preconditioner may not be powerful enough to speed up the convergence we shall give a generalization of Jacobi, e.g. Block Jacobi, which often gives better results.

3.3.2 Block-Jacobi

We explain the principle of Block-Jacobi preconditioners using the iterative process. Assume matrix A is sparse (consist of non-overlapping blocks):

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & & \\ A_{2,1} & A_{2,2} & A_{2,3} & \\ \ddots & \ddots & \ddots & \\ & A_{p-1,p-2} & A_{p-1,p-1} & A_{p-1,p} \\ & & A_{p,p-1} & A_{p,p} \end{pmatrix} ;\ x = \begin{pmatrix} \tilde{\xi_1} \\ \tilde{\xi_2} \\ \vdots \\ \tilde{\xi_p} \end{pmatrix} ;\ b = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{pmatrix}. \tag{3.11}$$

Here $A_{i,j}$ are block matrices, and we split the vector x, and right-hand-side b in p corresponding partitions. Therefore we have p diagonal blocks. Below we explain the reason to use $\tilde{\xi_i}$ instead of ξ_i. We define the sets S_i, which contain the indices of the partitions of x and b:

$$S_i = (j \mid l_i \le j \le r_i)\ \forall j \text{ and } \bigcup_{i=1}^{p} S_i = S = \{1, ..., n\}. \tag{3.12}$$

The sets S_i are defined such that the elements of $\tilde{\xi}_i$ are given by $(x_{l_i}, ..., x_{r_i})$. A similar relation holds for b and A. We must select the right blocks of A for the Block-Jacobi algorithm. For this we use the prolongation operator V_i^T and the restriction operator V_i. V_i is defined by:

$$V_i = [e_{l_i}, e_{l_i+1}, ..., e_{r_i}]. \tag{3.13}$$

The vector e_j is the j-th column of the $n \times n$ identity matrix. The blocks $A_{i,j}$ are defined by:

$$A_{i,j} = V_i^T A V_j, \tag{3.14}$$

with corresponding $\tilde{\xi}_i$,

$$\tilde{\xi}_i = V_i^T x \text{ such that } x = \sum_{i=1}^{p} V_i \tilde{\xi}_i. \tag{3.15}$$

In (3.11) $\tilde{\xi}_i$ denotes the partition, and we define $\xi_i = (0, ..., 0, \tilde{\xi}_i, 0, ..., 0)^T$ as a vector in the $n \times 1$ space, the same dimension as x. The Block-Jacobi method for the i-th part becomes,

$$\xi_i^{k+1} = \xi_i^k + V_i A_{i,i}^{-1} V_i^T (b - Ax^k), \tag{3.16}$$

for non-overlapping blocks, and for overlapping blocks:

$$\xi_i^{k+1} = \xi_i^k + V_i D_i A_{i,i}^{-1} V_i^T (b - Ax^k), \tag{3.17}$$

here D_i is a diagonal matrix with the weight contributions. By k overlapping elements the weight of D_i is $1/k$ on the overlapping elements, and by non-overlapping elements the value is 1. We combine all Block-Jacobi iteration parts, and we get:

$$x^{k+1} = x^k + \sum_{i=1}^{p} \xi_i^{k+1}. \tag{3.18}$$

Substitution of (3.17) in (3.18), we get the iterative process:

$$x^{k+1} = x^k + \sum_{i=1}^{p} V_i D_i A_{i,i}^{-1} V_i^T (b - Ax^k). \tag{3.19}$$

From (3.19), (3.7), and (3.10) it appears that the Block-Jacobi preconditioner M, is given by:

$$M = \sum_{i}^{p} V_i D_i A_{i,i}^{-1} V_i^T. \tag{3.20}$$

In the next paragraph the parallel aspects of a preconditioner in the GMRES method are given.

3.3.3 Parallel aspects of a preconditioning

The substitution of preconditioner M in the GMRES algorithm is very simple. Only two statements in the algorithm have to be changed.

- in the initial residual by $r^0 = M^{-1}(b - Ax^0)$,
- in the Arnoldi step by $v_{i+1} = M^{-1}Av_i$.

Jacobi The Jacobi preconditioning is easy to parallelize, because no communication is needed. Take the distribution of example 1. Each row is multiplied by the local inverse diagonal. We see that the preconditioning can be done in parallel without communication.

