V*O*Rtech Computing

Experts in Technisch Rekenwerk

Postbus 260
2600 AG DELFT

tel. 015-285 0125
fax. 015-285 0126
vortech@vortech.nl

MEMO	EV/M09.024
Datum	8 januari 2010
Auteur(s)	Edwin Vollebregt en Stef Hummel
Onderwerp	Parallel rekenen in OpenDA en OpenMI

Documentinformatie

Versie	Auteur	Datum	Opmerkingen	Review
0.1	EV	06-04-2009	Eerste opzet	
Bestandslokatie:		/v3/E05q_bo_simona/c88648-geo-ref-waqomi/discussie-arch		

1 Inleiding

De software-architectuur die in OpenDA en OpenMI wordt gebruikt is *componentengebaseerd*. Dat wil zeggen dat numerieke modellen die men in een OpenDA- of OpenMI-*applicatie* wil gebruiken zich moeten houden aan een voorgeschreven *interface* voor de zogenaamde *model-component*. Hierbij is een component niet meer dan een omvangrijk *object* uit de objectoriëntatie-technologie. OpenDA en OpenMI bieden zelf een generieke infrastructuur, waarmee verschillende model- en andere componenten in applicaties kunnen worden gecombineerd.

Bij de verdere uitwerking van OpenDA en OpenMI is parallel rekenen[1] een belangrijk aandachtspunt. Beide systemen zijn mede op grootschalige numerieke modellen gericht waarin parallel rekenen belangrijk is. Op dit moment wordt dit nog niet volledig ondersteund in OpenDA en OpenMI. Een moeilijkheid zit in het uitbreiden van het gehanteerde conceptuele model. Wat *is* een parallelle modelcomponent? Welke vragen kun je hem stellen? En op welke manier en met welke computerprocessen worden deze vragen vervolgens ingevuld?

In het kader van change c88648 van het SIMONA B&O-contract wordt geprobeerd deze vragen te beantwoorden. Stef Hummel en Edwin Vollebregt werken samen een kleine testcase uit aan de hand waarvan de problematiek wordt geïllustreerd en waarvoor mogelijke oplossingen kunnen worden uitgeprobeerd. Hieruit zou een geschikte route voor uitbreiding van OpenDA en OpenMI moeten kunnen worden gedestilleerd. In dit memo doen we hierover verslag.

[1] Naast parallel rekenen zijn ook domein decompositie en andere vormen van gekoppelde berekeningen van belang. Omwille van de compactheid wordt dit alles samen met parallel rekenen aangeduid.

2 Probleemstelling

We stellen ons een (OpenDA of OpenMI) applicatie voor die bestaat uit een hoofdprogramma en twee modelcomponenten. Het hoofdprogramma instantiëert beide modelcomponenten ieder één keer. De modelcomponenten zijn beiden grid-gebaseerd en gebruiken allebei parallel rekenen[2]. Vervolgens wordt via het hoofdprogramma een berekening in gang gezet. Bijvoorbeeld wordt via een "trigger" om de oplossing op het eindtijdstip gevraagd. In de berekening zijn de twee modelcomponenten om de beurt actief en vragen ze van elkaar gegevens op.

De vraag is hoe dit gedrag, dat voor sequentiële berekeningen al goed is uitgewerkt, ook in parallelle berekeningen kan worden geïmplementeerd. En wel zodanig dat er geen sequentiële bottle-neck wordt geïntroduceerd. De eenvoudige oplossing zou namelijk zijn om alle communicatie tussen de twee modellen te laten verlopen via master-modelcomponenten die alle uit te wisselen data verzamelen, aggregeren en doorsturen naar de andere model-component. Dat leidt echter tot een onacceptabele overhead wanneer er frequent hele velden worden verstuurd, zoals bijvoorbeeld in het RRSQRT Kalman filter gebeurt.

3 Parallel rekenen in Costa

In Costa is parallel rekenen ten dele goed uitgewerkt. Hier roept het hoofdprogramma (de Costa workbench) een sequentiëel filter-algoritme aan, worden er door het filter-algoritme meerdere model-instantiaties gecreëerd, en zijn er generieke voorzieningen waarmee parallel rekenen per model-instantiatie kan worden gebruikt. We gebruiken dit als voorbeeld voor hoe een en ander onder water kan worden geïmplementeerd, en voor het goed helder krijgen van de gebruikte terminologie.

In een parallelle berekening met Costa wordt de Costa-workbench executable via MPI op meerdere processoren tegelijk opgestart. Een van de executables gedraagt zich als master en voert het hoofdprogramma uit. De andere executables gedragen zich als workers en gaan in object-aanroep-mode staan. Dat wil zeggen: ze wachten af wat de master hen vraagt te doen, en die vragen betreffen objecten die lokaal worden aangemaakt en die met lokale data berekeningen doen. De in- en outputs van de vragen aan ieder object worden via MPI met het master-proces gecommuniceerd. Hiervoor kunnen plain data-arrays en ook objecten zoals een Costa treevector via MPI worden overgestuurd van het ene naar het andere proces.

Het abstracte model dat hierin wordt gehanteerd bestaat uit *distributed memory*, uit objecten die ieder onverdeeld ergens in een enkel lokaal geheugen bestaan, en uit handles naar andere objecten waaraan ("over het netwerk") opdrachten kunnen worden gegeven en vragen kunnen worden gesteld.

