# Verslag eindopdracht parallelle algoritmen

Albert-Jan Yzelman
Robin Zeeman

# *Inhoudsopgave*


**INHOUDSOPGAVE** 2
**INLEIDING** 3
**PROBLEEMBESCHRIJVING** 4
**UITLEG PROGRAMMA** 5
**ANALYSE** 8
*Communicatie analyse m2dson.c* 9
*Rekenkosten analyse m2dson.c* 10
*Communicatie analyse m3dson.c* 11
*Rekenkosten analyse m3dson.c* 12
**EXPERIMENTEN** 16
*Opmerkingen* 16
**CONCLUSIE** 17

# *Inleiding*

In het vak Parallelle Algoritmen hebben we de basisprincipes van, parallel programmeren geleerd en deze kennis hebben we in praktijk gebracht in deze eindopdracht. Het programmeren gebeurde in C, met behulp van het programmeermodel BSP (Bulk Synchronous Parallel). Dit model zorgt ervoor dat we parallel kunnen programmeren; het vult het gat tussen parallelle hardware en software. Het leren van het BSP-model als taal op zich lijkt niet al te moeilijk, het implementeren ervan is echter een ander verhaal.
Een parallel algoritme is namelijk gemaakt om op meerdere processoren tegelijk te lopen, wat essentieel anders is dan het 'gewone' sequentieel programmeren. Er dienen zich vele extra problemen aan, waar communicatieproblemen waarschijnlijk veruit de grootste zijn. Afgezien dat deze communicatie vaak niet gelijk vlekkeloos verloopt, streven we er ook nog naar deze communicatie tot een minimum te beperken, en zo onze processoren maximaal te benutten.

Op de komende pagina's vindt U een beschrijving, uitleg, en een analyse van twee parallelle algoritmes. Afgehandelde problemen en de efficiëntie van het algoritme worden uitvoerig doorgelicht, en suggesties voor verbeteringen zijn ook vermeld. Al met al valt in ieder geval te constateren dat deze algoritmes het sneller doen dan een sequentieel programma, wat op zich al een bevredigend resultaat is.

Robin Zeeman & Albert-Jan Yzelman
Januari 2004

## *Probleembeschrijving*

De opdracht is om matrixvermenigvuldiging parallel te programmeren, waarbij het uitvoeren van het programma zo kort mogelijk moet zijn. Dat wil dus zeggen: zo efficiënt mogelijk communiceren en rekenen.

Voor uitrekenen van het matrixproduct AB = C, waarbij A, B en C n x n-matrices voorstellen, maken we gebruik van de volgende methode:

1) verdeel de matrices A en B in deelmatrices ter grootte n/q x n/q (q | n)
2) deelmatrix C(s,t) wordt vervolgens berekend door C(s,t) = sum_(u=0)^(q-1) A(s,u)B(u,t) voor 0<=s, t<q
3) vervolgens plakken we de deelmatrices van C weer aaneen tot C zelf.

***Uitleg programma***

Om een programma te schrijven die matrices vermenigvuldigt, leek het ons handig om eerst in ieder geval een paar matrixbewerkingen te programmeren in een aparte module. Handig is om op te merken dat we alleen werken met vierkante matrices, iets dat het programmeren ietwat simpeler maakt. Het resultaat is de module squarepack, een complete listing hiervan is verderop dit verslag te vinden. We zullen hier alleen vermelden welke relevante functies er in zijn gedefinieerd en we zullen de definitie laten zien van de vermenigvuldigroutine, aangezien deze een primaire rol vervult in onze algoritmen. Laten we daar ook maar gelijk mee beginnen:

*Squarepack.c — matrixvermenigvuldigingsroutine — verm(x, y ,n)* Parallel of niet, deze routine zul je altijd nodig hebben bij vermenigvuldiging.