Block Jacobi By non-overlapping blocks, and the number of blocks equals to the number of used processors results in a parallel preconditioner with no communication. On each processor the inverse diagonal block, assuming the distribution of example 1, is multiplied with the rows on the processor.

4 Measuring tools and HPF syntax for parallel implementation

4.1 Details about parallel computations

We discussed already some basics about parallel computation. In this chapter we will derive methods for measuring the properties of the parallel algorithm. For the degree of parallelism, we look at the properties Speedup, Efficiency, and Scalability.

Vector computers can also be seen as parallel computers. We can speak here about fine parallelism because it is done on instruction level, see [5] for more details. So instructions are executed in parallel. In this report we are looking at coarse parallelism, this is on data structure level, and this involves whole algorithms instead of instructions. In this chapter we will discuss running (coarse) parallel programs on different machines.

4.1.1 CRAY T3E vs. Workstations

There are many types of parallel computers. We will discuss here only the behaviour of workstations in a network and the supercomputer, e.g. CRAY T3E. The parallel machines can be categorized by the structure of the memory organization, and interconnection networks of the CPU's, see [9] for more details.
The most important difference between the Cray and a workstation cluster is the communication time. This influences the construction of an optimal algorithm. For workstations the communication time is very large, due to the connection network between the workstations. The Cray communicates on an inter-network, which is very fast and where only data is transported. The workstations are connected in a Ethernet network, where communication signals between the processors are not the only signals. The Ethernet lets other people work on their computers as well.

Counterbalancing the low communication performance of a cluster of workstations, the price of a CRAY T3E is very high, and research have made clear that most of the time the workstations are idle or work with a small task e.g. word-processing. By using this wasted CPU-time we can build an abstract parallel computer. Programs like PVM, HPF can construct and use this CPU-time. For more details see [12].

We concentrate on a reduction of the wall-clocktime for practical problems. In some cases, these are relative small problems with a small grid, but with a large computation time caused by many SIMPLE iterations. Anyhow, sometimes the problem size is very large, e.g. 3D problems.

The advantages of HPF above PVM is that HPF takes care of all send and receive instruction implicitly, whereas using PVM the send and receive statements should be programmed by the user.

We have to get a high degree of parallelism in our algorithm, if 90% of the program can be executed in parallel, and 10% sequential then we can expect a speedup of at most 10 times. We can see this by the law of Amdahl,

$$R_A = \frac{1}{\frac{f}{v} + \frac{1-f}{s}}, \tag{4.1}$$

where R_A is the speed in Mflops, f is the fraction of parallel work, v is the speed of the parallel execution in Mflops, and s is the speed of sequential execution in Mflops. In the next section some measuring tools are derived.

4.1.2 Complexity analysis of parallel algorithms

The main reason for parallel computing is to achieve a small wall-clocktime. We derive tools for measuring the performance, which gives an indication how good the algorithm is, by computation of the same problem on one machine.

Speedup

The speedup is defined as the ratio of the time it takes to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p processors. Let T_1 and T_p be the time in which the problem is solved by a single processor or by p processors respectively. The speedup becomes,

$$S(p) = \frac{T1}{Tp}. \tag{4.2}$$

We construct the parallel algorithm such that it works well on a parallel machine. If we use it on a single machine it will not work optimal, due to the extra communication overhead of the parallel algorithm. Instead of this we are using the best sequential algorithm on the single machine for measuring the speedup. This is called the true speedup.

Efficiency

Our expectation about efficiency is that we have a linear speedup with the number of used processors. Due to load imbalance, communication overhead, and synchronisation the efficiency is less. We define the efficiency by,

$$E(p) = \frac{S(p)}{p}. \tag{4.3}$$

Scalability

A parallel algorithm is scalable when the efficiency is constant when increasing the problem size and the number of processors. This is not the case in practice, in general the communication overhead and load imbalance increases as the number of processors increases, so the efficiency decreases.

