TNO-rapport
FSP-RPT-980053

Parallellisatie van de lineaire solvers gebruikt in WISH3D

TNO Technisch Physische Dienst
TU Delft

Stieltjesweg 1
Postbus 155
2600 AD Delft

Telefoon 015 - 2692000
Fax 015 - 2692111

Datum
27 augustus 1998

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

Doctoraalopdracht
TU Delft
Technische Wiskunde
Vakgroep numerieke wiskunde

Examencommissie
Prof.dr.ir P. Wesseling (TU Delft)
Dr.ir. C. Vuik (TU Delft)
Dr.ir. P. Boerstoel (TNO-TPD)
Ir. W.J.A. Denissen (TNO-TPD)

Inhoud

1 Inleiding

De TPD (Technisch Physische Dienst) te Delft is één van de 15 gevestigde divisies van TNO (Nederlandse Organisatie voor Toegepast Natuurwetenschappelijk Onderzoek) in Nederland. De TPD is zelf ook weer onderverdeeld in een aantal subdivisies, waarvan Proces Fysica (PF) er een van is. PF heeft expertise op het gebied van modellering van warmte en stromingen in alle soorten van industriële processen.

Een voorbeeld hiervan zijn glasovens waar veelvuldig door PF aan gerekend wordt. Om het gedrag van glasovens te simuleren is er een software CFD (computational fluid dynamics) pakket WISH3D ontwikkeld. Het model dat voor de simulatie gebruikt wordt in WISH3D bestaat onder andere uit de Navier-Stokes vergelijkingen, enthalpie vergelijkingen etc. Deze vergelijkingen vormen een gekoppeld stelsel en kunnen niet analytisch worden opgelost voor de complexe geometrie van glasovens. WISH3D berust dan ook op numerieke methoden om een oplossing te bepalen. Het numerieke model bestaat uit de eindige volume methode (EVM) en solvers voor het oplossen van de gediscretiseerde stelsels. Vanwege de gekoppelde vergelijkingen bestaan de solvers uit een buiteniteratieproces de SIMPLE (Semi-Impliciet Method for Pressure-Linked equations) methode en een binneniteratieproces met klassieke iteratieve methoden.

De rekentijd voor het doorrekenen van glasovens is in de orde van dagen, het doel van dit onderzoek is dit terug te brengen tot een rekentijd in de orde van uren met behulp van parallellisatie van de WISH3D solvers in het binneniteratie proces.
Dit onderzoek berust deels op het werk van H. Tazammourti [1], die onderzoek heeft gedaan op het gebied van het versnellen van de WISH3D solvers en deels op een vooronderzoek met High Performance Fortran (HPF) door G.P. Boerstoel, A.J. Dalhuizen en W.J.A. Denissen [2].
Uit de aanbevelingen in [1] konden twee wegen worden bewandeld, de eerste manier was door het versnellen van het buiteniteratie proces dit is gedaan door A. Saghir, [3], wat geslaagd is en de tweede manier is dit onderzoek door middel van parallellisatie van de WISH3D solvers voor het binnen iteratie proces. Gemotiveerd door uitkomst in [2], gebruiken we voor de parallellisatie van de solvers HPF.

Met parallellisatie wordt het rekenwerk van WISH3D in stukken gesplitst en verdeeld over een aantal processoren of werkstations, die simultaan een stuk van het rekenwerk voltooien. Er zijn twee paradigma's om parallel te programmeren, nl. message passing en data parallel. In dit onderzoek zal de aandacht gevestigd worden op de data parallelle programmeerstijl en hiervoor zal HPF gebruikt worden. Bij HPF hoeft men niet expliciet de communicatie tussen de processoren te programmeren wat bij message passing talen wel het geval is. In een netwerk zijn werkstations met kleine taken bezig of staan te wachten. Met deze niet gebruikte CPU-tijd kan een abstracte parallelle computer gebouwd worden. Het voordeel is dat een goedkope parallelle computer ontstaat met een acceptabele performance voor parallelle code. De HPF-compiler zorgt voor de distributie en communicatie van de code zodat het parallelle programma gerund kan worden op de cluster van gekoppelde werkstations.

Aan de hand van een testprobleem, worden de parallelle eigenschappen van de solvers bepaald. Uiteindelijk is het de bedoeling de solvers in de al bestaande parallelle versie van WISH3D in te bouwen en te testen om te kijken hoe de solvers zich gedragen bij stromingsproblemen.

1.1 Overzicht van dit verslag

We geven een puntsgewijze opbouw van de hoofdstukken,

- **Wiskundig model gebruikt in WISH3D:** Hierin worden een aantal vergelijkingen opgesteld welke in WISH3D gebruikt worden en vervolgens worden deze gediscretiseerd.
- **HPF syntax:** Hier staat een korte beschrijving van de structuur en opbouw van de programmeertaal HPF.
- **Tijd en Performance-metingen:** Hier wordt een tijdmodel afgeleid voor bepaalde wiskundige bewerkingen om schattingen voor de Krylov solvers te maken. Verder geven we definities om de kwaliteit van parallelle programma's uit te drukken.
- **Parallellisatie van de solvers in WISH3D:** Allereerst bekijken we de parallelle eigenschappen van de huidige solvers van WISH3D. Deze vergelijken we met de geparalllelliseerde iteratieve Krylov solver.
- **Parallelle Preconditionering:** De ontwikkelde parallelle iteratieve Krylov solvers worden voorzien van een preconditionering om het convergentiegedrag te vernellen.
- **Praktijk Resultaten:** We proberen de ontwikkelde solvers te testen in een realistischer stromingsprobleem dan het testprobleem.
- **Conclusies en aanbevelingen:** Hier worden de conclusies gegeven van dit onderzoek en de aanbevelingen voor verder onderzoek.

2 Wiskundig model van WISH3D

2.1 Inleiding

In dit hoofdstuk zal het wiskundig model worden besproken voor het fysisch gedrag in een verbrandingskamer van (glas)ovens. Het model kan grofweg in twee delen worden opgesplitst namelijk, het beschrijven van de turbulente stroming en het verbrandingsproces. We gaan in dit hoofdstuk meer in op de turbulente stroming, meer details over het verbrandingsproces zijn te vinden in [4].

De stroming is turbulent en hierdoor kunnen we geen analytische oplossing bepalen. Met behulp van het $(k-\epsilon)$ turbulentie model voor een laag Mach getal en bijbehorende decompositie kan wel een gemiddelde oplossing worden gevonden. Meestal wordt Reynold decompositie toegepast, op de te bepalen grootheid ϕ, met de veronderstelling dat deze uit een gemiddeld deel en een fluctuerend deel bestaat,

$$\phi(x,t) = \bar{\phi}(x) + \phi^{'}(x,t). \tag{2.1}$$

Waarbij het gemiddelde bepaald wordt met,

$$\bar{\phi} = \lim_{T\to\infty} \int_0^T \phi(x,s)ds. \tag{2.2}$$

Voor de fluctuatie moet gelden,

$$\bar{\phi}^{'}(x,t) = 0. \tag{2.3}$$

Het blijkt dat bij dichtheidsvariatie in turbulente stromingen het handiger is om de Favre decompositie te gebruiken. Er ontstaan dan geen dichtheidscorrelaties in de transport vergelijking van het $(k-\epsilon)$ model. Favre decompositie bestaat ook uit een gemiddeld en een fluctuerend deel voor te bepalen grootheid ϕ,

$$\phi(x,t) = \tilde{\phi}(x) + \phi^{''}(x,t), \tag{2.4}$$

waarbij het Favre gemiddelde voldoet aan,

$$\tilde{\phi} = \frac{\overline{\rho\phi}}{\bar{\rho}}, \tag{2.5}$$

en wordt gebruik gemaakt van het Reynoldsgemiddelde, verder geldt voor de Favre fluctuatieterm,

$$\bar{\phi}^{''} = \frac{-\overline{\rho^{'}\phi^{'}}}{\bar{\rho}}. \tag{2.6}$$

In het vervolg van het verslag zal gebruik gemaakt worden van de Favre decompositie en ook wordt gebruik gemaakt van de Einstein sommatie conventie. Zoals eerder op gemerkt geven we nu de vergelijkingen met name voor de turbulente stroming. Voor het complete model verwijzen we naar [4].

2.2 Behoudswetten

De behoudswet voor massa geeft de continuïteitsvergelijking:

$$\frac{\partial \bar{\rho}}{\partial t} + \frac{\partial \bar{\rho}\tilde{u}_j}{\partial x_j} = 0, \tag{2.7}$$

met $\bar{\rho}$ het Reynolds gemiddelde dichtheid en $\tilde{u}_{ij}$ is de Favre gemiddelde snelheidscomponent in j-richting.
De behoudswet van impuls geeft de bewegingsvergelijking:

$$\frac{\partial \bar{\rho}\tilde{u}_i}{\partial t} + \frac{\partial \bar{\rho}\tilde{u}_j\tilde{u}_i}{\partial x_j} = -\frac{\partial \bar{p}}{\partial x_i} - \frac{\partial \tilde{\tau}_{ij}}{\partial x_j} + \bar{\rho} g_i, \tag{2.8}$$

hierin is $\bar{p}$ de druk, g_i is de zwaartekrachtcomponent in i-richting en $\tilde{\tau}_{ij}$ is de spanningstensor gegeven door:

$$\tilde{\tau}_{ij} = (\mu + \mu_t)\left(\frac{\partial \tilde{u}_i}{\partial x_j} + \frac{\partial \tilde{u}_j}{\partial x_i}\right) + \frac{2}{3}(\mu + \mu_t)\frac{\partial \tilde{u}_k}{\partial x_k}\delta_{ij} + \frac{2}{3}\bar{\rho}\tilde{k}\delta_{ij}. \tag{2.9}$$

Hierin is μ de moleculaire dynamisch viscositeit, δ_{ij} is de Kronecker delta, μ_t is de turbulente viscositeit en k is de turbulente kinetische energie afkomstig uit het turbulentie model.
Uit de energievergelijking gegeven door de enthalpie vergelijking volgt:

$$\frac{\partial(\bar{\rho}\tilde{h} - \tilde{p})}{\partial t} + \frac{\partial(\bar{\rho}\tilde{u}_j\tilde{h})}{\partial x_j} = \frac{\partial\left(\left(\frac{\lambda}{C_p} + \frac{\mu_t}{\sigma_h}\right)\frac{\partial \tilde{h}}{\partial x_j}\right)}{\partial x_j} + \tilde{S}_{rad}. \tag{2.10}$$

Waarbij $\tilde{h}$ staat voor de totale enthalpie, λ is de thermische conductiviteit, C_p is de specifieke warmte van het gasmengsel bij contante druk, σ_h is het turbulente Prandtl getal en S_{rad} is stralingsbronterm.
Turbulente grootheden volgen uit het standaard $(k - \epsilon)$ model, De transportvergelijking voor de turbulente kinetische energie $\tilde{k}$ is gegeven door:

$$\frac{\partial \bar{\rho}\tilde{k}}{\partial t} + \frac{\partial \bar{\rho}\tilde{u}_j\tilde{k}}{\partial x_j} = \frac{\partial}{\partial x_j}\left((\mu + \frac{\mu_t}{\sigma_k})\frac{\partial \tilde{k}}{\partial x_j}\right) + P_k - \bar{\rho}\tilde{\epsilon}, \tag{2.11}$$

hierin is P_k de produktie en $\tilde{\epsilon}$ is de dissipatie verhouding van de turbulente kinetische energie. Voor de dissipatie $\tilde{\epsilon}$ volgt:

$$\frac{\partial \bar{\rho}\tilde{\epsilon}}{\partial t} + \frac{\partial \bar{\rho}\tilde{u}_j\tilde{\epsilon}}{\partial x_j} = \frac{\partial}{\partial x_j}\left((\mu + \frac{\mu_t}{\sigma_\epsilon})\frac{\partial \tilde{\epsilon}}{\partial x_j}\right) + \frac{\tilde{\epsilon}}{\tilde{k}}(C_{\epsilon 1}P_k - C_{\epsilon 2}\bar{\rho}\tilde{\epsilon}). \tag{2.12}$$

Waarin $C_{\epsilon 1}$, $C_{\epsilon 2}$ empirische constanten zijn. Voor de produktie P_k geldt:

$$P_k = \left(\mu_t\left(\frac{\partial \tilde{u}_i}{\partial x_j} + \frac{\partial \tilde{u}_j}{\partial x_i}\right) - \frac{2}{3}\mu_t\frac{\partial \tilde{u}_k}{\partial x_i}\delta_{ij} - \frac{2}{3}\bar{\rho}\tilde{k}\delta_{ij}\right)\frac{\partial \tilde{u}_i}{\partial x_j}. \tag{2.13}$$

De vergelijkingen voor het turbulentie model bevatten de turbulente viscositeit, deze is gegeven door:

$$\mu_t = \frac{C_\mu - \mu\bar{\rho}\tilde{k}^2}{\tilde{\epsilon}}. \tag{2.14}$$

Hierin is C_μ een empirisch bepaalde constante. Voor de empirische constantes van het $(k - \epsilon)$ model nemen we de volgende waarden:

$C_\mu = 0.09,\ \sigma_k = 1.0,\ \sigma_\epsilon,\ C_{\epsilon 1} = 1.44$ en $C_{\epsilon 2} = 1.92.$

2.3 Discretisatie

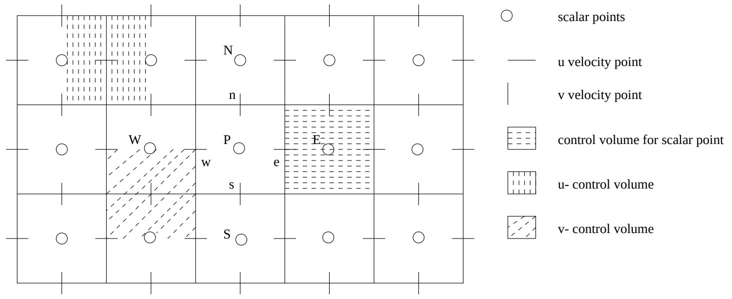
De convectie-diffusie vergelijkingen (2.7), (2.8) en (2.10) kunnen in de volgende algemene vorm geschreven worden,

$$\begin{aligned} &\frac{\partial \rho\phi}{\partial t} + \frac{\partial J_j}{\partial x_j} = \rho S_\phi, \\ &J_j = \rho u_j \phi - \Gamma_\phi \frac{\partial \phi}{\partial x_j} \quad , j = 1, 2, 3. \end{aligned} \tag{2.15}$$

Om de notatie te verkorten zijn de aanduidingen van Favre/Reynolds gemiddelde weggelaten. De variabele ϕ is de onbekende grootheid uit betrefende vergelijking. Termen die op het eerste gezicht niet meer in (2.15) voorkomen zijn verwerkt in de bronterm S_ϕ. De fluxterm J_j is de som van het convectieve deel $(\rho u_j \phi)$ en het diffusie deel $(\Gamma_\phi \frac{\partial \phi}{\partial x_j})$ in j-richting.

Het is in het algemeen niet mogelijk om (2.15) analytisch op te lossen. In WISH3D wordt met behulp van numerieke methoden een schatting gemaakt voor de oplossing. De discretisatie gebruikt in WISH3D berust op de beschreven methode in Patankar, voor meer details over discretisatie zie [5].

In WISH3D is (2.15) gediscretiseerd met de eindige volume methode (EVM). Hierbij is de verbrandingskamer van de oven verdeeld in een eindig aantal volume elementjes die elkaar niet overlappen. Ieder elementje bevat een gridpunt P waar de scalaire grootheden gedefinieerd zijn en op zijvlakken zijn de snelheden gedefinieerd. Hierdoor wordt een 'staggered' grid verkregen, zie figuur 2.1. Het voordeel hiervan is dat de stabiliteit van het drukveld gegarandeerd is.
Vervolgens integreren we (2.15) over het volume elementje, waarbij een schatting maken voor de integralen. Het is de bedoeling om de integralen zo te schatten dat we uitdrukkingen in buurgridpunten en zijvlakkenpunten van het betreffende volume elementje krijgen. Hieronder zullen we deze stappen verder uitwerken en we stellen de discretisatie voor een willekeurig (scalair) gridpunt P op. Voor het oplossen van het snelheidsveld op zijvlakken geldt een analoog verhaal.



Figuur 2.1: 2D staggered grid

Voordat we EVM kunnen toepassen hebben we tijdintegratie-schema nodig voor de niet stationaire term. We gebruiken een impliciet tijdschema, met het voordeel dat dit stabiel is, ook voor grotere tijdstappen. Voor (2.15) in het interval $[t, t + \Delta t]$ geldt:

$$\int_t^{t+\Delta t} \frac{\partial \rho\phi}{\partial \tau} d\tau + \int_t^{t+\Delta t} \frac{\partial J_j}{\partial x_j} d\tau = \int_t^{t+\Delta \tau} \rho S_\phi d\tau. \tag{2.16}$$

Beschouw $\frac{\partial J_j}{\partial x_j}$ en ρS_ϕ constant in het interval $[t, t + \Delta t]$, er volgt:

$$([\rho\phi]_P - [\rho\phi]_{P,0}) - \frac{\partial J_j}{\partial x_j} \Delta t = \rho S_\phi \Delta t. \tag{2.17}$$

Het subscript "0" duidt de vorige tijdstap t aan. Nu kunnen we EVM toepassen op (2.17),

$$\int_V ([\rho\phi]_P - [\rho\phi]_{P,0}) dV - \int_V \frac{\partial J_j}{\partial x_j} \Delta t dV = \int_V \rho S_\phi \Delta t dV. \tag{2.18}$$

We evalueren (2.18) termgewijs:
De bronterm S_ϕ wordt geschat door:

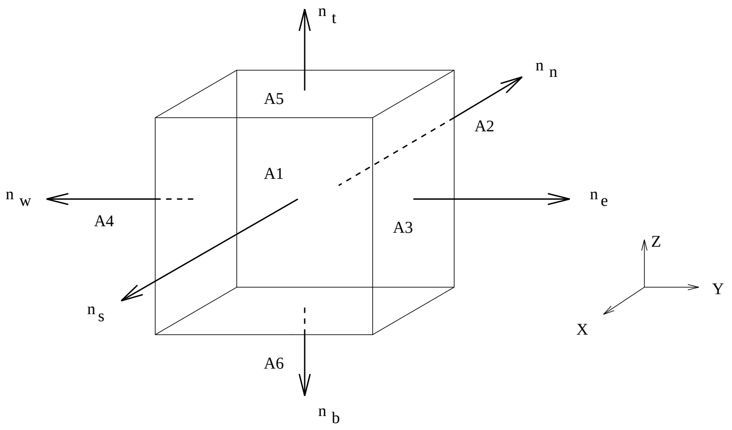
$$\int_V \rho S_\phi \Delta t dV \approx \rho S_\phi \Delta t \Delta V. \tag{2.19}$$

De eerste term van linker-lid wordt geschat met:

$$\int_V ([\rho\phi]_P - [\rho\phi]_{P,0}) dV \approx ([\rho\phi]_P - [\rho\phi]_{P,0}) \Delta V. \tag{2.20}$$

En voor de tweede term van het linker-lid gebruiken het Gauss theorema en krijgen:

$$\int_V \frac{\partial J_j}{\partial x_j} \Delta t dV = \int_A J_j n_j \Delta t dA = \sum_{i=1}^{6} \int_{A_i} J_j n_j \Delta t dA_i \approx \sum_{i=1}^{6} (J_j n_j)_i A_i \Delta t, \tag{2.21}$$



Figuur 2.2: 3D volume element

met $(n_1, n_2, n_3)^T$ de normaalvector op het zijvlak. Als eindig volume elementje nemen we een kubus, zie figuur 2.2, hierdoor splitst de oppervlakte integraal (2.21) op in zes stukken corresponderend met de zijoppervlakte A_i met $i = 1, ..., 6$ (of ook wel s,n,e,w,t,b) van de kubus. We gebruiken een Cartesisch assenstelsel en de kubus is hierlangs georiënteerd. Ieder oppervlak A_i heeft drie bijdragen in laatste term van (2.21) (Einstein sommatie) en hiervan vallen twee bijdragen af door gebruik van de oriëntatie van de kubus, immers de normaalvectoren vallen samen met de coordinaatas. Dus (2.21) wordt:

$$\sum_{i=1}^{6}(J_j n_j)_i A_i \Delta t = ((J_e A_e - J_w A_w) + (J_n A_n - J_s A_s) + (J_t A_t - J_b A_b))\,\Delta t. \tag{2.22}$$

Vul alle geëvalueerde termen in en we krijgen de dicretisatie voor het scalaire punt P,

$$\frac{([\rho\phi]_P - [\rho\phi]_{P,0})\,\Delta V_P}{\Delta t} + J_e A_e - J_w A_w + J_n A_n - J_s A_s + J_t A_t - J_b A_b = \rho S_\phi \Delta V_P. \tag{2.23}$$

Hierin zijn de hoofdletters E, W, N, S, T, B de buurgridpunten van P en de kleine letter e, w, n, s, t, b duiden de oppervlaktes aan van het volume elementje rond gridpunt P.
De fluxtermen moeten hierbij nog gediscretiseerd worden, dit laten we zien voor J_e, de rest gaat analoog. Bekend is dat bij een convectie diffusie vergelijking het convectieve deel kan gaan domineren, als men met centrale differenties de fluxterm J_e zou discretiseren kan dit numerieke oscillaties geven. Definieer het maas-Peclet getal,

$$Pe_e = \frac{\rho_e u_e \Delta x_e}{\Gamma_{\phi e}}. \tag{2.24}$$

Het maas-Peclet getal geeft de (lokaal) de verhouding tussen de convectie en de diffusie weer. Zolang $\|Pe_e\| < 2$ kan het centrale differentie schema gebruikt worden. Indien dit niet geldt moet men een ander schema gebruiken. Hiervoor wordt in WISH3D het Hybride schema toegepast, dit is een combinatie van centrale differentie en het upwind schema. Met het Hybride schema voor J_e geldt:

$$\begin{aligned} J_e =&(\rho u_1 A)_e \phi_E \quad (Pe)_e \quad \leq -2, \\ J_e =&(\rho u_1 A)_e \frac{\phi_P + \phi_E}{2} - (\Gamma_\phi A)_e \frac{\phi_E - \phi_P}{(\delta x_1)_e} \quad \|(Pe)_e\| < 2, \\ J_e =&(\rho u_1 A)_e \phi_P \quad (Pe)_e \quad \geq 2. \end{aligned} \tag{2.25}$$

Met het invoeren van $D_e = (\frac{\Gamma_\phi A}{\delta x_1})_e$, $F_e = \rho u_1 A$ en $a_E = D_e max\{0, -(Pe)_e, 1 - \frac{1}{2}(Pe)_e\}$ krijgen we voor J_e:

$$J_e = -a_E \phi_E + (a_E + (F)_e)\phi_P. \tag{2.26}$$

Nu kan (2.23) worden herschreven tot de vorm:

$$a_P \phi_P = \sum_{nb} a_{nb} \phi_{nb} + b_P, \tag{2.27}$$

waarbij de somatie loopt over de buurgridpunten E, W, N, S, T, B. De coeficienten a_P en b_P zijn gegeven door:

$$a_P = \sum_{nb} a_{nb} + (\rho_P \Delta t - S_P)\Delta V_P, \tag{2.28}$$

$$b_P = (\rho_{P,0} \phi_{P,0} \Delta t + S_C)\Delta V_P. \tag{2.29}$$

Hierin is de bronterm gelineairiseerd met $\rho S_\phi = S_C + S_P \phi_P$.