Dit model werkt goed voor een sequentiëel hoofdprogramma. Een model-component kan namelijk gemakkelijk uit een master-model en worker-modellen worden opgebouwd. Het

[2]Het gaat hierbij vooral om parallellisatie via aparte rekenprocessen, gekoppelde berekeningen; shared memory parallellisatie via OpenMP en data parallel rekenen leiden niet tot moeilijkheden voor OpenDA en OpenMI.

filter-algoritme instantiëert de master-modelcomponent. Deze instantiëert vervolgens remote worker-modelcomponenten. Iedere vraag die het filter-algoritme aan de master stelt, speelt deze direct naar de workers door. En de resultaten per worker worden eenvoudig in een hierarchische structuur samengevoegd en aan het filter-algoritme geretourneerd. Het filter-algoritme praat gewoon tegen een enkel "sequentiëel" model, en onder water worden de berekeningen parallel uitgevoerd.

Dit abstracte model kent twee beperkingen. Een praktische beperking is dat er nog niet zo veel vragen aan de worker-objecten kunnen worden gesteld. Je zou de model-states van twee model-instantiaties in de worker-processen van elkaar willen kunnen aftrekken, in plaats van dat deze bewerking in het master-proces wordt gedaan. Een meer conceptuele beperking is dat de aansturing van de worker-objecten vanuit een enkel master-object wordt gedaan. Deze vormt dan al gauw een sequentiële bottle-neck.

4 Gedistribueerde objecten

Een logische uitbreiding van het hierboven geschetste abstracte model is om een groep van een master-object plus meerdere worker-objecten samen te beschouwen als een "gedistribueerd object". Dit is iets wezenlijk anders dan een gewoon object. Met name is niet een-twee-drie duidelijk wat voor operaties je op zo'n gedistribueerd object kunt definiëren en hoe die worden geïmplementeerd. Hoe geef je de handle van zo'n gedistribueerd object weer en moeten alle sub-objecten altijd tegelijk hetzelfde doen of niet?

We kunnen vooralsnog uitgaan van collectief gedrag. Alsof verzoeken aan een gedistribueerd object steeds tegelijk aan alle betrokken sub-objecten worden doorgespeeld. Een SPMD-aanpak, single-program-multiple-data, alle sub-objecten van een gedistribueerd object voeren hetzelfde programma uit. Hierin kunnen gemakkelijk afspraken worden gemaakt over het aanroepen van andere objecten; zulke verzoeken hoeven door slechts een van de sub-objecten te worden gedaan. Het doorspelen van de handles van de sub-objecten lijkt ook niet het moeilijkste. Ergens onder water kan een lijstje worden aangelegd dat van een handle naar een gedistribueerd object naar de handles van de sub-objecten vertaalt.

De cruciale vraag is nu hoe we de data-stromen tussen twee gedistribueerde objecten kunnen regelen. Hoe krijgen we data van het grid van de ene groep op het gedistribueerde grid van de andere groep, zonder dat er tussendoor een globale data-structuur wordt gebruikt?

5 Uitwerking van communicatie voor een eenvoudige test-case

Stel dat het hoofdprogramma twee parallelle modelcomponenten instantiëert, "WAQUA" en "SWAN". Dit zijn gedistribueerde objecten. Het hoofdprogramma speelt dan, op de een of andere manier, de handles van deze objecten over en weer door, en roept bijvoorbeeld de SWAN-component aan.

SWAN weet dat hij de waterstand nodig heeft op zijn eigen grid. Hij maakt hiervoor een "gridfunctie" `swan_wl` aan, een gedistribueerd object, met kennis van het rooster van SWAN

en met een scalaire waarde per roosterpunt voor de in te vullen waterstand.

SWAN komt in zijn algoritme bij de stap "verkrijg de waterstand". Hierbij is onduidelijk hoe dit precies moet worden ingevuld. Misschien kan hij aan `swan_wl` de opdracht geven "ververs jezelf", anders roept hij een van zijn eigen subroutines aan met `swan_wl` als subroutine-argument.

In het verkrijgen van de waterstand is - op de een of andere manier - bekend dat deze door WAQUA geleverd wordt. WAQUA heeft ook een gedistribueerd object voor de waterstand: `wq_wl`. Voor het gemak nemen we aan dat de roosters van WAQUA en SWAN samenvallen, maar dat ze wel verschillend over de sub-objecten kunnen zijn gepartitioneerd.

Misschien is er een routine `waqua.get_handle("waterstand")`. Hiermee verkrijgt SWAN de handle van het object `wq_wl`. Let op dat "de handle" nog niet helder is gedefiniëerd, en dat ook "SWAN" een gedistribueerde entiteit betreft. Dit moet verder worden uitgewerkt.

We zijn nu zo ver dat het gedistribueerde object `swan_wl` aan het gedistribueerde object `wq_wl` kan vragen: "stuur me de waterstanden". Deze routine willen we invullen. De invulling is afhankelijk van de omstandigheden. Er zijn tenminste drie factoren van belang:

1. Heeft SWAN weet van parallel rekenen, kan hij met meerdere *peers* praten en data verzamelen of niet? Als SWAN niet op parallel rekenen is voorbereid dan moeten de data aan de WAQUA-kant worden samengevoegd.

2. Idem voor WAQUA; als WAQUA sequentiëel rekent dan moeten de data door de SWAN-master worden gedistribueerd.

3. Leven de verschillende sub-objecten in hetzelfde of in verschillende rekenprocessen? Kan een SWAN-routine een functie van een WAQUA-object direct aanroepen of alleen indirect via MPI?