x-Koninginnenprobleem, parallelle oplossing

Albert-Jan Yzelman

7-02-2004

Inhoudsopgave

Hoofdstuk 1

Inleiding

Het standaard (8-)koninginnenprobleem bestaat uit het plaatsen van 8 koninginnen op een schaakbord zonder dat deze elkaar kunnen slaan. Het x-koninginnen probleem is een simpele uitbreiding van dat probleem; we plaatsen nu x koninginnen op een vierkant bord met grootte x. Het is de bedoeling dat we x heel wat groter kiezen dan 8, en vervolgens het probleem oplossen met behulp van een parallel algoritme.
Hierbij komen verschillende zoekalgoritmen bij naar voren, alsmede meerdere technieken om een parallel algoritme te versnellen. Ook wordt er aandacht besteed aan het vergelijken van verchillende algoritmes, en wordt er kort geanalyseerd hoe die verschillen zijn ontstaan.
Op het internet zijn al vele programma's te vinden die een x-koninginnen probleem op kunnen lossen. Degene die ik heb bekeken maakten gebruik van de normale DFS zoals in hoofdstuk 3.2.1 beschreven zal worden. Echt snelle ben ik helaas niet tegengekomen, dus het uiteindelijke algoritme wat ik heb geschreven heb ik niet tegen een al bestaand algoritme getest.
Dit project is gedaan in het kader van het vak *High Performance Computing*, onder toeziend oog van de docent; Dr. Rob Bisseling.

Hoofdstuk 2

Representatie

We zouden de koninginnen in een matrix kunnen representeren, maar dit kost x^2 ruimte in het geheugen. Mooier is het om in de representatie er rekening mee te houden dat koninginnen nooit in dezelfde rij kunnen staan; we kunnen dus ook alleen de kolommen van de koninginnen opslaan. Op deze manier hebben we alleen x geheugen nodig om een bordsituatie op te slaan. Als we verder nog opmerken dat er nooit koninginnen op dezelfde kolom mogen staan, kunnen we ook eisen dat alle opgeslagen kolomnummers uniek zijn. Op deze manier representeert de array [1,2,3,4,5,6,7,8] dus een schaakbord waarop alle koninginnen op een diagonaal zijn geplaatst.

2.1 Diagonalen

Door de gekozen representatie is het al onmogelijk dat koninginnen elkaar horizontaal of verticaal kunnen slaan. Rest ons nu alleen nog het diagonaal slaan. Stel dat we een bordsituatie hebben weergegeven door een array A. Dan wordt de diagonaal van de koningin op A[x] gegeven door:
$D(t) = A[x] + t$, waar t de afstand is van x (in rijen).
Merk op dat we kunnen controleren of de koningin op A[x] de koningin op A[y] kan slaan, als
$D(y - x) = A[x] + y - x = A[y]$, of als
$D(x - y) = A[x] + x - y = A[y]$.
In de uiteindelijke implementatie is dit herschreven tot alleen maar 1 controle, en wel als volgt:
Als $D(y - x) = A[x] + y - x = A[y]$ geldt, geldt ook $A[x] - A[y] = x - y$.
$D(x - y) = A[x] + x - y = A[y]$ impliceert dat $A[x] - A[y] = y - x$.
Hieruit valt af te leiden dat, als $A[x]A[y]$ slaat, dat moet gelden:
$(A[x] - A[y])^2 = (x - y)^2$.

Als we nu een bordsituatie creeren door voor elke rij, beginnend bovenaan, willeukeurig een koningin op een bepaalde plek te zetten, dan kunnen we met bovenstaande formule bij voorbaat al plekken uitsluiten waar een koningin al op die diagonaal zit. Hiertoe hebben we een algoritme nodig dat controleert of een koningin op een bepaalde rij een andere koningin kan slaan.

2.2 Validatie-algoritme

We beginnen met een algoritme dat alleen controleert of een gegeven koningin een andere koningin slaat die in een hogere rij staat dan de huidige. Dit omdat de zoekalgoritmes die we later gebruiken, van boven naar beneden zoeken (in rijen). Het is dus niet nodig om te kijken naar wat onder de huidige koningin staat. Bovendien zal, als je dit algoritme voor alle koninginnen op het bord aanroept, alsnog alle koninginnen met alle andere hebben vergeleken. Het algoritme ziet er als volgt uit:

```
Algorithm validate(int *bord, int row, int positie):

 valid<-TRUE;

 for(j=(row-1); j>=0; j--) do {
  rowd <- (row-j);
  d <- (positie-bord[j]);
  if (d==0) {
   valid <- FALSE;
   j <- -1;
   }
  else {
   if ((d*d)==(rowd*rowd)) {
    valid <- FALSE;
    j <- -1;
    }
   }
  }
 return valid;

End Algorithm
```

De variabele *bord* is de array met kolomposities van de koninginnen, netzoals de array A hierboven. *row* geeft weer in welke rij de koningin staat, waarvan we de *positie* willen valideren. Dus in principe zegt het algoritme of $bord[row] = positie$ mogelijk is. Standaard gaat het algoritme ervan uit dat de positie klopt, en zoekt het van *row* tot 0 naar een reden waarom dit niet zou koppen. In de *for*-loop, slaan we in de variabele *rowd* de afstand (distance) tussen de huidige rij (*row*) en de rij waarop we aan het controleren zijn (j). In d slaan we de afstand op tussen de kolomposities. Als eerst merken we op dat als $d = 0$ dat de kolomposities gelijk zijn; $bord[row] = positie$ is dus niet mogelijk. Het algoritme zet *valid* op FALSE en termineert de loop. Vervolgens controleert het algoritme of $d^2 = rowd^2$, wat precies hetgeen is wat we in de vorige paragraaf hadden afgeleid. Als dit waar is dan kruisen de koninginnen elkaar diagonaal, en dus wordt *valid* op FALSE gezet en de loop gestopt.

