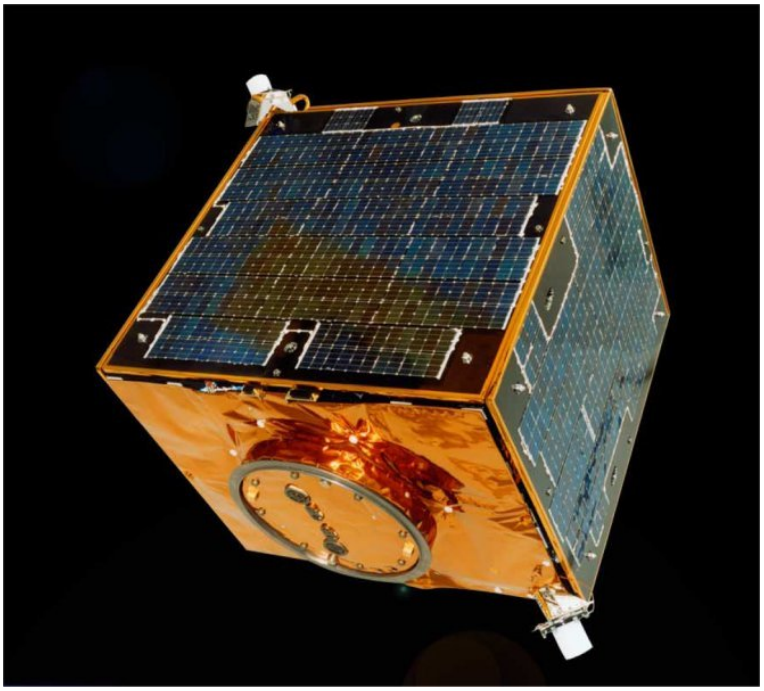
rijksuniversiteit groningen
faculteit wiskunde en natuurwetenschappen

Parallellisatie van COMFLO met OpenMP

Matthew van der Zwan



Bachelorscriptie Technische Wiskunde / Informatica

Mei 2009

Parallellisatie van COMFLO met OpenMP

Samenvatting

De hedendaagse computers krijgen steeds meer processoren en kernen per processor. Door een programma zodanig aan te passen dat het hier gebruik van maakt, kan het aanzienlijk versneld worden. Dit proces heet parallellisatie.

Deze scriptie gaat over het parallelliseren van het programma ComFlo. Comflo is een pakket voor het simuleren van vrije oppervlakte stroming, ontwikkeld en onderhouden aan de afdeling wiskunde van de Rijksuniversiteit Groningen.

Voor het parallelliseren van ComFlo is gebruik gemaakt van OpenMP, dit is een Application Programming Interface die speciaal ontwikkeld is om eenvoudig programma's te kunnen parallelliseren. De resultaten die hiermee zijn behaald, worden vergeleken met de resultaten van het Master Onderzoek van Jan Feitsma, die ComFlo heeft geparallelliseerd met behulp van MPI.

Het parallelliseren van ComFlo met OpenMP is gelukt. Het resultaat is minder snel dan de met MPI geparallelliseerde versie, maar parallelliseren met OpenMP is vele malen eenvoudiger. Vergeleken met de investering die gedaan moet worden is de parallellisatie met OpenMP zeer geslaagd.

Bachelorscriptie Technische Wiskunde / Informatica
Auteur: Matthew van der Zwan
Begeleiders: R. Luppes, A.E.P. Veldman en A. Meijster
Datum: Mei 2009

Instituut voor Wiskunde en Informatica
Postbus 407
9700 AK Groningen

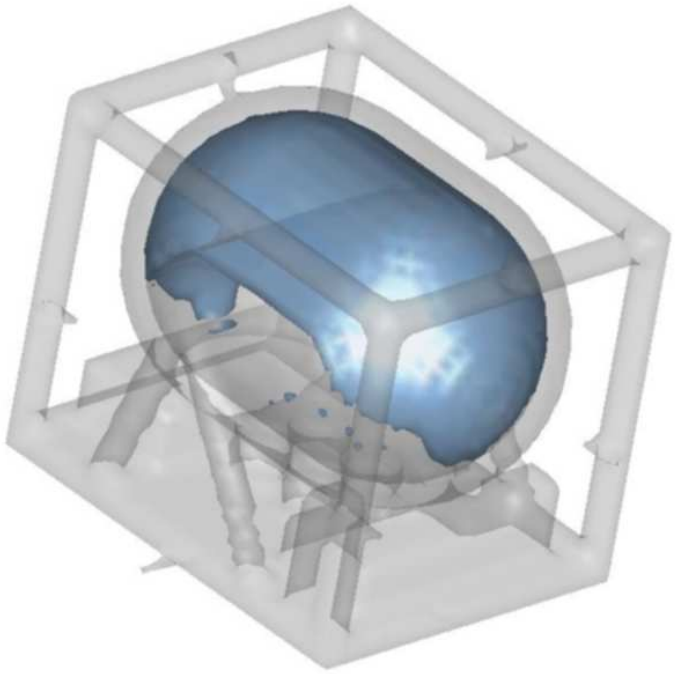
Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Wat is ComFlo

ComFlo is een verzameling programma's voor het simuleren van stroming met vrij oppervlak in zowel aardse als micro-zwaartekracht omgevingen. Het wordt gebruikt door een aantal grote onderzoekscentra in en buiten Nederland. ComFlo wordt ontwikkeld en onderhouden aan de afdeling wiskunde van de Rijksuniversiteit Groningen.



Figuur 1.1: Voorbeeld van de verwerkte uitvoer van een ComFlo simulatie

1.2 Parallellisatie

Het draaien van een ComFlo simulatie kost veel tijd, daarom wordt onderzocht of de simulaties versneld kunnen worden door middel van parallellisatie. Dit is het aanpassen van het programma zodat het werkt op meerdere processoren tegelijk. Doordat deze processoren allemaal een deel van het rekenwerk krijgen zal dit resulteren in snelheidswinst.

1.3 Doel van het onderzoek

Er zijn verschillende manieren om een programma te parallelliseren. In dit onderzoek is gekeken naar parallellisatie van ComFlo met behulp van OpenMP. Het uiteindelijke streven is het programma zo aan te passen dat het minstens 4 keer sneller werkt wanneer we het laten werken op 8 processoren, dan het nu doet op 1 processor. Eerder heeft Jan Feitsma al onderzocht hoe ComFlo geparallelliseerd kan worden met MPI (Feitsma [3]), hieruit bleek dat dit resultaat gehaald kan worden met MPI. Verwacht wordt dat dit met OpenMP ook mogelijk moet zijn, maar dan met minder inspanning.

1.4 Indeling verslag

Hoofdstuk 2 beschrijft de structuur van het ComFlo programma en onderbouwt de keuze voor de te parallelliseren procedure. Verder wordt uitgelegd wat er moet gebeuren om deze procedure te kunnen parallelliseren. Hoofdstuk 3 geeft een aantal mogelijkheden om een programma te parallelliseren, zoals OpenMP en MPI. In hoofdstuk 4 wordt uitleg gegeven over het gebruik van OpenMP en wordt de voor dit onderzoek gebruikte aanpak belicht. Hoofdstuk 5 en 6 geven de resultaten van dit onderzoek, waarna deze in hoofdstuk 7 worden besproken.

Hoofdstuk 2

ComFlo

ComFlo is geschreven in Fortran, het ComFlo programma waar in dit onderzoek mee is gewerkt, is SloshDP, wat geschreven is in Fortran 77. Uit eerder onderzoek is gebleken dat in dit programma vooral de procedure `slag` veel rekentijd verbruikt (Feitsma [3]), zie ook tabel 2. Deze procedure maakt deel uit van de Poisson solver en voert een SOR iteratie voor het oplossen van de Poisson vergelijking uit.

Subroutine	60x40x40	120x80x80
`slag`	57.0%	86.8%
`fluidforce`	8.0%	3.1%
`tilde`	6.7%	2.4%
`velbc`	3.9%	1.1%
`vfhn`	2.7%	0.7%

Tabel 2.1: De routines in ComFlo die de meeste tijd gebruiken voor een simulatie van 1 seconde van een flat-spin manoevre, overgenomen uit Feitsma [3]

Omdat de te parallelliseren code in ComFlo zich bevindt in de Poisson solver, is er voor gekozen eerst een alleenstaande Poisson solver te maken en deze te parallelliseren. Dit neemt eventuele invloeden van de rest van de ComFlo code op de parallellisatie van deze code weg, zodat er zo zuiver mogelijke meetresultaten ontstaan. De methode die hier als beste uit naar voren komt, wordt later toegepast in de ComFlo code.

Voor het oplossen van de Poisson vergelijking wordt in de Poisson solver gebruik gemaakt van de SOR methode. Voor deze methode hangt de waarde in een roosterpunt niet alleen af van de waarde op het vorige tijdstip, zoals bij Jacobi, maar ook van de nieuwe waarde in de buurcellen. Deze afhankelijkheid zorgt voor de uitdaging bij het parallelliseren van deze methode, omdat er, indien we dit zouden negeren, geheel verkeerde antwoorden zouden ontstaan. In feite is het algoritme dan geen goede SOR meer.

Om deze afhankelijkheden weg te nemen wordt de zogenaamde red/black-methode gebruikt. De roosterpunten worden zodanig opgedeelt dat elk punt in één van deze verzamelingen alleen afhankelijk is van punten in andere verzamelingen. Door het verdwijnen van afhankelijkheden binnen de verzamelingen is het mogelijk om de iteraties door deze verzamelingen te parallelliseren. In 2 dimensies geeft dit 2 verzamelingen die er uit zien als een schaakbord-patroon. Hierbij grenst elk zwart vlak alleen aan rode vlakken en omgekeerd. Wanneer de red/black-methode wordt toegepast op een 3 dimensionaal rooster krijgen we 8

lussen die elk parallelliseerbaar zijn. Dit is te zien in de listing van de `slag` code in appendix B.