Verschil is natuurlijk wel dat deze routine eigenlijk praktisch het hele sequentiële programma is; en maar een klein deel van zijn parallelle broertje. Om een matrix te vermenigvuldigen hebben we als variabelen natuurlijk 2 matrices (x en y) nodig, en handig zou ook zijn de grootte van die matrices (n). De routine ziet er dan als volgt uit:

```
double **verm(double **x, double **y, int n){ double **r){
        double **r;
        int a,i,j;

        r=matallocd(n,n);

        for(i=0; i<n; i++){
                for(j=0; j<n; j++){
                        r[i] [j]=x [i][0] * y[0][j];
                        for(a=1; a<n; a++){
                                r[i][j] += (x[i][a]) * (y[a][j]);
                                }
                        }
                }
return r;
}
```

Een matrix in c valt te programmeren als een ‘dubbele’ rij. Een rij in c definieer je bijvoorbeeld met:

int *x**;**

Dit heeft tot gevolg dat men toegang kunt krijgen tot delen van x door x[0], x[1], enz. op te vragen. De waardes in x[0], x[1],.. zijn hier allemaal integers. Na een definitie moet er nog op gelet worden dat de variabele x nog *gealloceerd* moet worden; een stuk geheugen van een bepaalde grootte moet nog aan x worden toegewezen. Logischerwijs bepaalt dit ook gelijk de grootte van x.

Een matrix definiëren kon dus door 2 rijen te gebruiken, op de volgende manier:

double **x;

Toegang krijg je dan door x[a] [b] te schrijven. Ook dit moet gealloceerd worden, dit kan met behulp van de methode matallocd, gedefinieerd in de module bspedupack geschreven door Dr. Bisseling. Het alloceren gebeurt door middel van het handig gebruiken van pointers, jets waar we nu niet op zullen ingaan. Na het declareren en alloceren kunnen we de matrix gaan vullen met waarden; in onze vermenigvuldigingsroutine wordt dit gedaan in de 2 for-loops. De code die er staat implementeert vrij direct de bekende methode voor matrixvermenigvuldiging. Een kleine optimalisering is te vinden waar de $_3$e for-loop niet bij

a=0 begint, maar bij a=l. Dit maakt het mogelijk net voor de 3e loop r[i][j] de waarde te geven voor a=0. Was dit niet zo gedaan, dan had er in plaats daarvan r[i][j]=0 gestaan, en dat zou dus net 1 stapje meer rekenwerk zijn geweest voor de computer. Het ziet er misschien uit als een hele kleine verbetering, maar als men bijvoorbeeld werkt met 1000 bij 1000 matrices, dan scheelt het al 10002 rekenstapjes. De routine eindigt met “return r;”, wat het gevonden resultaat dus terug geeft.

*Squarepack.c — random matrices genereren — genmat(n)*

Wat ook handig is in ons geval is een methode die snel een matrix bedenkt die we mogen vermenigvuldigen. We maken hier gebruik van de ingebouwde functie van c om random getallen te geven. Hier komen vaak extreem grote getallen uit, wat op zich niet veel uitmaakt. Echter in het begin van het programmeren en tijdens het debuggen wilden we de matrices nog wel eens op het scherm printen; zulke hoge getallen maken dat de matrices scheef in het beeld kwamen. Dit was de reden dat we in de functie de randomgetallen met rest bij deling door 20 gebruiken; alle matrices hebben dus waardes onder de 20.

*Squarepack.c — matrices vergelijken — equals(x,y,n)*

Om te kijken of een uitkomst wel klopt, kunnen we ervoor kiezen ons programma ook het sequentieel algoritme te volgen op processor 0, waar ook de originele 2 matrices nog staan. Na berekening van het sequentiële deel moeten de 2 matrices met elkaar vergeleken worden of ze wel met elkaar stroken. Daarvoor is deze functie gemaakt; in de listing ziet men dat het een simpele procedure betreft die 1 teruggeeft als het strookt, en 0 als het dat niet doet. Een leuke tweak zou nog zijn om in plaats van “r=0” gelijk “return 0” te zetten; de procedure houdt dan gelijk op en hoeft de loop dus niet af te maken. Maar de ervaring is dat deze methode eigenlijk ook al snel zat is, wij vonden zelfs verrassend -snel; computers houden blijkbaar erg van vergelijken...

*Squarepack.c — matrices optellen — som(x,y,n)*

In beide algoritmen is het nodig om deelmatrices bij elkaar op te tellen. Deze methode handelt het optellen af. Dit gebeurt vrij direct met 2 for-loops en door gewoon 2 elementen op te tellen. Een verbetering zoals in verm(x,y,n) is hier niet nuttig; er worden immers direct waardes toegekend aan r[a][b], en niet een waarde erbij opgeteld zoals bij het vermenigvuldigen.