Hoofdstuk 3

Het oplossen

Nu er een manier van valideren is opgezet kan er worden nagedacht over een algoritme dat zoekt naar een mogelijke oplossing. Hiertoe zijn er verschillende methodes. Wegens voorgaande representatie ligt het misschien voor de hand om op een directe manier te puzzelen met permutaties van de kolom-lijst van koninginnen. We kunnen dit doen door telkens willekeurig permutaties uit te voeren; we gebruiken dan een soort evolutionair algoritme. Dit houdt in dat we op een bepaalde kolom-lijst een reeks verschillende permutaties uit te proberen. Van elke nieuwe kolom-lijst wordt dan vervolgens bepaald welke de beste is (in dit geval welke de minste aantal elkaar slaande koninginnen bevat). Op die kolom-lijst voeren we dan hetzelfde proces uit, totdat er een lijst onstaat waar er geen koninginnen worden geslagen. Op dit principe kunnen vele varianten worden bedacht; we kunnen bijvoorbeeld uitgaan van de 3 beste uit een lijst permutaties, of 10 beste, of welk aantal je ook maar wilt. Misschien bestaat er zelfs een slimme manier van combineren waardoor twee goede kolomlijsten gecombineerd kunnen worden tot een nog betere.
Echter, een evolutionair algoritme heeft als nadeel dat het mogelijk zeer lang duurt totdat er een oplossing wordt gevonden. Bovendien, als er geen oplossing bestaat, dan blijft dit algoritme oneindig lang doorzoeken. Met dit in gedachte, kunnen we beter op zoek gaan naar een methode die wel zekerheid geeft over het al dan niet bestaan van een oplossing, en waarvan we dus ook zekerheid hebben dat de oplostijd van het probleem eindig is. Om een dergelijk algoritme te schrijven, is het handig om het probleem als een boom te representeren.

3.1 Bomen

Stel dat we een bordsituatie creeren door eerst op de bovenste rij een koningin te zetten. Laat er daarna gekeken worden naar welke kolommen op de volgende rij daardoor afvallen. Op een van die overgebleven kolommen wordt een koningin gezet. Vervolgens herhalen we dit principe voor alle andere rijen. We kunnen volgens dit principe een zogenaamde zoekboom opstellen. Deze begint bij een leeg speelbord. Vervolgens kunnen we een koningin op de eerste rij in elke kolom zetten, wat dus leidt tot x vertakkingen in de boom. Voor elke vertakking kunnen we weer een koningin in de tweede rij zetten, op elke kolom, exclusief diegenen die kruizen met de eerste koningin, en de kolom waar de eerste koningin al in stond. Dit geeft dus weer een verdere vertakking voor de boom. Dit principe kunnen we herhalen voor elke rij. Merk op dat de grootte van deze zoekboom erg snel groeit. Het aantal mogelijke paden, zonder rekening te houden met diagonalen, komt neer op $\Pi_{i=0}^{x}(x-i)$.

3.2 Zoeken

Afhankelijk van hoe je de boom doorloopt, komt men uit op een oplossing, of op een dood punt; een eindpunt die niet op de laatste rij staat, omdat er geen relevante mogelijkheden meer zijn om een koningin kwijt te kunnen op zijn volgende rij. Op een dood punt kan er op verschillende manieren worden verder

gegaan; men zou van voor af aan een ander pad door de boom proberen bijvoorbeeld. De manier waarop een pad door een zoekboom wordt gekozen en de manier waarop wordt omgegaan met een 'dood punt', is karakteristiek voor een zoekalgoritme. De nu volgende zoekalgoritmes zijn elk geimplementeerd en getest op het x-koninginnenprobleem.

3.2.1 Depth-First Search Algorithm (DFS)

Het eerste wat geprogrammeerd was, is een "Depth-First Search Algorithm (DFS)". Dit algoritme zoekt een pad door de boom, en als die doodloopt en geen oplossing heeft, dan neemt het een stapje terug en zoekt het kiest het een ander pad. Dit herhaalt zich totdat er een oplossing wordt gevonden. Als dit algoritme geen oplossing vindt, dan is er ook geen oplossing. Een simpel DFS-algoritme ziet er zo uit:

```
Algorithm search(int *bord, int row):

for(i=0;i<x;i++) do
 searchOnRow(bord,row,i);

End Algorithm


Algoritm searchOnRow(int *bord, int row, int i):

if(validate(bord,row,i)==TRUE) {
 bord[row]=i;
 if (row<(x-1))
  search(bord, (row+1));
 else
  we hebben een oplossing
 }

End Algorithm
```

Het search-algoritme vraagt als invoer een array van integers, welk het bord moet representeren, en een integer row. Het is de bedoeling dat als men wilt beginnen met zoeken, de opdracht $search(bord, 0)$ uitvoert. Het algoritme loopt dan recursief door totdat het alle mogelijke bordcombinaties heeft gehad (en geen oplossing heeft gevonden) of totdat er een oplossing is gevonden. We zien dat het search algoritme in feite gewoon alle koninginposities (van 0 tot x) op row achter elkaar uitprobeert. searchOnRow bekijkt vervolgens of de positie $bord[row] = i$ eigenlijk wel geldig is. Zo ja, dan wordt er daadwerkelijk een koningen op $bord[row]$ gezet, en roepen we het search-algoritme weer aan, dit keer op het volgende rij. Echter, als we al op de laatste rij zaten te zoeken, hebben we een oplossing en houdt het algoritme op.