De discretisatie is bepaald voor een willekeurig (scalair) gridpunt P.
In WISH3D worden de gridpunten (globaal) genummerd met 3 indices i, j en k. Stel dat gridpunt ϕ_P met lokale referentie P overeenkomt met $\phi_{i,j,k}$ dan wordt (2.27):

$$(a_p)_{i,j,k}\phi_{i,j,k} =(a_E)_{i,j,k}\phi_{i+1,j,k} + (a_W)_{i-1,j,k}\phi_{i-1,j,k} + ... + (a_B)_{i,j,k}\phi_{i,j,k-1} + b_{i,j,k}, \tag{2.30}$$

$$\text{met } i = 1, \ldots, NI \quad j = 1, \ldots, NJ \quad k = 1 \ldots, NK.$$

Voor alle gridpunten geldt (2.30), deze vergelijkingen worden samengevat met matrix-vector notatie:

$$Ax = b. \tag{2.31}$$

Met de matrix A bestaande uit elementjes $a_E, .., a_B$ voor gridpunt i, j, k, x is de onbekende vector bestaande uit de gridpunten $\phi_{i,j,k}$ en b is vector met bekende elementjes $b_{i,j,k}$.

2.4 De SIMPLE Methode

In vorige paragraaf is het gekoppelde stelsel niet-lineaire differentiaalvergelijkingen afgeleid. De oplossing van het stelsel geeft problemen als het drukveld niet van te voren bekend is. Hiervoor wordt de SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) gebruikt, die beschreven is in Patankar [5].

In de SIMPLE methode wordt de continiteitsvergelijking (2.7) herschreven in een drukcorrectie vergelijking. We beginnen met een beginschatting voor de druk in bewegingsvergelijkingen (2.8) om de snelheidscomponenten van het snelheidsveld op te lossen. De snelheidscomponenten zijn benaderingen en zullen niet aan de continuiteisvergelijking voldoen. Met de drukcorrectievergelijking wordt geprobeerd om het snelheidsveld te verbeteren, ook de druk wordt hiermee verbeterd. We laten nu de andere grootheden het snelheids en drukveld beïnvloeden. En herhalen bovenstaand proces met het verbeterde drukveld, net zolang totdat het snelheidsveld voldoet aan de continuiteitsvergelijking. De druk zal dan ook naar het juiste drukveld geconvergeerd zijn.

De SIMPLE methode is een iteratief proces, waarmee het gekoppelde stelsel wordt opgesplitst in een aantal lineaire stelsels, welke in de (SIMPLE) iteratie stap worden opgelost. Deze lineaire stelsels worden in WISH3D ook weer met iteratieve methoden opgelost.
Het SIMPLE proces is tijdrovend, en in dit verslag wordt gekeken om het oplossen van de lineaire stelsel te versnellen door de iteratieve methoden te parallalliseren om zo de wall-clocktijd te verlagen.

3 High Performance Fortran

3.1 Inleiding

In dit verslag is gekozen voor een data parallelle programmeerstijl met behulp van HPF. Voordat we verder gaan wordt het verschil tussen een data parallelle programmeerstijl en message passing programmeerstijl nog eens aangehaald. Een data parallelle aanpak heeft betrekking op de parallellisatie van collectieve operaties op arrays of gedeelten hiervan, die verdeeld zijn over een aantal processoren. De operaties zijn vaak al optimaal geïmplementeerd door de fabrikant, zodat de programmeur niet steeds het wiel opnieuw hoeft uit te vinden. Als een te parallelliseren probleem uit te drukken valt in dit soort operaties op arrays dan kan een efficient programma worden verwacht. Bij data parallelle aanpak met HPF worden de details beperkt voor wat betreft communicatie, geheugenbeheer en distributie van data objecten. Zodat programmeur zich volledig bezig kan houden met de parallellisatie van het probleem.

Bij message passing programmeerstijl moet wel expliciet de communicatie en distributie verzorgt worden, dit geeft meer vrijheid. Maar de programmeur is wel verantwoordelijk voor minimalisatie van communicatie tussen processoren en juiste distributie zodat iedere processor evenveel werk verzet, load-balancing. Dus naast parallellisatie van het probleem moet hij tegelijkertijd zorgen dat alles goed verloopt voor wat betreft de communicatie en distributie.

Er wordt nu verder gegaan met data parallelle stijl en in het bijzonder met HPF.

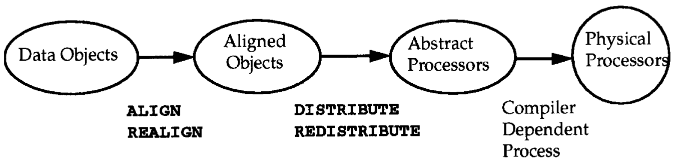
3.2 HPF struktuur

3.2.1 Geschiedenis

In 1991 is de basis voor High Performance Fortran (HPF) gelegd door verschillende mensen uit de industrie, universiteiten en overheidslaboratoria. Het doel was om een standaardtaal te ontwikkelen voor het schrijven van parallelle programma's. Ondanks dat er vele andere talen bestaan, maar deze zijn vaak te machine afhankelijk en hebben bij een grote omvang het probleem om het geheel efficient te houden (denk hierbij aan opmerkingen over de message passing stijl). Veel ideeën uit HPF zijn tegenwoordig geïmplementeerd in Fortran 90 & 95. Met HPF kan de gebruiker op een gemakkelijke manier het parallellisme tot uitdrukking brengen. De onderliggende details van de architectuur blijft hierbij buiten schot voor de gebruiker en op deze manier wordt een machine onafhankelijke code gegenereerd. De paragrafen over HPF zijn gebaseerd op [6], [7].

3.2.2 Data mapping

Bij een data parallelle programmeerstijl zijn bewerkingen op arrays belangrijk, waarbij de arrays verdeeld zijn. Hierdoor kunnen de operaties parallel verlopen. De distributie moet zo worden gekozen dat het lokale werk gemaximaliseerd wordt. Met de HPF compiler wordt het verdelen van de data objecten, de arrays, bestuurd. In figuur 3.1 wordt het data mapping model grafisch weergegeven. Bij de eerste fase van het parallel programmeren worden alle data objecten gegroepeerd en gekoppeld aan elkaar, ook wel gealigned, met behulp van het Align-directieve.
Deze groep van data objecten die gealigned zijn worden op een abstracte verzameling van processoren afgebeeld, met het DISTRIBUTE-directieve. De set van abstracte processoren wordt uiteindelijk afgebeeld op de physische processoren, niet zichtbaar voor de programmeur. Het beheer van de dataobjecten en communicatie daar tussen , wat systeem afhankelijk is, wordt door de HPF compiler afgehandeld. In de volgende paragrafen zullen de belangrijkste directieven besproken worden.



Figuur 3.1: Data Mapping Model

3.2.3 Compiler Directieven

Op dit moment zijn de compilers nog niet slim genoeg om zelf voor de distributie van data objecten te zorgen zodat een acceptabele performance gehaald wordt. In HPF besturen we de data mapping met behulp van directieven. Deze extra informatie helpt de compiler om efficiente code te genereren. Tijdens de eerste fase van de executie van het programma wordt de data mapping en communicatie routines klaar gezet door de compiler. Het is belangrijk om de data objecten op de juiste manier te partitioneren en deze zo te distribueren zodat de juiste gepartitioneerde objecten zo dicht mogelijk bij elkaar op elke processor bevinden om zo de communicatie tussen de processoren te minimaliseren.

Alle directieven beginnen met het zogeheten *directieve origin*:
!HPF$

Dit is het teken voor de HPF-compiler dat er een directieve statement volgt. Maar voor Fortran 90 of Fortran 95 compilers wordt het gelezen als commentaar. Zo blijft het mogelijk, met een zeer klein aantal aanpassingen aan de code met deze compilers te compileren.
De directieven zijn in volgende twee klassen in te delen nl., *specificatie directieve* en *executie directieve.* De belangrijkste specificatie directieven zijn:

PROCESSORS
DISTRIBUTE
ALIGN
TEMPLATE

deze worden tijdens het compileren uitgevoerd. De belangrijkste executie-directieve zijn:
INDEPENDENT
REDISTRIBUTE

en werken tijdens de executie fase. Hieronder bespreken we het gebruik en de syntax van de specificatie directieven.

PROCESSOR Directieve

In HPF wordt de verdeling van de abstracte processor gespecificeerd door PROCESSORS directieve. De syntax is als volgt:
!HPF$ PROCESSORS *name (decl.list.shape)*

- name, we kunnen meerdere processorsvormen naast elkaar gebruiken daarvoor heeft ieder zijn unieke naam nodig.
- decl.list.shape, dit vertelt de compiler hoeveel processoren er gebruikt worden en tevens wordt aangeduid hoe deze met elkaar verbonden zijn. Het aantal processoren en wijze van koppeling is hetzelfde als het definieren van de afmetingen van arrays.

Voorbeelden:
!HPF$ PROCESSORS p(N)

er zijn N processoren, die gerangschikt zijn als een rechte lijn.

!HPF$ PROCESSORS procs(3,3)

de processors vormen een 2 dimensionale grid, met 9 processoren.

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

op executie tijd krijgen we het aantal gebruikte processoren en deze worden gerangschikt als een rechte lijn.

!HPF$ PROCESSORS scalar

er is geen specificatie opgegeven dit geeft een sequentieel programma met 1 processor.

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

deze rangschikking kan gebruikt worden op de Cray T3E met 64 processoren, de abstracte processoren zijn dan in een 3 dimensionale kubus gerangschikt. De physische processors op de Cray zijn ook op een 3 dimensionale torus structuur met elkaar verbonden. We kunnen dit soort verdelingen ook wel op andere machines toepassen maar de motivatie komt dan niet van de machine structuur.

Het is ook mogelijk naast het aantal processoren ook de vorm tijdens executie op te geven met behulp van de intrinsieke functies: **Number-of-processors()** en **Processor-shape()**. De preciese afbeelding van de abstracte processor-vorm naar physische processoren is niet zichtbaar in HPF.

DISTRIBUTE Directieve

Het directieve specificeert de verdeling van de data objecten zoals bijvoorbeeld: matrices, vectoren. De syntax is als volgt:

!HPF$ DISTRIBUTE (dist.list) ONTO *target* :: *distributee*

- target, geeft aan naar welke abstracte processor de verdeling van het data object afgebeeld moet worden.
- distributee, geeft data objecten aan.
- dist.list, specificeert hoe we het data object in partities opdelen om deze te verdelen over de abstracte processoren. Hiervoor gebruiken we de statements: BLOCK, CYCLIC of *. Hiermee zijn we in staat om het opdelen in partities te besturen. Opmerking: bij partitioneren is de dimensie van de partitie in overeenstemming of kleiner dan de dimensie van de abstracte processor verdeling.

Om het gebruik van deze directieve te laten zien, hebben we de volgende voorbeelden:

!HPF$ PROCESSORS procs(4)

!HPF$ DISTRIBUTE (BLOCK,*) ONTO procs :: A

We hebben de matrix A en 4 processoren met elkaar verbonden in een rij. Iedere processor heeft twee buren behalve de randprocessoren die hebben er maar 1. Met (BLOCK,*) delen we de matrix in de eerste dimensie op in een gelijk aantal rijen, zodat op iedere processor ongeveer evenveel rijen krijgt toegewezen. Er wordt geprobeerd om de rijen zodanig te verdelen dat er zo min mogelijk communicatie nodig zal zijn. Dus BLOCK gaf de opdeling in de eerste dimensie en * slaat op de tweede dimensie van de matrix. Een * geeft aan dat er geen verdeling in deze richting is. De verdeling van de matrix is in overeenstemming met de dimensie van de processoren.

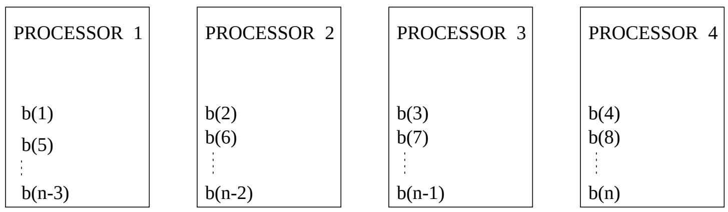
Een ander voorbeeld, laat het gebruik van CYCLIC zien, in dit voorbeeld delen we de vector b (nx1) op in een cyclische verdeling met,
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: b

Zie ook figuur 3.2 voor het uiteindelijke resultaat van de CYCLIC distributie, het eerste element van b staat op de eerste processor en tweede element van b staat op de tweede processor, dit loopt door tot de laaste processor en dan beginnen we weer van vooraf aan.

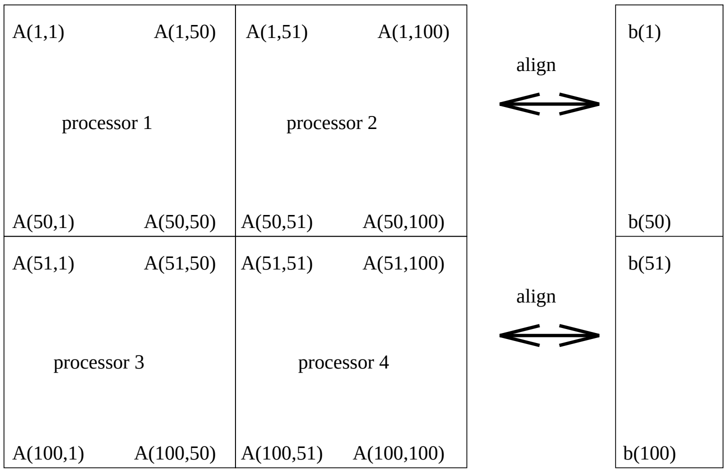
Als laatste voorbeeld een (BLOCK,BLOCK) distributie, zie HPF implementatie,

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

Stel we hebben 4 abstracte processoren met een 2 dimensionale vorm en een matrix A (100×100). De matrix A wordt op een (BLOCK,BLOCK) wijze verdeeld over de 4 procesoren, dat wil zeggen de eerste dimensie van de matrix A bevat 100 rijen deze wordt in tweeën gesplitst in overeenstemming met dimensie van de processors, hetzelfde wordt gedaan voor de kolommen van A. We krijgen dan de volgende situatie, zie figuur 3.3, iedere processor bevat 2500 elementen.



Figuur 3.2: CYCLIC distributie van de vector b, met 4 processoren



Figuur 3.3: Een (BLOCK,BLOCK) distributie van A met b gealigned aan A

ALIGN Directieve

Het ALIGN directieve zorgt ervoor dat data objecten bij elkaar op 1 processor komen. Tijdens compileren is al bekend welke objecten bij elkaar horen en dit hoeft tijdens de executie niet meer worden uitgezocht, hierdoor wordt performance verhoogt. De syntax is als volgt:

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

- alignee, het data object wat gealigned moet worden, deze mag dus niet voorkomen in DISTRIBUTE directieve.
- target, is het data object wat al gealigned of gedistribueerd is.
- list, specificeert hoe we het object verbinden aan het al verdeelde object.

Ter illustratie het volgende voorbeeld, de matrixvector vermenigvuldiging $A*b$

```
!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
```

de vector b is hierbij in twee delen opgesplitst en ieder stuk is toegekend aan 2 rij-processoren, zie figuur 3.3.

TEMPLATE Directieve

Uit ALIGN directieve proberen we data te koppelen, meestal kiezen we een data structuur en "align"-en hiermee alles. Dit kost extra geheugenruimte. Omdit te vermijden wordt een TEMPLATE gebruikt. Een TEMPLATE geeft alleen de vorm aan van een dataobject zonder dat deze ruimte gereseveerd wordt. Vervolgens verdelen we het TEMPLATE data object. Nu kunnen we de TEMPLATE gebruiken als ALIGN target. Dus we hebben een vast data object wat alleen de afmetingen weergeeft met bijbehorende verdeling, waar alle andere objecten aan gealigned zijn. De syntax ziet er als volgt uit:

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

- templee, de naam van TEMPLATE data object.
- templ-dec-list, specificeert de dimensies van dataobject, of de distributie.

Als voorbeeld, defineer indexspace ter grote van 100,

!HPF$ TEMPLATE dimension(100) :: A

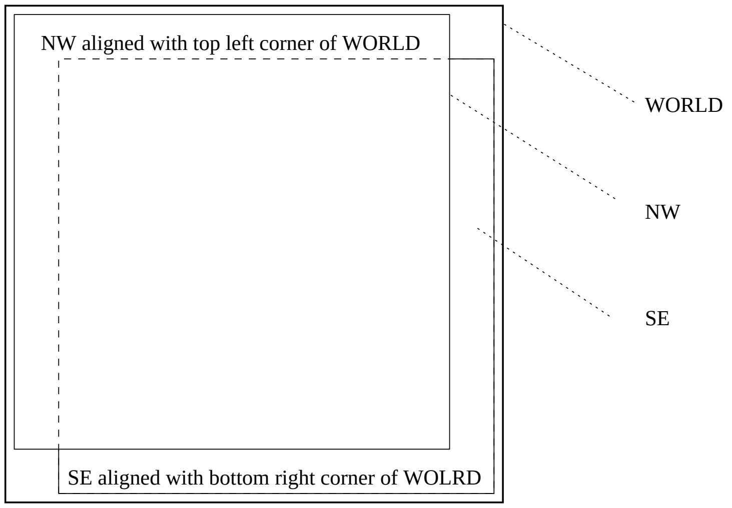
Het volgende voorbeeld is het align-en van 4 kleinere arrays met afmeting N bij N aan de hoekpunten van een groter aray met afmeting $N+1$ bij $N+1$, maar zonder geheugenruimte voor de array in beslag te hoeven nemen,

```
!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)
```

Merk op dat declaratie van de template WORLD geen geheugenruitme kost voor het array. De relatie tussen array is getoond in figuur 3.4 voor de arrays NW en SE.