*Squarepack.c — matrices kopiëren — copy(x,n), copyin(x,y,n)*

Als we een matrix willen gelijkstellen aan een andere (x=y), dan gebeurt dat in c niet helemaal zoals we altijd willen. Het is namelijk zo dat als c inderdaad een commando x=y zou krijgen, waar x en y dus matrices zijn die zoals hierboven waren gedefinieerd en gealloceerd, dan *point* x gewoon naar y. Met andere woorden, x IS écht y geworden. Dit noemen we ook wel een dynamische toekenning, wat nu dus inhoudt dat als de matrix y verandert, x ook mee verandert. Niet altijd even handig dus. Eerst schreven we om dit op te vangen de methode copy(x,n); als je naar de listing kijkt lijkt het heel goed te doen wat we wilden; een statische toewijzing bewerkstelligen. Dit gebeurt ook wel, maar na wat nadenken zagen we in dat je als je zeg maar y=copy(x,n) intypt, y alsnog zit te pointen naar een ‘andere’ matrix; copy(x,n) heeft gewoon een hele nieuwe matrix gemaakt gelijk aan matrix x, en y=copy(x,n) laat y pointen naar die matrix. Normaal gesproken is het eigenlijk geen probleem, behalve dat het wat geheugen verspilt. Maar in ons programma zorgde dit voor behoorlijk raar gedrag. Om dit te begrijpen zullen we eerst wat moeten vertellen over hoe verschillende processoren met elkaar communiceren in BSP.

BSP heeft 2 simpele procedures om te communiceren:
bsp_get(pid, source, offset, dest, nbytes); en

bsp_put(pid, source, dest, offset, nbytes);

Zoals we weten werken we met meerdere processoren. Die processoren hebben allemaal een nummer gekregen, en die kunnen we hier gebruiken voor pid (processor-id). Een processornummer vraag je in het programma op door gebruik te maken van de functie bsp_pid();. Source staat voor bron, en dest voor bestemming (destination). nbytes stelt de grootte voor van de variabele die ge-get of ge-put gaat worden. Zoals de namen doen vermoeden 'haal' je een variabele uit een processor met bsp_get en stop je een waarde in een processor met behulp van bsp_put. Het zou mooi zijn als je bij source en dest gewoon simpel variabelen kon zetten. Helaas is dit niet waar; source en dest vragen beide om *geheugenlocaties*, ofwel *pointers.* Offset geeft aan hoeveel je 'naast' die geheugenlocaties wil zitten; dit kan handig zijn in sommige situaties.

Verder zit er nog een addertje onder het gras. Het communiceren met get en put kan alleen als de geheugenlocaties voor beide processoren voor elkaar bekend zijn. Eerst moet er geregistreerd worden met behulp van bsp_push_reg(variable, nbytes);. Dit zorgt ervoor dat iedere processor van elke andere processor weet waar bij hun de variabele is opgeslagen in het geheugen, en wat de lengte ervan is. Deze variabele is trouwens wederom een pointer. -

Het is gewoonlijk om in een parallel programma (met BSP als model) aan het begin alle variabelen te registreren waarmee je later wilt gaan communiceren. Dit houdt dus in dat je ook de pointers in het begin vaststelt, en later diezelfde pointers gebruikt voor communicatie. Dit werkt allemaal prima, totdat je het hiervoor besproken copy(x,n);procedure gebruikt...

Want wat gebeurt er dan? De oude pointer die in het begin is gemaakt point naar een plaats in het geheugen waar ooit matrix y stond. Het stukje geheugen ter grootte van nbytes wat begon op die plaats wordt ge-put of ge-get. Maar op die plaats staan helaas sinds die copy-opdracht alleen nog maar pointers. Pointers naar plaatsen die op de andere processor geen interessante gegevens bevatten. Op deze manier lijkt de communicatie dus te mislukken, een probleem waar we een tijdje mee bezig waren geweest. Na constatering van dit probleem was een oplossing snel gevonden, in de vorm van copyin(x,y,n);. Deze functie kopieert alle waardes van x in y, en deze keer is y=x dus eindelijk goed geïmplementeerd.

# Analyse

Parallelle programma's kunnen op communicatie en rekentijd worden geanalyseerd. Dit wordt gebruikt om te bekijken hoe algoritmes zich gedragen als je met invoerwaardes speelt, en in ons geval kan het helpen bepalen welk algoritme handiger is met bepaalde invoerwaardes.