3.2.2 Random DFS

Voorgaand DFS-algoritme werkt prima, maar het is duidelijk dat dat algoritme een specifiek zoekpatroon volgt. Er wordt altijd een koningin gezet op de eerste plaats waar dit kan. Mogelijk is dit geen handige manier, en kan men beter telkens een willekeurig gekozen positie gebruiken voor het neerzetten van de koningin. Een handige manier om dat te implementeren is door een random array $tosearch$ te defenieren, welk unieke elementen bevat uit $[0, x]$, in een willekeurige volgorde. Vervolgens passen we het search-algoritme een beetje aan:

```
Algorithm search(int *bord, int row):

tosearch<-random array;

for(i=0; i<x; i++) do
 searchOnRow(bord,row,tosearch[i]);

End Algorithm
```

Deze implementatie heeft als voordeel dat de functie niet meer herschreven hoeft te worden voor andere zoek-strategieen. Het originele DFS-algoritme werkt nu bijvoorbeeld precies hetzelfde als $tosearch = [0, 1, ..., x]$.
Random search heeft als voordeel dat er geen bepaald zoekpatroon gevolgd wordt. Hierdoor is het te verwachten dat de oplostijd die nodig is beschreven kan worden als een stochast; er bestaat een gemiddelde oplostijd. Sequentieel gezien is dit eigenlijk niet zo interessant; het enige wat dit betekent is dat de oplostijd onvoorspelbaar is geworden. Waarschijnlijk loopt het programma random soms sneller dan DFS, maar soms ook langzamer. Random search wordt pas interessant als het algoritme wordt geparalelliseerd. De oplosruimte wordt dan opgedeeld tussen meerdere processoren, en er wordt een random search op elk van die subsets uitgevoerd. Uit de kansrekening weten we dat in zo'n geval de totale oplostijd sneller convergeert naar het gemiddelde; de standaarddeviatie wordt kleiner. Parallel gezien is de oplostijd van een random search dus redelijk stabiel, en (zoals later experimenteel zal blijken) sneller dan standaard DFS.

3.2.3 Random Array

Om een random array te verkrijgen gebruiken we 3 verschillende algoritmes. Ze zijn redelijk simpel opgebouwd:

```
Algorithm coinFlip():

random <- rand()%100;
if (random<50)
 return TRUE;
else
 return FALSE;

End Algorithm


Algorithm *transpose(int *a, int b, int c):

temp <- a[c];
a[c] <- a[b];
a[b] <- temp;
return a;

End Algorithm


Algorithm *randomArray(int length):
```

```
for (i=0; i<length; i++)
 toReturn[i] <- i;

for (i=0; i<length; i++)
 if (coinFlip()==TRUE)
  transpose(toReturn,i,(rand()%length));
return toReturn;

End Algorithm
```

De functie *coinflip()* geeft met 50 procent kans TRUE terug, en anders FALSE. *Transpose(a,b,c)* draait in de rij a de elementen b en c om. Vervolgens maken we een random array met de functie *randomArray* door eerst een rij te maken van $[0, 1, ..., x]$, en vervolgens elk element met 50

```
Algorithm *randomArray(int length):

for (i=0; i<length; i++)
 toReturn[i] <- i;

random <- rand()%100;
chance <- (1-1/x)*100;

for (i=0; i<length; i++)
 if (random<chance)
  transpose(toReturn,i,(rand()%length));
return toReturn;

End Algorithm
```

Dit algoritme geeft elke elementpositie van de random array $1 - 1/x$ kans om verplaatst te worden. In de uiteindelijke implementatie wordt echter nog steeds gebruik gemaakt van de eerste methode; ik had geen tijd gehad uitvoerig te bekijken welke van deze methodes beter was. De eerste paar testjes (op een sequentieel algoritme) wezen uit dat de eerste methode sneller leek, maar de testruimte was redelijk klein (10 runs), en het snelheidsverschil miniem.

Hoofdstuk 4

Parallelisering

Het koninginnenprobleem is voor kleine waarden (8 koninginnen) goed te doen voor elke computer gebruik makend van een sequentieel DFS of Random-DFS algoritme. Er bestaan ook evolutionaire algoritmes die voor die probleemgrootte goed werken. Maar de oplostijd stijgt behoorlijk naarmate we het aantal koninginnen op een x bij x bord vergroten. Dit is terug te vinden in het experimentele gedeelte van dit verslag. De volgende stap is nu om te kijken hoe we dit probleem parallel kunnen oplossen. We zouden willen dat zowel normale DFS als random-DFS parallel sneller zouden lopen dan hun sequentiele versies. Het is zoals we straks zullen zien wel makkelijk om een algoritme te bedenken dat minstens zo snel is als een sequentieel equivalent. Maar we zullen ook zien dat, met wat meer moeite, we veel sneller kunnen zijn dan een sequentieel algoritme.