Hoofdstuk 3

Parallellisatie

Zoals in de inleiding vermeld is parallellisatie het aanpassen van een programma, zodanig dat het programma gaat werken op meerdere processoren. Parallellisatie van een programma is op een hoop manieren te bewerkstelligen. Dit hoofdstuk beschrijft de parallellisatie methoden die gebruikt worden in dit onderzoek.

3.1 OpenMP

OpenMP is een API[1] die het mogelijk maakt door toevoeging van zogenaamde compiler directives een programma te parallelliseren[2]. Het doet dit door gebruik te maken van speciale OpenMP libraries die aan het programma gekoppeld worden. OpenMP is ontworpen om te werken op shared memory machines, hoewel het tegenwoordig ook te gebruiken is op clusters door middel van ClusterOpenMP.

3.2 Automatische parallellisatie

Een aantal compilers, waaronder de bij dit onderzoek gebruikte IBM compiler, kan een programma automatisch parallelliseren. De verwachting is dat deze methode wel resultaat zal hebben, dat wil zeggen, dat versnelling behaald zal worden, maar dat met behulp van OpenMP een beter resultaat behaald kan worden. Dit omdat het met OpenMP beter mogelijk is om de compiler bij te sturen, wat naar verwachting nodig is omdat het probleem zeker niet triviaal is.

3.3 MPI

MPI is net als OpenMP een API voor het parallelliseren van programma's. In tegenstelling tot openMP gebeurt dit echter niet aan de hand van compiler statements, maar door het zelf uitschrijven van de communicatie tussen de verschillende threads (of machines). Het parallelliseren van een programma met behulp van MPI kost dan ook veel meer tijd dan met eerder genoemde methodes. Een voordeel van MPI is dat het ondersteund wordt door een hoop platforms, waaronder ook de BlueGene die in Groningen staat. Maar een met MPI geparallelliseerd programma draait ook op shared memory machines.

[1] Application Programming Interface

Zoals in de inleiding gemeld gaat het masteronderzoek van Jan Feitsma (Feitsma [3]) over het parallelliseren van de Poisson solver in ComFlo met behulp van MPI.

3.4 Metingen

Bij het parallelliseren van het een programma is het de bedoeling dat het programma sneller wordt omdat er meer rekenkracht beschikbaar komt. De meest gebruikte grootheid om te bepalen of een programma ook daadwerkelijk sneller wordt is de speedup. Deze speedup is de factor die het programma sneller is geworden voor n processoren ten opzichte van het programma op 1 processor. Of in formulevorm:

$$\text{speedup(n)} = \frac{t_1}{t_n} \tag{3.1}$$

Omdat een programma vaak aangepast wordt om het goed te kunnen parallelliseren zijn er twee keuzes te maken voor de tijd op 1 processor. Zo is het mogelijk om de tijd van het seriële programma hier te gebruiken, maar vaak wordt ook het parallelle programma, uitgevoerd op 1 processor, gebruikt. Door de eerder genoemde aanpassing aan het programma, is de parallelle versie die draait op 1 processor vaak langzamer dan de originele versie. Daarom zullen beide methodes een andere speedup opleveren. In dit verslag wordt de tijd van het geparallelliseerde programma op 1 processor gebruikt als basis.

Hoofdstuk 4

OpenMP

4.1 Voorbeeld

Als voorbeeld voor het parallelliseren met OpenMP kijken we naar het volgende programma, dat de som van de elementen in een array berekent.

```
integer, dimension(maxI) :: a
integer sum

sum = 0

do i = 1, maxI
  sum = sum + a(i)
end do
```

Met OpenMP kan de lus als volgt geparallelliseerd worden

```
!$omp parallel
!$omp do reduction(+:sum)
do i = 1, maxI
  sum = sum + a(i)
end do
!$omp end do
!$omp end parallel
```

De regel `!$omp parallel` begint een nieuwe parallelle regio in de code, alles wat binnen deze regio valt wordt parallel uitgevoerd. Door hierbinnen de regel `!$omp do` op te nemen wordt de do lus parallel uitgevoerd. Daarbij wordt met het statement `reduction(`:sum)+ aangegeven dat alle processen zelf een sum mogen bijhouden, maar dat deze aan het eind bij elkaar opgeteld dienen te worden. Aan het einde van de lus moet aangegeven worden dat deze geëindigd is en ook de parallelle regio moet afgesloten worden. Dit gebeurd met de statements `!$omp end do` en `!$omp end parallel`.

In plaats van het reduction statement te gebruiken voor het optellen van de waarden van sum van de individuele processen kan dit ook handmatig gedaan worden. Dit heeft voordelen als de som bijvoorbeeld uit meerdere lussen bestaat, zoals in het geval van het bepalen van de norm in de routine SLAG van ComFlo (zie hoofdstuk 2). Het volgende voorbeeld geeft aan hoe dit mogelijk is.

```
integer, dimension(maxI) :: a
integer sum, mySum

sum = 0
```

```
mySum = 0

!$omp parallel firstprivate(mySum)
!$omp do
do i = 1, maxI
  mySum = mySum + a(i)
end do
!$omp end do

!$omp critical
  sum = sum + mySum
!$omp end critical
!$omp end parallel
```

In deze versie is er een nieuwe variabele `mySum` geïntroduceerd, waarin elk proces de som van zijn gedeelte van de taak bij kan houden. Met behulp van het statement `firstprivate(mySum)` wordt ervoor gezorgd dat elk proces zijn eigen versie van de variabele `mySum` krijgt. Als beginwaarde krijgen al deze nieuwe `mySum` variabelen de waarde die er eerder aan toegekend is, dus in dit geval 0.

Omdat de waarden van `mySum` nu niet meer automatisch opgeteld worden tot de waarde van de gehele som doen we dit na uitvoering van de lus. Hierbij wordt eerst de waarde van de som ingelezen, daar wordt de waarde van `mySum` bij opgeteld en deze wordt weer opgeslagen als de nieuwe `sum`. Dit proces levert niet de correcte som op als de waarde van sum tussendoor aangepast wordt. Daarom wordt het aanpassen van de variabele sum uitgevoerd in een zogeheten *kritische omgeving*. De code die hierin staat wordt door alle processen uitgevoerd, maar nooit tegelijkertijd. Het process dat bezig is met rekenen in de kritische sectie moet deze afmaken voordat het volgende proces toegelaten wordt. Dit zorgt ervoor dat de som wel correct wordt uitgerekend.

4.2 SOR Poisson Solver

Zoals beschreven in hoofdstuk 2 bestaat de te parallelliseren code van de `SLAG` routine uit 8 bijna identieke lussen, verdeeld over 4 grote lussen met daarbinnen weer 2 lussen. Omdat deze 4 grote lussen identiek geparallelliseerd worden, wordt in de voorbeelden hieronder steeds maar 1 lus gegeven. Tevens wordt van de tweede binnenste lus de body weggelaten, aangezien deze gelijk is aan de body van de eerste binnenste lus.

4.2.1 Naïeve parallellisatie

Wanneer we de eerste lus van de routine `SLAG` parallelliseren met behulp van OpenMP is dit het resultaat:

```
!$omp parallel private(k, j, i, pklad, diff)
!$omp&firstprivate(delta, omega)
!$omp do
do k = 2, (kmax - 1), 2
  do j = 2, (jmax - 1), 2
    do i = 2, (imax - 1), 2
      pklad = rl(i,j,k)
            - cxn(i,j,k) * p(i-1,j,k) - cxp(i,j,k) * p(i+1,j,k)
            - cyn(i,j,k) * p(i,j-1,k) - cyp(i,j,k) * p(i,j+1,k)
            - czn(i,j,k) * p(i,j,k-1) - czp(i,j,k) * p(i,j,k+1)
```

```
        diff = pklad - p(i,j,k)
        p(i,j,k) = p(i,j,k) + omega * diff
        delta = delta + diff * diff
      end do
    end do

    do j = 3, (jmax - 1), 2
      do i = 3, (imax - 1), 2
        ...
      end do
    end do
  end do
!$omp end do

!$omp critical
deltaSum = deltaSum + delta
!$omp end critical

!$omp end parallel

c Delta terugzetten op 0
delta = 0.0

c Volgende lussen
```

Aangezien de variabelen `pklad` en `diff` alleen nodig zijn voor het berekenen van de nieuwe waarde in een punt heeft elke thread zijn eigen versie van deze variabelen. De variabelen `k`, `j` en `i` zijn ook uniek per thread, elke thread verwerkt een deel van het gehele domein en gebruikt daarvoor deze eigen `k`, `j` en `i`. Hoewel OpenMP automatisch de `k` een privé-variabele zal maken omdat over deze lus wordt geparallelliseerd, geldt dit niet voor de `i` en de `j`. Wanneer deze niet handmatig privé gemaakt worden, zal dit dan ook leiden tot onverwachte, en incorrecte, uitvoer van het programma.

Verder heeft elke thread een variabele `delta`, die na elke lus opgeteld worden in de variabele `deltaSum`. Deze `deltaSum` is de norm voor het stopcriterium van de Poisson solver.