3.2.4 Voorbeeld HPF-programma

We geven een voorbeeld met gebruik van tot nu toe gegeven directieven in een HPF progamma. Een veel gebruikte operatie in wetenschappelijke berekeningen is de matrix-matrix vermenigvuldiging $C = A * B$. Een manier om dit aan te pakken is om de i-de rij van A te vermenigvuldigen met j-de kolom om zo het element $C(i,j)$ te bepalen. Bij parallellisatie is het dus van belang zoveel als mogelijk de rijen (of gedeelte) van A en kolommen (of gedeelte) van B bij elkaar te zetten op zelfde processor.
Stel dat we een even aantal processoren hebben, die we op een 2 dimensionale manier rangschikken, in voorbeeld implementatie hieronder nemen we 4



Figuur 3.4: Template WORLD waar aan gealigned wordt

processoren.

```
    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
```

In dit programma zijn de rijen van A gealigned met de kolommen van B zoals voorgesteld. Hierdoor is zelf geen communicatie nodig, de vermenigvuldiging van de rij en kolom, oftewel inprodukt wordt lokaal berekend. Hiervoor is gebruik gemaakt van de parallelle functie DOT_PRODUCT geïmplementeerd door de fabrikant.

3.2.5 Loops in HPF

We hebben tot nu toe de verdeling van vectoren en matrices laten zien, maar om de data objecten te manipuleren gebruiken we vaak loops. In deze paragraaf bespreken we het gebruik van de FORALL-loop. De FORALL-loop zijn geen generalliseerde parallelle DO-loop, met de DO-loop bijvoorbeeld uit Fortran 77. Het kan beter gezien worden als een array toekenning. Een groot verschil tussen een DO en FORALL-loop is de executie volgorde, de DO-loop loopt in vaste richting, terwijl deze van de FORALL-loop niet gespecificeerd is en dus resultaten die afhangen van de doorlopen volgorde, kunnen niet gespecificeerd worden met FORALL-loops in HPF, voorbeeld:

FORALL (I=1:N) S=X(I) ! voldoet niet aan HPF

omdat de toekenning aan S afhankelijk is van de volgorde van I.

Interpretatie van een FORALL-loop

De syntax voor de FORALL ziet er als volgt uit:

FORALL (header) (FORALL assignment)

- header, bevat index waarden van data elementen waarop bewerkingen op worden uitgevoerd.
- FORALL assignment, bestaat uit een expressie met assignment.

Het FORALL statement wordt in 4 stappen uitgevoerd, wat we hieronder zullen toelichten aan de hand van een voorbeeld: neem A een array met afmeting (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. **bereken de geldige verzameling van loop index waarden :** In FORALL statement is als eerste een expressie voor de loop index va-

riabele gegeven. Hieruit worden de indices bepaald, in willekeurige volgorde.

2. **Definieer de actieve verzameling van indices:** Door eventuele maskering van indices kunnen we nog bepaalde voorwaarden opleggen, alleen die indices waarvoor dat waar is komen in de uiteindelijke actieve verzameling. Dit kan dus weer in willekeurige volgorde gebeuren. In het voorbeeld behoren alleen de indices 1,2 en 3 tot de actieve verzameling, hiervoor is het masker A(I).NE.0.0 waar.

3. **Evalueer rechter lid van expressie:** Na het definieren van de indices komt de uiteindelijke expressie. In ons voorbeeld is dit $B(I) = 1.0/A(I)$. We bereken in willekeurige volgorde voor de actieve verzameling indices het rechterlid hiervan, dus $1.0/A(1),\ 1.0/A(2),\ 1.0/A(3)$.

4. **Toekenning aan linker lid:** Nu volgt de toekenning aan het linker lid (in willekeurig volgorde voor de actieve verzameling indices).

In de executie verloop van het voorbeeld zien we grote overeenkomsten met dat van matrix toekenningen (assignments) : steeds moet een stap afgerond zijn voordat met de volgende begonnen kan worden, impliciete synchronisatie. In iedere stap is geen vaste doorlooprichting, maar geheel willekeurig. Dit verklaart waarom we voor de FORALL-loop niet kunnen spreken van een gegeneraliseerde DO-loop. De FORALL heeft meer weg van een toekenningsstatement (assignment statement).

Visualisatie van een FORALL-loop

We zullen het verschil tussen de doorlooprichting van de FORALL en DO-loop grafisch weergeven. Om de executie te visualiseren gebruiken we een uitvoergraaf. Deze graaf laat alle berekeningen van de FORALL zien waar deze moet eindigen voordat een andere kan starten. De berekeningen van de expressie staan in de ellipsen, en in de graaf zijn vier stappen van de FORALL zichtbaar: actieve indexverzameling, evaluatie, toekenning en beeindiging. Aan de hand van een voorbeeld laten we zien hoe doorlooprichting van de FORALL en DO-loop is en dit wordt nog uitgebreid met het INDEPENDENT-directieve.

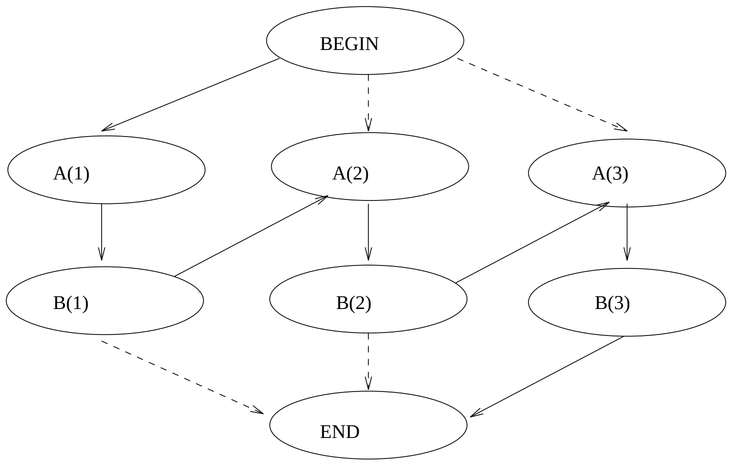
Stel dat we de volgende DO-loop hebben:

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

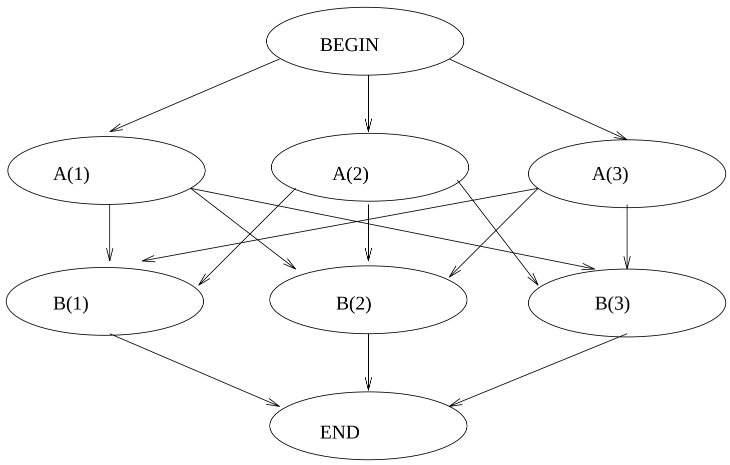
In figuur 3.5 zien we de doorlooprichting van de DO-loop. De gehele lijnen geven de doorlooprichting aan. De gestippelde lijnen geven een mogelijke richting aan, maar deze kunnen niet doorlopen worden omdat dit niet tot juiste resultaat leidt.

Stel dat we de volgende FORALL-loop hebben:
FORALL (I=1:3) B(I)=1.0/A(I)

In figuur 3.6 geven het "worst case" scenario van alle mogelijke doorlooprichtingen. In dit geval vindt synchronisatie plaats, eerst moet het rechter lid worden bepaald voordat de toekenning plaats vindt aan het linker lid.



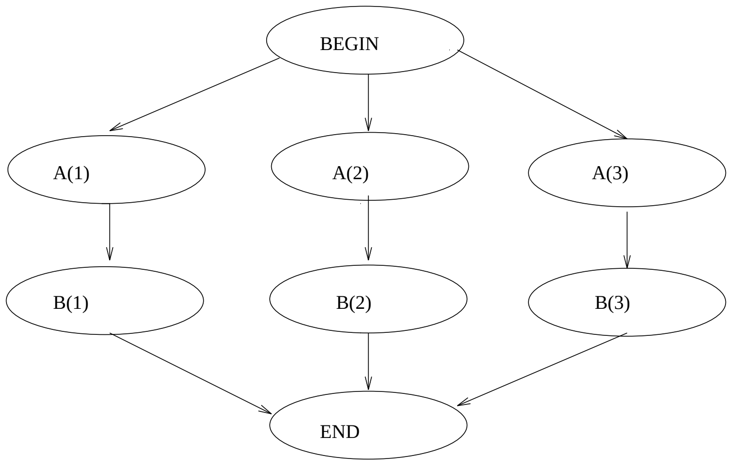
Figuur 3.5: Uitvoergraaf van de DO-loop



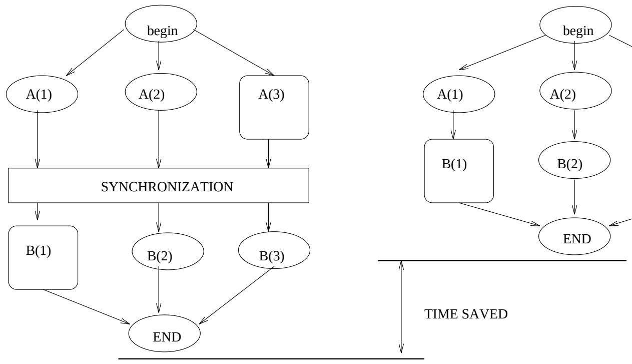
Figuur 3.6: Uitvoergraaf van de FORALL-loop

INDEPENDENT-Directieve met de FORALL

Als we voor een loop het INDEPENDENT directieve plaatsen geven we aan de compiler extra informatie dat de loop parallel doorlopen kan worden, dit is nodig in situaties waar compiler geen duidelijk parallellisme kan ontdekken in de loop. We hebben het INDEPENDENT directieve toegepast op vorige voorbeeld voor de DO en FORALL-loop, zie figuur 3.7, de doorlooprichting is geheel parallel voor beide loops. Ook de DO-loop kan in dit geval geparallelliseerd worden. Dit is in het algemeen niet altijd mogelijk. De programmeur is hier verantwoordelijk bij het gebruik van deze directieve. De FORALL-loop kan best zonder dit extra directieve geparallelliseerd worden, deze loop zoekt dit zelf uit. Maar bij het gebruik van INDEPENDENT directieve, hoeft de FORALL-loop geen testen meer te doen, met name het synchronisatie na iedere fase hoeft niet meer, zie figuur 3.8, zo dat de doorlooptijd in dit geval kleiner wordt. In figuur 3.8 hebben we een unbalanced loop, om het voordeel te benadrukken.



Figuur 3.7: Uitvoergraaf voor DO-loop en FORALL met het INDEPENDENT directieve



Figuur 3.8: Het verschil in synchronisatie in de FORALL-loop

4 Tijdmodel en Performance

4.1 Tijdmodel

Om een indruk te krijgen van de performance van HPF worden eenvoudige wiskundige bewerkingen zoals het inprodukt, matrixvectorvermenigvuldiging (matvec) en update in HPF geïmplementeerd. Geprobeerd wordt om voor de bewerkingen een tijdmodel op te stellen. Aangezien deze bewerkingen de bouwstenen vormen voor de iteratieve Krylov methode, in het bijzonder GMRES, kunnen we aan de hand van het tijdmodel voor de bouwstenen inzicht krijgen in het gedrag van deze iteratieve methoden.

De communicatietijden zijn echter lastig om direct te meten, wel kunnen we de totaaltijd T_{tot} voor de bouwsteen op een verschillend aantal processoren p meten. Uit deze metingen wordt geprobeerd om de communicatietijden te fitten. Voor de bouwstenen nemen we het volgend tijdmodel:

$$T_{\mathrm{tot}} = \frac{T_1}{p} + T_{\mathrm{up}} + k * T_{\mathrm{m}}, \tag{4.1}$$

hierin is T_1 de executietijd op 1 processor, T_{up} is de opstarttijd om communicatie op te zetten, k zijn het aantal boodschappen wat verzonden moet worden en T_{m} is de tijd om één boodschap te sturen.
Het aantal boodschappen k van een bouwsteen is bekend. Door T_{tot} te meten op een verschillend aantal processoren p krijgen we een overbepaald stelsel. Met de kleinste kwadraten methode kunnen we dan uit het overbepaalde stelsel de parameters T_{up} en T_{m} fitten. Hierbij kan gekozen worden of T_1 gefit of gelijk genomen wordt aan de meting op 1 processor. Het overbepaalde stelsel is te schrijven in de volgende vorm:

$$Ax = b, \tag{4.2}$$

hierin is $x = (T_1, T_{\mathrm{up}}, T_{\mathrm{m}})$. De kleinste kwadraten formulering zegt dat er een oplossing $\bar{x}$ bestaat zodat $\|A\bar{x} - b\|$ minimaal is, $\bar{x}$ is een oplossing van

$$A^T A\bar{x} = A^T b. \tag{4.3}$$

De oplossing $\bar{x}$ bevat de 'gefitte' waarden voor de parameters, details van het kleinste kwadraten probleem zie [8].

4.1.1 Bouwstenen

Voor de tijdmeting is gekozen voor de CoSy-HPF compiler. De HPF code wordt getest op een cluster van werkstations. Het cluster, de Babylon genaamd,

bestaat uit 6 werkstations met een Intel pentium processor (p5 200mhz) , welke met elkaar verbonden zijn op een afgeschermd netwerk. De CoSy compiler is een research compiler ontwikkeld door TNO en verkeert nog in de ontwikkelingsfase, hierdoor zijn nog niet alle HPF statements geïmplementeerd. Bij de ontwikkeling van de CoSy compiler is de aandacht uitgegaan naar efficiente implementatie van de FORALL loop. De FORALL loop spelen een grote rol bij het ontwikkelen van de bouwstenen.
Verder kiezen we voor de afmetingen van de array's $(n \times n \times n)$ met $n = 36$. Hier kunnen we een 3D probleem mee modeleren. Dit is aantrekkelijk omdat in WISH3D de vectoren in dit soort array's worden opgeslagen. Er wordt gedistribueerd in één van de array richtingen. Hogere orde distributies zijn nog niet mogelijk met de CoSy compiler. We zullen nu per bouwsteen het tijdmodel opstellen:

Inwendig produkt:

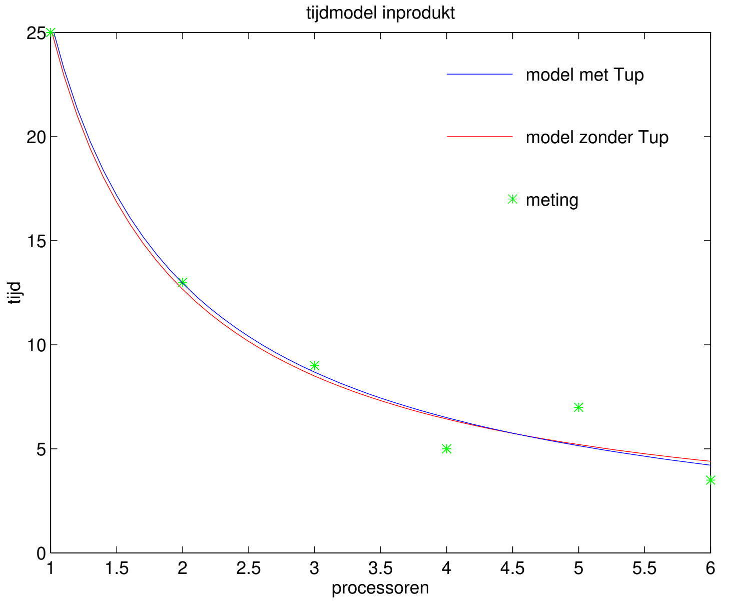
Het inwendig produkt $c = (x, y)$ is geïmplementeerd met de volgende code,

FORALL (k, j, i)
$c(i,j,k) = x(i,j,k) * y(i,j,k)$
END FORALL
$ans = sum(c)$

De code bevat geen standaard functie *c= DOT_PRODUCT(x,y)* omdat deze functie alleen werkt voor $(n \times 1)$ dimensionale array als vectoren en aangezien in WISH3D $(n \times n \times n)$ dimensionale array's als vectoren gebruikt worden moeten we het inwendig produkt zelf implementeren.

Het geïmplementeerde inprodukt bestaat uit 2 delen, nl. de FORALL loop en een sommatie. In de FORALL hoeft niet gecommuniceerd te worden de array's x en y kunnen lokaal met elkaar vermenigvuldigd worden. In de sommatie met sum wordt wel gecommuniceerd, eerst wordt lokaal de sommatie uit gevoerd voor de lokale elementen van c. Deze deelsommen worden vervolgens op 1 processor verzameld (gather). Hiervoor zijn dan $(p-1)$ berichten verzonden. De deelsommen worden op deze processor opgeteld en via 1 bericht (scatter) naar iedere processor terug gezonden. Dus totaal zijn er $k = p$ berichten verzonden. De gemeten totaaltijden T_{tot} voor een verschillend aantal processoren staan vermeld in Tabel 4.1.

Met de totaaltijden en $k = p$ ingevuld in (4.1) krijgen we uit het kleinste kwadraten probleem de gefitten waarden gegeven in Tabel 4.2. De combinatie van (4.1) met parameters uit Tabel 4.2 is het tijdmodel voor het inprodukt. We zullen dit het model met T_{up} noemen, in figuur 4.1 is het model met T_{up} geplot naast de gemeten waarden. Bij gebruik van 5 processoren wordt de gemeten totaaltijd in Tabel 4.1 groter, dit heeft te maken met de load-balancing. We komen hier bij het behandelen van de GMRES nog op terug.



Figuur 4.1: Totaaltijd model Inprodukt, CoSy compiler op de Babylon

PE's	T_{tot}
1	25
2	13
3	9
4	5
5	7
6	4

Tabel 4.1: *Totaaltijden (msec) voor het inprodukt, met CoSy compiler en afmetingen array's 36x36x36 op de Babylon*

T_1	25.0
T_{up}	0.646
T_m	-0.995

Tabel 4.2: *Gefitte waarden voor het inprodukt, met CoSy compiler en afmetingen array's 36x36x36*

In Tabel 4.2 is T_{m} negatief. Doordat T_{up} hoog is wordt T_{m} af en toe negatief bij het fitten. De negatieve waarde maakt het model onbruikbaar, omdat de tijden altijd positief horen te zijn. Bij werkstations is het realistischer om T_{up} als een functie van het aantal processoren te beschouwen. We combineren T_{up} met T_{m} in het volgende model :

$$T_{\mathrm{tot}} = \frac{T_1}{p} + p * T_{\mathrm{m}} \quad p \geq 2. \tag{4.4}$$

Dit zullen we het model zonder T_{up} noemen. Met de gefitte waarden gegeven in Tabel 4.3. Hierdoor wordt T_{m} positief en kunnen met het model werken. In Figuur 4.1 is te zien dat beide modellen ongeveer gelijk zijn.

T_1	25.6
T_{m}	0.0328

Tabel 4.3: *Gefitte waarden voor het inprodukt, met CoSy compiler en afmetingen array's 36x36x36 op de Babylon*

Matrix-vector vermenigvuldiging:

De matrix-vector vermenigvuldiging, $v2 = Av1$, waarbij A ijl is en bestaat uit 7 diagonalen (3D probleem). Deze diagonalen zijn in WISH3D opgeslagen in 3 dimensionale array's (vectoren). De vectoren $v1$ en $v2$ zijn ook opgeslagen in 3 dimensionale array's. De implementatie in HPF is als volgt:

FORALL (k, j ,i)
$v_2(i,j,k) = (a_p)_{i,j,k} v1_{i,j,k} + (a_e)_{i,j,k} v1_{i+1,j,k} + ...$
$...+(a_b)_{i,j,k} v1_{i,j,k-1}.$
END FORALL

In de code evalueert de FORALL-loop, vanwege de stencil structuur in het argument, mogelijke afhankelijkheden in elementen in de gedistribueerde richting. Op iedere processor wordt precies bepaald welke elementen nodig zijn op de buurprocessen om te kunnen vermenigvuldigen. Deze elementen worden ghostcellen genoemd. Vervolgens worden de ghostcellen uitgewisseld naar betreffende buurprocessoren, dus communicatie. Maar voordat de ghostcellen worden opgestuurd, worden ze ingepakt, dat wil zeggen met een bepaald algoritme compacter opgeslagen. Nadat ze opgestuurd zijn worden de ghostcllen eerst uitgepakt en kan het resterende werk van de FORALL-loop, de vermenigvuldiging, lokaal worden afgehandeld. Bij p processoren zijn er $(p-1)$ interfaces tussen de processoren, er worden dan 2(p-1) pakketjes tussen buurprocessoren verstuurd (bij 1 dimensionale distributie). Dus $k = 2(p-1)$.