4.1 Superlineare speedup

Het verschil tussen een sequentieel algoritme en een parallele, is dat een parallele meerdere processoren gebruikt om tot een oplossing te komen. De verschillende processoren opereren zoveel mogelijk zelfstandig – immers, communicatie tussen processoren kost tijd, en daar willen we zo weinig mogelijk van gebruiken. Om het verschil goed uit te leggen is het handig om een voorbeeld te gebruiken.
Stel dat we een sequentieel algoritme hebben dat een oplostijd heeft van T_s. Stel ook dat we een parallele versie hebben van dit algoritme, en dat we P processoren beschikbaar hebben voor dat algoritme. Van een normaal parallel algoritme kunnen we verwachten dat de oplostijd wordt verlaagd naar $T_p = T_s/P$; we hebben P processoren die elk een even groot deel van het sequentiele algoritme tegelijkertijd uitvoeren.
Maar soms kan de oplostijd nog lager uitvallen. Een relevant voorbeeld hiervan zijn zoekproblemen. Als we een zoekprobleem verdelen over meerdere processoren, is er meer kans snel een oplossing te vinden. Stel dat het sequentiele algoritme ontworpen was om van een rij getallen te zeggen of er een 7 in stond. De sequentiele versie zou er zo uit kunnen zien:

```
Algorithm SearchSeven(int *getallenlijst, int lijstlengte):

for(i=0; i<lijstlengte; i++) do {
 if (getallenlijst[i]==7) do {
  return TRUE;
 }
}

return FALSE;
```

```
End Algorithm
```

Stel dat we een lijst van 10 getallen invoeren, en dat alleen het laatste cijfer in de rij een 7 is. Het sequentiele algoritme doorloopt de hele rij in $T_s = 10$ tijd en vindt de zeven. Stel dat we dit algoritme paralleliseren over twee processoren die elk de helft van de originele rij krijgen, en vervolgens het normale sequentiele algoritme uitvoert op elk van die processoren. De tweede processor vindt dan de zeven na $T_p = 5$ tijdseenheden, waarna het parallele algoritme stopt. In dit geval klopt de voorspelling dat $T_p = T_s/P$ precies. Maar stel nu dat de 7 op $getallenlijst[5]$ stond, en niet op de laatste plaats. Dan is $T_s = 5$. Als we nu de getallenlijst splitsen, staat de zeven op de eerste plaats van de lijst op de tweede processor. Het algoritme stopt dus gelijk na de eerste iteratie, en dus geldt $T_p = 1$. Dit is heel wat sneller dan de voorspelde $T_s/2 = 3,5$, en het is dit verschijnsel dat een parallel algoritme sneller is dan liniear te verwachten was, dat we *superlinieare speedup* noemen.

4.2 Het BSP-model

Bij een parallel algorime is het de bedoeling dat elke processor hetzelfde programma draait. Hierdoor is het gewenst dat er gecommuniceerd kan worden tussen de processoren. Bij dit project wordt het *BSP*-model gebruikt. Dit model bevat een klein aantal opdrachten die communicatie mogelijk maken. De uitleg hier is beperkt tot wat nodig is om de algoritmes die gaan komen begrijpbaar te maken; meer informatie over de werking van BSP is (bijvoorbeeld) te vinden in het boek *Parallel Scientific Computation* van R.H. Bisseling. Om gegevens van processoren op te vragen danwel gegevens in andere processoren op te slaan, heeft het BSP-model twee primitieven:

```
bsp_get(pid,source,offset,destination,size);
```

en

```
bsp_put(pid,source,destination,offset,size);
```

pid is het processor-identificatie nummer. Elke processor in een parallel algoritme onder BSP heeft een unieke nummer toegewezen gekregen. Dit nummer is op te vragen via de functie

```
bsp_pid();
```

De variabelen *source* en *destination* verwijzen naar de bron-variabelen respectievelijk de aankomst-variabele. We versturen dus de waardes van en naar de respectievelijke variabelen. $offset$ kunnen we gebruiken om een aantal bytes naast de opgegeven variabele te lezen (bij get) danwel te schrijven (bij put). Dit is handig te gebruiken in combinatie met getallenlijsten, zoals later zal blijken. *size* geeft aan hoeveel informatie er moet worden gelezen danwel geschreven. Hierdoor kan het BSP-model integers lezen en schrijven, maar ook lijsten van integers, of zelfs delen van lijsten, enzovoorts.
Er is echter een 'maar' verbonden aan het gebruik van de get en put functies. Op het moment dat een algoritme een van die functies tegenkomt, wordt er niet direct gecommuniceerd. Het daadwerkelijk lezen en/of schrijven gebeurt pas op het moment van een synchronisatie. Bij een synchronisatiepunt wacht een processor totdat alle andere processoren ook willen synchroniseren. Dit is het enige moment waarop er gecommuniceerd kan worden, en een synchronisatie kan in een algoritme worden aangeroepen met behulp van de functie

```
bsp_sync();
```

Het stuk algoritme dat tussen twee synchronisatie-stappen staat wordt ook wel een *superstep* genoemd. De beperking van communicatie tot alleen de synchronisaties heeft grote gevolgen over hoe een parallel algoritme moet worden geschreven. Een extra moeilijkheid nog is dat de waarden pas op het moment van synchronisatie worden gelezen; dus als de waarde van een variabele in een put danwel get functie wordt gewijzigd voordat er een synchronisatie is geweest, wordt de gewijzigde variabele gelezen/weggeschreven. Vooralsnog zijn deze vier BSP-functies de enige die ik zal gebruiken in de psuedo-code dat nog volgt, dus kunnen we verder gaan met het paralleliseren van de zoekalgoritmes.

4.3 Simpele parallelisering

We hebben al gezien hoe DFS sequentieel werkt (Hoofdstuk 3.2). We kunnen dat algoritme op verschillende manieren paralleliseren. Het simpelste is om de zoekboom op te splitsen tussen de verschillende processoren. We kunnen bijvoorbeeld, bij een 8-koninginnenprobleem en een parallel DFS-algoritme op 2 processoren, de ene processor laten zoeken op $bord[0] = [0, 1, 2, 3]$ en de ander op $bord[0] = [4, 5, 6, 7]$. Bedoeld word dus dat processor 0 de methode searchOnRow(bord,0,i) aanroept voor achtereenvolgens $i = 0, 1, 2, 3$, totdat er een oplossing is gevonden. Processor 1 doet hetzelfde, maar dan voor $i = 4, 5, 6, 7$.