Communication time

We shall give some model of the times T_1 and T_p. Assume that the algorithms are completely parallelizable. T_1 is the execution time when the algorithm works on 1 processor. When we run it on more processors the time will decrease proportionally. For T_p we get:

$$T_p = \frac{T_1}{p} + T_{\text{commun}}. \tag{4.4}$$

T_{commun} stands for the communication overhead. We have to communicate the results on specific times, and also the synchronisation is important where processors have to wait on each other. The communication time consist of the startup time and a message time. The message time is the time that is needed for sending one double precision multiplied by the number of double precisions to be sent. In GMRES we use inner products with global communication and matrix-vector multiplications with local communication (if sparse), herefore we get the following model,

$$T_{\text{commun}} = n_{\text{mv}} T_{\text{mv}} + n_{\text{dot}} T_{\text{dot}}, \tag{4.5}$$

in which n_{mv} is the number of matrix-vector multiplication, n_{dot} is the number of inner products. Both consist of a startup time which depends on the machine. As an example T_{dot} can be written as:

$$T_{\text{dot}} = T_{\text{up}} + k * T_{\text{mes}}, \tag{4.6}$$

where T_{up} is the startup time, T_{mes} is the message time for sending one double precession, and k is length of the message, e.g. the number of double precisions. If we know this values beforehand we can calculate the behaviour of a parallel algorithm. To estimate these values we can do small experiments with sending messages between the processors.

4.2 HPF structure

4.2.1 History

In 1991 the basis of High Performance Fortran (HPF) was defined by a working group containing members from industry, universities and government laboratories. The goal was to develop a standard language to construct parallel programs. Although a number of languages has been developed, which target parallel computers,(usually extensions of well-known conventional languages e.g. Fortran, C++), it is still a massive job to convert large and often long evolved code to run efficiently on a parallel machine. Many aspects are implemented nowadays and become a standard in Fortran 90 and 95. With HPF the user is able to express parallelism in a relative simple manner, hiding the details of the underlying architecture from the programmer to provide a code which is not machine specific. The paragraphs about HPF are based on [13], [14].

4.2.2 Data Mapping

The main item by a data parallel paradigm is to process and manipulate large arrays. The array operations work on parts of the distributed array, therefore the array operation can be done in parallel. The distribution is chosen in a way that maximizes the ratio of local work. With HPF compiler directives the mapping of the distributive data is controlled. Consider the following data mapping model to control the data, see figure 4.1.
In the first stage of programming in parallel all groups of data objects are aligned relative to each other, using the **ALIGN**-directive. This group of aligned objects is mapped onto a set of abstract processors, with the **DISTRIBUTE**-directive. The abstract processors are mapped onto the physical (real) processors. This is system dependent and the compiler takes control of this. Many arrangements of the abstract processors are possible, which give different communication overheads for the aligned objects between the processors. In the next paragraph we will give some of the main directives.

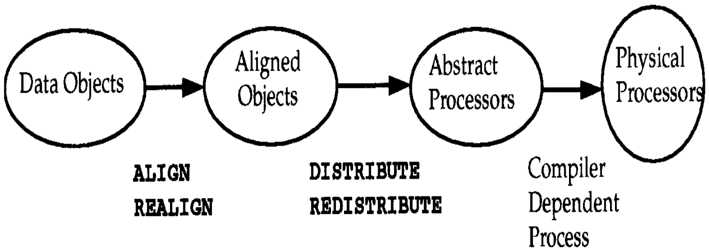

Figure 4.1: Data Mapping Model

4.2.3 Directives

At present compilers are not smart enough to distribute data arrays over a set of processors to obtain an acceptable performance. In HPF the data mapping is controlled with compiler directives. This extra information helps the compiler to produce efficient code. During the execution of the program the compiler generates the communication overhead, depending on the data mapping. We see the importance of partitioning in the right way and we try to keep pieces of closely interacting data elements together on each processor to reduce communication.

All directives begin with the *directive origin*:
!HPF$

This is the sign for the HPF compiler that there is a directive statement, but for other compilers e.g. Fortran 90 or 95 it is read as comment. There are two main classes of HPF directives e.g. specification-directives and executable-directives. The main specification-directives are:

PROCESSORS
DISTRIBUTE
ALIGN
TEMPLATE

And the main executable-directives are:

INDEPENT
REDISTRIBUTE

Specification-directives are involved in the compiling process and the executable-directives work on run-time. Below we discuss the syntax and the use of the specification-directives.

The PROCESSOR Directive

In HPF the arrangement of the abstract processors is specified by the **PROCESSORS** directive. The syntax is as follows:

!HPF$ PROCESSORS *name (decl.list.shape)*

- name, the name has to be unique and may not appear as variable during the program. By different names we can specify different processor arrangements in the program.
- decl.list.shape, this will tell the compiler how and how many processors are used. The shape of the arrangements is given in a way similar to describe the dimension of a matrix.