Het model met T_{up} voor matvec ziet er als volgt uit:

$$\begin{aligned} T_{\mathrm{tot}} &= \frac{T_1}{p} + T_{\mathrm{up}} + 2*(p-1)*T_{\mathrm{m}} \\ T_{\mathrm{tot}} &= \frac{T_1}{p} + T_{\mathrm{up}} + p*\tilde{T}_{\mathrm{m}} \qquad p \geq 2. \end{aligned} \tag{4.5}$$

Om het model beter te kunnen vergelijken, hebben we de laatste term veranderd. Voor de $\tilde{T}_m$ hebben we nu een p afhankelijkheid. In vervolg noemen we $\tilde{T}_m$ weer T_m. De gemeten totaaltijden staan in Tabel 4.4.
Dit geeft de volgende waarden in Tabel 4.5 voor de te fitten grootheden.

PE's	T_{tot}
1	65
2	31
3	37
4	17
5	16
6	24

Tabel 4.4: *Totaaltijden (msec) voor matvec, met CoSy compiler en afmetingen array's 36x36x36 op de Babylon*

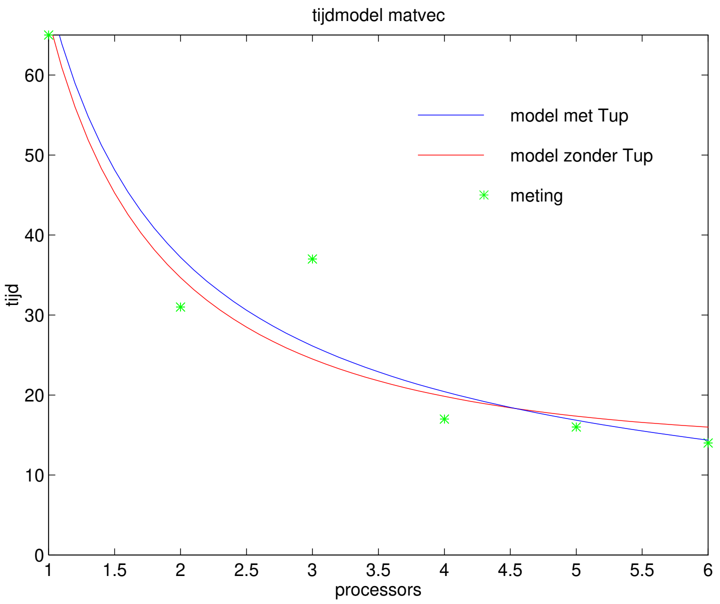
Het blijkt dat het in- en uitpakken van ghostcellen te verwaarlozen is tegen

T_1	64.42
T_{up}	5.7
T_{m}	-0.346

Tabel 4.5: *Gefitte waarden voor matvec, met CoSy compiler en afmetingen array's 36x36x36*

de communicatietijden. Ook hier ondervinden we het zelfde probleem van negatieve waarden met het fitten van parameters. We stellen een nieuw model op waarbij we er van uitgaan dat T_{up} op een cluster van werkstation ook afhankelijk is van het aantal processoren p. Combineren we dit met T_{m} waarbij het aantal te versturen boodschappen gelijk is aan p. We krijgen dan model zonder T_{up}:

$$T_{\mathrm{tot}} = \frac{T_1}{p} + p*T_{\mathrm{m}} \qquad p \geq 2. \tag{4.6}$$



Figuur 4.2: Tijdmodel matvec, CoSy compiler op de Babylon

Dit geeft de volgende waarden in Tabel 4.6.

T_1	66.02
T_{m}	0.833

Tabel 4.6: *Gefitte waarden voor matvec, met CoSy compiler en afmetingen array's 36x36x36*

Zie in figuur 4.2 beide modellen voor matvec, waarin model 1 geen realistisch gedrag vertoont. Vanwege de negatieve waarden van T_m zal de totaaltijd bij toename van het aantal processoren uiteindelijk zelfs negatief worden. Model 2 is wat dat betreft realistischer, hierbij begint de totaaltijd juist toe te nemen bij gebruik van meer processoren.

Vector Update:

De bouwsteen update $x = x + ay$, met x en y vectoren opgeslagen in een 3 dimensionale array's en a een scalar. Dit geeft de volgende implementatie in HPF:

FORALL (k, j, i)
$x(i,j,k) = x(i,j,k) + a * y(i,j,k)$
END FORALL

Er wordt bij de update geen communicatie verwacht, alles kan lokaal op iedere processor bepaald worden. Hieruit volgt dat we de communicatietermen in (4.1) kunnen verwaarlozen, voor de zekerheid wordt nog wel een parameter T_α

voor de mogelijke overhead aangenomen, zodat het model blijft kloppen:

$$T_{\text{tot}} = \frac{T_1}{p} + T_\alpha \quad p \geq 1. \tag{4.7}$$

De gefitte waarden staan in Tabel 4.8.

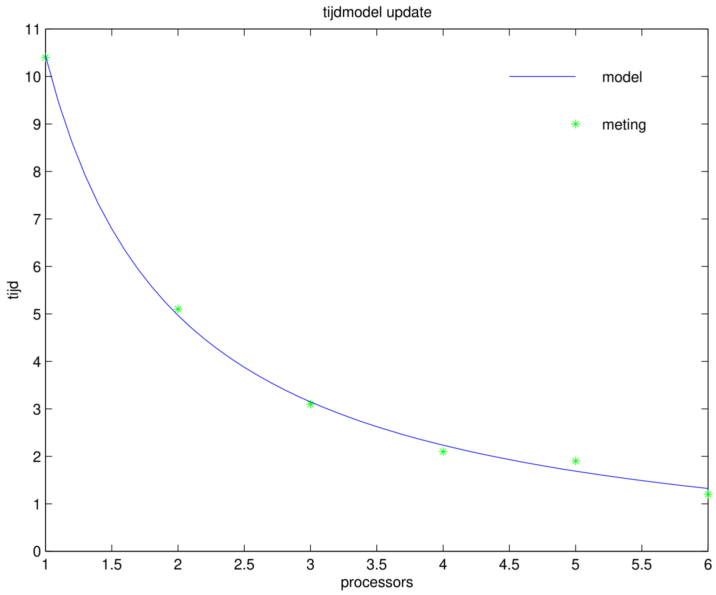
PE's	T_{tot}
1	11
2	5
3	3
4	3
5	2
6	1

Tabel 4.7: *Totaaltijden (msec) voor update, met CoSy compiler en afmetingen array's 36x36x36 op de Babylon*

T_1	11.4
T_α	-0.5

Tabel 4.8: *Gefitte waarden voor update, met CoSy compiler en afmetingen array's 36x36x36 op de Babylon*

Uit figuur 4.3 blijkt het model prima overeen te komen met de metingen.



Figuur 4.3: Tijdmodel update, CoSy compiler Babylon

Samenvattend

Bij het inprodukt en matvec hebben we het model (4.1) aangepast voor werkstations zodat T_{up} ook afhankelijk van p is. Hierdoor krijgen we een bovengrens voor de tijdschatting. De parameters van de tijdmodellen zijn bepaald door metingen op clusters van maximaal 6 werkstations. Toch proberen we met behulp van het tijdsmodel erachter te komen hoe een solver (iteratieve methode), bestaande uit de diverse bouwstenen, zich zal gedragen bij meer dan 6 processoren. Verderop in het verslag zullen we de resultaten van het tijdsmodel opgebouwd uit de bouwstenen vergelijken met de direct gemeten tijd.

4.2 Performance HPF

Het doel om parallel te rekenen is om de rekentijd te bekorten (wallclock tijd). Om een indicatie te krijgen hoe de prestatie van de ontwikkelde parallelle solver is, hebben we een 'meetinstumenten' nodig. Hieronder geven we definities waarmee we de kwaliteit kunnen uitdrukken.

Speedup

De **Speedup** is gedefinieerd als de verhouding van de tijd om een probleem op te lossen op 1 processor met de tijd die nodig is met p processoren. Laat T_1 en T_p respectievelijk de tijd op 1 processor en de tijd op p processoren zijn. De speedup volgt uit:

$$S(p) = \frac{T_1}{T_p}. \tag{4.8}$$

Om T_1 te bepalen kunnen we het parallelle programma op 1 processor draaien. Vanwege de struktuur van het programma voor parallellisme is extra overhead in de tijdmeting aanwezig, waardoor de speedup beter zal uitvallen. Door de tijd van het beste sequentiële algoritme als T_1 te gebruiken, krijgen we een beter beeld, dit wordt ook wel de **True Speedup** genoemd.

Efficientie

Een goed werkend parallelprogramma noemen we **Efficient** indien het een lineaire speedup heeft met het aantal gebruikte processoren. Vanwege communicatie, en geen goede load-balance en synchronisatie wordt de efficientie minder. We definiëren de **Efficientie** met:

$$E(p) = \frac{S(p)}{p}. \tag{4.9}$$

Schaalbaarheid

Een parallel algoritme heet **Schaalbaar** als de efficientie constant blijft bij het opvoeren van de probleemgrootte en het vergroten van het aantal processoren. In de praktijk zal de efficientie kleiner worden doordat in het algemeen de communicatie overhead en load inbalance groter worden als het aantal processoren toeneemt.

5 Parallelle Solvers

5.1 Inleiding

In dit hoofdstuk worden de parallelle eigenschappen van solvers bekeken. In een testprobleem, te vinden in APPENDIX A, worden de solvers getest. Dit testprobleem gebruikt dezelfde discretisatie als in WISH3D. Er wordt in het testprobleem een lineair stelsel opgelost vergelijkbaar met de stelsels die bij het SIMPLE proces ontstaan in WISH3D. We gaan uit van vectoren in 3D array opgeslagen en voor de afmetingen nemen we $(36 \times 36 \times 36)$, waarbij in de k richting is gedistribueerd. De HPF code voor het testprobleem wordt met de CoSy compiler gecompileerd en de parallelle code testen we uit op de Babylon, zie hoofdstuk 4 voor meer details.

De CoSy compiler levert efficiente parallelle code op, maar de compiler is beperkt in HPF statements en nog steeds in ontwikkeling. Het is de bedoeling WISH3D met een andere HPF compiler te compileren, namelijk met de PGHPF compiler van de Portland Group. Dit is een volledige HPF compiler, maar de CoSy compiler geeft vaak betere resultaten met FORALL loops dan de PGI compiler, vandaar de keuze voor de CoSy compiler om het maximale eruit te halen. De CoSy compiler draait op een afgeschermde cluster (Babylon), hierdoor hebben we geen invloed van andere processen. Dit is voor tijdmetingen van groot belang. De PGI compiler draait op een cluster van SUN-werkstations, deze machines zijn minder snel dan de machines van de Babylon en niet afgeschermd.

We beginnen de testen met de TDMA solvers en daarna komt een iteratieve Krylov solver aanbod.

5.2 TDMA-solver

In het WISH3D pakket wordt het stelsel (2.31) met o.a. de TDMA-solvers opgelost. TDMA solvers bestaat uit drie varianten: Line TDMA, Plane TDMA en Space TDMA. In [1] worden TDMA solvers besproken en in [9] paragraaf 3.2.1 pag.12 worden te verwachten problemen besproken van de Plane TDMA-solver bij parallellisatie. In deze paragraaf zal Line TDMA worden uitgediept.

5.2.1 Parallel line-TDMA

In vergelijking (2.31) is A onder te verdelen in blokken, zodanig dat een blok precies een lijn voorstelt in het grid. Hieronder wordt de linx solver uitgelegd in een 2D situatie, deze lijnen waar de oplossing van bepaald wordt liggen dan evenwijdig aan de x-as (i-richting). Het oplossen in deze richting wordt ook wel een i-sweep genoemd. A wordt globaal op de volgende wijze in blokken opgesplitst:

$$A = D - L - U, \tag{5.1}$$

waarbij D, L, en U de volgende blokken bevatten $D = diag\{A_1, A_2, ..., A_{NJ}\}$, $L = diag\{L_2, L_2, ..., L_{NJ}\}$, en $U = diag\{U_1, U_2, ..., U_{NJ-1}\}$. De blokken zijn als volgt verdeeld, voor het blok A_j geldt:

$$A_j = \begin{pmatrix} (a_p)_{1,j} & -(a_e)_{1,j} & & & \\ -(a_w)_{2,j} & (a_p)_{2,j} & -(a_e)_{2,j} & & \\ \ddots & \ddots & \ddots & & \\ & -(a_w)_{NI-1,j} & (a_p)_{NI-1,j} & -(a_e)_{NI-1,j} & \\ & & -(a_w)_{NI,j} & (a_p)_{NI,j} & \end{pmatrix}, \tag{5.2}$$

Hierin stelt $(.)_{i,j}$ het i-de punt van de j-de lijn voor. De blokken L_j en U_j zijn gedefineerd als,

$$L_j = \begin{pmatrix} (a_s)_{1,j} & & & \\ & (a_s)_{2,j} & & \\ & & \ddots & \\ & & & (a_s)_{NI,j} \end{pmatrix}, \quad U_j = \begin{pmatrix} (a_n)_{1,j} & & & \\ & (a_n)_{2,j} & & \\ & & \ddots & \\ & & & (a_n)_{NI,j} \end{pmatrix}, \tag{5.3}$$

Ook x is opgesplitst per blok (lijn), net zoals b,

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_{NJ} \end{pmatrix}, \text{ per lijn } x_j = \begin{pmatrix} x_{1,j} \\ \vdots \\ x_{NI,j} \end{pmatrix}. \tag{5.4}$$

De orginele line-TDMA solver in WISH3D, bepaalt per lijn een oplossing via een Block Gauss-Seidel structuur:

$$A_j x_j^m = L_j x_{j-1}^m + U_j x_{j+1}^{m-1} + b_j. \tag{5.5}$$

Hierin staat m voor de m-de iteratie. Voor de parallelle oplossing voor de linx solver is hier gekozen om de moeilijker te parallelliseren Block Gauss-Seidel structuur te vervangen door de Block Jacobi, dit resulteert in $L_j x_{j-1}^{m-1}$ de oude waarden. Dit heeft gevolgen voor de convergentie snelheid. De matrix A_j is tridiagonaal en de stelsels (5.5) worden opgelost met de methode van Crout, zie [10].

Met de methode van Crout wordt voor de blokken A_j de volgende LU-decompositie gepostuleerd,

$$A_j = LU = \begin{pmatrix} \alpha_1 & & & \\ -(a_w)_{2,j} & \alpha_2 & & \\ & \ddots & \ddots & \\ & & -(a_w)_{NI,j} & \alpha_{NI} \end{pmatrix} \begin{pmatrix} 1 & \beta_1 & & & \\ & 1 & \beta 2 & & \\ & & \ddots & \ddots & \\ & & & 1 & \beta_{NI-1} \\ & & & & 1 \end{pmatrix} \tag{5.6}$$

Stel dat we de juiste distributie gevonden hebben, dan worden de coefficiënten α, en β op onderstaande wijze berekend:

DO $i = 1...NI$
FORALL $j = 1...NJ$
$\alpha_i = (a_p)_{i,j} + (a_w)_{i,j}\beta_{i-1}$
$\beta_i = -(a_e)_{i,j}/\alpha_i,$
END FORALL
END DO

Hierin is $\beta_0 = 0$. Het i^e element (punt) wordt voor alle lijnen j parallel (tegelijk) bepaald. De volgende stap wordt het oplossen van het onderdriehoeksstelsel $Ly = b$ en daarna het bovendriehoeksstelsel $Ux = y$. Voor een onderdriehoeksstelsel geldt:

DO $i = 1...NI$
FORALL $j = 1...NJ$
$y_{i,j} = (b_i + (a_s)_{i,j}x_{i,j-1} + (a_n)_{i,j+1}x_{i,j+1} + (a_w)_{i,j}y_{i-1,j})/\alpha_i,$
END FORALL
END DO

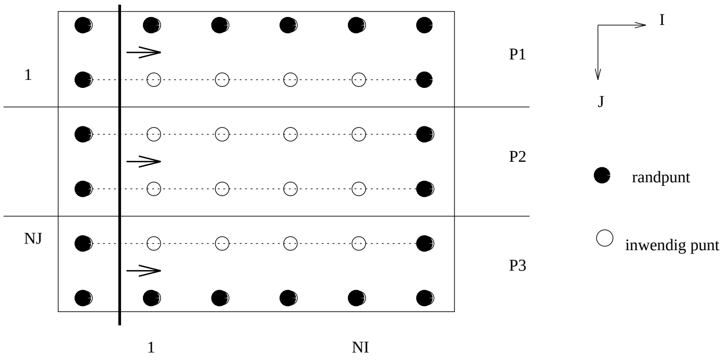
Volgens Crout is $y_{0,j} = 0$, maar in de implementatie is gekozen voor de voorwaarde $y_{0,j} = x_{rand}$, op deze manier is de randvoorwaarde verwerkt. Als laatste volgt dan de terugsubstitutie van het bovendriehoekstelsel:

DO $i = NI...1 \ \ (-1)$
FORALL $j = 1...NJ$
$x_{i,j} = y_{i,j} + \beta_i x_{i+1,j},$
END FORALL
END DO

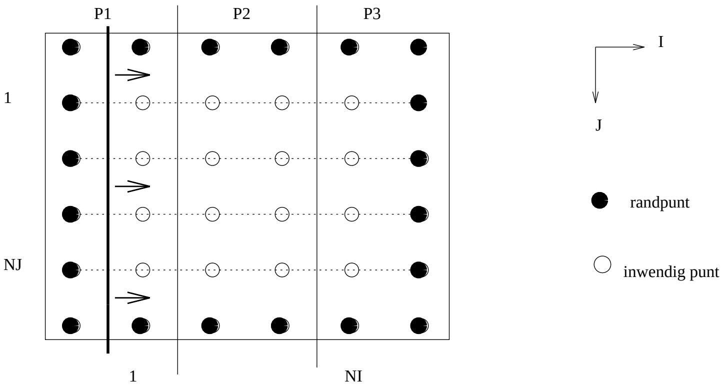
Ook hier is volgens methode van Crout $x_{NI+1,j} = 0$, maar in de implementatie is dat weer de randvoorwaarde $x_{NI+1,j} = x_{rand}$. Merk op dat zo de randvoorwaarden weer in de berekening komen.

Hieronder zullen we het gedrag bespreken van linx solver aan de hand van twee distributies in i en j-richting.
Stel de distributie in j-richting, zie 5.1. Linx bepaalt via bovenstaand algoritme de elementen van Crout α en β in i-richting per lijn. De lijnen zijn

echter door de distributie verdeeld over het aantal processoren. Hierdoor kunnen de elementen parallel worden berekend, hetzelfde geldt voor het oplossen van boven- en onderdriehoekstelsel, dit gaat ook per element in i-richting en en door de distributie gaat dit ook weer parallel. Bij de laatste stap, de terugsubstitutie, komt dan ook een lijn van oplossingen (punten) per i-stap beschikbaar, aangezien de elementen i van de lijn parallel bepaald worden.



Figuur 5.1: 2D grid, distributie in j-richting, sweep in i-richting



Figuur 5.2: 2D grid, distributie in i-richting, sweep in i-richting

Maar een distributie in i-richting, zie 5.2, gaat dit bepalen van de elementen i niet meer parallel omdat de distributie nu de elementen i over de processoren verdeelt zijn i.p.v de lijnen j. Hieruitvolgt dat maar *één* processor tegelijk aan het werk is, dus een sequentieel proces.

Generalisatie naar 3D

Bij het testprobleem is een 3D probleem geïmplementeerd, hierbij worden de parallelle versie van de TDMA solver: linx, liny, en linz gebruikt voor het oplossen. Met resp. een z.g. sweep-richting in i, j, en k.
Zolang de sweep-richting loodrecht op de distributie-richting staat werkt de solver parallel. Dus de linx solver (in 3D) met de sweep in i-richting werkt voor distributies in j, k-richting parallel. En voor een distributie in i-richting

werkt de parallel geïmplementeerde linx solver voor een deel sequentieel (het algoritme is wel sequentieel, maar in de implementatie kan nog wat parallel worden afgehandeld). De liny en linz solvers hebben vergelijkbare eigenschappen.