4.3.1 Normale DFS

In het sequentiele algoritme werd het zoeken begonnen met de opdracht search(bord,0). In het parallelle geval is het het makkelijkst om die regel te vervangen met het volgende:

```
searchLength<-x/P;
pid<-processor_nummer;

for(i=pid*searchLength; i<(pid+1)*searchLength; i++) do
 searchOnRow(bord,0,i);
```

Op deze manier hoeven we in het begin alleen de waarde x door te sturen, en elke processor kan gelijk aan de slag. Het enige probleem hier is dat zodra een processor een oplossing heeft, dat deze de andere processoren moet laten stoppen. Laat elke processor een variabele *solved* hebben, die bijhoudt of het algoritme een oplossing heeft gevonden. We kunnen met kleine aanpassingen ervoor zorgen dat als *solved* =TRUE, het zoekalgoritme stopt. Als een processor nu een oplossing vindt, hoeven we alleen de *solved*-variabelen op alle andere processoren ook op TRUE zetten. Zoals we weten uit het BSP-model, kunnen we dit doen met een 'put'-opdracht. Nu krijgen we te maken met een typisch parallelisatie-probleem; een put opdracht wordt pas uitgevoerd na een sync. Aangezien we niet kunnen voorspellen wanneer er een oplossing gevonden wordt, moeten we van tijd tot tijd alle processoren laten syncen. Syncen kost veel tijd naargelang er veel processoren worden gebruikt, dus willen we het aantal syncs klein houden. Echter, als er een oplossing is, willen we ook niet te lang wachten op een sync. Het vinden van een balans hiertussen is een onderwerp waar in Hoofdstuk 5 verder op wordt in gegaan.
Voorlopig kunnen we ons beter eerst bezighouden met wat we gebruiken om te bepalen wanneer we gaan syncen. We zouden dit kunnen doen door een tijdsinterval in te stellen, maar een betere oplossing lijkt het om de sync-momenten af te laten hangen van hoeveel werk een processor heeft verricht. Door ons probleem te herkennen als een zoekboom, zien we dat het plaatsen van een koningin op een geldige plek gezien kan worden als het doen van werk. We gaan dan immers een tak verder omlaag in de boom. Om nu de hoeveelheid werk bij te houden introduceren we de variabele *counter*, die bij elke plaatsing van een koningin op het bord wordt geincrementeerd. Het algoritme synct nadat er een vooraf ingesteld aantal werkverrichtingen zijn gepasseerd. Een geschikte waarde voor dit algoritme bleek 10000 te zijn. We kunnen zien of er 10000 werkverrichtingen zijn gepasseerd als *counter* modulo $10000 = 0$ geldt.

4.3.2 Implementatie

In het licht van parallelisering hebben we dus een nieuwe algoritme moeten introduceren, en de sequentiele algoritmes iets moeten aanpassen. We hebben gezien dat we, als we een oplossing hebben, het algoritme willen laten stoppen door de variabele *solved* op elke processor op TRUE te zetten. Dit doen we met de volgende methode:

```
Algorithm SendTerminate()

P<-Number of processors in use

for(i=0; i<P; i++)
 bsp_put(i,&solved,&solved,0,SZINT);

End Algortihm
```

Nu moet dit algoritme natuurlijk nog worden aangeroepen in het zoekalgoritme. Gewijzigd ziet deze er zo uit:

```
Algorithm search(int *bord, int row):

for(i=0;i<x;i++) do {

 searchOnRow(bord,row,i);

 if (solved==TRUE)
  i=x;
 }

End Algorithm


Algoritm searchOnRow(int *bord, int row, int i):

if (solved==TRUE)
 return;

if (counter%10000==0)
 sync();

if (validate(bord,row,i)==TRUE) {
 counter++;
 bord[row]=i;
 if (row<(x-1))
  search(bord, (row+1));
 else {
  solved=TRUE;
  igotit=TRUE;
  solution=bord;
  sendTerminate();
  }
 }

End Algorithm
```

In deze implementatie zijn er twee nieuwe variabelen te vinden. *igotit* houdt bij of de huidige processor daadwerkelijk een oplossing heeft gevonden (en dus niet gestopt is door een andere processor), en *solution* slaat de oplossing op. Ook is er te zien dat op verschillende plaatsen wordt gecontroleerd of de variabele *solved* waar is. Zo ja, dan wordt het algoritme zo snel mogelijk gestopt.
Tot slot, nog de even het basis-algoritme ter volledigheid. Deze ziet er nu als volgt uit:

```
Algorithm main():

pid<-processor nummer;

if (pid==0) do {
 Vraag x van de gebruiker (grootte van het probleem);
 Vraag het aantal processoren (P) van de gebruiker;
}

Vraag x en P van processor 0;

searchLength<-x/P;

for(i=pid*searchLength; i<(pid+1)*searchLength; i++) do
 searchOnRow(bord,0,i);

End Algorithm
```

4.3.3 Random DFS

Voorgaande implementatie van DFS hoeft gelukkig maar zeer weinig te worden aangepast om Random-DFS parallel te krijgen. In essentie was het enige verschil tussen normale DFS en random DFS het gebruik van een array *tosearch*. Het eerste verschil zit hem dus in het search-algoritme:

```
Algorithm search(int *bord, int row) {

deeperSearch=randomArray(x);

for(i=0;i<x;i++) {
 searchOnRow(bord,row,deeperSearch[i]);

 if (solved==TRUE)
  i=x;
 }

End Algorithm
```