Examples:

!HPF$ PROCESSORS p(N)

the N processors are arranged as a line.

!HPF$ PROCESSORS procs(3,3)

the processors forms a 2 dimensional grid, with 9 processors.

!HPF$ PROCESSORS jan(number_ of_ processors())

on run-time we get the exact number of processors used and arrange them as a line.

!HPF$ PROCESSORS scalar

no specification about decl.list.shape will give a one processor program.

!HPF$ PROCESSORS cray(4,4,4)

this arrangement can be used on the CRAYT3E with 64 processors, this is possible because the interconnection network of the CRAY3TE is an 3D torus. On other machines is also possible to use this arrangement, but the motivation comes not from the machine architecture.

It is possible to give the actual number of processors at run-time and their shape, using the intrinsic functions: **Number-of-processors()** and **Processor-shape()**. The actual mapping of the abstract processors to physical is not visible in HPF.

DISTRIBUTE Directive

This directive takes care of partitioning the data objects, e.g. matrices and vectors. The syntax is:
!HPF$ DISTRIBUTE (dist.list) ONTO *target* :: *distributee*

- target, specify on which abstract processor arrangement the data object is partitioned.
- distributee, gives the data object.
- dist.list, specify how we want to partition the distributee on the abstract processors. Herefore we use special statements, e.g. BLOCK, CYCLE, and *. With these statements we can manage the partitioning. Note the dimension of the partitions used for the data objects is the same or less as the dimension of the processor arrangement.

To make the use of a dist.list clear we give first some examples:
!HPF$ PROCESSORS procs(4)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO procs :: A

Here we want to distribute a matrix A on a array of 4 processors. With the use of (BLOCK,*) this will partition the rows of A in equal parts such each processor gets a part, this is done by the statement BLOCK, the * means that this dimension is not considered, because we have only a one dimensional array of processors.

Another distribution is with the statement CYCLE, here we will force a cyclic distribution, for example with a vector b of $(n \times 1)$:
!HPF$ DISTRIBUTE (CYCLE,*) ONTO procs :: b

See figure 4.2 on the next page, for the CYCLE distribution.

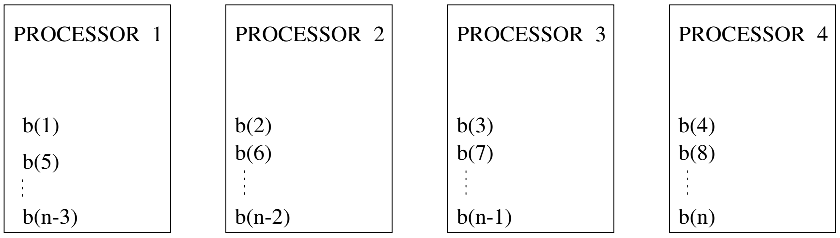

Figure 4.2: CYCLIC distribution with a vector b on 4 processors

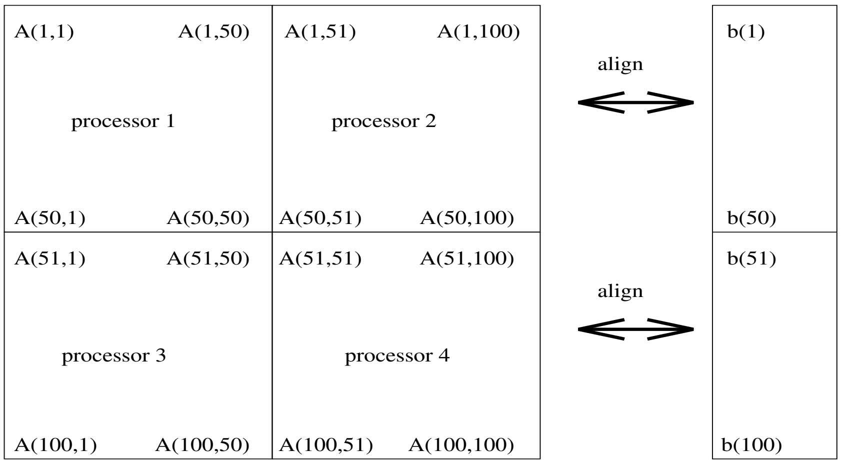

Figure 4.3: a (BLOCK,BLOCK) distribution of A with the alignment of b

For example, suppose we have four abstract processors and a two dimensional (100 $\times$ 100) array A(100,100). Then the array is distributed such that each processor will contain 2500 elements, see figure 4.3. The code fragment of this implementation is,

```
!HPF$ PROCESSORS procs(2,2)
      INTEGER, DIMENSION(100,100) :: A
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs :: A
```