De matrices van het testprobleem zijn in k-richting gedistribueerd, dit houdt in dat linx, en liny parallel werken en linz deels sequentieel werkt. Het testprobleem wordt allereerst opgelost met linx, liny, en linz door ze cyclisch af te wisselen. Hiervan zijn metingen verricht om de speedup en efficientie te berekenen, zie Figuur 5.3 en 5.4. Hieruit volgt een minder efficient parallel gedrag voor deze combinatie van de solvers. In Figuren 5.5 en 5.6 is speedup en efficiëntie berekend voor alleen de linz solver, we zien hierin een slecht parallel gedrag. De speedup daalt vrij snel ten opzichte van het aantal processoren, bij 6 processoren hebben we een speedup 3.5. Hieruit volgt dat linz sequentieel gedrag bezit. Met behulp van het volgende model kan de verhouding tussen parallel en sequentieel gedeelte zichtbaar worden gemaakt:

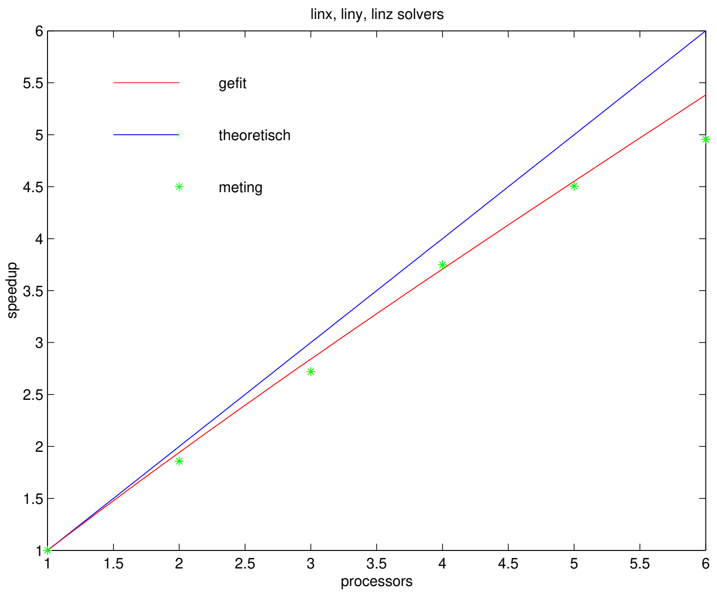
$$T_{\rm tot} = T_{\rm s}(\alpha + (1-\alpha)\frac{1}{p}). \tag{5.7}$$

Hiermee is $T_{\rm tot}$ de gemeten parallelle tijd, $T_{\rm s}$ is de tijd gemeten op 1 processor, en p is het aantal processoren, waarbij α de sequentiële verhouding aangeeft. Het gemiddelde voor α van linz bedraagt 0.16, d.w.z dat ongeveer 84% parallel is en 16 % sequentieel. Zie opmerking in [9] paragraaf 4.1.1 pag.22 met de wet van Amdahl kan de speedup in dit geval nooit groter worden dan ongeveer 6.

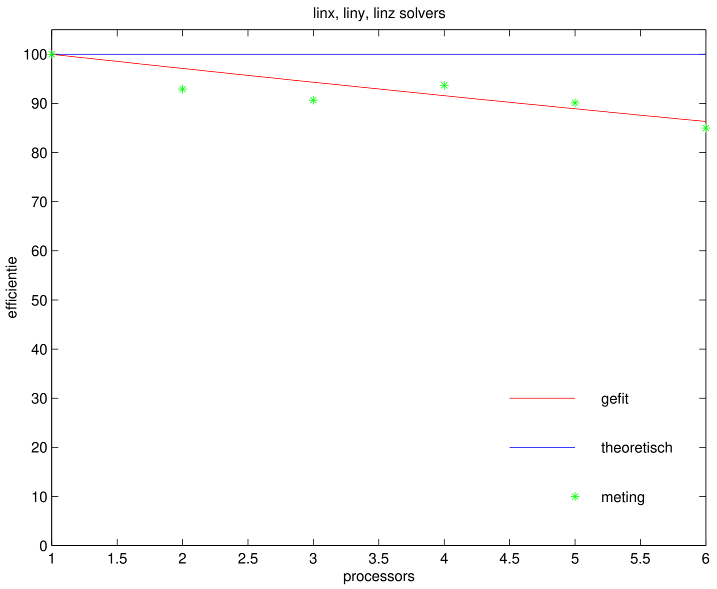
Een oplossing zou zijn door linz te verbieden als de distributie in k-richting is, hiervoor zijn ook weer de speedup, en efficientie berekend, zie Figuur 5.7 en 5.8. Dit laat zeer goede parallelle eigenschappen zien, maar de solver combinatie zal wel minder robuust zijn dan met linx, liny, en linz combinatie.
De laatste variant van de TDMA heeft zeer goede parallelle eigenschappen, maar is minder robust geworden. In Tabel 5.1 staan van de bovenstaande besproken combinaties de tijden op een verschillend aantal processoren. Mogelijk kan bv. Krylov-solver (GMRES) met deze TDMA-variant versneld worden, dit zal in paragraaf 6 verder worden uitgewerkt.

pe	linx, liny, linz	linx,liny	linz
1	446.1	450.9	411.6
2	240	230.2	242.9
3	164	150.2	190.5
4	119.6	111.2	160.7
5	99	99.4	121.8
6	88.8	73.4	116.5

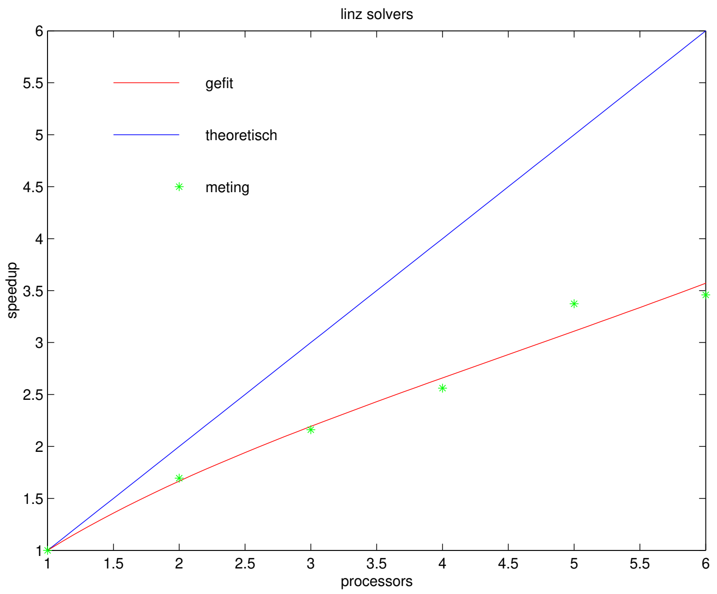
Tabel 5.1: *Totaaltijden (sec) voor linxyz, linxy en linz solver bij een distributie in k richting, voor de CoSy compiler op Babylon*



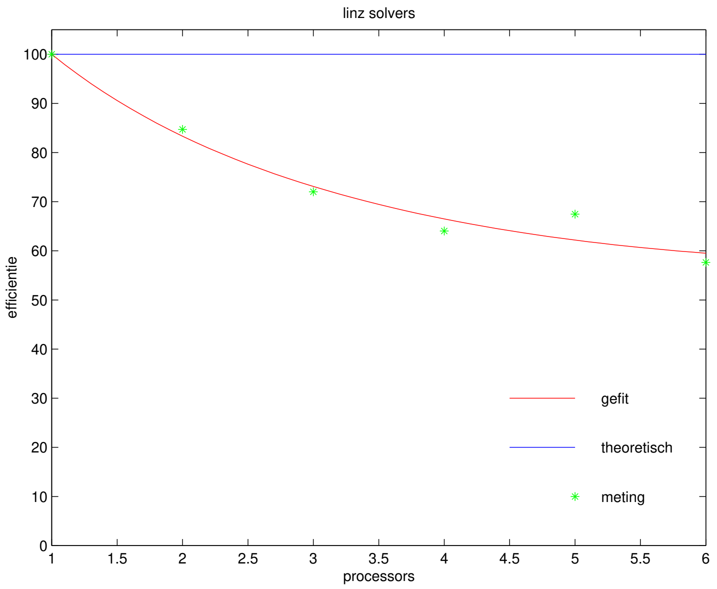
Figuur 5.3: Speedup, distributie in k-richting, sweeps in i, j, k-richting



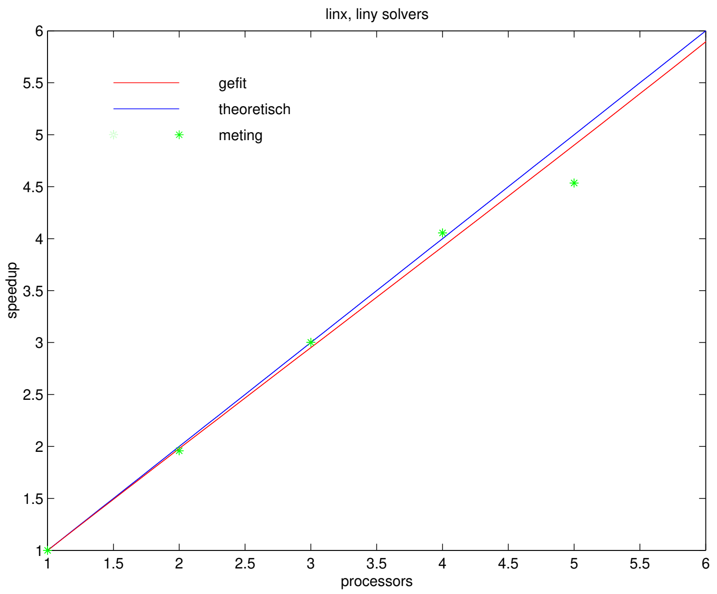
Figuur 5.4: Efficiëntie, distributie in k-richting, sweeps in i,j,k-richting



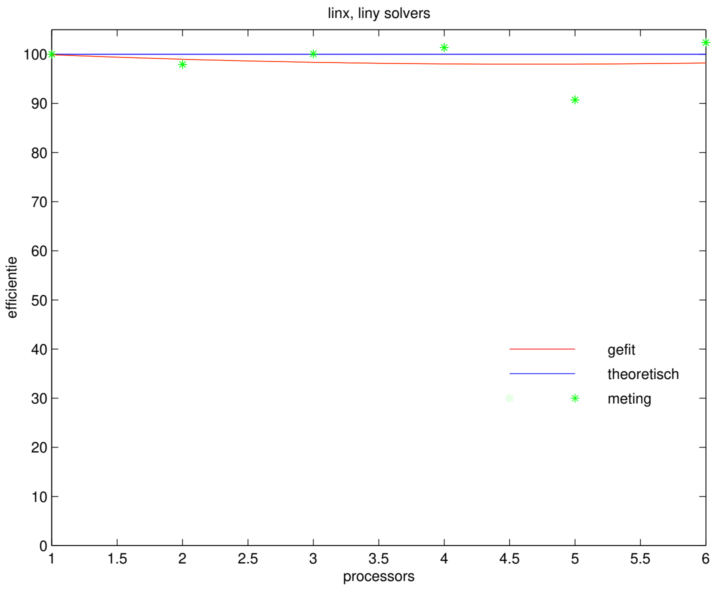
Figuur 5.5: Speedup, distributie in k-richting, sweep in k-richting



Figuur 5.6: Efficiëntie, distributie in k-richting, sweep in k-richting



Figuur 5.7: Speedup, distributie in k-richting, sweeps i,j-richting



Figuur 5.8: Efficiëntie, distributie in k-richting, sweeps in i,j-richting

5.3 Incomplete LU decompositie (ILU)

De ILU methode is te vergelijken met de SPACE-TDMA solver zoals opgemerkt in [1]. Hieronder zullen we de ILU-methode uitleggen en de parallelle eigenschappen analyseren. De SPACE-TDMA solver is robust en convergeert snel in de tijd en is daarom veel gebruikt in WISH3D. Vandaar het animo om deze solver te parallelliseren.

ILU principe

We lossen het stelsel (2.31) op, waarbij A ijl is. Dit kunnen we doen met een directe methode, Gauss decompositie. Het probleem hierbij is dat ijle structuur verloren gaat, de L en U matrices lopen in het proces vol, z.g. fill-in. Het principe van ILU methode is deze fill-in te verwaarlozen. Splits A op in de volgende vorm,

$$A = LD^{-1}U - R. \tag{5.8}$$

waarbij L, D, U en R aan de volgende eisen voldoen:
L : strikt onderdriehoeksstelsel (L^s) van A + diagonaal matrix D
U : strikt bovendriehoeksstelsel (U^s) van A + diagonaal matrix D
R : $R = LD^{-1}U - A$.
D : is een diagonaal matrix, met elementen $d_{i,j,k}$, deze volgen uit:

$$\text{Stel} \;\; diag(LD^{-1}U) \; = \; diag(A). \tag{5.9}$$

We schrijven L en U hierbij als $L = (L^s + D)$ en $U = (U^s + D)$. Ingevuld in (5.9) krijgen we,

$$\begin{aligned}
diag\left((L^s + D)D^{-1}(U^s + D)\right) &= (a_p)_{i,j,k}, \\
diag\left(L^s D^{-1} U^s + U^s + D + L^s\right) &= (a_p)_{i,j,k}, \\
diag\left(L^s D^{-1} U^s\right) + d_{i,j,k} &= (a_p)_{i,j,k},
\end{aligned} \tag{5.10}$$

waarbij $diag(L^s) = \mathbf{0}$, en voor U^s analoog. Werk de eerste term van (5.10) uit, er volgt:

$$d_{i,j,k} = (a_p)_{i,j,k} - \frac{(a_w)_{i,j,k}(a_e)_{i-1,j,k}}{d_{i-1,j,k}} - \frac{(a_s)_{i,j,k}(a_n)_{i,j-1,k}}{d_{i,j-1,k}} - \frac{(a_b)_{i,j,k}(a_t)_{i,j,k-1}}{d_{i,j,k-1}}. \tag{5.11}$$

Met $(a_e)_{0,j,k} = 0$, $(a_n)_{i,0,k} = 0$, $(a_t)_{i,j,0} = 0$, hieruitvolgt dat $d_{1,1,1} = (a_P)_{1,1,1}$. We lossen nu het stelsel $LD^{-1}Ux = b + Rx$ op, waarbij we de term Rx verwaarlozen.

ILU-Algoritme

Met iteratief naverfijnen wordt ILU recursief gemaakt, meer details zie [11] p28. We kunnen dan als extra optie de verfijning instellen. Het algoritme ziet er als volgt uit,

maak $d_{i,j,k}$
for $m = 1..Max$ **do**
  $r = b - Ax^m$
  $LD^{-1}U\delta = r$
  $x^{m+1} = x^m + \delta$
end for

Uit de literatuur is bekend dat ILU moeilijk te parallelliseren is. Bij parallellisatie wordt meestal gebruik gemaakt van ontkoppeling. Ontkoppelen heeft tot gevolg dat de eigenschappen van de solver worden veranderd. Zogeheten red-black ordening is veel gebruikt, het probleem in HPF is dat we de data-matrices anders moeten verdelen, dit kan door aanroep van ILU, maar dat betekent dat iedere ILU iteratieslag een copie gemaakt moet worden, dit blijkt dus niet efficient. Het volledig doorvoeren van red-black ordening is ook een optie, maar wordt in dit verslag niet onderzocht. Dit kan voor ILU efficient zijn, maar voor andere solvers of algoritmes binnen WISH3D niet. In [12] wordt de ILU gerealiseerd met CYCLE distributie, maar resultaten vielen hierbij tegen.

Het idee is om een ontkoppelde Block ILU te maken, waarbij de matrix A in blokken wordt opgedeeld zodanig dat iedere processor een blok bezit. Lokaal wordt dan de ILU-methode toegepast. Dit houdt overigens wel in dat we een processor afhankelijke ILU solver hebben.

Parallellisatie

Uitgaande van het 3D testprobleem, met 3D array's waarin k-richting is gedistribueerd. We maken gebruik van de kennis hoe de FORALL loop door de CoSy compiler behandeld wordt, voor de parallellisatie.
De FORALL wordt door de CoSy compiler voor de lokale processen omgezet in DO loops. In dit geval waar 3D array's gebruikt worden en de FORALL met drie indices door de array's loopt, resulteert dit lokaal in drie DO loops. De index range voor deze DO loops is aangepast voor het behandelen van de lokale arrayelementen. Lokaal gebruiken we dus DO loops dat wil dan ook zeggen dat de doorlooprichting (lokaal) vast is, waar indices in oplopende richting worden doorlopen.

Bij afhankelijkheden (recursies) in de expressie in de FORALL-asignment, van de array's welke verdeeld zijn, wordt door de CoSy compiler een hulpvariabele ingevoerd. Hierin wordt de recursie variabele tijdelijk bewaard. Met deze hulpvariabele worden dan de oude waarden gebruikt in de expressie, waardoor de lokale doorlooprichting er toch niet tot doet. Zodat alles geheel in overeenstemming is met de afspraken over de FORALL in HPF. Met de $-u$ compiler optie voor de CoSy compiler kan men het testen op recursie in gedistribueerde richtingen uitzetten, waardoor geen hulpvariabele gemaakt wordt bij recursies. De programmeur is dan ook zelf verantwoordelijk om ervoor te zorgen dat in de FORALL expresies geen recursie voorkomt en als dat wel het geval is moet hij zelf een hulpvariabele invoeren. Met het principe dat we lokaal over DO loops beschikken en het (lokaal) toelaten van recursie bij afhankelijkheden in verdeelde richtingen (dus geen hulpvariabele invoeren) kunnen we de ontkoppelde BLOCK ILU implementeren.

De eerste stap in het ILU algoritme is het maken van de matrix D met elementjes $d_{i,j,k}$ met behulp van (5.11). De FORALL loop herkent wel de stencil structuur in (5.11) en bepaalt daarom voor iedere buurprocessor de ghostcellen. Door nu juist de waarden op de randen van de partitie heel erg groot te maken, krijgen de ghostcellen een grote waarde. Hierdoor valt de laatste term in (5.11) voor buurgridpunten te verwaarlozen. Op randen worden namelijk de buurpunten in de recursie gebruikt. Dus de gridpunten op verschillende processoren zijn onafhankelijk van elkaar gemaakt. Lokaal laten we de recursie toe met gebruik van de DO loops.

De tweede stap is het berekenen van het residu voor het iteratief naverfijnen, dit levert geen problemen op bij parallellisatie. De expressie voor het residu bevat geen recursie. De elementen van het residu kunnen met het FORALL bepaald worden

De derde stap is het oplossen van het stelsel $LD^{-1}U\delta = r$, beschouw hierbij eerst het onderdriehoekstelsel:

$$Lz = r, \tag{5.12}$$

Het oplossen van een onderdriehoekstelsel is recursief, we beginnen met het eerste element van z te bepalen, vervolgens kunnen we het tweede element bepalen (met gebruik van eerste elementje), etc. Het oplossen van (5.12) willen we op iedere processor lokaal uitvoeren. Bij gebruik van de FORALL, met de compiler optie, op (5.12) kunnen we dan lokaal deze onderdriehoekstelsels met behulp van de DO loops oplossen. De FORALL loop blijft wel voor de ghostcellen zorgen, door z op nul te initialiseren zijn de buren ontkoppeld.
Nu beschouwen we het bovendriehoekstelsel:

$$U\delta = Dz. \tag{5.13}$$

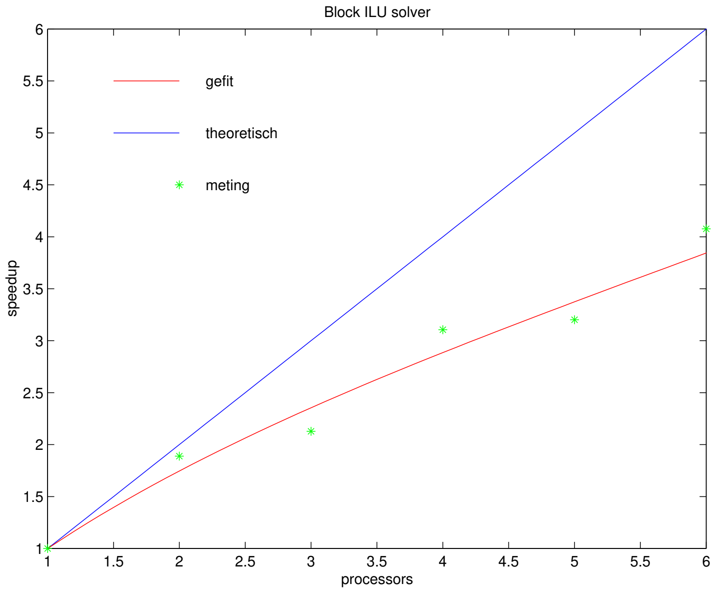
Bij oplossen van dit stelsel, wordt begonnen bij het laatste element, daarna het ene laatste element etc., dit doen we in het sequentiële geval met DO loops in teruglopende volgorde. Nu kunnen we geen FORALL loop gebruiken omdat we deze niet kunnen sturen. Om toch (5.13) benaderend op te lossen veranderen we U door de afhankelijkheid in de gedistribueerde richting k te verwaarlozen. Dit komt neer op het nul maken van het betreffende diagonaalelement. Nu kan voor deze richting een FORALL loop gebruikt worden omdat de richting er toch niet toe doet. De overige richtingen blijven wel door de DO loops lopen. Op dit punt hebben we de ILU nog een keer afgezwakt, eerst door ILU globaal te ontkoppelen en nu door lokaal bij terugsubstitutie een extra ontkoppeling toe te passen, we kunnen dit verantwoorden omdat het iteratieve naverfijnen de fout in oplossing probeert te beperken.