We hebben 4 analyses opgesteld, voor beide algoritmes een communicatieanalyse en een rekentijdanalyse. Elke analyse bekijkt eerst per superstep wat de zgn. 'cost' van een opdracht is. Als we bijvoorbeeld een loop hebben, laten we zeggen eentje van 0 tot p (aantal processoren), waar in de body van de loop een put staat voor een n bij n matrix, dan is de cost gelijk aan: p*n*n. Er wordt p keer een n bij n matrix doorgegeven. De eenheid is natuurlijk niet erg duidelijk gedefinieerd; de snelheid van de communicatie hangt sterk van het systeem af. Het kan echter wel goed verhoudingen weergeven, en meer vragen we in principe ook niet. Voor rekentijdanalyse wordt een cost op eenzelfde manier bepaald. Stel we hebben dezelfde loop, maar in die loop vinden 2 bewerkingen plaats: 1 optelling en 1 vermenigvuldiging. We tellen deze bewerkingen als 1 rekeneenheid, en dus wordt de cost: 2p.

Bij de laatste analyse staat ook nog een kleine voorlopige conclusie, die we later in de experimenten proberen te bevestigen. Na de experimenten volgt de definitieve conclusie.

Communicatie analyse m2dson.c

Superstep 1:

$0 < d < p$ loop, 2 puts in de loop, totale cost:
$(p-1)*2*n*n = 2 * n^2 * (q^2 - 1)$

Superstep 3:

$0 \leq d < p$ loop, 1 put in de loop, totale cost:
$q^2 *(n/q)*(n/q) = n^2$

---

Totaal communicatie:

$$n^2 * ( 2 * ( q^2 - 1 ) + 1) = n^2 ( 2q^2 - 1 )$$

---

Stel we nemen bijvoorbeeld n=100 en q=2. De totale communicatie wordt dan:
70000 tijdseenheden. Verdubbelen we n, dan verviervoudigt de communicatietijd.
Verdubbelen we q, dan verviervoudigt de communicatietijd ook; het verdubbelt zelfs iets meer voor lage q. Dit komt doordat voor relatief lage q de –1 een welkome rol speelt. De rol is echter snel vergeten als q vergroot wordt; dan wordt de communicatie nagenoeg gelijk is met $2q^2n^2$.

Rekenkosten analyse m2dson.c

Superstep 1:

Deelalgoritme 2 keer:
Dit algoritme (zowel deelmrij als deelmkol) doorloopt q keer de deelm-functie. Deze functie doorloopt $(n/q)^2$ een bewerking. De totale cost ligt dus op:
$2 * q* (n/q)^2 = 2 * n^2 / q$

Superstep 2:

2 keer Vermenigvuldigen:
We vermenigvuldigen hier deelmatrices. Het algoritme bevat dus effectief 3 loops ter lengte van (n/q). Cost: $2*(n/q)^3$

q-1 keer optellen:
Er worden matrices bij elkaar opgeteld. De functie som bevat 2 loops, in dit geval ter lengte van (n/q). Cost: $(q-1)*(n/q)^2$

De totale cost ligt dus op:
$(n/q)^2 * ( 2 * (n/q) * (q-1) ) \leq (n/q)^2 * 2n = (2*n^3) / (q^2)$

Superstep 3:

Construct:
De construct methode doorloopt 2 loops ter lengte n. Totale cost:
$n^2$

---

Totale rekenkosten:

$n^2 * ( 2 / q + 2n / q^2 + 1 )$

---

We zien dat dit ver onder de $n^3$ ligt voor het sequentiële algoritme. Gebruiken we bijvoorbeeld q=2 en n=100:
Sequentieel = 1000000, en
m2dson.c = 10000 * ( 1 + 50 + 1) = 520000

Het parallelle rekenwerk ligt dus al op bijna de helft dan dat voor het sequentiële rekenwerk! Maken we n wat groter, laten we zeggen, n=500; dan wordt het verschil nog groter:
Sequentieel = 125.000.000, en
m2dson.c = 63.000.000

Dit ligt dus nog meer richting de helft van het sequentiële deel. We kunnen natuurlijk ook gewoon de vergelijking $n^3 = n^2 * ( 2 / q + 2n / q^2 + 1 )$ oplossen. Als we aan beide kanten niet schuiven (het gaat ons tenslotte om de verhouding) dan zien we dat het gelijk staat aan:

$1 = 2/(q*n) + 2/q^2 + 1/n.$

Als n groter wordt zien we dat 2/(qn) en 1/q snel klein worden. De rekentijd is dus, voor grote matrices, een factor $2/q^2$ kleiner dan de sequentiële rekentijd.