Het tweede verschil zit in de aanroep van het parallelle gedeelte. Als we het simpel willen houden kunnen we processor 0 een globale *tosearch*-array laten maken en doorsturen naar alle processoren, voordat het algoritme begint. Vervolgens passen we de search-aanroep, gedeeltelijk op dezelfde manier als bij de normale DFS:

```
searchLength<-x/P;
pid<-processor_nummer;
if (pid==0)
 tosearch <- randomArray(x)
tosearch <- vraag tosearch array van processor 0

sync;

for(i=pid*searchLength; i<(pid+1)*searchLength; i++) do
 searchOnRow(bord,0,tosearch[i]);
```

Nu wordt simpelweg de random-lijst in opgedeeld tussen de processoren. Dit is het enige verschil met het normale parallelle DFS algoritme. De effecten zijn echter goed te merken; sommige processoren hebben dankzij de random-methode meer geluk dan andere en vinden (soms veel) eerder een oplossing dan een andere. En als er een processor is die per ongeluk erg goed gokt en snel klaar is, krijgen we een veel lagere oplostijd dan een sequentieel random algoritme, die iets minder goed gokt. Naar verwachting is het dus ook zo dat een dit parallel algoritme het heel goed zal doen met een groot aantal processoren. Hierover meer in hoofdstuk 5.

4.4 Global Parallel Search

Een andere manier om te paralleliseren is door meerdere processoren aan een kleine deelboom te laten werken. Een manier om dit te doen is via een *rundown*-methode. We laten dan eerst een processor via een DFS methode de zoekboom doorlopen, totdat deze ergens compleet vastloopt en dus in de boom naar boven zou moeten gaan om dan een ander pad te onderzoeken. In plaats van sequentieel verder te gaan om dat pad te onderzoeken, kunnen we vanaf hier beginnen te paralleliseren. We zouden alle mogelijke paden die nog te onderzoeken zijn vanuit dat punt in de boom, kunnen verdelen onder de beschikbare processoren. Dit is essentieel een andere soort parallelisering dan eerder beschreven, want hier draagt elke processor wezenlijk zijn deel bij aan de uiteindelijke oplossing. Er treden echter heel wat moeilijkheden op bij deze manier van paralleliseren, welke ik hier een voor een zal toelichten.

4.4.1 Zoekmatrix

Eerst moet er beslist worden hoe precies we de zoektocht over de processoren verdelen. Als het Global Parallel Search (GPS) algoritme na het sequentiele 'rundown' deel vastloopt, willen we dat er vanaf dat punt geparalleliseerd wordt. We zouden dit kunnen doen door een rij terug te gaan in de zoekboom, en alle andere mogelijke paden, te verdelen tussen de processoren. Als nu alle processoren weer vastlopen, gaan we weer een rij terug in de boom en doen we hetzelfde. Deze methode vereist dat we voor elke rij bijhouden waar we zijn met zoeken. We moeten weten welke we paden we al hebben uitgeprobeerd (of nog worden uitgeprobeerd door een processor), en welke er nog uitgeprobeerd moeten worden.
Hiervoor gebruik ik een zoekmatrix. De zoekmatrix is een x bij $(x+1)$ matrix, waarin elke rij een *tosearch*-array bevat, plus nog een getal dat bijhoudt waar het algoritme real-time in de array zit. Bijvoorbeeld, als het algoritme op rij 5 aankomt, en het wil dieper zoeken, dan genereert het eerst een *tosearch*-array. Het algoritme doorloopt deze array van begin tot eind, dus de variabele die bijhoudt waar het algoritme real-time zit wordt geinitialiseert op 0. Dus geldt dat $searchmatrix[5] = [tosearch + +0]$, waar $++$ een concatinatie-teken is; de array tosearch is uitgebreid met een element 0.

4.4.2 Master/slave

De zoekmatrix is goed te vullen met *tosearch*-arrays tijdens de rundown-stap van het GPS-algoritme. We zien dat niet de hele searchmatrix gevuld wordt; we gaan er immers vanuit dat het rundown-deel op een bepaalde rij vastloopt. Dit is echter geen probleem omdat we alleen op dezelfde rij of hoger gegevens halen uit de matrix.
Omdat het handig is de searchmatrix te vullen tijdens het rundown-gedeelte, wordt het voor ons aantrekkelijk een master/slave model te gebruiken voor de parallelisatie. Vanwege de grootte van de matrix wordt het namelijk inefficient om deze meerdere malen door te sturen naar andere processoren, dus gebruiken we die matrix alleen op de 'master'-processor. Deze master zal vervolgens al het werk verdelen als het parallelle deel van het algoritme begint. Het rundown-deel laten we sequentieel lopen op de master-processor. We hoeven dit niet te paralleliseren, omdat het vastlopen op dit probleem best snel gebeurt. Als we x echt extreem groot kiezen, wordt het misschien interessant om over parallelisatie na te gaan denken, maar omdat we alle tosearch-arrays in de searchmatrix willen opslaan zou dat wel eens heel complex kunnen worden.