In Tabel 5.2 staan de doorlooptijden voor diverse processoren voor de ontkoppelde BLOCK ILU versie met iteratief naverfijnen voor de CoSy compiler op de Babylon.

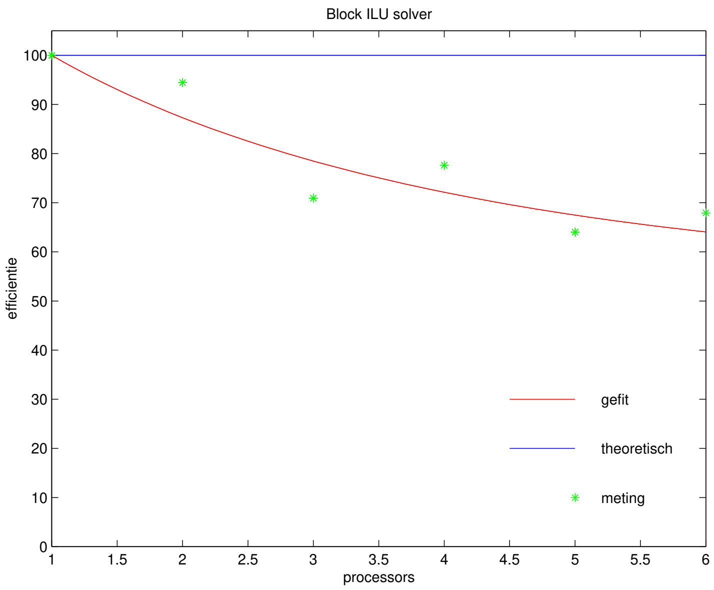
pe	tijd	iteraties
1	341.6	277
2	180.8	284
3	160.5	288
4	110.0	291
5	106.7	293
6	83.8	299

Tabel 5.2: *Totaaltijd ten opzichten van het aantal processoren en iteraties voor ontkoppelde BLOCK ILU gecombineerd met iteratief naverfijnen, voor de CoSy compiler op de Babylon*

In de Tabel 5.2 zien we dat ILU afhankelijk is van het aantal processoren. Het aantal iteraties neemt toe naarmate het aantal processoren groter wordt. Hieruit volgt dat de convergentie beïnvloed wordt door het aantal processoren. In Figuur 5.9 en 5.10 staan respectievelijk de speedup en de efficientie. De ontwikkelde ontkoppelde BLOCK ILU solver met iteratief naverfijnen blijkt niet zo efficient. Dit komt vermoedelijk vanwege het vele initialiseren van de ghostcellen tijdens het naverfijnen. De toename van het aantal iteraties valt voor het testprobleem erg mee. Voor andere problemen en bij gebruik van veel processoren kan dit misschien problemen opleveren. Bij vergelijking van Tabel 5.2 met Tabel 5.1 zien we dat de ontkoppelde BLOCK ILU versie met iteratief naverfijnen wel sneller is.



Figuur 5.9: Speedup voor ontkoppelde BLOCK ILU met iteratief naverfijnen, met CoSy compiler op de Babylon

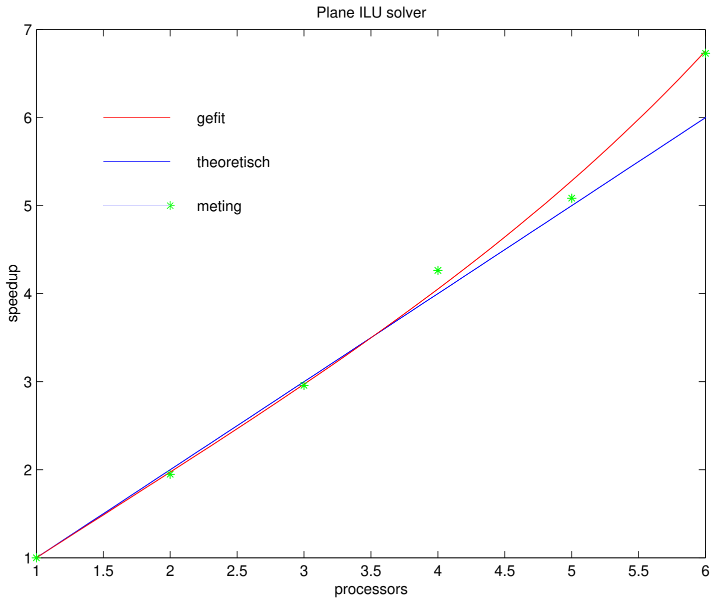


Figuur 5.10: Efficientie voor ontkoppelde BLOCK ILU met iteratief naverfijnen, met CoSy compiler op de Babylon

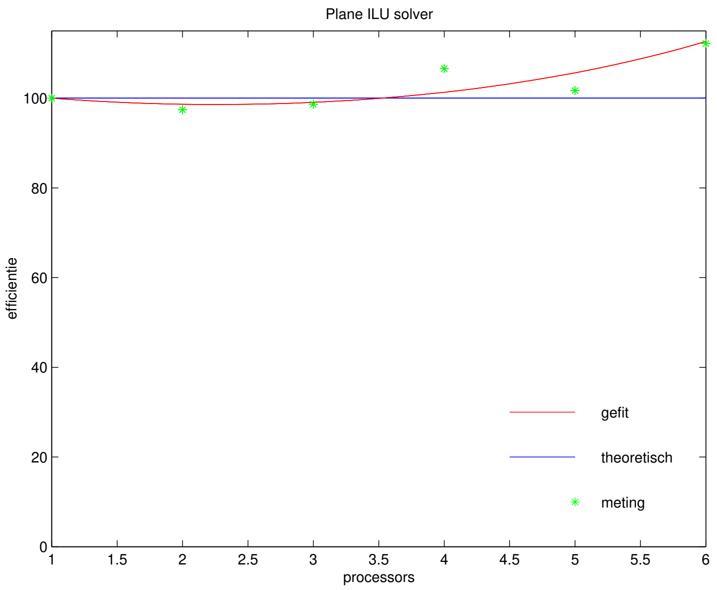
Een processor afhankelijke solver gaat regelrecht in tegen de principes van HPF, waarbij het de bedoeling is dat een parallel programma hetzelfde antwoord oplevert (zelfde convergentie gedrag en iteraties) als het sequentiële programma. Het bovenstaande ontkoppelde ILU algoritme werkt niet met de door PGI ontwikkelde HPF compiler. Deze ontdekt het processor afhankelijk resultaat. Om toch een ILU methode voor deze compiler te maken, passen we het ILU algoritme toe op vlakken evenwijdig aan de distributie. We zullen dit Plane ILU solver noemen. Hierbij hebben we geen afhankelijkheid in de k richting meer. De vlakken kunnen nu parallel berekend worden. Op de vlakken zelf lossen we het stelsel op met behulp van DO loops. In Tabel 5.3 staan de totaaltijden weergegeven. Het aantal iteraties is nu constant, en speedup en efficientie van het Plane ILU algoritme staan in de Figuren 5.11 en Figuur 5.12. Voor de speedup en efficientie zien we zelfs een super-lineair gedrag. De totaaltijd om een oplossing te bepalen is groter geworden op 1 processor, vergelijk Tabel 5.2 met Tabel 5.3, maar Plane ILU is zeer efficient te parallelliseren. Een verklaring voor het super-lineaire gedrag is moeilijk te geven, maar het vermoeden is dat begintijden enigszin te groot zijn.

pe	tijd
1	509.5
2	261.5
3	172.2
4	119.5
5	100.2
6	75.7

Tabel 5.3: *Tijd ten opzichten van het aantal processoren voor Plane ILU, met 405 iteraties, voor de CoSy compiler op de Babylon*



Figuur 5.11: Speedup voor ontkoppelde Plane ILU met iteratief naverfijnen, met CoSy compiler op de Babylon



Figuur 5.12: Speedup voor ontkoppelde PLane ILU met iteratief naverfijnen, met CoSy compiler op de Babylon

5.4 Generalized Minimal Residual methode (GMRES)

De GMRES methode is een iteratieve methode om grote stelsels op te lossen, zoals (2.31), waarbij $A \in \mathbb{R}^{n \times n}$ niet-symetrisch is. Als A wel symetrisch is kan men beter de Geconjugeerde Gradienten (CG) methode gebruiken. Voor meer details over iteratieve methoden zie [11] en [13].

De GMRES methode start met een beginschatting x_0 van de oplossing en hiermee wordt het begin residu $r_0 = b - Ax_0$ berekend. In de m-de iteratie, is een correctie z_m bepaald in de Krylov deelruimte, zodat $x_m = x_0 + z_m$. De Krylov deelruimte is gedefinieerd als:

$$K_m(A; v_1) = span\{v_1, Av_1, ..., A^{m-1}v_1\}. \tag{5.14}$$

De Krylov deelruimte is opgebouwd met het Arnoldi proces, wat een orthonormale basis $\{v_1, ..., v_n\}$ construeert voor $K_m(A; v_1)$.

ARNOLDI METHODE:

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

In het Arnoldi proces, herkennen we de Gram-Schmidt (GS) methode voor het orthonormaliseren. Vanwege de afrondfouten met gebruik van GS, kunnen we beter de gemodificeerde GS (MGS) gebruiken voor het orthonormaliseren. Hiervoor worden de twee for-loops van de Arnoldi methode gecombineerd:

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

Als $h_{i+1,i} = 0$ dan stoppen we, de basis kan niet worden uitgebreid. De vectoren $v_1, ..., v_i$ spannen dan een invariante deelruimte op voor de Krylov deelruimte en een exacte oplossing is berekend.

We kunnen het Arnoldi proces samenvatten met:

Laat V_i een $n \times i$ matrix zijn met de vectoren $v_1, v_2, ..., v_i$ als kolommen. De Hessenberg matrix $H_i \in \mathbb{R}^{(i+1)\times i}$ bevat coefficienten $h_{j,i}$ van het Arnoldi proces:

$$H_i = \begin{pmatrix} h_{1,1} & h_{1,2} & ... & h_{1,i} \\ h_{2,1} & h_{2,2} & & \\ & h_{3,2} & \ddots & \vdots \\ & & \ddots & h_{i,i} \\ & & & h_{i+1,i} \end{pmatrix}. \tag{5.15}$$

Het Arnoldi proces is equivalent met:

$$AV_i = V_{i+1}H_i \;\; en \;\; V_{i+1}^T V_{i+1} = I. \tag{5.16}$$

De correctie z_m, $x_m = x_0 + z_m$ wordt als volgt bepaald. Neem z_m als lineaire combinatie van $v_1, ..., v_m$ dus $z_m = V_m y$ voor $y \in \mathbb{R}^m$, waarbij y de opossing is van het onderstaande kleinste kwadraten probleem :

$$\min_{z \in K_m(v_1)} \|b - A(x_0 + z)\|_2 = \min_{y \in \mathbb{R}^m} \|r_o - AV_m y\|_2 = \min_{y \in \mathbb{R}^m} \|\|r_0\|e_1 - H_m y\|_2. \tag{5.17}$$

Hierin is $e_1 = (1, 0, ..., 0)^T \in \mathbb{R}^m$. Omdat in iedere iteratie een nieuwe basisvector v_i berekend en opgeslagen wordt, nemen rekencomplexiteit en geheugenruimte toe met het aantal iteraties. Daarvoor herstarten we het GMRES-proces na een vast aantal iteraties, zeg m iteraties. GMRES met herstarten wordt genoteerd met GMRES(m), het totale algoritme ziet er als volgt uit,

GMRES(m) algoritme

x_0 gegeven :$r_0 = b - Ax_0$;$v_1 = \frac{r_0}{\|r_0\|}$
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
  $h_{i+1,i} = \|v_{i+1}\|_2$
  if $h_{i+1,i} = 0$ **then**
    neem $m = i$ goto los kleinste kwadraten probleem op
  else
    $v_{i+1} = v_{i+1}/h_{i+1,i}$
  end if
end for
kleinste kwadraten probleem
$\min_{y \in \mathbb{R}^m} \|\|r_0\|e_1 - H_m y\|_2$ bereken x_m

if *voldoet* **then**
stop
else
zet $x_0 = x_m$ en start opnieuw
end if

De GMRES solver wordt in de SIMPLE methode aangeroepen om een stelsel vergelijkingen op te lossen. Dit stelsel is niet lineair en is afhankelijk van de oplossing. De GMRES bepaalt een benaderende oplossing voor een constante matrix, het is niet de bedoeling om een exacte oplossing te bepalen tijdens een SIMPLE iteratie, maar een benadering te maken. We spreken ook wel van een buiten en een binnen iteratie, waarmee respectievelijk een SIMPLE iteratieslag en een GMRES iteratie wordt aangeduid.

Uit het testprobleem blijkt een extra herhaling van de GMRES(m) in de binneniteratie het convergentie gedrag te versnellen. Hiervoor voeren we een extra parameter h in, die het aantal herhalingen van de GMRES(m) aangeeft. Samenvattend hebben we de GMRES(h,m) methode, waar h voor het aantal herhalingen staat en m het aantal GMRES iteraties aangeeft (herstarten).

5.4.1 Controle van het tijdmodel voor de GMRES

In paragraaf (4.1) hebben we een tijdmodel afgeleid voor de bouwstenen matvec, inprodukt en update. De GMRES methode is opgebouwd met deze bouwstenen en met behulp hiervan proberen we een voorspelling van de tijd op p processoren te geven.
Defineer c als het aantal cycles, de cycles zijn het aantal keer dat de GMRES(h,m) wordt aangeroepen door de buiten iteratie totdat deze geconvergeerd is. Aan de hand van de resultaten van de GMRES op 1 processor, zoals T_1 en het aantal cycles c, kunnen we een voorspelling maken. In Tabel 5.4 zijn aantallen van de bouwstenen in de GMRES beschreven.

	aantal
matvec	(m+1).h.c
update	(1/2 m(m+1)+1).h.c
inprd	(1/2 m(m+1)+2).h.c

Tabel 5.4: *De aantallen voor de bouwstenen van c SIMPLE iteraties met GMRES(h,m) in de binnenloop*

Het afgeleide tijdmodel voor de bouwstenen is:
matvec : $T^m_{\text{tot}} = \frac{T^m_1}{p} + p * T^m_m$,
update : $T^u_{\text{tot}} = \frac{T^u_1}{p}$,
inw.prd: $T^i_{\text{tot}} = \frac{T^i_1}{p} + p * T^i_m$.

De parameters hiervan staan in volgende Tabel 5.5

	T_1	T_m
matvec	66.4	0.833
update	11.4	–
inprd	25.6	0.0328

Tabel 5.5: *De tijden van de bouwstenen (msec) voor array's van 36x36x36 met de CoSy-compiler*

Voor de GMRES(h,m) methode met c cycles, kan het volgende (totale) tijdsmodel worden opgezet:

$$T_{\rm tot} = aant_{matvec}T^m_{\rm tot} + aant_{upd}T^u_{\rm tot} + aant_{inpr}T^i_{\rm tot}, \tag{5.18}$$

$$\begin{aligned} T_{\rm tot} &= (m+1)h.c(\frac{T^m_1}{p} + pT^m_m) + (\frac{1}{2}m(m+1)+1)h.c(\frac{T^u_1}{p}) + \\ &+ (\frac{1}{2}m(m+1)+2)h.c(\frac{T^i_1}{p} + pT^i_m)). \end{aligned} \tag{5.19}$$

In Tabel 5.6 geven we met behulp van (5.18) een aantal voorspellingen van de doorlooptijd T_{tot} en de efficientie E van GMRES(h,m) met c cycles. Het aantal cycles c is afhankelijk van de keuze h en m in GMRES(h,m).

PE	GMRES(2,4) c=110		GMRES(2,8) c=32		GMRES(2,10) c=22	
	T_{tot}	E	T_{tot}	E	T_{tot}	E
1	167.6	100	126.8	100	123.6	100
2	85.3	98.2	64.2	98.7	62.5	98.8
4	45.7	91.8	33.8	93.8	32.7	94.4
6	33.8	82.7	24.4	86.6	23.4	87.9
12	25.9	53.9	17.2	61.3	16.1	64.0
24	31.0	22.5	18.7	28.3	16.8	30.7
32	37.3	14.0	21.9	18.1	19.4	20.0

Tabel 5.6: *Voorspelling tijd (sec) en efficientie van GMRES(h,m) voor aantal processoren met CoSy compiler (Babylon)*

Uit Tabel 5.6 kunnen we aflezen dat voor 6 processoren de GMRES(2,4), dus 2 maal herstarten van 4 gmres iteraties, een executie tijd $T_{tot} = 33.8\ sec$ heeft met een efficientie $E = 82.7$ % . Tabel 5.7 bevat de gemeten tijden van de GMRES. Dit is gebaseerd op het test-probleem met 36x36x36 gridpunten en met de CoSy-compiler gecompileerd. Wat opvalt in de Tabel 5.7 is dat bij 5 processoren, dat de GMRES(h,m) minder efficient wordt, terwijl bij 6 processoren de efficientie hoger ligt, hier komen we nog op terug.

PE	GMRES(2,4) c=110		GMRES(2,8) c=32		GMRES(2,10) c=22	
	T_{tot}	E	T_{tot}	E	T_{tot}	E
1	201.0	100	146.6	100	140.5	100
2	103.6	97.0	74.5	98.4	71.4	98.4
3	73.9	90.6	52.4	93.3	48.7	96.2
4	51.6	97.4	37.9	96.7	35.3	99.5
5	48.7	82.5	34.3	85.5	33.0	84.8
6	37.1	90.3	26.8	91.2	25.3	92.6

Tabel 5.7: *Gemeten tijden (sec) en efficientie van GMRES(h,m) voor 1 t/m 6 processoren met CoSy compiler (Babylon)*

Uit Tabel 5.6 kunnen we ongeveer de minimale wallclocktijd (T_{tot}) bepalen. We kunnen de volgende vraag stellen: kunnen we het optimaal aantal processoren op voorhand bepalen? Voor het optimaal aantal vectoren geldt $\frac{\partial T_{tot}}{\partial p} = 0$.

$$\rightarrow \ (m+1)h.c(\frac{-T_1^m}{p^2} + T_m^m) + (\frac{1}{2}m(m+1)+1)h.c(\frac{-T_1^u}{p^2}) + \\ + (\frac{1}{2}m(m+1)+2)h.c(\frac{-T_1^i}{p^2} + T_m^i)) = 0. \tag{5.20}$$

Als we alle onbekende parameters in (5.23) invullen, kunnen we hieruit de optimale p oplossen, zie Tabel 5.8.

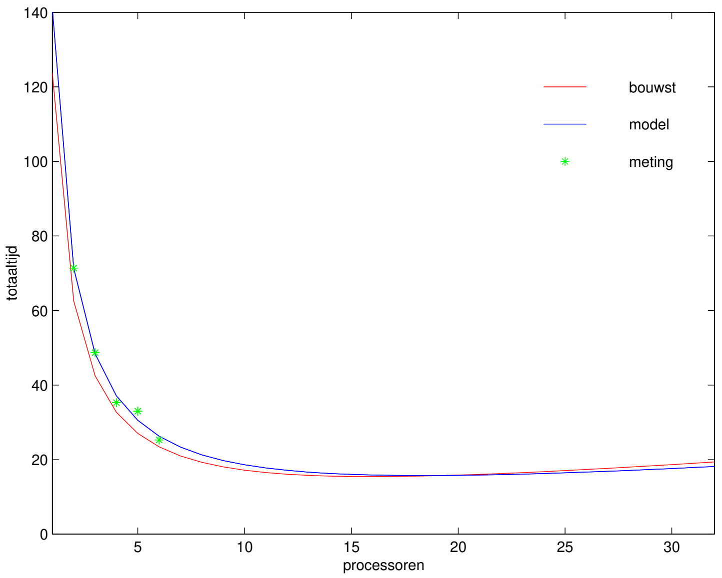
gmres(h,m)	aant. proc
(2,4)	13
(2,8)	15
(2,10)	16
(2,15)	18

Tabel 5.8: *Optimaal aantal processoren*

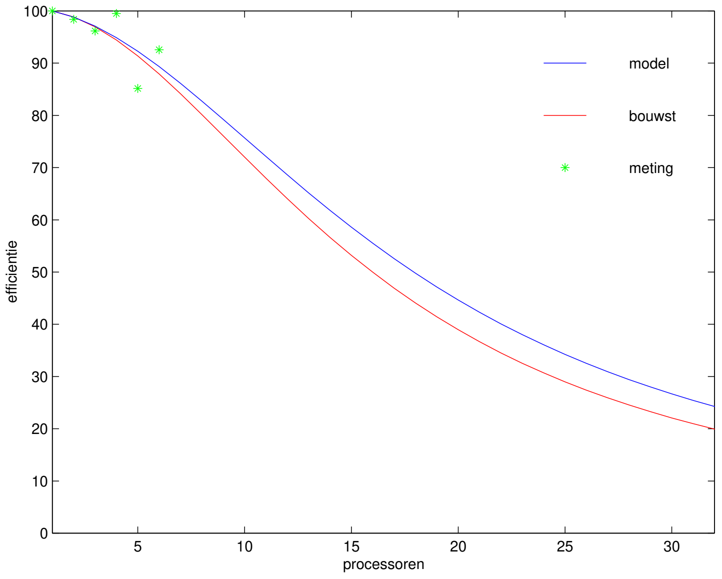
Voor de GMRES(2,10) hebben we (5.22) ook grafisch weergegeven in Figuur 5.13 . Hierin laten we ook model (5.24) zien, welke direct gefit is aan de 6 gemeten totaaltijden.

$$T_{tot} = \alpha + \frac{\beta}{p} + p * \gamma. \tag{5.21}$$

Met $\alpha = 0.422$, $\beta = 140.1$ en $\gamma = 0.211$ zodat T_{tot} de tijd in sec aangeeft. Dit levert een extra controle op en beide modellen zijn dus in overeenstemming met elkaar. Hetzelfde hebben we voor de efficientie gedaan in Figuur 5.14. Opmerkelijk is dat de efficientie grote sprongen maakt. We hadden dit voor 5 processoren ook al vastgesteld. Voor 4 processoren hebben we een te grote waarde voor de efficientie.