Communicatie analyse m3dson.c

Superstep 0:

$0 \leq s,t,u < q$ loop, 2 puts in de loop, totale cost:
$q^3*2*(n/q)*(n/q) = 2 * n^2 * q$

Superstep 1:

$0 < u < q$ loop, 1 put in de loop, totale cost:
$(q-1)*(n/q)*(n/q) = n^2 * (q-1)/(q^2)$
*wat voor grote q ongeveer gelijk is aan: $(n^2)/q$*

Superstep 2:

$0 \leq s,t < q$ loop, 1 put in de loop, totale cost:
$q^2 * (n/q) * (n/q) = n^2$

---

Totaal communicatie:

$$(2 * n^2 * q + (q-1)/(q^2) * n^2 + n^2 ) = n^2 * ( 2q + (q-1)/(q^2) +1)$$

---

Stel we nemen bijvoorbeeld n=100 en q=2. De totale communicatie wordt dan:
85000 tijdseenheden. Verdubbelen we n, dan verviervoudigt de communicatietijd.
Verdubbelen we q, dan verdubbelt de communicatietijd ongeveer; het verdubbelt iets minder voor lage q. Dit komt doordat voor grote q de communicatie nagenoeg gelijk is aan $2q * n^2$.
Voor relatief lage q spelen de + 1 en zelfs de breuk $(q-1)/(q^2)$ een tijdsrovende rol, een rol die natuurlijk verergert met grote n.
Dit alles staat natuurlijk in schril contrast met m2dson; een grote q heeft daar een kwadratisch gevolg voor de communicatietijd, terwijl hier de tijd ‘maar’ verdubbelt. m3dson lijkt dus handiger als er veel processoren beschikbaar zijn, en het is voor beide algoritmes leuker als er grote matrices zijn. (In de zin dat de rekentijd t.o.v. een sequentiële methode steeds lager wordt). We hopen dit te kunnen bevestigen met experimenten.

Rekenkosten analyse m3dson.c

Superstep 0:

Deelalgoritme $2 * q^3$ keer:
Dit algoritme doorloopt $2q^3$ keer de deel-functie. Deze functie doorloopt $(n/q)^2$ een bewerking. De totale cost ligt dus op:
$3 * q^2 * (n/q)^2 = 3 * n^2$

NB: Deze kost ligt veel hoger dan m2dson, omdat hier p(0,0,0) het delen in zijn eentje doet.

Superstep 1:

1 keer Vermenigvuldigen:
We vermenigvuldigen hier deelmatrices. Het algoritme bevat dus effectief 3 loops ter lengte van (n/q). Cost: $(n/q)^3$

q-1 keer optellen:
Er worden matrices bij elkaar opgeteld. De functie som bevat 2 loops, in dit geval ter lengte van (n/q). Cost: $(q-1)*(n/q)^2$

De totale cost ligt dus op:
$(n/q)^2 * ( (n/q) * (q-1) ) \leq (n/q)^2 * n = n^3 / q^2$

Superstep 3:

Construct:
De construct methode doorloopt 2 loops ter lengte n. Totale cost:
$n^2$

---

Totale rekenkosten:

$n^2 * ( 3 + n / q^2 + 1 )$

---

Laten we een kleine vergelijking maken. Stel n=500:
Sequentieel = 125.000.000,
m2dson.c = 63.000.000, en
m3dson.c = 32.250.000

We zien dat m3dson weer bijna de helft is van m2dson qua rekenwerk! Dit was ook te zien aan de formules zelf, berekenen we de verhouding op een zelfde manier als bij m2dson voorgaand, dan zien we dat de rekentijd voor m3dson een factor $1/q^2$ zo klein is (voor grote n). Precies de helft van m2dson.