4.4.3 Werkverdeling

Nu komen we aan bij de daadwerkelijke parallelisatie. We herinneren ons van de 'simpele parallelisatie' dat we daar elke processor het searchonrow-algoritme lieten aanroepen, met verschillende parameters. Omdat we hier, vanwege de searchmatrix, continu werken met tosearch-arrays, is het handig dit in te bouwen in het search-algoritme zelf. We herschrijven het algoritme als volgt:

```
Algorithm search(int *bord, int row, int *tosearch) {

for(t=0;t<x;t++) {
 searchOnRow(bord,row,tosearch[t]);

 if (solved==TRUE)
  t=x;
 }

End Algorithm
```

Dankzij deze wijziging moeten we ook een regeltje in het algoritme searchOnRow veranderen. In de oude regel klopt de aanroep naar de functie search namelijk niet meer:

```
search(bord, (row+1))
```

Deze regel zorgde voor het dieper zoeken in de boom, maar het bevat nu een argument te weinig. We moeten hier nu ook een tosearch-array meegeven. Om makkelijk zowel normale DFS als random DFS te ondersteunen, schrijven we een nieuw algoritme *genSearch()* dat een tosearch-array teruggeeft naargelang welke methode wordt gevolgd. Dus, als we normale DFS gebruiken, dan geeft genSearch() altijd de array $[0, 1, ..., x]$ terug. Bij random DFS geeft het een random array terug. Op deze manier kunnen we ook makkelijk andere zoekstrategieen implementeren; we hoeven dan alleen het *genSearch*-algoritme aan te passen. Al met al, de nieuwe regel in searchOnRow wordt dus:

```
search(bord,(row+1),genSearch());
```

Nu we een duidelijke structuur hebben aangebracht kan het starten van het parallelle gedeelte abstract ontworpen worden. De aanroep zal er als volgt uit gaan zien:

```
while (solved==FALSE) {

 clock();

 if (outofwork==FALSE) {
  search(bord,(currentrow+1),genSearch());
  if (solved==FALSE)
   outofwork=TRUE;
  }
 askforwork();
 }
```

De variabele *outofwork* geeft aan of de processor werk heeft ja of nee. Als een processor werk heeft, dan heeft het van de master processor een *bord* doorgestuurd gekregen, en een *currentrow*. Dat zijn de enige twee variabelen die nodig zijn om een al deels opgelost koninginnenprobleem verder op te lossen. Als de zoektocht is afgelopen en geen oplossing is gevonden (*if solved==FALSE*), dan heeft de processor geen werk meer en wordt outofwork op TRUE gezet. Vervolgens vraagt de processor om werk, en begint alles weer van voor af aan.
Het algoritme dat werk verdeelt tussen processoren, en de synchronisaties regelt tijdens het zoeken, is het *clock*()-algoritme. Dit algoritme wordt als eerste opgeroepen als de parallelisatie begint; dit omdat alle processoren (behalve de master) nog geen werk hebben. Hier begint echter al een moeilijkheid. Omdat de functie clock() ook synchroniseert, moet gelden dat alle processoren precies evenveel keer clocken (het aantal synchronisaties moet gelijk blijven, anders loopt het algoritme natuurlijk vast). Gelukkig is dit probleem makkelijker op te lossen dan sommige andere die we al zijn tegen gekomen... Het enige wat moet gelden is dat, terwijl er nog geen oplossing is, de functie clock van tijd tot tijd wordt aangeroepen door elke processor. Er mag tijdens de zoektocht nergens anders worden gesynchroniseerd dan in de clock-functie. Nu merken we op dat als een van de processoren een sendTerminate() functie aanroept, dat alle processoren dat pas weten na de eerstvolgende clock. Als we er dus verder voor zorgen dat als er een terminatie-signaal is ontvangen (dus als solved=TRUE), het hele zoekalgoritme gelijk stopt, zodat het niet weer bij de clock-functie aankomt. Het search en searchOnRow algoritme deden dit al, en in de aanroep zorgt de regel *while (solved==FALSE)* precies voor het gewenste gedrag.
Het werk verdelen in de functie clock zelf is opzich nog redelijk ingewikkeld. We hebben gezien dat een processor dat geen werk meer heeft de functie *askforwork* aanroept. Deze functie bevat alleen het volgende commando:

```
bsp_put(pta,true,wrq,bsp_pid()*SZINT,SZINT);
```

Dit commando 'put' de waarde TRUE in een array *wrq*, op plaats *i*. *pta* is de nummer van de master-processor. De array *wrq* (work requests) op de master-processor houdt bij welke processor werk nodig heeft. Als *wrq*[*i*] =TRUE, dan betekent dat dat processor *i* werk wilt hebben. *askforwork()* bevat geen sync-opdracht, omdat dat alleen gedaan mocht worden in het clock-algoritme. Hier komen we nog een probleem tegen. Als het clock-algoritme alleen maar eenmaal zou synchroniseren, dan duurt het 1 clock voor de masterprocessor erachter te komen dat een processor werk nodig heeft, en als hij het direct zou versturen, zou dit pas de 2e clock aankomen. Dit is de reden waarom ik besloten heb dit clock-algoritme twee keer te laten synchroniseren; een keer aan het begin, om alle binnenkomende berichten te krijgen (en als bijeffect mogelijk een terminatiesignaal te versturen), en aan het eind een keer om het toebedeelde werk te versturen.
Het uitdelen van werk is dankzij de searchmatrix zeer makkelijk geworden. Echter, hier moeten we ook erg oppassen met wat we doen. We kunnen bijvoorbeeld gewoon de rij waarvoor het rundown-algoritme was vastgelopen, gewoon verder doorlopen. Maar wat als daar ook geen oplossing te vinden is? Gaan we dan gewoon weer een rij terug om daar hetzelfde te doen? Dit klinkt wel logisch om te doen, en is ook het eerste wat ik in het kader van GPS had geprogrammeerd.