Figuur 5.13: Wallclocktijd (sec) voor de CoSy compiler op de Babylon



Figuur 5.14: Efficientie (%) voor de CoSy compiler op de Babylon

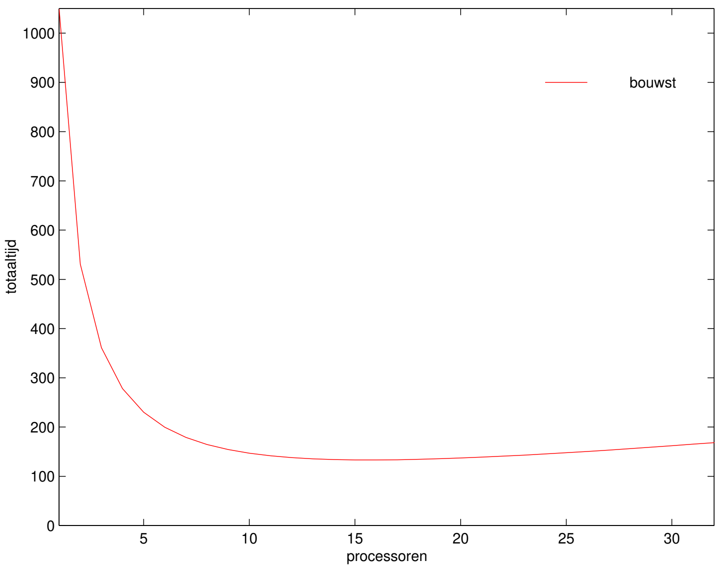
PGI compiler met GMRES

Het testprobleem is met de CoSy-compiler gecompileerd, nu herhalen we bovenstaande test voor GMRES(2,10) met de PGI-compiler. In Tabel 5.9 staan de tijden voor de bouwstenen voor de PGI-compiler en we gebruiken een model analoog aan (5.22).

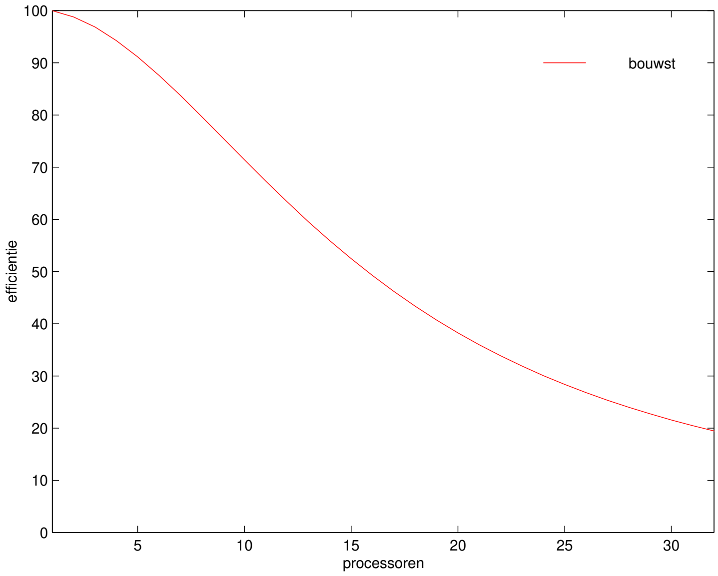
	T_1	T_m
matvec	247.8	8.06
update	88.2	–
inprd	282.2	0.136

Tabel 5.9: *Tijden voor de diverse bouwstenen*

Ondanks andere verhoudingen tussen matvec en inprodukt blijft het gedrag hetzelfde. Zie Figuur 5.15 voor de totaaltijd en de efficientie staat in Figuur 5.16. Wel zien we dat de tijden een stuk hoger liggen. Dit komt omdat we het testprogramma o.a. op een andere cluster van werkstations draaien.



Figuur 5.15: Wallclock-tijd (sec) voor GMRES(2,10) met de PGI compiler (cluster van sun werkstations)



Figuur 5.16: Efficiencie (%) voor GMRES(2,10) PGI compiler (cluster van sun werkstations)

Samenvattend voor GMRES

Uit bovenstaande blijkt dat de GMRES(h,m) efficienter wordt naarmate m, het aantal GMRES iteraties, toeneemt. De voorspellingen met het tijdmodel (5.22), zie Tabel 5.6, komen ongeveer overeen met de gemeten tijden uit Tabel 5.7. Hierbij dient de voorspelling als bovengrens voor de wallclock tijd te worden genomen, omdat de gemeten tijden beter uitvallen (voor de efficientie als ondergrens). Om het gedrag op de Cray T3E te voorspellen zullen we een nauwkeuriger tijdmodel moeten hanteren, zie ook opmerkingen in hoofdstuk 4. Wel kunnen we bovenstaand tijdmodel als grove ondergrens gebruiken op de Cray.

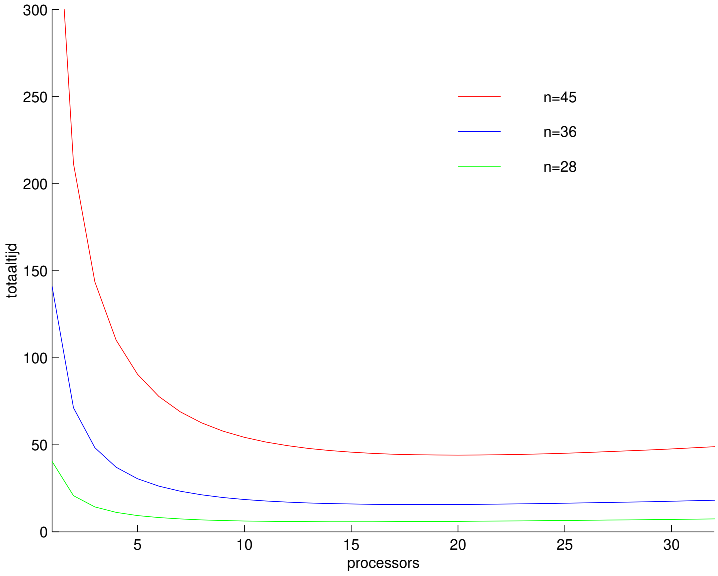
Als laatste test vergroten we de afmetingen van het testprobleem. Er zijn twee redenen hiervoor, ten eerste om te kijken of GMRES(h,m) schaalt en ten tweede voor de invloed van loadbalancing in het model.

Ad 1, GMRES schaalt als er een factor te vinden is tussen de toename in afmeting en in de toename van processoren. In Figuur 5.17 zien we dat het optimale aantal processoren meeschaalt met de afmeting. In Tabel 5.10 staan het optimaal aantal processoren.

n	≈ gridpunten	$p_{optimaal}$
12	1700	1
28	23000	15
36	46000	18
45	93000	20
48	110000	24

Tabel 5.10: *Afmeting n x n x n, met optimum*

Bij verdere toename komt het model in tegenspraak, omdat $T_{tot} = T_1$ op $p = 1$ te hoog is, krijgen we bij direct fitten een optimum van 12 processoren. Terwijl uit metingen de efficientie een superlineair gedrag vertoont, zie Tabel 5.11. We berekenen bij de hoogste efficientie de bij behorende T_1 in dit geval is $p = 5$, dus $5 * 365.9\ s = 1829.5\ s$. Als we nu deze waarden gebruiken ligt de efficientie op $p = 5$ nog steeds op 100% maar het optimum ligt verder als 12 processoren. We geven hier geen uitspraak over dit optimum omdat dit geen voorspellende waarde meer heeft. De gemeten tijd T_{tot} op $p = 1$ is groter dan verwacht. Dit komt omdat we tegen de fysieke geheugengrens aan lopen van het cluster van werkstations waarop we de meting uitvoeren. Hierdoor kan het programma minder snel worden uitgevoerd.



Figuur 5.17: GMRES(2,10) bij verschillende probleem groottes met de CoSy compiler (Babylon)

p	T_{tot}	E
1	2001.1	100
2	942.2	106.2
3	618.8	107.8
4	514.1	97.3
5	365.9	109.4
6	347.1	96.1

Tabel 5.11: *Efficientie en tijden bij n=60*

Ad 2, We krijgen geen goede loadbalancing als de afmetingen van het testprobleem niet deelbaar is door het aantal processoren. Voor het testprobleem geldt dat met behulp van de CoSy compiler maar in 1 richting verdeeld kan worden. De afmeting van de gedistribueerde richting is n en als het aantal gebruikte processoren p niet deelbaar is, zorgt de compiler ervoor dat een van de processoren een stukje meer krijgt dan de andere processoren. Met als gevolg dat het testprobleem minder efficient zal draaien, omdat andere processoren moeten wachten op de ene processor met meer werk voordat het programma verder kan. In Tabel 5.7 is dit te zien met $n = 36$ en $p = 5$ we zien dat het programma minder efficient werkt, terwijl bij $p = 1, 2, 3, 4, 6$ het beter gaat.

Een oplossing voor dit probleem zou kunnen zijn om de betreffende totaaltijd niet mee te nemen bij fitten in model (5.25). Dus voor $n = 36$ laten we de totaaltijd voor $p = 5$ weg, bij direct fitten aan de overgebleven 5 metingen krijgen we voor $\alpha = 1.53$, $\beta = 139.1$ en voor $\gamma = 0.039$. Er zit dus een factor 5 tussen de waarden in γ met wel of niet mee fitten van T_{tot} op $p = 5$. Het model geeft nu een veel efficienter gedrag, het optimum van het aantal processoren is ongeveer verdubbeld. De invloed van load-balancing kan men beperken door de afmetingen in de gedistribueerde richting te vergroten. Hoe dan ook blijft het model met alle metingen meegenomen nog steeds een bovengrens geven voor de wallclock-tijd en een ondergrens voor de efficientie.

Samenvattend voor tijdmodel voor GMRES

Het tijdmodel geeft een indicatie van het gedrag voor de GMRES methode op een (dedicated) cluster van werkstations. (op de Babylon met de CoSy compiler). Het model is gebaseerd op 6 metingen dus we kunnen geen precieze uitspraken doen over bijvoorbeeld 60 processoren, het geeft eerder een richting aan. We zagen dat effecten zoals, loadbalancing en schaling, problemen op leverden. Het model is zeker niet volledig. Men kan het model als boven/ondergrens voor tijdschattingen/ efficientie hanteren.

In de grafieken van de totaaltijden is te zien dat bij het optimaal aantal processoren de grafiek relatief vlak is. Als het aantal gebruikte processoren groter is dan het optimum aantal geeft dit gelukkig geen grote tijdstoename.

6 Parallelle Preconditionering

6.1 Inleiding

Preconditionering is nodig om de iteratieve Krylov methode te versnellen. De preconditionering transformeert het stelsel (2.31) zodanig dat het eenvoudiger is op te lossen voor de iteratieve methode. De preconditionering ziet er als volgt uit:

$$M^{-1}Ax \;=\; M^{-1}b, \tag{6.1}$$

met M de preconditioneringsmatrix, de oplossing x wordt niet beïnvloed door door de preconditionering.

Een iteratieve methode is equivalent met een preconditionering. We kunnen het iteratieve methode schrijven als een preconditionering. Een iteratieve methode is te schrijven in de volgende klassieke standaard vorm, zie [11],

$$x_{i+1} = x_i + M^{-1}(b - Ax_i), \tag{6.2}$$

waar M zo wordt gekozen dat (6.2) gemakkelijk te bereken is. Herschrijf (6.2) tot:

$$x_{i+1} = (I - M^{-1}A)x_i + M^{-1}b = Gx_i + f. \tag{6.3}$$

Men kan (6.3) opvatten als techniek om het volgend stelsel op te lossen:

$$(I - G)x = f. \tag{6.4}$$

Substitueer nu de definitie van G en f in (6.4) en er volgt:

$$M^{-1}Ax = M^{-1}b, \tag{6.5}$$

en we vinden het gepreconditioneerde stelsel terug. Hiermee hebben we de relatie tussen preconditionering en iteratieve methode laten zien.

Het blijkt handiger te zijn om rechts te preconditioneren in plaats van links zoals in (6.1), omdat het residu $r = b - Ax$ dan niet veranderd wordt. Bij links preconditioneren krijgen we in de GMRES:

$$\begin{aligned} M^{-1}x = M^{-1}b, \Rightarrow& \\ r =& M^{-1}(b - Ax). \end{aligned} \tag{6.6}$$

en bij rechts preconditioneren blijft het residu hetzelfde:

$$\begin{aligned} AM^{-1}Mx = b \Rightarrow& \\ r = (b - AM^{-1}Mx) = b - Ax.& \end{aligned} \tag{6.7}$$

Met de rechter preconditionering blijft het GMRES algoritme de fout in de zelfde ruimte berekenen en dit is dan in overeenstemming met foutmeting zoals bedoeld in het hoofdprogramma. We geven hieronder het parallelliseren van de preconditioneringen. Door de rechter preconditionering werkt deze op de vectoren $x = M^{-1}y$:

6.1.1 Diagonal preconditioning (Jacobi)

Bij Jacobi is $M = diag(A)$, A is gedistribueerd over het aantal processoren en de vector y is hiermee gealigned.
Dit betekent dat $x = M^{-1}y = \frac{1}{diag((a_p)_{i,j,k}})y \quad \forall i,j,k$ lokaal kan worden uitgevoerd en hiervoor is geen communicatie nodig.

6.1.2 Linx preconditionering

Bij de Linx solver preconditioneren we op een BLOCK Jacobi manier, dat wil zeggen $M = A_{\text{tridiagonaal}}$ stellen de lijnen evenwijdig aan i-richting voor.
De preconditioneringsmatrix lijkt nu meer op A en de verwachting is dat dit een beter resultaat geeft. Uit hoofdstuk 5 weten we dat de lijnsolver, bij juiste distributie, zeer goede parallelle eigenschappen heeft. De lijnen (blokken) zijn zo over de processors verdeeld zodat geen communicatie nodig is, tussen de lijnen (blokken). Wel is er meer rekenwerk in vergelijking met Jacobi preconditionering.

6.1.3 Block ILU preconditionering

We verdelen de matrix A in blokken, waar een blok een vlak voorstelt evenwijdig aan de distributie. Op de blokken wordt de ILU decompositie toegepast, $LD^{-1}U$ bestaat voor de 3D situatie uit 5 diagonalen voor de blokken (vlak), dit nemen als preconditionerings matrix $M = LD^{-1}U$. M lijkt nu nog meer op A met de verwachting dat dit de GMRES nog meer versneld.
De blokken (vlakken) zijn over het aantal processoren verdeeld, met het Plane ILU algoritme worden voor p vlakken tegelijk (p is aantal gebruikte processoren) de matrix D gemaakt, vervolgens kunnen we dan $x = (LD^{-1}U)^{-1}y$ voor p vlakken tegelijk bepalen. Totdat lokaal alle vlakken zijn behandeld. In Plane ILU wordt wel communicatie verwacht.

In het volgende hoofdstuk zullen we de preconditioneringen toepassen. Bij het testprobleem bleek dit niet zo zinvol. Het aantal iteraties nam wel af, echter de benodigde rekentijd bleef ongeveer gelijk.

7 Resultaten in WISH3D

7.1 Inleiding

Het was de bedoeling om een glasoven met WISH3D parallel door te rekenen, om zo het gedrag van de geïmplementeerde solvers te bekijken. Dit is helaas niet gelukt.
Er kon geen executable worden verkregen door WISH3D te compileren met de nog in ontwikkelde zijnde CoSy compiler. Hierdoor zijn we overgestapt op een andere (commerciële) HPF compiler van de Portland Group.

In eerste instantie hebben we WISH3D met het ovenprobleem (3D probleem) sequentieel gecompileerd met de PGI compiler. De verkregen executable gaf tijdens het runnen niet de te verwachten resultaten weer, met name voor de x-richting. Waarschijnlijk is er een bug tijdens het parallelliseren ingeslopen.

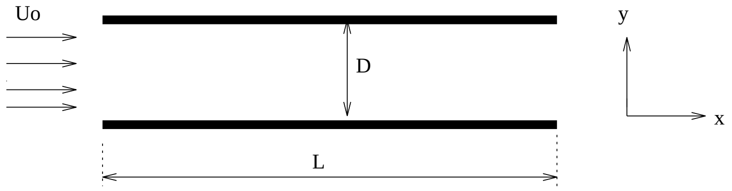
Dit gaf aanleiding om een vlakke plaatstroming te ontwikkelen (2D probleem), dit hebben we ook weer sequentieel gecompileerd. Nu klopten de resultaten met de te verwachten resultaten. Vervolgens hebben we dit probleem parallel met de PGI compiler gecompileerd. De verkregen executable crashte tijdens het uitvoeren.
We kregen hierbij sterk het idee dat dit wegens een bug in de compiler gebeurde. We hebben geprobeerd om deze bug te lokaliseren, dit werd bemoeilijkt door de gebruikte scope-regels in WISH3D. Bij debuggen van de parallelle code kan er naar één proces worden gekeken. Het probleem hierbij is dat de dataobjecten verdeeld zijn en de waarden die de debugger verschaft over een bepaalde variablele kunnen daarom onzin zijn. Hierover hebben we contact gehad met de Portland Group. Die raden ons aan hun debugger aan te schaffen om dit probleem op te lossen. Die weg zijn we niet ingeslagen.

Om toch een idee te krijgen van het gedrag in WISH3D van GMRES gebruiken we als test de stroming tussen twee vlakke platen (in het sequentiële geval). We kijken wat de invloed van de SIMPLER methode op de GMRES heeft (SIMPLER is een uitbreiding van SIMPLE, zie voor meer details in [5]). We kunnen het gedrag dan enigzins terugvertalen naar een parallelle situatie.

7.2 Stroming tussen 2 vlakke platen

7.2.1 Probleemomschrijving

Beschouw een stationaire luchtstroming in de x-richting tussen 2 vlakke platen, zie figuur 7.1. De platen hebben hierbij een lengte $L = 5\ m$ en de afstand tussen de platen is hierbij $D = 0.1\ m$. Andere materiële eigenschappen zijn:



Figuur 7.1: Stationaire luchtstroming in x-richting tussen 2 vlakke platen

dichtheid van lucht $\rho = 1.16\ kg/m^3$ en de dynamische viscositeit van lucht $\mu = 1.824\ 10^{-5}\ kg/ms$. De stroming tussen de platen zal laminair zijn.

Randvoorwaarde:

De randvoorwaarde zijn samengevat in Tabel 7.1:

variabele	instroomrand	uitstroomrand	wand
u	$u = u_0 = 0.01$	$\frac{\partial u}{\partial x} = 0$	$u = 0$
v	$v = 0$	$v = 0$	$v = 0$
p	$\frac{\partial p}{\partial x} = 0$	$p = 0$	$\frac{\partial p}{\partial y} = 0$

Tabel 7.1: *Randvoorwaarde*

7.2.2 Numerieke oplossing

We berekenen een oplossing aan de hand van de numerieke grootheden zoals gegeven in Tabel 7.2 voor een stroming tussen 2 vlakke platen.

Er zullen verschillende solvers voor dit probleem gebruikt worden om een

rekenrooster	(25×10)
Discretisatie methode	Eindige volume metode + Hybride schema
Oplosmethode	SIMPLER, zie [5]

Tabel 7.2: *Numerieke grootheden*

oplossing te bepalen tot $O(10^{-9})$ nauwkeurig. Wij zijn met name geïnteresseerd in de tijd waarin de solver een oplossing bepaald. Hiervoor meten we de tijd over het gehele SIMPLE(R) proces. De totaaltijd van WISH3D bestaat dan ook uit deze solvertijd en resterende tijd voor initialisatie en dergelijke. De resterende tijd blijkt toch schaalbaar bij parallellisatie, dus deze tijd laten we in onze beschouwing achterwegen.