Nu we dit hebben behandeld, kunnen we makkelijker beginnen aan het beschrijven van de programma's. We beginnen met het programma voor 2-dimensionale matrixvermenigvuldiging. Alhoewel het leuk zou zijn het hele programma in detail te behandelen lijkt ons dit niet erg efficiënt, we beperken ons daarom tot een wat meer globaal niveau. Het programma (m2dson.c) genaamd, is op te delen in verschillende stappen. Na initialisatie van variabelen en het doorsturen van basisvariabelen zoals de grootte van de matrices en het getal q wat staat voor in hoeveel delen de matrices zijn ingedeeld, zijn er nog 3 superstappen, en daarna is er nog een sequentieel deel en een controle-deel, die optioneel uitgevoerd kunnen worden. Elke stap eindigt met een sync; een punt waarvandaan pas verder wordt gegaan als alle andere processoren ook bij dat punt zijn. Belangrijk is om te weten dat pas ná zo'n sync, alle puts en gets zijn uitgevoerd.

*Superstep 0:*
Dit is het deel dat de variabelen n en q doorstuurt naar alle andere processoren. Ook vinden in deze step de registraties plaats.

*Superstep 1:*
In deze stap maakt processor 0 willekeurig een matrix a en b. Deze matrices worden vervolgens in al hun totaliteit verstuurd naar alle andere processoren.

*Superstep 2:*
Zoals uitgelegd in het vorige hoofdstuk, heeft elke processor 2*q deelmatrices nodig; om $C_{(0,0)}$ uit te rekenen op p=0 bijvoorbeeld, moet de som worden berekend van $A_{(0,0)}*B_{(0,0)} + A_{(o,1)}*B_{(1,0)}$ (bij q=2). Handig leek het ons om hiervoor een rij van matrices te gebruiken; in de rij deela slaan we alle deelmatrices van matrix a op, die de processor nodig heeft. Een rij van matrices is op zich een raar ding; we definiëren het in C als:
double ***deela;
Het alloceren hiervan is ook weer een verhaal met pointers; na de methode matallocd van bspedupack te hebben bestudeerd was het niet al te moeilijk om te komen met een functie matrowallocd(n,q,p). Deze functie is nog heel gericht op deze specifieke toepassing, en vraagt ook expliciet naar de n,q,p die in het programma bekend zijn (p staat voor het aantal processoren). De functie alloceert vervolgens een rij ter lengte p, bestaande uit vierkante matrices ter grootte van (n/q).
Om het programma wat overzichtelijker te maken, en aangezien we deze matrowallocd en andere functies die te maken hebben met rijen van matrices, zijn al deze functies samengevoegd in matrow.c. Ook de functie die de matrices opdeelt en in deelmatrices terugstuurt zit in dat pakket. Echter, we hebben daar twee varianten van nodig, één voor het sturen van rijen van matrices (nodig voor matrix 'a') en eentje voor het sturen van kolommen van matrices (nodig voor matrix 'b'). De functies deelmrij en deelmkol hebben we hiervoor ontworpen, deze staan wel in m2dson.c. Ze hebben beide als invoer nodig de oorspronkelijke matrix (oa), de getallen n en q, en bun processornummer (s). Aan de hand van n,q en s bekijken deze functies hoe ze de matrix a resp. b moeten opknippen, en welke delen de processor nodig heeft. Deze delen worden vervolgens in een rij teruggegeven, in superstep 2 worden ze opgeslagen in deela en deelb.
Het voordeel om ze in die rijen op te slaan wordt gelijk erna duidelijk; om de som te berekenen van de vermenigvuldiging van de 2 matrixrijen, kunnen we een routine schrijven die zeer efficiënt alle vermenigvuldigingen optelt in een uitkomstmatrix (c). Het algoritme bevat dezelfde tweak als het vermenigvuldigingsalgoritme door eerst:

```
c=verm(deela[0] ,deelb [0] ,(n/q));
```

uit te rekenen, en daarna pas te sommeren met:

```
for(d=1; d<q; d++){
        c=som(c,verm(deela[d] ,deelb [d],(n/q)),(n/q));
        }
```

Na deze loop eindigt de superstep.