ALIGN Directive

The ALIGN directive keeps the data objects together on each processor for a better performance. For example, for a matrix vector multiplication, the matrix is partitioned with the DISTRIBUTE directive and the vector is partitioned in the same way with the ALIGN directive. The syntax of the ALIGN directive:

!HPF$ ALIGN *list* WITH *target :: alignee*

- alignee, the alignee may not be the distributee in a DISTRIBUTE directive, but some object which should be aligned with another.

- target, is the data object which is already aligned or distributed.
- list, specifies how to align the object to the target.

For example, the matrix vector multiplication $A * b$

```
!HPF$ PROCESSORS procs(2,2)
   INTEGER, DIMENSION(100,100) :: A
   INTEGER, DIMENSION(100,1) :: b
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs :: A
!HPF$ ALIGN (i) WITH A(i,*) :: b
```

The vector b is split in two parts and each part is aligned to each of the two row-processors, see figure 4.3.

TEMPLATE Directive

A template is simply an abstract space of index positions. It can be considered as an array of ”nothing” (as compared with ”integers”). A template may be used as an abstract align-target that may be distributed.

!HPF$ TEMPLATE *templ-dec-list* :: *templee*

- templee, is the name of the template.
- templ-dec-list, specifies the dimensions in matrix notation of the template, or the distribution.

For example, declaring an index space of size 100,

```
!HPF$ TEMPLATE dimension(100) :: A
```

Another example is, when we wish to align four smaller arrays of size N by N with the four corners of a larger array of size $N + 1$ by $N + 1$, but without declaring the fullsized array,

```
!HPF$ TEMPLATE DIMENSION(N+1,N+1) :: world
   REAL DIMENSION(N,N) :: NW, NE, SW, SE
!HPF$ ALIGN NW(i,j) WITH WORLD(i,j)
!HPF$ ALIGN NE(i,j) WITH WORLD(i,j+1)
!HPF$ ALIGN SW(i,j) WITH WORLD(i+1,j)
!HPF$ ALIGN SE(i,j) WITH WORLD(i+1,j+1)
```

Note that the declaration of the template WORLD does not cause any memory allocation as it is an index space. The relationship is shown in figure 4.4 for the arrays NW and SE.