4.2.2 Een intelligentere versie

Wat bij bovenstaande methode opvalt is dat na elke buitenste lus de waarden van de norm opgeteld worden en dat tevens voor elke van die lussen een nieuwe parallelle sectie wordt geopend en weer gesloten. Tussen de lussen gebeurt echter bijna niets, dus is het ook mogelijk om alle lussen in dezelfde parallelle sectie uit te voeren. Dit zorgt ervoor dat er maar één keer nieuwe threads afgesplitst en weer bijeengevoegd hoeven worden. Dit in tegenstelling tot de 4 keer dat dit met bovenstaande oplossing moet gebeuren. Ook hoeft het optellen van de delta's nog maar één keer te gebeuren, namelijk helemaal op het einde van de parallelle sectie. Dit zorgt ervoor dat de code niet meer onderbroken wordt door de kritische secties, wat het geheel ook sneller maakt. In code ziet dit er als volgt uit:

```
!$omp parallel private(k, j, i, pklad, diff)
!$omp&firstprivate(delta, omega)
!$omp do
do k = 2, (kmax - 1), 2
  do j = 2, (jmax - 1), 2
    do i = 2, (imax - 1), 2
      pklad = rl(i,j,k)
```

```
            - cxn(i,j,k) * p(i-1,j,k) - cxp(i,j,k) * p(i+1,j,k)
            - cyn(i,j,k) * p(i,j-1,k) - cyp(i,j,k) * p(i,j+1,k)
            - czn(i,j,k) * p(i,j,k-1) - czp(i,j,k) * p(i,j,k+1)
      diff = pklad - p(i,j,k)
      p(i,j,k) = p(i,j,k) + omega * diff
      delta = delta + diff * diff
    end do
  end do

  do j = 3, (jmax - 1), 2
    do i = 3, (imax - 1), 2
      ...
    end do
  end do
end do
!$omp end do

Volgende lussen

!$omp critical
deltaSum = deltaSum + delta
!$omp end critical

!$omp end parallel
```

4.3 Jacobi Poisson Solver

Tot het analyseren van het gedrag van Jacobi is besloten toen de speedup van SOR tegen leek te vallen op de machine waar toen op getest werd. Jacobi is eenvoudiger dan SOR en uit eerdere tests al gebleken was dat Jacobi goed parallelliseerbaar was op deze machine. Op deze manier was het mogelijk om te zien of het sneller worden van de geparallelliseerde versie van SOR een menselijke fout of een machineprobleem is. Het bleek een probleem van de machine te zijn, meer hierover in hoofdstuk 7, in de discussie over dit onderzoek.

In tegenstelling tot de Poisson Solver met SOR heeft die met Jacobi maar 1 lus waarin gerekend wordt. Dit komt omdat SOR de nieuwe waarden gebruikt en de lus gesplitst wordt om afhankelijkheden te verwijderen. Jacobi rekent met oude waardes en kent deze afhankelijkheden dus niet. Het parallelliseren van de methode die Jacobi gebruikt kent dan ook maar één aanpak, die hetzelfde is als de naïeve parallellisatie van de SOR code, behalve dat er nu maar één lus is.

Hoofdstuk 5

Analyse Poisson Solver

5.1 Metingen

Voor het uitvoeren van de metingen is de solver zodanig aangepast dat hij elke keer exact 100 iteraties uitvoert. Dit zorgt ervoor dat er een eerlijke vergelijking mogelijk is tussen de verschillende methoden, omdat deze door afronding net iets andere resultaten kunnen geven. Deze afrondingen zorgen ook voor een klein verschil in het aantal iteraties dat nodig is voor convergentie. Om dit verschil weg te nemen is er voor gekozen het aantal iteraties vast te zetten.

De tijd die gemeten wordt is de tijd die nodig is om de eerder genoemde 100 iteraties uit te voeren. Om te zorgen dat er zuivere resultaten gevonden worden, wordt dit proces 5 keer uitgevoerd. De meting die in het midden ligt van deze 5 metingen wordt gebruikt in de tabel. Hierdoor worden eventuele outliers in de metingen uitgefilterd. Verder wordt de speedup aangegeven, zoals eerder aangegeven berekend ten op zichte van het parallelle programma op 1 processor. Behalve dat de speedup te zien is in de tabel, is deze ook nog uitgezet in een grafiek met de ideale lineaire speedup ter vergelijking.

In de komende analyse wordt gekeken naar de tijden op de volgende roosters:
50x50x50 Het kleinste rooster waarop is gerekend
100x100x100 Qua grootte het middelste rooster waarop is gerekend
200x200x200 Het grootste rooster waarop is gerekend

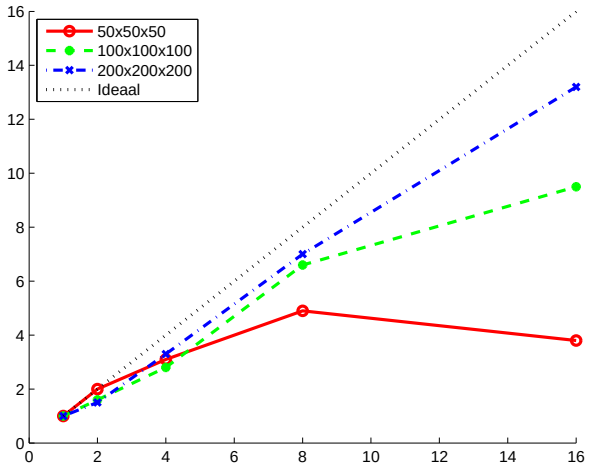
Er is gerekend met 1,2,4,8 en 16 threads op de Huygens supercomputer [1]. Dit is een IBM pSeries 575 systeem. Het systeem bestaat uit 104 nodes met elk 16 IBM Power6 dual core processoren met een snelheid van 4.7 Ghz. Elke node heeft ten minste 128 GB geheugen. Elke thread krijgt een eigen processor toegewezen.

Als compiler is de op de Huygens aanwezige IBM compiler `xlf` gebruikt. Voor het vertalen van de programma's met OpenMP gebruikt deze de compileroptie `-qsmp=omp`, voor het automatisch parallelliseren is de compileroptie `-qsmp=auto`. In alle gevallen is ook de compiler optie `-O3` gebruikt om de compiler optimalisaties aan te zetten.

Het volledige overzicht van de tijden en bijbehorende speedups is te vinden in appendix A.

5.2 Jacobi

Wanneer we bovenstaande tabel bekijken valt op dat het programma met het groter worden van het rooster beter gaat schalen. Wanneer we kijken naar de gehele tabel (appendix A) zien



Figuur 5.1: Speedup van Jacobi met OpenMP op de Huygens

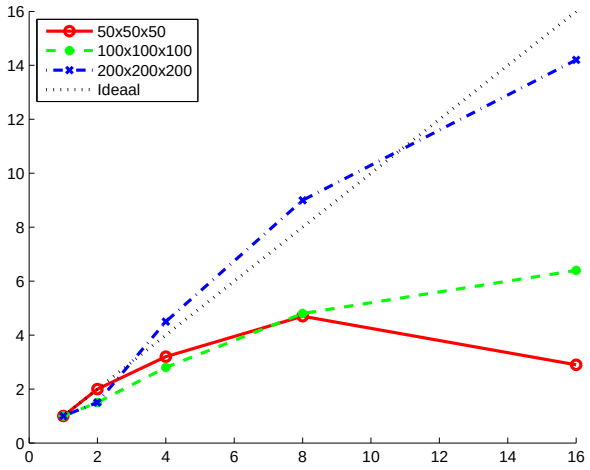
Threads	50x50x50	100x100x100	200x200x200
1	0.07 (1.0)	0.51 (1.0)	5.10 (1.0)
2	0.04 (2.0)	0.31 (1.6)	3.43 (1.5)
4	0.02 (3.1)	0.18 (2.8)	1.54 (3.3)
8	0.01 (4.9)	0.08 (6.6)	0.73 (7.0)
16	0.02 (3.8)	0.05 (9.5)	0.39 (13.2)

Tabel 5.1: Timings voor Jacobi met OpenMP op de Huygens. De tijden zijn in seconden, met daarachter (tussen haakjes) de speedup ten op zichte van het programma op 1 processor.

we dat dit niet helemaal waar is. Verder is opvallend dat bij het kleinste rooster de stap van 8 naar 16 processoren juist een vertraging introduceert in plaat van een grotere versnelling. Dit komt waarschijnlijk omdat het creëren van de benodigde threads meer tijd kost dan het rekenproces zelf.

Threads	50x50x50	100x100x100	200x200x200
1	0.04 (1.0)	0.26 (1.0)	3.55 (1.0)
2	0.02 (2.0)	0.18 (1.5)	2.40 (1.5)
4	0.01 (3.2)	0.10 (2.8)	0.79 (4.5)
8	0.01 (4.7)	0.05 (4.8)	0.40 (9.0)
16	0.01 (2.9)	0.04 (6.4)	0.25 (14.2)

Tabel 5.2: Timings voor Jacobi met automatische parallellisatie op de Huygens. De tijden zijn in seconden, met daarachter de speedup ten op zichte van het programma op 1 processor.

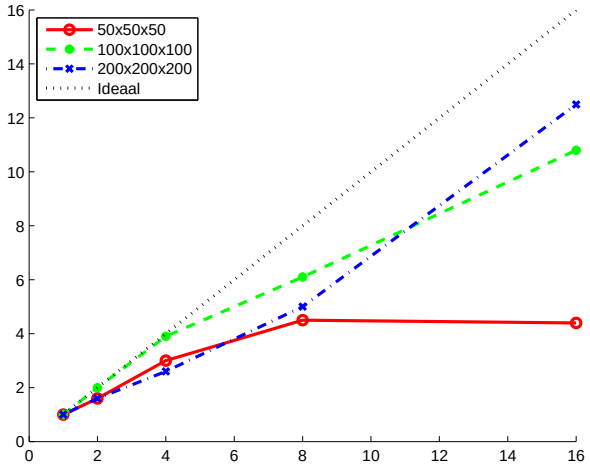


Figuur 5.2: Speedup van Jacobi met automatische parallellisatie op de Huygens

Het vergelijken van de tabel voor automatische parallellisatie met de tabellen voor het parallelliseren met OpenMP levert een zeer interessante observatie op. Hoewel de speedup niet altijd groter is, is deze versie wel sneller dan de OpenMP versie op de twee grotere roosters. Op het kleine rooster lijkt weinig te gebeuren, de tijd op 1 processor ligt hier ook lager dan bij de OpenMP versies, maar deze tijd neemt niet noemenswaardig af wanneer er meer processoren gebruikt worden.