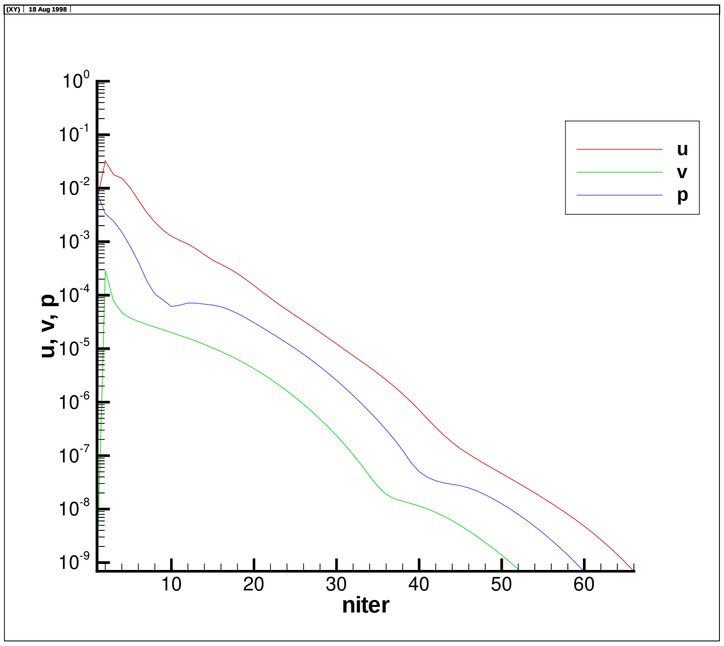
TDMA solvers

We zullen het probleem eerst met de TDMA solvers oplossen, in Tabel 7.3 staan de gebruikte testen met betreffende solver en de gebruikte relaxatieparameter. Verder staan het aantal benodigde iteraties en de solvertijd om de oplossing te bepalen.

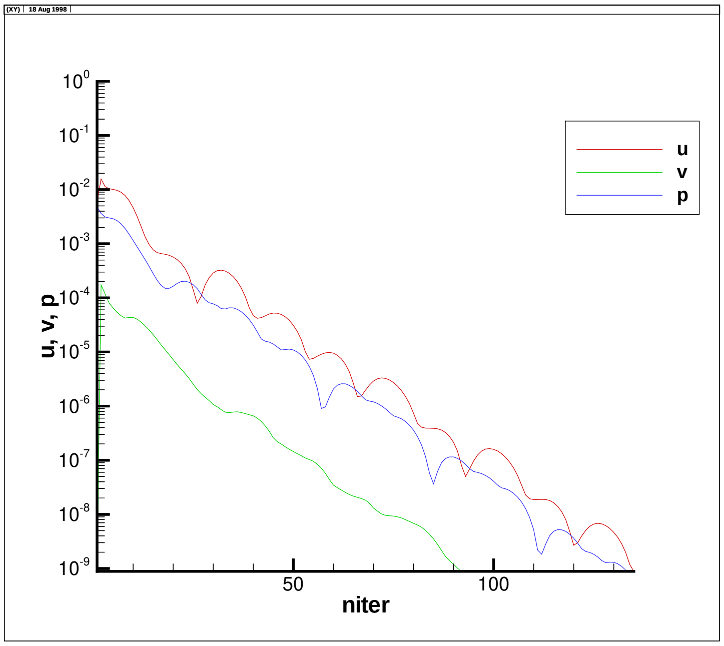
Test	solver	niter	tijd	relax u en v
1	linxyx	66	8.16	0.8
2	plane	135	12.12	0.8

Tabel 7.3: *SIMPLER met TDMA solvers voor vlakke plaatstroming*

In test1 gebruiken we de Line TDMA solver met sweeps in x en y richting. In [1] wordt opgemerkt dat de linxyz solver een exacte oplossing voor u en v wordt bepaald. Dit resulteert dan ook in een zeer snelle solvertijd. In test 2 gebruiken we de Plane TDMA solver. In de SIMPLER methode worden voor u en v nog relaxaties toegepast, uit testen blijkt dat met 0.8 een goede convergentie gehaald kan worden voor dit probleem. In Figuur 7.2 en Figuur 7.3 laten we van beide testen het bijbehorende convergentie gedrag zien.



Figuur 7.2: Test1, convergentiegedrag van linxyz TDMA solver met PGI compiler sequentieel gecompileerd op SUN werkstations



Figuur 7.3: Test2, convergentiegedrag van Plane TDMA solver met PGI compiler sequentieel gecompileerd op SUN werkstations

GMRES solvers

Het is bekend uit de conclusies in [1] voor het sequentieel doorrekenen dat de Krylov solver, in dit geval GMRES, meer tijd nodig heeft dan de TDMA solver om de oplossing te bepalen in WISH3D. De GMRES blijkt een factor 2 à 3 langzamer te zijn. De GMRES is een iteratieve methode met lange recurrenties, zie voor meer detail in [11]. Dit houdt in dat veel werk nodig is om een oplossing te bepalen. Met behulp van preconditioneren proberen we het aantal iteraties te beperken. Wat resulteert in een snellere oplostijd.

Uit [1] volgt ook dat het aantal iteraties van een Krylov methode niet veel uitmaakt voor de convergentie binnen het SIMPLER proces. Dit geeft het volgende idee: we voeren een maximum en een minimum in voor het aantal GMRES iteraties. Hiermee beperken we het aantal GMRES iteraties, maar er is wel een minimum vereist om te convergeren. Door de GMRES nu een relatieve reductie op het residu te laten bepalen hopen we iteraties uit te sparen. De GMRES solver bepaalt hierbij zelf per variabele u, v en p het aantal GMRES iteraties begrensd tussen het minimum en het maximum.

Het herstarten laten we wegens bovenstaande opmerkingen weg, $h = 1$. Dit alles geeft een flexibel te hanteren GMRES solver op, waarbij de gebruiker minder aan parameters hoeft te tunen. Wel blijft de relaxatieparameter bestaan.

We gaan nu de flexibele GMRES variant toepassen. In Tabel 7.4 gebruiken we de GMRES met een mimimun van $min = 4$ en een maximum van $max = 10$ GMRES iteraties. We laten de GMRES per SIMPLE iteratiestap een reductie van 10^{-2} op het residu bepalen. Er zijn verschillende preconditioneringen toegepast voor het oplossen van de stroming tussen de 2 vlakke platen.

Test	precon	min	max	niter	tijd	relax u,v	gem
3	Jacobi	4	10	297	71.74	0.8	4 8 10
4	Linx	4	10	415	101.7	0.8	4 7 10
5	Plane ILU	4	10	97	32.5	0.8	4 4 10
6	Liny	4	10	91	22.28	0.8	4 5 10

Tabel 7.4: *De GMRES met minimaal 4 en maximaal 10 GMRES iteraties, voor verschillende preconditionering voor de vlakke plaatstroming*

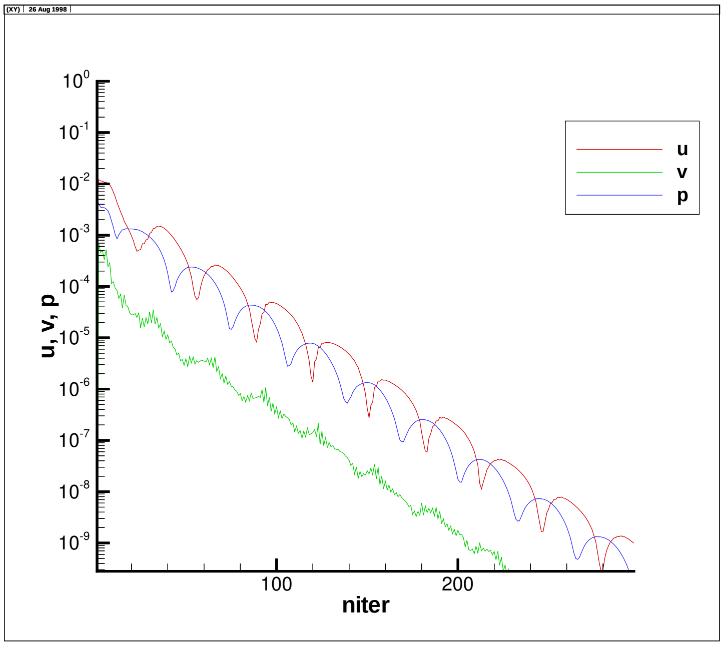
In Tabel 7.4 staan het aantal SIMPLER iteraties en de oplosttijd, waarbij in het SIMPLER proces nog gerelaxeerd wordt met 0.8. In de laatste kolom staan het gemiddeld aantal GMRES iteraties voor respectievelijk u, v en p, die nodig zijn geweest bij het oplossen. In figuur 7.4 is het convergentie gedrag geplot van GMRES versnelt met Jacobi, de andere testen geven ongeveer een gelijk convergentiegedrag weer. De druk heeft het maximaal aantal ingestelde GMRES iteraties nodig. Uit [1] weten we ook dat de druk p de convergentie domineert. Deze solvertijden van Tabel 7.4 liggen op zijn minst een factor 3 hoger ten opzichten van resultaten uit Tabel 7.3.

De gemiddelde GMRES iteraties van u, v en p in de laatste kolom van Tabel 7.4 zeggen iets over de convergentie van het probleem. Het blijkt dat als we het maximum ongeveer gelijk maken aan het gemiddelde van v (in dit geval), we GMRES iteraties kunnen besparen. Dit hebben we voor iedere GMRES combinatie gedaan en hierbij staan de snelste combinaties in Tabel 7.5 (met hoogst mogelijke relaxatiefactor voor deze gevallen). We zien dan we nu nog een factor 2 langzamer zijn dan linxyx TDMA, zie Tabel 7.3.

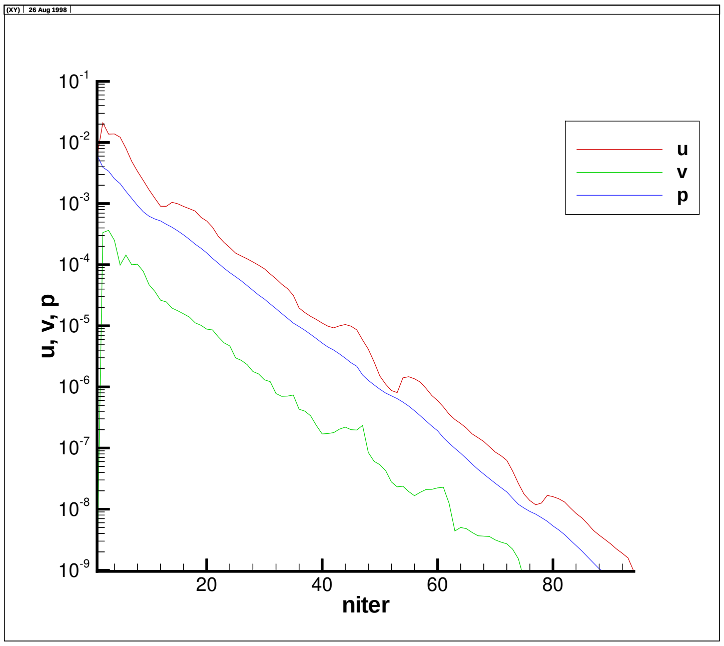
Test	precon	min	max	niter	tijd	rel u,v	gem
7	Plane ILU	2	3	87	15	0.85	2 3 3
8	Liny	2	3	95	14.5	0.85	2 3 3

Tabel 7.5: *SIMPLER met GMRES(1,3) met minimum 2 GMRES iteratie, voor verschillende preconditionering voor de vlakke plaatstroming*

Dus deze beperking voor het aantal GMRES iteraties levert een goede solvertijd. Vergelijk Figuur 7.4 en Figuur 7.5 qua convergentiegedrag. De te verwachten toename in het aantal SIMPLER iteraties blijft hierbij achterwegen.



Figuur 7.4: Test 3, convergentiegedrag van GMRES met Jacobi preconditionering, met PGI sequentieel gecompileerd op SUN werkstations



Figuur 7.5: Test 7, convergentiegedrag van GMRES met Plane ILU preconditionering, met PGI sequentieel gecompileerd op SUN werkstations

De druk speelt bij het beperken van de GMRES iteraties een grote rol. Als het maximum aantal GMRES iteraties voor de druk te klein is, wordt het totale convergentieproces vertraagd. Dit is probleem afhankelijk.

Voor de TDMA solver is 0.8 min of meer de grootst mogelijke relaxatie factor. Voor GMRES hadden we in Tabel 7.5 ongeveer de grootst mogelijke relaxatiefactoren voor de flexibele GMRES variant met Plane ILU en Liny als preconditionering. In Tabel 7.6 staat GMRES met Jacobi preconditonering. We kunnen deze bijna met 1 relaxeren.

min	max	niter	tijd	relax u, v
4	10	721	158.2	0.4
4	10	297	71.7	0.8
4	10	382	87.4	0.7
4	10	195	49.7	0.95
4	10	196	49.4	0.98

Tabel 7.6: *Verschillende relaxatiefactoren van GMRES met Jacobi preconditionering waarbij min 4 en max 10 GMRES iteraties*

Dus bij de GMRES solver moeten we opnieuw op zoek naar de optimale relaxatiefactoren.

7.3 Conclusies

De TDMA solver zijn sequentieel ongeveer twee keer zo snel dan GMRES solver. Bij parallellisatie zullen de TDMA solvers twee keer zo langzaam convergeren, omdat Gauss Seidel structuur vervangen wordt door Jacobi. Hierdoor zal het convergentiegedrag in het parallelle geval van de TDMA solvers en GMRES ongeveer gelijk zijn. In [1] waren er problemen met de druk, het stelsel voor de druk bleek singulier. Hiervoor werd een referentie punt voor de druk ingevoerd, waardoor de CG methode weer convergeerde. Dit effect hebben we bij de GMRES solver niet gezien. Dit valt misschien te verklaren omdat we een eenvoudiger probleem hebben beschouwd.

8 Conclusies en aanbevelingen

8.1 Conclusies

In het afstudeerwerk is onderzocht of de wallclock-tijd gereduceerd kan worden door parallellisatie van de lineaire solvers in WISH3D met behulp van High Performance Fortran. Behandelde onderwerpen met bijbehorende conclusies zijn:

1. Het beschrijven van de HPF syntax, conclusies:
   - eenvoudig om parallelle programma's mee te ontwikkelen, waarbij de compiler de communicatie en distributie opzich neemt.
   - schaalbaar van parallelle code
2. Het beschrijven en parallelliseren van de lineaire solvers, met conclusies:
   - Line TDMA solvers, zijn te parallelliseren, hierbij moet men BLOCK Gauss Seidel iteratie opgeven voor een BLOCK Jacobi iteratie. Hierdoor kan in plaats van 1 lijn, alle lijnen parallel worden doorgerekend, bij een distributie in 1 richting. Er wordt een zeer efficiente solver verkregen als de sweep-richting van de Line TDMA loodrecht staat op de distributie richting.
   - De ILU solver (SPACE TDMA) is moeilijk te parallelliseren, door 1 afhankelijkheid op te geven bij een distributie in één richting gaat de ILU solver over in een Plane ILU, deze is sterk verwant aan de Plane TDMA solver, die wel goed te parallelliseren is.
   - De GMRES en in het algemeen de Krylov solvers, laten zich goed parallelliseren onder HPF. Met het afgeleide tijdmodel kan het gedrag van de GMRES worden geschat. De GMRES methode blijft hetzelde convergentiegedrag houden.
3. Preconditionering is nodig bij de GMRES, vooral bij een 2D vlakke plaatstroming kwam dit duidelijk naar voren. De volgende precondi-tioneringen zijn hiervoor gebruikt: Jacobi, Line TDMA en Plane ILU. Het is jammer dat geen 3D probleem doorgerekend is, Het is moeilijk om te zeggen hoe GMRES in die situaties convergeert.
4. De HPF compilers zijn duidelijk nog in een ontwikkelingsfase, dit kwam naar voren naarmate het onderzoek vorderde. WISH3D kan daarom het beste met een message passing taal geparallelliseerd worden. HPF is vrij strikt en men is erg gebonden aan collectieve array operaties, terwijl een message passing taal flexibeler is.

5. Het is moeilijk om de beste parallelle solver aan te wijzen. De vlakke plaatstroming in het sequentiële geval laat zien dat de TDMA solvers het snelst zijn. De GMRES methode is ongeveer een factor 2 langzamer. Bij parallellisatie met HPF van de TDMA solvers leveren we de Gauss Seidel structuur in voor Jacobi, waarmee de convergentie snelheid gehalveerd wordt. Hierdoor komen de TDMA solvers en GMRES in het parallelle geval dicht bij elkaar qua convergentiesnelheid (voor de vlakke plaatstroming).

8.2 Aanbevelingen

Tijdens het onderzoek zijn een aantal zaken naar voren gekomen die in de toekomst bij eventuele parallellisatie gebruikt kunnen worden.

- Stap over op een message passing taal zoals bv. MPI. Probeer bij parallellisatie van TDMA solver de Gauss Seidel structuur vast te houden eventueel in ontkoppelde vorm.
- Update de code van WISH3D, maak daarbij gebruik van zoveel mogelijk standaard (bibliotheek) functies. Deze functies zijn vaak zeer efficient geïmplementeerd.
- Door het werk van A. Saghir te parallelliseren, dwz Het SIMPLE proces wordt herschreven als een distributieve iteratieve methode deze kan bij toepassing van de Krylov iteratieve methoden, voor het inprodukt en matvec geparallelliseerd worden. Het oplossen van de lineaire stelsels kan hierbij met geparallelliseerde TDMA solvers. Men heeft dan ook het buiten iteratieproces geparallelliseerd.

APPENDIX

3D Testprobleem

Het volgende diffussie probleem,

$$\frac{\partial}{\partial x}\phi\frac{\partial \phi}{\partial x} + \frac{\partial}{\partial y}\phi\frac{\partial \phi}{\partial y} + \frac{\partial}{\partial z}\phi\frac{\partial \phi}{\partial z} = S,$$

met grootheid ϕ en diffusie werkend in een volume $V = [0,1] \times [0,1] \times [0,1]$.

In het testprogramma beschouwen we de volgende oplossing: $\phi = x + y + z$, hiermee is de bronterm bepaald $\quad \Rightarrow \quad S = 3$.
De discretisatie wordt op Patankar methode van de volgende vorm,

$$a_P\phi_P = \sum_{nb} a_{nb}\phi_{nb} + b_P.$$

Waarin nb de buurgridpunten E, W, N, S, T, B van punt P zijn, en a_{nb} zijn gedefinieerd als:

$$\begin{aligned}
a_E &= \frac{\frac{1}{2}(\phi_E + \phi_P)}{\delta_E} \\
a_W &= \frac{\frac{1}{2}(\phi_W + \phi_P)}{\delta_W} \\
a_N &= \frac{\frac{1}{2}(\phi_N + \phi_P)}{\delta_N} \\
a_S &= \frac{\frac{1}{2}(\phi_S + \phi_P)}{\delta_S} \\
a_T &= \frac{\frac{1}{2}(\phi_T + \phi_P)}{\delta_T} \\
a_B &= \frac{\frac{1}{2}(\phi_B + \phi_P)}{\delta_B} \\
b_P &= -\Delta V_P S.
\end{aligned}$$

A Referenties

[1] H. Tazammourti, *(Parallelle) Solvers voor CFD-code WISH3D*, TNO-rapport FSP-RPT-970041 ,1997.
http://ta.twi.tudelft.nl/users/vuik/numanal/tazammourti.html

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

[3] A. Saghir, *Versnellen van SIMPLE(R) voor WISH3D met Krylov methoden*, TNO-rapport FSP-RPT-980039, 1998.
http://ta.twi.tudelft.nl/users/vuik/numanal/saghir.html

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

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

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

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

[8] G. Strang, *Linear Algebra and its Applications*, HBJ Publishers, Massachusetts, 1986.

[9] S.M.C.M. van Oudenaarde, *Parallelization of the linear solvers used in WISH3D*, TNO-rapport FSP-RPT-980017, 1998.
http://ta.twi.tudelft.nl/users/vuik/numanal/oudenaarde.html

[10] H.A. van der Vorst, P. Wesseling, P. Wilders, *Numerieke methode voor Differentiaalvergelijkingen*, Dictaat, TU Delft, 1995.

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

[12] G.R. Lueeke, J.J. Coyle, *High Performance Fortran versus explicit message passing on the IBM SP-2 for the parallel LU, QR, and Cholesky factorizations*, Supercomputing, 68 XIII-2, 1996.

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