*Superstep 3:*

In deze stap worden alle uitkomstmatrices verstuurd naar processor 0, met behulp van puts. Op zich leek het ons logischer gets te gebruiken, maar dat bleek niet helemaal te lukken. Gelukkig maakt een put niks uit qua efficiëntie. We hebben ervoor gekozen de aparte uitkomsten uit de rijen te putten in een resrij op processor 0. Dit heeft als voordeel dat we een functie construct konden schrijven (in matrow.c) die vanuit een matrixrij met deelmatrices de echte matrix weer kan terugconstrueren. Ook zouden we zo gauw geen andere manier vinden om zoveel matrices gestructureerd weg te schrijven in het geheugen. De put werkt trouwens wel met een tussenmatrix; we kunnen niet direct in de resrij putten. Althans dat werkte niet helemaal toen het geprobeerd was. De geputte matrix wordt erna pas in een resrij gedaan, nog via een copy-opdracht. Het heeft niet zin cm het te veranderen naar copyin, aangezien dat het programma niet sneller maakt. Ook is het niet zeker of copyin goed overweg kan met delen uit matrixrijen.

Na deze communicatiestap begint processor 0 aan het construeren van de uitkomstmatrix via de functie construct(resrij,n,q);. Een beschrijving van die functie staat in matrow.c.

*Sequentieel Algoritme:*

Dit bestaat uit enkel 1 regel:
cc=verm(oa,ob,n);
Het resultaat cc wordt vervolgens gebruikt in de uitkomstcontrole:

*Controle:*

De controle maakt gebruik van de equals(x,y,n); functie in squarepack.c. Als de matrices overeenkomen wordt hiervan melding gedaan, en als ze niet overeenkomen evenzo. De uitkomstcontrole en het sequentiële deel worden overgeslagen, tenzij en aan het begin van het programma kenbaar is gemaakt dat deze stappen moeten worden doorlopen.

Dit concludeert het programma m2dson.c. Nu zullen we de 3d-variant gaan behandelen.
De 3d-variant lijkt erg veel op de 2d-versie. Het verschil ligt natuurlijk bij de processorverdeling; elke processor rekent een product uit, en daar houdt het bij op. Bij de 2d variant berekenden we nog een som van matrixvermenigvuldigingen. De processorcoördinaten hebben we in dit programma wat meer prominent naar voren gebracht; het product van $A_{(s,u)}*B_{(u,t)}$ wordt berekend door de processor P(s,t,u). Om dit coördinatenstelsel goed te gebruiken hebben we een functie pnr(s,t,u,q); gemaakt die het (s,t,u)-stelsel omrekent naar een ‘1-tallig’ stelsel. Of anders gezegd, een functie die aan een coördinaat (s,t,u) een processornummer toekent. We hebben q nodig omdat we weten dat elke coördinaat nooit groter wordt dan q. Op die manier kunnen we volgens $p = s + t*q + t*q^2$ aan elke coördinaat een unieke nummer toekennen.
Tijdens het programmeren bleek het ook handig te zijn om terug te rekenen; om van een processornummer tot het juiste coördinaat te komen. Hiertoe zijn de functies pcs, pct en pcu opgesteld (processor-coördinate-s,t,u).
Voor dit programma hebben we geen matrixrijen meer nodig om mee te rekenen; immers we

hoeven geen sommen meer te berekenen. Hiertoe is het deelalgoritme veranderd in eentje die uit de ‘a’ of ‘b’ matrix één deelmatrix pakt, afhankelijk van de processorcoördinaten.
De structuur van het programma blijft echter wel ongeveer hetzelfde; we zullen weer per superstep bekijken wat het programma precies doet.

*Superstep 0:*

Het registreren van variabelen en het doorsturen van n en q naar alle processoren is al gedaan. Rest en alleen nog in deze stap om 2 matrices te genereren en deelmatrices te versturen naar de andere processoren. Dit gebeurt ook anders dan in m2dson. Hier pakt processor 0 veer elke processorcoördinaat de bijbehorende deelmatrix en put die bij de bijbehorende processor. Dit gebeurt veer zowel de deelmatrix van a en van die van b. De sync staat in de loop omdat hij anders de verkeerde ta en tb in de processor put.

*Superstep 1:*

Hier worden de matrices bij elke processor met elkaar vermenigvuldigd. Dit gebeurt hier met behulp van copyin, omdat we anders problemen krijgen met het putten later. Na het vermenigvuldigen konden we natuurlijk gelijk alles sturen naar processor 0 en deze alles laten opsommen, maar we hebben gekozen voor wat optimalisatie, en zorgden dat alle processoren hun uitkomsten stuurden naar P(s,t,0). Vervolgens bepaalt die processor de som. Nu krijgen we dat processor P(s,t,0) dezelfde matrix bevat als die in de m2dson.c.