5.3 SOR

De volgende tabellen geven een overzicht van de tijden die nodig zijn om het SOR programma uit te voeren zoals boven beschreven.

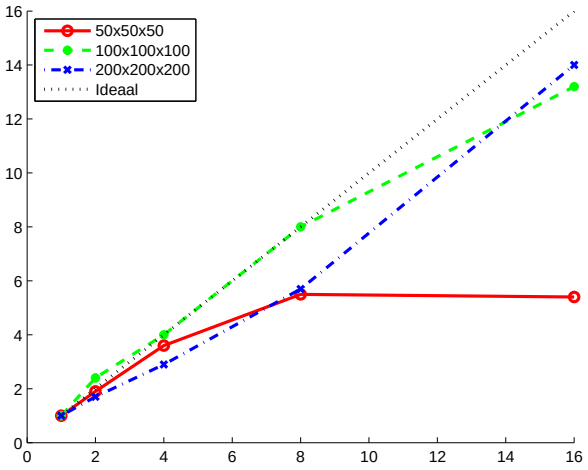


Figuur 5.3: Speedup van SOR met OpenMP op de Huygens

Threads	50x50x50	100x100x100	200x200x200
1	0.16 (1.0)	1.67 (1.0)	17.84 (1.0)
2	0.10 (1.6)	0.82 (2.0)	11.50 (1.6)
4	0.05 (3.0)	0.43 (3.9)	6.79 (2.6)
8	0.04 (4.5)	0.27 (6.1)	3.56 (5.0)
16	0.04 (4.4)	0.15 (10.8)	1.43 (12.5)

Tabel 5.3: Timings voor SOR met OpenMP op de Huygens. De tijden zijn in seconden, met daarachter (tussen haakjes) de speedup ten op zichte van het programma op 1 processor.

Tabel 5.3 geeft een zelfde beeld voor de parallellisatie van de SOR methode als eerder gevonden werd voor Jacobi. Ook hier lijkt de schaalbaarheid van het probleem beter te worden als het rooster groeit. De volledige tabellen (zie wederom appendix A) laten zien dat dit weer niet overal geldt, maar dat overal wel de doelstelling van dit onderzoek gehaald wordt. De SOR methode laat ook het terugvallen van de snelheid zien als er teveel processoren worden gebruikt voor het kleinste rooster.



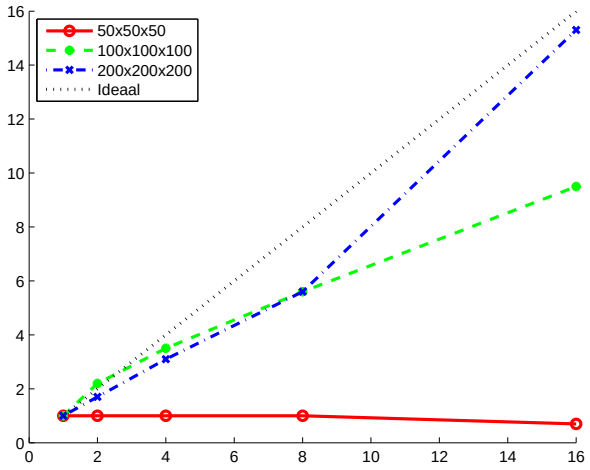
Figuur 5.4: Speedup van SOR met “slimmere” OpenMP op de Huygens

Threads	50x50x50	100x100x100	200x200x200
1	0.17 (1.0)	1.86 (1.0)	19.68 (1.0)
2	0.09 (1.9)	0.79 (2.4)	11.33 (1.7)
4	0.05 (3.6)	0.46 (4.0)	6.78 (2.9)
8	0.03 (5.5)	0.23 (8.0)	3.47 (5.7)
16	0.03 (5.4)	0.14 (13.2)	1.40 (14.0)

Tabel 5.4: Timings voor SOR met “slimmere” OpenMP op de Huygens. De tijden zijn in seconden, met daarachter de speedup ten op zichte van het programma op 1 processor.

Bij het vergelijken van de tabel voor de “slimmere” OpenMP aanpak met de tabel voor de simpele OpenMP aanpak zien we als we naar de speedup kijken een verbetering. Wanneer we echter kijken naar de tijd die nodig was voor het rekenen, zien we dat de tijden op 8 en 16

processoren niet veel verschillen. Het verschil in speedup wordt vooral gemaakt doordat de gemeten tijd op 1 processor hier hoger ligt. Ook hier is weer te zien dat het kleinste rooster te klein is om mee te rekenen op 16 processoren, aangezien de speedup daar weer naar beneden gaat.



Figuur 5.5: Speedup van SOR met automatische parallellisatie op de Huygens

Threads	50x50x50	100x100x100	200x200x200
1	0.09 (1.0)	1.14 (1.0)	15.10 (1.0)
2	0.09 (1.0)	0.52 (2.2)	9.04 (1.7)
4	0.08 (1.0)	0.33 (3.5)	4.88 (3.1)
8	0.09 (1.0)	0.20 (5.6)	2.69 (5.6)
16	0.12 (0.7)	0.12 (9.5)	0.99 (15.3)

Tabel 5.5: Timings voor SOR met automatische parallellisatie op de Huygens. De tijden zijn in seconden, met daarachter de speedup ten op zichte van het programma op 1 processor.

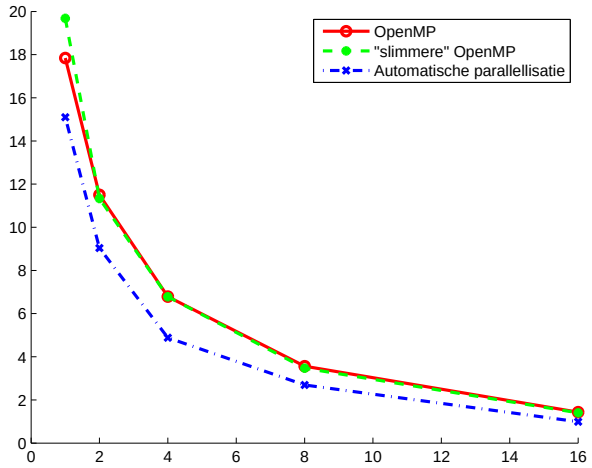
Bij de automatische parallellisatie van de SOR methode zien we net als bij Jacobi dat de speedup niet altijd beter is, maar de rekentijd wel weer lager ligt.

5.4 Conclusies

De meest opvallende conclusie is dat voor zowel SOR als Jacobi alle parallellisatie-methoden moeite hebben met het allerkleinste rooster. Voor de automatisch geparallelliseerde versie geldt dit ook nog voor een aantal grotere roosters. Wanneer er gewerkt gaat worden met deze kleine roosters lijkt OpenMP dus een betere aanpak. Voor SOR ontlopen de twee OpenMP aanpakken elkaar hier niet veel, al is de “slimmere” aanpak vaak net een fractie sneller. Voor kleine roosters lijkt dit voor SOR dan ook de beste oplossing. Ook voor Jacobi lijkt voor kleine roosters OpenMP de beste keus.

Voor grote roosters is behalve de speedup ook de tijd belangrijk. Zowel voor SOR als Jacobi leidt dat voor de Huygens tot de keuze voor automatische parallellisatie als aanbevolen

methode. Deze methode heeft namelijk voor alle grote roosters minder tijd nodig dan elk van de OpenMP methodes voor SOR met hetzelfde aantal processoren. En ook voor Jacobi is deze parallellisatie-methode wat dat betreft het beste.



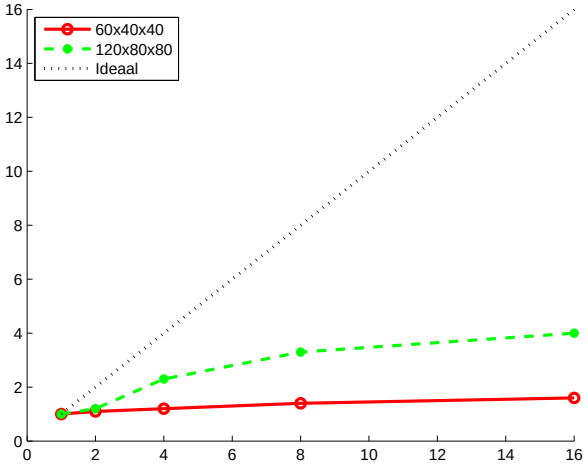
Figuur 5.6: Tijd voor SOR op het grootste rooster voor de verschillende parallellisatiemethoden op de Huygens

Hoofdstuk 6

Analyse ComFlo

Omdat er maar beperkte rekentijd beschikbaar was op de Huygens, is er voor gekozen geen hele grote roosters te gebruiken voor de analyse van het geparallelliseerde ComFlo programma. In het voorgaande hoofdstuk is gebleken dat de "slimmere" OpenMP aanpak van de OpenMP methoden het beste was op kleine roosters. Daarom is besloten deze parallellisatie-methode toe te passen op de Poisson solver in het ComFlo programma. De resultaten worden vergeleken met het MPI-geparallelliseerde programma van Jan Feitsma (Feitsma [3]) en een automatisch geparallelliseerd programma die op dezelfde machine zijn uitgevoerd.