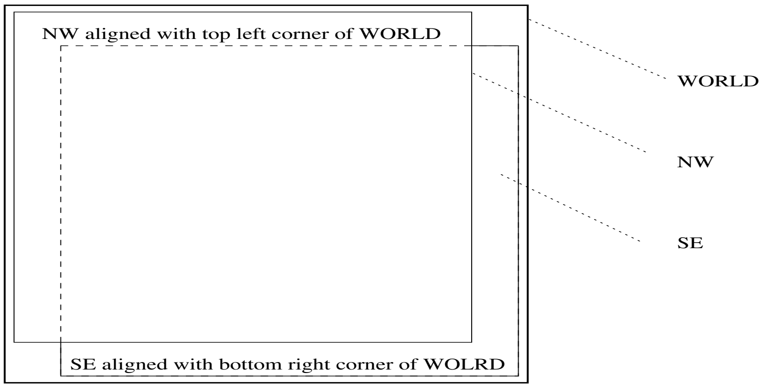

Figure 4.4: alignment with the template WORLD

4.2.4 Example HPF-program

We give an example to use all directives in one example program. A widely used operation in scientific computing is the matrix matrix multiplication of A and B to form a matrix C. One way to do this operation is to multiply the i-th row of A by the j-th column of B to form element $C(i, j)$. To do this on a parallel machine, it is essential that we keep many of the rows of A and columns of B on the same processor.
Suppose we have even number of processors, we arrange the processor in 2D grid (for example we take 4 processors).

```
  PROGRAM ABmultiply
  IMPLICIT NONE
  INTEGER PARAMETER :: N=100
  INTEGER DIMENSION(N,N) :: A, B, C
  INTEGER :: i,j

!HPF$ PROCESSORS procs(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs :: C
!HPF$ ALIGN (i,*) WITH C(i,j) :: A
!HPF$ ALIGN (*,j) WITH C(i,j) :: B

! whole array operation
  A = 1
  B = 2
  C = 0


  DO i =1,N
    DO j = 1,N
       All the work is local due to ALIGNs
```

```
      C(i,j) = DOT_ PRODUCT(A(i,:), B(:,j))
    END DO
  END DO

  WRITE(*,*) C
  END
```

This program aligns the rows of A and the columns of B as we require. Therefore we can do local the inner product (DOT_PRODUCT implemented by the manufacturer).

4.2.5 Parallel feature of HPF

We partitioned the matrices and vectors, and distributed them over the processors, but to manipulate them we often use loops. We explain here the use of a FORALL-loop, this is one of the extra features in HPF.
The FORALL-loop is not just a parallel generalization of the DO-loop as used in many other languages, e.g. Fortran 77. It can better be seen in the way of array assignments. The DO-loop is executed in a defined order. However the FORALL statement is not executed in a defined order, the order is implementation dependent. Hence with array assignment kept in mind, FORALL-loops where the result depends on the order of execution are not defined for parallel execution and thus undefined in HPF. For example:

```
FORALL (I=1:N) S=X(I) ! not-conforming
```

because it depends on the order of execution.

Interpretation of a FORALL loop

FORALL statements defines operations on a range of index values where an optional mask expression is true. The operation is executed in four stages, which we outline below through an example: take A as an array of (6×1).

```
A=0
A(1)=2 ; A(2)=3 ; A(3)=4
FORALL (I=1:6, A(I).NE.0.0) B(I)=1.0/A(I)
```

1. **Compute the valid set of index values:** This is defined in the first part of the FORALL statement for the running loop variables, which may be specified for example as vectors. And this can be done in any order of the index range 1 to 6.

2. **Define the active set of index values:** Because of the masking the active set are the indices where the masking value is .TRUE. , again this can be done in any order. Here are 1, 2, and 3 in the active set, because 4 to 6 are zero.

3. **Evaluate the right hand side:** We evaluate, only the active set members, the expression of the assignment, in any order. In the example we have $1.0/A(I),\ 1.0/A(2),\ 1.0/A(3).$

4. **Make assignment of the left hand sides:** For each member of the active set make the assignment of the precalculated values to the corresponding left hand sides of the expression. The assignments may be in any order.

This process (example) of execution is similar to that of array assignments: all right-hand sides are evaluated and then assigned to the left hand side. This explains why a FORALL-loop is not a generalized parallel loop. The main point is that each stage of the execution must be completed before the next stage can begin, hence implicit synchronization. In the stages the data elements can be executed in any order.

Visualising a FORALL-loop

To visualize the execution of the DO, FORALL statement a precedence graph is used. In here arrows are linked to ellipses indicate a possible order of computation at run time, and they do not show data dependencies. We show again with an example the possible order of execution for the DO-loop, FORALL-loop, and both with the use of INDEPENT-directive.
Suppose we have the following DO-loop:

```
DO i=1,3
  B(i)=1.0/A(i)
end DO
```

In the figure 4.5 we see the order of execution, the solid lines give the order of execution, and the dashed line just indicates a order of execution, but they cannot run in that direction to get the right answer.
Suppose we have the following FORALL-loop:

```
FORALL (I=1:3) B(I)=1.0/A(I)
```

Here is given the ”worst case” scenario, see figure 4.6, showing all possible orderings. In this case, these dependency arrows show that the execution of the FORALL synchronizes at each step. All the evaluations of the right hand side elements must be completed before assignments to left hand side take place.