*Superstep 2:*

Logischerwijs leidt dit tot een stap die veel lijkt op superstep 3 in m2dson. Er is alleen een verschil dat er 2 for-loops worden gebruikt, dit met het oog op het gebruikte coördinatenstelsel. Voor de rest wordt alles weer in een resrij geput. Deze rij heeft de lengte van q*q, dezelfde lengte als in m2dson.c dus.

*Superstep 3:*

In deze superstep wordt de construct methode aangeroepen, net zoals in het vorige programma. Ook het *Sequentieel algoritme* en de *uitkomstcontrole* blijven hetzelfde. Mogelijke verbeteringen zitten hem waarschijnlijk in het verder opdelen van het sommeren en doorsturen voor erg grote matrices. Dit kan door, nadat alle uitkomsten van P(s,t,*) zijn opgeteld bij P(s,t,0), hetzelfde te doen veer P(s,*,0) en daarvan alle uitkomsten op te slaan in P(s,0,0). Vervolgens moet processor 0, ofwel P(0,0,0) alle matrices van P(*,0,0) samenstellen. Dit valt ook te gebruiken in de 2-dimensionale versie. Helaas zijn we en niet aan toegekomen deze verbeteringen te implementeren. Verder zijn er in superstep 1 en 2 heel veel sync’s gebruikt. Dit vanwege het probleem dat we met get hebben; het schijnt niet echt te willen functioneren met matrices. Als we gets konden gebruiken, dan hadden we bijvoorbeeld in superstep 1 een loop kunnen maken en alle deeluitkomsten laten getten, en in een matrixrij kunnen stoppen. Vervolgens een loop uitvoeren om te sommeren, en dat was dat geweest. Op die manier hebben we maar 2 sync’s nodig 1 na het communiceren en 1 na het rekenen.

***Experimenten***
We hebben de programma's m2d en m3d voor verschillende waarden van n, p en q uitgevoerd en vervolgens de tijd die nodig was om de matrixvermenigvuldiging uit voeren. Bij elk van de gekozen waarden hebben we ter vergelijking ook de matrixvermenigvuldiging sequentieel uitgevoerd.
De tijd is in seconden.

*10x10-matrix*

| *methode* | | | *tijd* |
|---|---|---|---|
| m2d | 2 | 4 | .03269 |
| m3d | 2 | 8 | .024128 |
| sequentieel | | | .000133 |

*96x96-matrix*

| *methode* | | | *tijd* |
|---|---|---|---|
| m2d | 3 | 9 | .07152 |
| m3d | 2 | 8 | .076487 |
| sequentieel | | | .064812 |

*525x525-matrix*

| *methode* | | | *tijd* |
|---|---|---|---|
| m2d | 5 | 25 | 53.891750 |
| m3d | 3 | 27 | 2.270404 |
| sequentieel | | | 13.245914 |

*l000xl000-matrix*

| *methode* | | | *tijd* |
|---|---|---|---|
| m2d | 8 | 64 | >5min. |
| m3d | 4 | 64 | 12.650868 |
| sequentieel | | | 165.404784 |

**Opmerkingen**
We zien dat voor kleine matrices de sequentiële methode beduidend sneller is. Voor de parallelle methoden is de verhouding communicatie en rekenen natuurlijk verstoord: communicatie kost veel meer tijd dan rekenen.

Bij grotere matrices haalt sequentieel het niet bij m3d, de hoeveelheid rekenen neemt hard toe en dat is wat m3d juist verdeeld ever zijn processoren. Daarentegen kost het voor m2d flink wat meer tijd dan sequentieel om een 1000 bij 1000 matrixvermenigvuldiging uit te voeren. De manier van communiceren is bij m2d natuurlijk ook belabberd: er worden hele matrices van processor 0 naar de andere doorgegeven. Je kunt wet nagaan dat het doorgegeven van een miljoen doubles wel even duurt...

***Conclusie***
Grote matrices vermenigvuldigen gaat parallel snellen omdat de rekentijd dan minimaal gemaakt wordt. Kleine matrices kunnen het beste sequentieel vermenigvuldigd worden: de rekentijd is toch al kort en parallel zorgt in dergelijk geval alleen veer extra communicatie.