De tabellen geven de tijd in seconden voor het uitvoeren van het gehele programma met als eindtijd voor de simulatie 0.2 seconden. Voor het 60x40x40 rooster zijn dit 46 tijdstappen, voor het 120x80x80 rooster zijn dit er 62. Achter de tijden staat wederom de speedup vermeld. De grafieken geven wederom de speedup aan voor de gebruikte roosters.

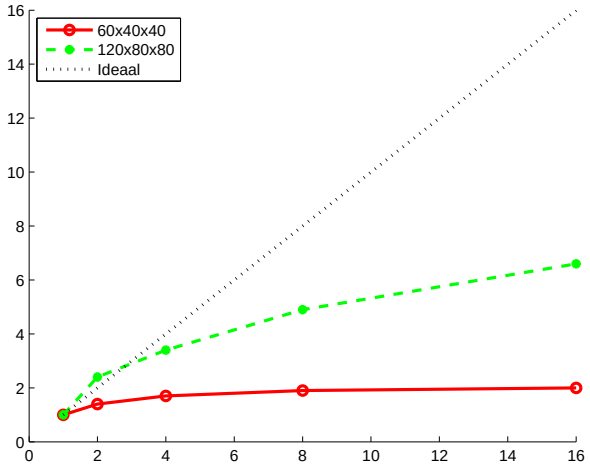


Figuur 6.1: Speedup van ComFlo met OpenMP op de Huygens

In tabel 6.1 is te zien dat, hoewel er nu wel enige winst behaald wordt, de resultaten op het kleinste rooster nog steeds erg tegenvallen. Ook voor het grotere rooster valt het resultaat enigzins tegen. Hoewel er een behoorlijke snelheidswinst geboekt wordt, blijft deze onder de gewenste speedup van 4 bij 8 processoren.

Proc.	60x40x40	120x80x80
1	62 (1.0)	1617 (1.0)
2	59 (1.1)	1299 (1.2)
4	51 (1.2)	710 (2.3)
8	43 (1.4)	496 (3.3)
16	38 (1.6)	400 (4.0)

Tabel 6.1: Timings voor ComFlo met OpenMP op de Huygens. De tijden zijn in seconden, met daarachter (tussen haakjes) de speedup ten op zichte van het programma op 1 processor.



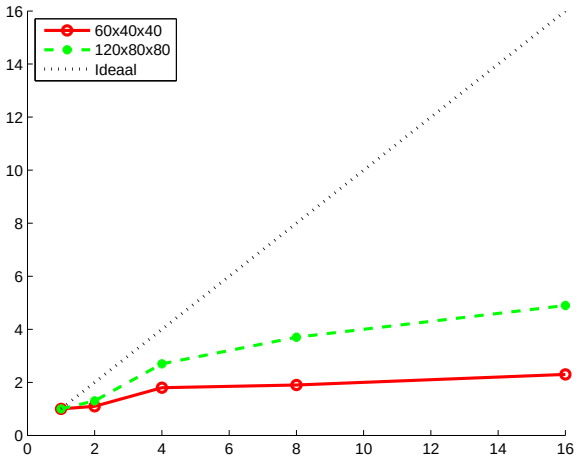
Figuur 6.2: Speedup van ComFlo met MPI op de Huygens

Proc.	60x40x40	120x80x80
1	80 (1.0)	2216 (1.0)
2	58 (1.4)	895 (2.4)
4	47 (1.7)	647 (3.4)
8	43 (1.9)	452 (4.9)
16	40 (2.0)	338 (6.6)

Tabel 6.2: Timings voor ComFlo met MPI op de Huygens. De tijden zijn in seconden, met daarachter de speedup ten op zichte van het programma op 1 processor.

De MPI geparallelliseerde code heeft duidelijk ook problemen met het kleine rooster. Op het grote rooster haalt deze versie echter veel betere schalings resultaten dan de OpenMP versie. Opvallend is dat het programma een speedup haalt van 2.4 op 2 processoren, of anders gezegd, meer dan 2 keer zo snel wordt bij een verdubbeling van het aantal processoren. Dit zou kunnen duiden op het feit dat de rekentijd voor 1 processor te hoog ligt, waardoor de gehele speedup berekening niet meer zuiver is. Desondanks is het MPI programma op het grote rooster (een beetje) sneller dan het OpenMP programma wanneer er meer dan één processor gebruikt wordt. Het parallelliseren met MPI kost echter veel meer tijd en is veel foutgevoeliger dan het parallelliseren met OpenMp, terwijl de snelheidswinst niet enorm veel

groter is. Het parallelliseren met OpenMP is dus zeker een goede methode om dit programma sneller te maken.



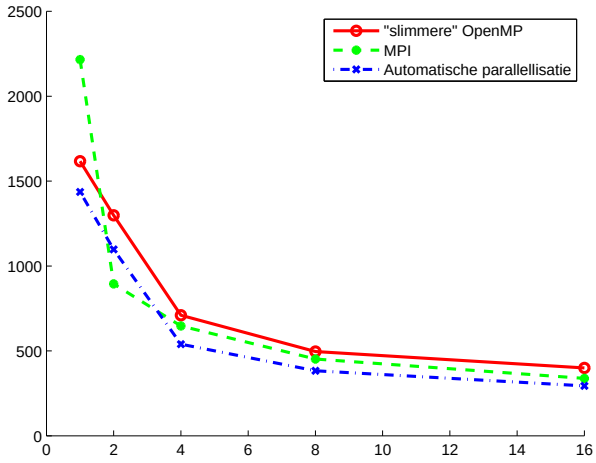
Figuur 6.3: Speedup van ComFlo met automatische parallellisatie op de Huygens

Proc.	60x40x40	120x80x80
1	57 (1.0)	1436 (1.0)
2	50 (1.1)	1098 (1.3)
4	32 (1.8)	540 (2.7)
8	29 (1.9)	383 (3.7)
16	25 (2.3)	294 (4.9)

Tabel 6.3: Timings voor ComFlo met automatische parallellisatie op de Huygens. De tijden zijn in seconden, met daarachter de speedup ten op zichte van het programma op 1 processor.

Zoals bovenstaande tabel laat zien is de speedup van het automatisch geparallelliseerde programma hoger dan de speedup voor het met OpenMP geparallelliseerde programma. Ook liggen de rekentijden lager. Beide kan veroorzaakt worden doordat in het geval van automatische parallellisate meer delen van het programma geparallelliseerd zijn dan bij handmatige parallellisatie. Maar omdat de routine `slag` het grootste gedeelte van de rekentijd verbruikt en dit de routine is die handmatig is geparallelliseerd, is een vergelijking nog wel mogelijk.

De automatisch geparallelliseerde code haalt weliswaar een lagere speedup dan de MPI geparallelliseerde code, maar is wel sneller dan de MPI code. Het verschil in speedup wordt dan ook vooral veroorzaakt doordat de automatisch geparallelliseerde code veel sneller is op 1 processor.



Figuur 6.4: Tijd van ComFlo op het grootste rooster voor de gebruikte parallellisatiemethoden op de Huygens

Hoofdstuk 7

Conclusie & Discussie

7.1 Parallellisatie

Het uiteindelijke doel van dit onderzoek was het versnellen van het ComFlo programma door middel van parallellisatie met behulp van OpenMP. Uit eerder onderzoek was gebleken dat het niet nodig was het gehele programma te parallelliseren, maar alleen de Poisson solver. Dit gedeelte van de code blijkt qua tijdsverbruik namelijk het duurste onderdeel van de ComFlo code.

Het parallelliseren van de code met OpenMP is gelukt en hoewel de gewenste speedup niet gehaald is, komt het toch behoorlijk in de buurt. De eerder door Jan Feitsma met MPI geparallelliseerde versie werkt beter op de testmachine, maar hier staat tegenover dat het parallelliseren met behulp van MPI veel ingewikkelder is, een hoop meer tijd kost en veel foutgevoeliger is. Het andere alternatief, het gebruik van de mogelijkheid van de compiler tot automatische parallellisatie, is zeer veelbelovend. Op de Huygens levert deze beter speedup resultaten dan het met OpenMP geparallelliseerde programma en de rekentijden zijn zelfs korter dan die bij MPI. Hiervoor is echter gebruik gemaakt van de speciale IBM compiler die bij deze machine hoort. Dit geeft dan ook geen garantie dat hetzelfde resultaat behaald kan worden met de automatische parallellisatie optie van een andere compiler.

7.2 Compiler en machine keuze

Tijdens het onderzoek is naar voren gekomen dat de belangrijkste factoren de keuze van de compiler en de machine (waarop gerekend wordt) zijn. Eerdere tests zijn uitgevoerd op de SI01, deze machine heeft 8 Intel Xeon X7350 Quad Core processoren die werken op 2.93 Ghz en een totaal geheugen van 128 GB. Hierbij leverde alleen de code gecompileerd met de speciale Intel compiler op deze machine enig resultaat op. De resultaten bleven echter ver achter bij die berekend op de Huygens, waar ook de versie op 1 processor veel sneller was. Dit leidt tot de conclusie dat de keuze van machine iets is wat in gedachten gehouden moet worden bij het parallelliseren van programma's zoals ComFlo. Doordat ComFlo echter als algemeen programma wordt uitgegeven, zal het niet mogelijk zijn om het programma met de speciale compilers van deze machines te vertalen. Dit zorgt er ook voor dat eventuele speciaal voor deze machines geoptimaliseerde (OpenMP-) libraries niet gebruikt kunnen worden. Dit geldt echter zeker ook voor MPI. Er zal onderzoek nodig zijn om te bepalen wat dit voor impact heeft op het ComFlo programma en de distributie daarvan.