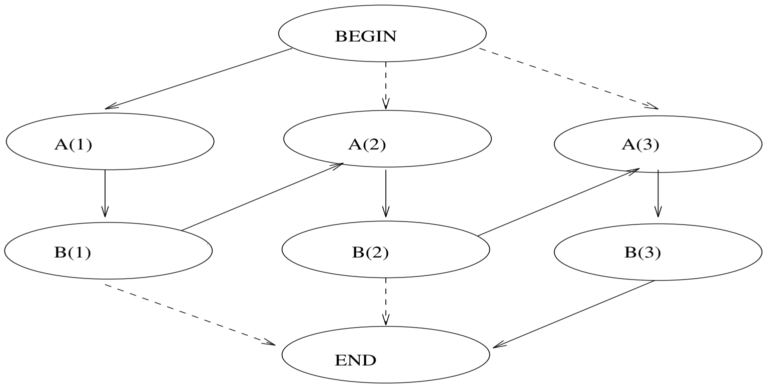

Figure 4.5: precedence graph of the DO-loop

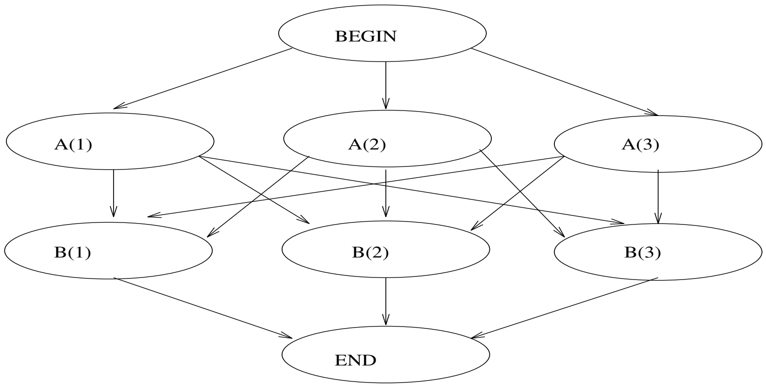

Figure 4.6: precedence graph of the FORALL-loop

Indepent-Directive

We give the compiler extra additional information with the INDEPENT-directive about the execution of a DO-loop, or FORALL-loop. The INDEPENT-directive is an executable directive and works at run time. This is intended to allow the compiler to generate a parallel loop in situations where parallelism is not obvious, see figure 4.7, for both loops. Although the FORALL statement could be parallelized without the use of the INDEPENT-directive. The FORALL-loop is executed in lock-step, for synchronization. This is not necessary with the use of the INDEPENT-directive, this can be seen in figure 4.8. This figure gives especially the synchronization between object, and we also express an unbalanced situation in the statements to get a realistic behaviour. The advantage is that the execution of the FORALL-loop with use of INDEPENT-directive is faster.

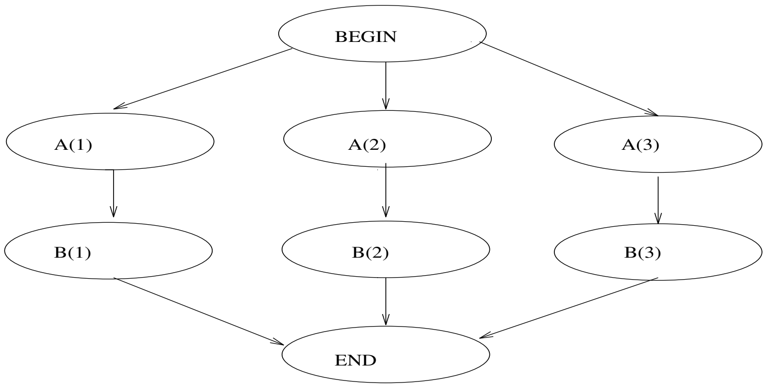

Figure 4.7: precedence graph for the DO-loop and FORALL with the INDEPENT directive

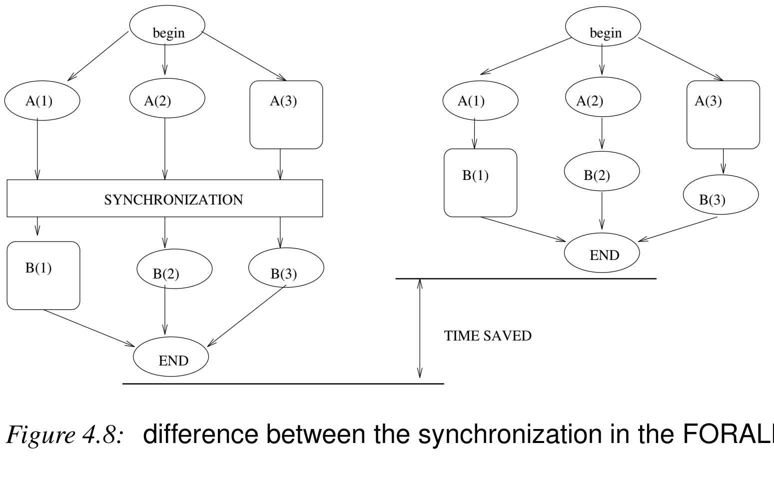

Figure 4.8: difference between the synchronization in the FORALL-loop

5 Suggestions and test problems

In this report some interesting questions and suggestions came up, which we convert into the following topics for further research:

Mapping and communication

We investigate the balance between mapping with HPF directive on various processor arrangements and communication time on own test software routines. Maybe it is possible to make a model for the workstation cluster and Cray machine.

Build in new WISH3D solvers

- **Krylov solvers**, we construct a Krylov solver. We shall implement this, and test it in the ”WISH3D-kernel” version. If the test shows that the solver has a good parallel behaviour we implement it in the ”WISH3D-almost” version with only velocity field and pressure component (cold flow, no burning model).
- **TDMA solvers**, We have done some research about the TDMA solvers. We will look if it is possible to parallize some sweep-directions to get still good convergence of the TDMA solvers.

Build in the preconditioners

We build in the (parallel) preconditioners and look if the resulting methods converge faster. If this is not the case we shall implement better preconditioners, e.g. an ILU-preconditioner, because in 3D the space TDMA is an ILU solver (and in 2D the plane TDMA is ILU solver), so we use the ’parallel’ TDMA-solver as preconditioner.

Test-cases

To test the solvers we will run test-problems, which are related to practical problems.

- 2D problem to solve the cold flow of a furnace, with solvers like, line-TDMA, plane-TDMA ($\equiv$ ILU-solver), and GMRES method.
- 3D problem to solve the cold flow of a furnace, with solvers like, plane-TDMA, space-TDMA ($\equiv$ ILU-solver), and GMRES method.

6 References

[1] H. Tazammourti, *(Parallelle) Solvers voor CFD-code WISH3D*, Afstudeerverslag, TU Delft, 1997. http://dutita0.twi.tudelft.nl/users/vuik/numanal/tazammourti.html

[2] G.P. Boerstoel, *Modelling of Gas-Fired Furnaces*, Thesis, TU Delft, 1997.

[3] S.V. Patankar, *Numerical Heat Transfer and fluid Flow*, Hemisphere, New York, 1980.

[4] J. van Kan, A. Segal, *Numerieke methode voor PDV* , D.U.M., Delft, 1993.

[5] H. van der Vorst *Parallel rekenen en super computers*, Academic Service, Schoonhoven, 1988 .

[6] G.P. Boerstoel, J.A.M. Denissen, A.J. Dalhuijsen, *HPCN-NICE Sector programma industriële stroming*, Eindrapportage, TNO-TPD / TU Delft, 1997.

[7] K.Vuik, H. van der Vorst, *Voortgezette Numerieke Lineaire Algebra*, Dictaat, TU Delft, 1996.

[8] G.H. Golub, C.F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989.

[9] H.X. Lin, *Parallel Rekenen*, Dictaat, TU Delft, 1996.

[10] Y. Saad, *Iterative methods for sparse linear systems*, PWS Publishing Company, Boston, 1996.

[11] A.J.C. van Gemund, Hai-Xiang Lin, *Prototype Parallelization of the WISH Application*, Delft, 1996.

[12] C.H. Cap, V. Strumpen, *Efficient parallel computing in distributed workstation environments*, Parallel Computing, 19:1235-1257, 1993.

[13] High Performance Fortran Forum, *High Performance Fortran Language Specification*, Houston, 1996. http://www.crpc.rice.edu/HPFF/hpf2/index.html

[14] A.K. Ewing, H. Richardson, A.D. Simpson, R. Kulkanni, *Writing Data Parallel Programs with High Performance Fortran*, studentnotes v1.3.1. http://www.epcc.ed.ac.uk/epcc-tec/hpf

[15] R.R.P. van Nooyen, K. Vuik, P. Wesseling, *Parallelism in GMRES applied to the computation of incompressible flows*, TUD Report 96-109, TU Delft, 1996. http://dutita0.twi.tudelft.nl/users/vuik/pub96.html