Bijlage A

Timings Poisson Solver

Deze appendix bevat de volledige tabellen met de verzamelde resultaten van de tijdsmetingen uitgevoerd op de Poisson solver. Zoals beschreven in hoofdstuk 5 zijn alle metingen uitgevoerd op de Huygens met 1, 2, 4, 8 en 16 threads.

De tabellen geven de tijden voor het uitvoeren van 100 iteraties. Achter de tijden staat (tussen haakjes) de speedup ten op zichte van het programma op 1 processor. De roosters waarop is gerekend zijn van de vorm `i x j x k`, waarbij k wordt aangegeven door de bovenste rij in de tabel. De rij daaronder geeft de waarde voor j en in de linker kolom staat de waarde voor i.

	50			100			200		
	50	100	200	50	100	200	50	100	200
50	0.16 (1.0) 0.1 (1.6) 0.05 (3.0) 0.04 (4.5) 0.04 (4.4)	0.37 (1.0) 0.18 (2.0) 0.1 (3.8) 0.06 (5.9) 0.06 (6.1)	0.77 (1.0) 0.41 (1.9) 0.2 (3.8) 0.13 (5.9) 0.09 (8.1)	0.37 (1.0) 0.18 (2.0) 0.12 (3.2) 0.06 (6.0) 0.06 (6.5)	0.78 (1.0) 0.48 (1.6) 0.19 (4.0) 0.12 (6.7) 0.08 (10.1)	1.73 (1.0) 0.86 (2.0) 0.44 (3.9) 0.27 (6.4) 0.15 (11.6)	0.77 (1.0) 0.47 (1.6) 0.2 (3.9) 0.12 (6.4) 0.08 (9.8)	1.8 (1.0) 0.87 (2.1) 0.47 (3.8) 0.25 (7.1) 0.15 (11.9)	5.14 (1.0) 2.86 (1.8) 1.22 (4.2) 0.61 (8.5) 0.37 (14.0)
100	0.35 (1.0) 0.23 (1.5) 0.11 (3.2) 0.06 (5.9) 0.05 (6.5)	0.73 (1.0) 0.41 (1.8) 0.19 (3.9) 0.11 (6.5) 0.09 (8.1)	1.68 (1.0) 0.82 (2.0) 0.46 (3.6) 0.24 (7.1) 0.17 (9.9)	0.74 (1.0) 0.39 (1.9) 0.23 (3.3) 0.12 (6.0) 0.08 (9.4)	1.67 (1.0) 0.82 (2.0) 0.43 (3.9) 0.27 (6.1) 0.15 (10.8)	5.56 (1.0) 2.48 (2.2) 1.08 (5.2) 0.57 (9.7) 0.34 (16.3)	1.66 (1.0) 0.78 (2.1) 0.42 (3.9) 0.22 (7.7) 0.14 (11.5)	5.08 (1.0) 2.56 (2.0) 1.06 (4.8) 0.57 (8.9) 0.35 (14.6)	10.83 (1.0) 6.2 (1.7) 2.4 (4.5) 1.06 (10.2) 0.6 (18.2)
200	0.66 (1.0) 0.37 (1.8) 0.18 (3.6) 0.1 (6.3) 0.08 (7.8)	1.48 (1.0) 0.72 (2.1) 0.45 (3.3) 0.23 (6.4) 0.18 (8.1)	4.59 (1.0) 2.18 (2.1) 1.11 (4.1) 0.56 (8.3) 0.39 (11.8)	1.49 (1.0) 0.71 (2.1) 0.42 (3.5) 0.22 (6.8) 0.12 (12.3)	4.85 (1.0) 2.13 (2.3) 0.97 (5.0) 0.56 (8.7) 0.35 (13.7)	8.75 (1.0) 5.16 (1.7) 2.31 (3.8) 1.01 (8.7) 0.54 (16.1)	4.78 (1.0) 1.93 (2.5) 0.82 (5.8) 0.5 (9.6) 0.28 (16.9)	8.67 (1.0) 4.91 (1.8) 2.5 (3.5) 1.2 (7.2) 0.53 (16.4)	17.84 (1.0) 11.5 (1.6) 6.79 (2.6) 3.56 (5.0) 1.43 (12.5)

Tabel A.1: Volledige timings voor SOR met OpenMP op de Huygens

	50			100			200		
	50	100	200	50	100	200	50	100	200
50	0.17 (1.0) 0.09 (1.9) 0.05 (3.6) 0.03 (5.5) 0.03 (5.4)	0.37 (1.0) 0.21 (1.8) 0.1 (3.7) 0.05 (6.8) 0.04 (8.5)	0.76 (1.0) 0.43 (1.8) 0.25 (3.0) 0.12 (6.2) 0.09 (8.2)	0.37 (1.0) 0.2 (1.8) 0.1 (3.8) 0.05 (7.3) 0.04 (9.8)	0.78 (1.0) 0.46 (1.7) 0.23 (3.4) 0.11 (6.9) 0.07 (11.1)	1.81 (1.0) 0.84 (2.2) 0.51 (3.5) 0.28 (6.4) 0.17 (10.7)	0.81 (1.0) 0.41 (2.0) 0.22 (3.6) 0.11 (7.2) 0.06 (12.8)	1.88 (1.0) 0.94 (2.0) 0.48 (3.9) 0.23 (8.1) 0.14 (13.8)	5.5 (1.0) 2.6 (2.1) 1.16 (4.7) 0.68 (8.1) 0.36 (15.4)
100	0.35 (1.0) 0.17 (2.1) 0.09 (3.9) 0.05 (6.6) 0.04 (8.0)	0.73 (1.0) 0.43 (1.7) 0.22 (3.3) 0.11 (6.7) 0.08 (9.0)	1.68 (1.0) 0.82 (2.0) 0.46 (3.7) 0.24 (7.1) 0.18 (9.3)	0.73 (1.0) 0.37 (2.0) 0.22 (3.4) 0.11 (6.9) 0.07 (10.2)	1.86 (1.0) 0.79 (2.4) 0.46 (4.0) 0.23 (8.0) 0.14 (13.2)	5.44 (1.0) 2.46 (2.2) 1.12 (4.9) 0.56 (9.8) 0.32 (17.3)	1.73 (1.0) 0.84 (2.1) 0.45 (3.8) 0.24 (7.1) 0.13 (13.8)	5.62 (1.0) 2.36 (2.4) 1.06 (5.3) 0.63 (8.9) 0.33 (17.2)	10.56 (1.0) 6.66 (1.6) 3.19 (3.3) 1.54 (6.9) 0.57 (18.5)
200	0.66 (1.0) 0.41 (1.6) 0.2 (3.3) 0.1 (6.6) 0.07 (8.8)	1.49 (1.0) 0.8 (1.9) 0.42 (3.5) 0.21 (7.2) 0.15 (10.1)	4.76 (1.0) 2.38 (2.0) 0.98 (4.9) 0.57 (8.4) 0.39 (12.1)	1.48 (1.0) 0.76 (2.0) 0.39 (3.7) 0.21 (7.2) 0.12 (12.1)	4.86 (1.0) 2.03 (2.4) 0.95 (5.1) 0.53 (9.1) 0.34 (14.2)	9.5 (1.0) 5.82 (1.6) 2.37 (4.0) 1.04 (9.1) 0.62 (15.4)	4.92 (1.0) 2.36 (2.1) 0.97 (5.1) 0.57 (8.6) 0.32 (15.3)	9.45 (1.0) 4.64 (2.0) 2.96 (3.2) 1.11 (8.5) 0.54 (17.4)	19.68 (1.0) 11.33 (1.7) 6.78 (2.9) 3.47 (5.7) 1.4 (14.0)

Tabel A.2: Volledige timings voor SOR met “slimmere” OpenMP op de Huygens

	50			100			200		
	50	100	200	50	100	200	50	100	200
50	0.09 (1.0) 0.09 (1.0) 0.08 (1.0) 0.09 (1.0) 0.12 (0.7)	0.23 (1.0) 0.14 (1.6) 0.11 (2.0) 0.11 (2.0) 0.12 (1.8)	0.49 (1.0) 0.36 (1.4) 0.15 (3.2) 0.11 (4.4) 0.17 (2.8)	0.23 (1.0) 0.19 (1.2) 0.11 (2.2) 0.12 (1.9) 0.13 (1.8)	0.49 (1.0) 0.24 (2.0) 0.13 (3.8) 0.1 (4.8) 0.17 (2.9)	1.18 (1.0) 0.53 (2.2) 0.39 (3.0) 0.22 (5.4) 0.12 (9.7)	0.49 (1.0) 0.43 (1.2) 0.13 (3.8) 0.14 (3.6) 0.18 (2.7)	1.17 (1.0) 0.54 (2.2) 0.31 (3.8) 0.23 (5.1) 0.11 (10.4)	4.12 (1.0) 1.88 (2.2) 1.0 (4.1) 0.5 (8.2) 0.3 (14.0)
100	0.22 (1.0) 0.13 (1.7) 0.1 (2.2) 0.12 (1.9) 0.12 (1.8)	0.46 (1.0) 0.26 (1.8) 0.17 (2.8) 0.11 (4.1) 0.17 (2.7)	1.1 (1.0) 0.5 (2.2) 0.34 (3.2) 0.18 (5.9) 0.16 (6.7)	0.46 (1.0) 0.3 (1.5) 0.18 (2.5) 0.1 (4.5) 0.17 (2.8)	1.14 (1.0) 0.52 (2.2) 0.33 (3.5) 0.2 (5.6) 0.12 (9.5)	4.16 (1.0) 2.01 (2.1) 0.65 (6.4) 0.42 (9.8) 0.31 (13.6)	1.17 (1.0) 0.61 (1.9) 0.38 (3.1) 0.24 (4.9) 0.16 (7.4)	4.26 (1.0) 1.82 (2.3) 0.7 (6.1) 0.47 (9.1) 0.29 (14.9)	8.39 (1.0) 4.5 (1.9) 2.83 (3.0) 0.72 (11.7) 0.58 (14.5)
200	0.39 (1.0) 0.2 (2.0) 0.13 (3.1) 0.14 (2.7) 0.17 (2.3)	0.98 (1.0) 0.44 (2.2) 0.29 (3.4) 0.18 (5.3) 0.16 (6.3)	3.75 (1.0) 1.6 (2.3) 0.64 (5.9) 0.48 (7.7) 0.31 (12.3)	0.98 (1.0) 0.44 (2.2) 0.26 (3.7) 0.15 (6.4) 0.14 (7.1)	3.67 (1.0) 2.05 (1.8) 0.71 (5.1) 0.39 (9.3) 0.26 (13.9)	7.53 (1.0) 4.07 (1.8) 1.46 (5.1) 0.99 (7.6) 0.42 (18.1)	3.7 (1.0) 1.74 (2.1) 0.6 (6.2) 0.36 (10.3) 0.26 (14.2)	7.42 (1.0) 5.06 (1.5) 1.56 (4.8) 0.84 (8.9) 0.48 (15.4)	15.1 (1.0) 9.04 (1.7) 4.88 (3.1) 2.69 (5.6) 0.99 (15.3)

Tabel A.3: Volledige timings voor SOR met automatische parallelisatie op de Huygens

	50			100			200		
	50	100	200	50	100	200	50	100	200
50	0.07 (1.0) 0.04 (2.0) 0.02 (3.1) 0.01 (4.9) 0.02 (3.8)	0.14 (1.0) 0.07 (1.9) 0.04 (3.3) 0.03 (4.8) 0.02 (5.5)	0.31 (1.0) 0.15 (2.1) 0.08 (4.0) 0.04 (7.3) 0.04 (8.1)	0.15 (1.0) 0.08 (1.8) 0.03 (4.4) 0.02 (6.6) 0.03 (5.8)	0.27 (1.0) 0.14 (2.0) 0.08 (3.5) 0.05 (6.0) 0.03 (9.1)	0.49 (1.0) 0.33 (1.5) 0.17 (2.8) 0.09 (5.2) 0.06 (8.3)	0.27 (1.0) 0.15 (1.8) 0.07 (3.7) 0.05 (5.5) 0.04 (6.2)	0.51 (1.0) 0.29 (1.8) 0.16 (3.1) 0.1 (5.1) 0.05 (9.7)	1.13 (1.0) 0.67 (1.7) 0.35 (3.2) 0.17 (6.5) 0.11 (10.6)
100	0.1 (1.0) 0.07 (1.5) 0.04 (2.9) 0.02 (4.3) 0.02 (4.7)	0.22 (1.0) 0.14 (1.6) 0.09 (2.5) 0.05 (4.7) 0.05 (4.5)	0.46 (1.0) 0.28 (1.7) 0.16 (2.8) 0.08 (5.5) 0.07 (6.7)	0.22 (1.0) 0.14 (1.6) 0.09 (2.4) 0.04 (5.9) 0.03 (7.5)	0.51 (1.0) 0.31 (1.6) 0.18 (2.8) 0.08 (6.6) 0.05 (9.5)	1.06 (1.0) 0.62 (1.7) 0.33 (3.2) 0.23 (4.6) 0.15 (7.2)	0.45 (1.0) 0.3 (1.5) 0.17 (2.6) 0.07 (6.0) 0.05 (9.6)	0.94 (1.0) 0.62 (1.5) 0.3 (3.1) 0.19 (5.0) 0.11 (8.4)	2.55 (1.0) 1.18 (2.2) 0.71 (3.6) 0.35 (7.2) 0.21 (11.9)
200	0.28 (1.0) 0.13 (2.1) 0.07 (3.7) 0.04 (6.7) 0.04 (7.0)	0.44 (1.0) 0.26 (1.7) 0.16 (2.7) 0.09 (4.9) 0.08 (5.3)	0.98 (1.0) 0.5 (2.0) 0.31 (3.2) 0.21 (4.6) 0.16 (6.3)	0.45 (1.0) 0.26 (1.7) 0.15 (3.1) 0.1 (4.6) 0.06 (8.2)	0.95 (1.0) 0.6 (1.6) 0.31 (3.0) 0.16 (6.1) 0.11 (8.3)	2.5 (1.0) 1.62 (1.5) 0.71 (3.5) 0.37 (6.7) 0.22 (11.3)	1.12 (1.0) 0.53 (2.1) 0.33 (3.4) 0.18 (6.3) 0.1 (11.5)	2.51 (1.0) 1.51 (1.7) 0.67 (3.7) 0.36 (7.0) 0.21 (11.9)	5.1 (1.0) 3.43 (1.5) 1.54 (3.3) 0.73 (7.0) 0.39 (13.2)

Tabel A.4: Volledige timings voor Jacobi met OpenMP op de Huygens

	50			100			200		
	50	100	200	50	100	200	50	100	200
50	0.04 (1.0) 0.02 (2.0) 0.01 (3.2) 0.01 (4.7) 0.01 (2.9)	0.06 (1.0) 0.04 (1.4) 0.03 (2.2) 0.02 (3.3) 0.02 (2.8)	0.13 (1.0) 0.06 (2.1) 0.05 (2.7) 0.03 (3.8) 0.03 (4.6)	0.06 (1.0) 0.04 (1.4) 0.02 (2.6) 0.01 (4.4) 0.02 (3.1)	0.13 (1.0) 0.09 (1.4) 0.05 (2.7) 0.03 (4.6) 0.02 (6.0)	0.28 (1.0) 0.18 (1.5) 0.11 (2.5) 0.06 (4.3) 0.04 (6.8)	0.13 (1.0) 0.06 (2.2) 0.05 (2.9) 0.02 (5.3) 0.03 (4.3)	0.28 (1.0) 0.21 (1.3) 0.12 (2.2) 0.06 (4.8) 0.03 (7.9)	0.57 (1.0) 0.36 (1.6) 0.23 (2.5) 0.13 (4.3) 0.07 (7.8)
100	0.05 (1.0) 0.04 (1.4) 0.03 (2.0) 0.02 (3.0) 0.02 (3.1)	0.12 (1.0) 0.06 (2.1) 0.04 (2.8) 0.03 (3.8) 0.03 (4.3)	0.27 (1.0) 0.13 (2.0) 0.13 (2.1) 0.07 (3.7) 0.05 (5.1)	0.12 (1.0) 0.09 (1.4) 0.05 (2.6) 0.03 (4.3) 0.02 (5.3)	0.26 (1.0) 0.18 (1.5) 0.1 (2.8) 0.05 (4.8) 0.04 (6.4)	0.6 (1.0) 0.37 (1.6) 0.21 (2.9) 0.12 (4.9) 0.08 (7.8)	0.26 (1.0) 0.17 (1.6) 0.1 (2.7) 0.05 (5.0) 0.04 (7.2)	0.59 (1.0) 0.33 (1.8) 0.21 (2.8) 0.12 (4.9) 0.08 (7.7)	1.76 (1.0) 0.61 (2.9) 0.39 (4.5) 0.22 (8.1) 0.14 (12.7)
200	0.11 (1.0) 0.08 (1.3) 0.05 (2.5) 0.03 (3.6) 0.03 (3.9)	0.24 (1.0) 0.13 (1.8) 0.08 (3.1) 0.06 (4.1) 0.05 (4.6)	0.5 (1.0) 0.27 (1.8) 0.18 (2.8) 0.11 (4.6) 0.1 (5.2)	0.25 (1.0) 0.13 (1.9) 0.1 (2.5) 0.05 (4.6) 0.04 (6.2)	0.67 (1.0) 0.26 (2.6) 0.2 (3.4) 0.1 (6.5) 0.07 (9.3)	1.55 (1.0) 0.73 (2.1) 0.41 (3.8) 0.2 (7.6) 0.16 (9.9)	0.5 (1.0) 0.37 (1.3) 0.2 (2.4) 0.1 (5.0) 0.06 (7.7)	1.58 (1.0) 0.83 (1.9) 0.36 (4.4) 0.19 (8.2) 0.13 (12.3)	3.55 (1.0) 2.4 (1.5) 0.79 (4.5) 0.4 (9.0) 0.25 (14.2)

Tabel A.5: Volledige timings voor Jacobi met automatische parallellisatie op de Huygens

Bijlage B

Listing Slag Code

```
      SUBROUTINE SLAG(DELTA,MAXDIFF)

      IMPLICIT NONE

      INCLUDE 'COMMONS/paramdp.inc'
      INCLUDE 'COMMONS/coefpdp.inc'
      INCLUDE 'COMMONS/gridardp.inc'
      INCLUDE 'COMMONS/labelsdp.inc'
      INCLUDE 'COMMONS/numerdp.inc'
      INCLUDE 'COMMONS/physdp.inc'

C  Local variables
      INTEGER I, J, K
      REAL*8  DELTA, NORM, DIFF, MAXDIFF
C
C  Compute one SOR-iteration and return the (scaled) Euclidean norm
C  of the difference of two successive pressure solutions. A red-black
C  ordering is used. Note that the Poisson equation can be solved in
C  every single cell (i.e. also non F-cells) which is faster on
C  vector computers.
C
      ITER = ITER + 1
      NORM = ZERO
      MAXDIFF = ZERO

C
C  Red points, even z-planes.
C
      DO 10, K = 2, KMAX-1, 2
        DO 11, J = 2, JMAX-1, 2
        DO 11, I = 2, IMAX-1, 2
          IF ((PLABFS(I,J,K) .EQ. 1) .OR. (PLABFS(I,J,K) .EQ. 3)) THEN
            DIFF = - P(I,J,K) + DIV(I,J,K)
     &             - CXL(I,J,K) * P(I-1,J,K) - CXR(I,J,K) * P(I+1,J,K)
     &             - CYL(I,J,K) * P(I,J-1,K) - CYR(I,J,K) * P(I,J+1,K)
     &             - CZL(I,J,K) * P(I,J,K-1) - CZR(I,J,K) * P(I,J,K+1)
            P(I,J,K) = P(I,J,K) + OMEGA * DIFF
            PRIVNORM = PRIVNORM + DIFF * DIFF
            IF (ABS(P(I,J,K)).LT.SMALL) THEN
               print*,'ZERO PRESSURE ??'
               PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF))
```

```
            ELSE
               PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF/P(I,J,K)))
            ENDIF
          ENDIF
 11     CONTINUE

        DO 12, J = 3, JMAX-1, 2
        DO 12, I = 3, IMAX-1, 2
          IF ((PLABFS(I,J,K) .EQ. 1) .OR. (PLABFS(I,J,K) .EQ. 3)) THEN
            DIFF = - P(I,J,K) + DIV(I,J,K)
     &             - CXL(I,J,K) * P(I-1,J,K) - CXR(I,J,K) * P(I+1,J,K)
     &             - CYL(I,J,K) * P(I,J-1,K) - CYR(I,J,K) * P(I,J+1,K)
     &             - CZL(I,J,K) * P(I,J,K-1) - CZR(I,J,K) * P(I,J,K+1)
            P(I,J,K) = P(I,J,K) + OMEGA * DIFF
            PRIVNORM = PRIVNORM + DIFF * DIFF
            IF (ABS(P(I,J,K)).LT.SMALL) THEN
               print*,'ZERO PRESSURE ??'
               PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF))
            ELSE
               PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF/P(I,J,K)))
            ENDIF
          ENDIF
 12     CONTINUE
 10   CONTINUE

C
C  Red points, odd z-planes.
C
      DO 20, K = 3, KMAX-1, 2
        DO 21, J = 3, JMAX-1, 2
        DO 21, I = 2, IMAX-1, 2
          IF ((PLABFS(I,J,K) .EQ. 1) .OR. (PLABFS(I,J,K) .EQ. 3)) THEN
            DIFF = - P(I,J,K) + DIV(I,J,K)
     &             - CXL(I,J,K) * P(I-1,J,K) - CXR(I,J,K) * P(I+1,J,K)
     &             - CYL(I,J,K) * P(I,J-1,K) - CYR(I,J,K) * P(I,J+1,K)
     &             - CZL(I,J,K) * P(I,J,K-1) - CZR(I,J,K) * P(I,J,K+1)
            P(I,J,K) = P(I,J,K) + OMEGA * DIFF
            PRIVNORM = PRIVNORM + DIFF * DIFF
            IF (ABS(P(I,J,K)).LT.SMALL) THEN
               print*,'ZERO PRESSURE ??'
               PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF))
            ELSE
               PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF/P(I,J,K)))
            ENDIF
          ENDIF
 21     CONTINUE

        DO 22, J = 2, JMAX-1, 2
        DO 22, I = 3, IMAX-1, 2
          IF ((PLABFS(I,J,K) .EQ. 1) .OR. (PLABFS(I,J,K) .EQ. 3)) THEN
            DIFF = - P(I,J,K) + DIV(I,J,K)
     &             - CXL(I,J,K) * P(I-1,J,K) - CXR(I,J,K) * P(I+1,J,K)
     &             - CYL(I,J,K) * P(I,J-1,K) - CYR(I,J,K) * P(I,J+1,K)
     &             - CZL(I,J,K) * P(I,J,K-1) - CZR(I,J,K) * P(I,J,K+1)
            P(I,J,K) = P(I,J,K) + OMEGA * DIFF
            PRIVNORM = PRIVNORM + DIFF * DIFF
            IF (ABS(P(I,J,K)).LT.SMALL) THEN
```

```
            print*,'ZERO PRESSURE ??'
            PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF))
          ELSE
            PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF/P(I,J,K)))
          ENDIF
        ENDIF
 22   CONTINUE
 20 CONTINUE

C
C  Black points, even z-planes.
C
      DO 30, K = 2, KMAX-1, 2
        DO 31, J = 3, JMAX-1, 2
        DO 31, I = 2, IMAX-1, 2
          IF ((PLABFS(I,J,K) .EQ. 1) .OR. (PLABFS(I,J,K) .EQ. 3)) THEN
            DIFF = - P(I,J,K) + DIV(I,J,K)
     &             - CXL(I,J,K) * P(I-1,J,K) - CXR(I,J,K) * P(I+1,J,K)
     &             - CYL(I,J,K) * P(I,J-1,K) - CYR(I,J,K) * P(I,J+1,K)
     &             - CZL(I,J,K) * P(I,J,K-1) - CZR(I,J,K) * P(I,J,K+1)
            P(I,J,K) = P(I,J,K) + OMEGA * DIFF
            PRIVNORM = PRIVNORM + DIFF * DIFF
            IF (ABS(P(I,J,K)).LT.SMALL) THEN
              print*,'ZERO PRESSURE ??'
              PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF))
            ELSE
              PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF/P(I,J,K)))
            ENDIF
          ENDIF
 31     CONTINUE

        DO 32, J = 2, JMAX-1, 2
        DO 32, I = 3, IMAX-1, 2
          IF ((PLABFS(I,J,K) .EQ. 1) .OR. (PLABFS(I,J,K) .EQ. 3)) THEN
            DIFF = - P(I,J,K) + DIV(I,J,K)
     &             - CXL(I,J,K) * P(I-1,J,K) - CXR(I,J,K) * P(I+1,J,K)
     &             - CYL(I,J,K) * P(I,J-1,K) - CYR(I,J,K) * P(I,J+1,K)
     &             - CZL(I,J,K) * P(I,J,K-1) - CZR(I,J,K) * P(I,J,K+1)
            P(I,J,K) = P(I,J,K) + OMEGA * DIFF
            PRIVNORM = PRIVNORM + DIFF * DIFF
            IF (ABS(P(I,J,K)).LT.SMALL) THEN
              print*,'ZERO PRESSURE ??'
              PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF))
            ELSE
              PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF/P(I,J,K)))
            ENDIF
          ENDIF
 32     CONTINUE
 30   CONTINUE

C
C  Black points, odd z-planes.
C
      DO 40, K = 3, KMAX-1, 2
        DO 41, J = 2, JMAX-1, 2
        DO 41, I = 2, IMAX-1, 2
          IF ((PLABFS(I,J,K) .EQ. 1) .OR. (PLABFS(I,J,K) .EQ. 3)) THEN
```

```
          DIFF = - P(I,J,K) + DIV(I,J,K)
     &           - CXL(I,J,K) * P(I-1,J,K) - CXR(I,J,K) * P(I+1,J,K)
     &           - CYL(I,J,K) * P(I,J-1,K) - CYR(I,J,K) * P(I,J+1,K)
     &           - CZL(I,J,K) * P(I,J,K-1) - CZR(I,J,K) * P(I,J,K+1)
          P(I,J,K) = P(I,J,K) + OMEGA * DIFF
          PRIVNORM = PRIVNORM + DIFF * DIFF
          IF (ABS(P(I,J,K)).LT.SMALL) THEN
              print*,'ZERO PRESSURE ??'
              PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF))
          ELSE
              PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF/P(I,J,K)))
          ENDIF
        ENDIF
 41   CONTINUE

      DO 42, J = 3, JMAX-1, 2
      DO 42, I = 3, IMAX-1, 2
        IF ((PLABFS(I,J,K) .EQ. 1) .OR. (PLABFS(I,J,K) .EQ. 3)) THEN
          DIFF = - P(I,J,K) + DIV(I,J,K)
     &           - CXL(I,J,K) * P(I-1,J,K) - CXR(I,J,K) * P(I+1,J,K)
     &           - CYL(I,J,K) * P(I,J-1,K) - CYR(I,J,K) * P(I,J+1,K)
     &           - CZL(I,J,K) * P(I,J,K-1) - CZR(I,J,K) * P(I,J,K+1)
          P(I,J,K) = P(I,J,K) + OMEGA * DIFF
          PRIVNORM = PRIVNORM + DIFF * DIFF
          IF (ABS(P(I,J,K)).LT.SMALL) THEN
              print*,'ZERO PRESSURE ??'
              PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF))
          ELSE
              PRIVMAXDIFF = DMAX1(PRIVMAXDIFF,ABS(DIFF/P(I,J,K)))
          ENDIF
        ENDIF
 42   CONTINUE
 40   CONTINUE

      DELTA = SQRT(NORM)

      RETURN
      END
```

Bibliografie

[1] Gegevens Huygens. https://subtrac.sara.nl/userdoc/wiki/huygens/description.

[2] OpenMP website. http://www.openmp.org.

[3] J.C. Feitsma. MPI parallelization of the Poisson solver in ComFlo. 2008. Master Thesis, Rijksuniversiteit Groningen, http://www.math.rug.nl/~veldman/Scripties/Feitsma-Master.pdf.