De resultaten waren echter niet om over naar huis te schrijven. Dit allemaal vanwege een denkfout; als we telkens een rij terug gaan en dan verder zoeken, dan volgen we in principe precies het sequentiele DFS algoritme, alleen verdelen we het werk onder meerdere processoren. Een speedup is dan logisch gezien alleen te halen als een van die processoren eerder dan de andere processoren op de oplossing stuit. Maar dit gebeurt in ons probleem nooit; de oplossing zit altijd helemaal onderin de zoekboom. Voordat een processor daar komt zijn de andere processoren al vastgelopen en zijn ze verder gaan zoeken in de volgende deelbomen. Om het nog erger te maken, zijn deze deelbomen steeds groter geworden. Dus, als er een oplossing wordt gevonden, zijn de andere processoren *volgens het sequentiele DFS algoritme* al verder dan eigelijk nodig is geweest! Als dit algoritme gewoon sequentieel uitgevoerd was dan had het *minder* werk verricht dan het parallelle algoritme. Hier komt nog bij dat het parallelle algoritme nog extra werk heeft dankzij de clocks en het werk versturen. Het sequentiele algoritme is dus eigenlijk efficienter. Dit nadeel valt op te lossen door de deebomen redelijk klein te houden; als er dan een oplossing gevonden wordt is er een minimum aan teveel werk verricht.
Het volgende idee was om de rij waarvoor het rundown-algoritme '(de *stoprow*) was vastgelopen, op te slaan. Als we nu een rij omhoog zouden gaan in voorgaande werkverdeling, gaan we nu ook weer een rij naar beneden, zodat de rij waarop processoren zoeken altijd die zelfde rij is als waarop het rundown-algoritme vast liep. Hierdoor blijven de deelbomen waarop processoren zoeken relatief klein, en als er nu een oplossing gevonden wordt, is er ook weinig overbodig werk gedaan door de andere processoren (weinig in relatie tot de vorige werkverdeling althans). Nu gaat er helaas wat anders de mist in. Telkens een rij omhoog gaan en weer naar beneden gaan zijn bewerkingen die allemaal door de masterprocessor moeten worden gedaan. Als er dus heel veel work-requests zijn, bijvoorbeeld door gebruik van veel processoren, vertraagt het algoritme aanzienlijk. Experimentatie wees ook uit dat de masterprocessor soms moeilijk terug kon komen op de stoprow; het komt voor dat het probleem op een hogere rij al vast loopt. De masterprocessor zit dan een tijdje sequentieel te zoeken, terwijl de slave-processoren allemaal op hem wachten. Een oplossing hiervoor is om, als de masterprocessor op deze zoektocht op een hogere rij dan stoprow vastloopt, de stoprow met 1 te verlagen. Dit levert een essentiele snelheidswinst op.
Het probleem dat het algoritme vertraagt als er veel meer processoren worden gebruikt wordt hiermee echter niet opgelost. Er is wel een manier om te voorkomen dat dit probleem een bottleneck wordt; de masterprocessor moet pas veel werk verrichten voor workrequests, als deze een rij hoger moet gaan. Met veel processoren gebeurt dit sneller, tenzij het probleem evenredig veel groter wordt. Het idee is dus dat er met dit algoritme, nooit teveel processoren op een probleem van bepaalde grootte moet worden gezet. 8 processoren kunnen bijvoorbeeld nog prima werken op een probleem van 100 koninginnen, maar als er hier 64 processoren voor worden gebruikt, kan het algoritme extreem veel langer duren.

4.4.4 Rundown

We hebben het nu al veelvuldig gehad over het rundown-gedeelte, en na voorgaande overwegingen zijn we zover om er een te ontwikkelen. Hiervoor kunnen we niet gebruik maken van de al geschreven search en searchOnRow algoritmes; deze gaan, zodra vastgelopen, recursief verder met zoeken volgens het DFS-principe. Ook moeten we rekening houden met de searchmatrix. Er zit niets anders op dan een hele nieuwe algoritme te bedenken, en soms een beetje af te kijken van de search en searchOnRow algoritmes. Het uiteindelijke algoritme ziet er zo uit:

```
Algorithm rundown(int *bord, int row, int *tosearch):

searchmatrix[row] <- onlyValid(tosearch,bord,row)
searchmatrix[row][x] <- 0;

stoprow <- row;

for(t=0;t<x;t++) {
 if (validate(bord,row,tosearch[t])==TRUE) {
```

```
  bord[row] <- tosearch[t];
  if (row<(x-1))
   stoprow <- rundown(bord,(row+1),genSearch());
  else {
   solved <- TRUE;
   igotit <- TRUE;
   solution <- bord;
   stopSeq <- TRUE;
   sendTerminate();
   }
  }
 else
  if (t==(x-1))
   stopSeq <- TRUE;

 if (solved==TRUE)
  t <- x;

 if (stopSeq==TRUE)
  t <- x;
 }

return stoprow;

End Algorithm
```

Het eerste wat het algoritme doet is het updaten van de searchmatrix. In plaats van het direct zetten van de *tosearch*-array wordt eerst de functie onlyValid opgeroepen. Hierover meer in de volgende paragraaf. Na het updaten slaat het algoritme de huidige rij op in de variabele *stoprow*, zodat we aan het einde weten waar het algoritme is vastgelopen. Daarna zien we een gecombineerde versie van de algoritmes search en searchOnRow, met een paar grote verschillen. Het eerste verschil dat we zien is dat het algoritme rekening houdt met de rij waarop het zoekt; stoprow wordt geupdate met een nieuwe waarde als het algoritme zichzelf aanroept. Het 2e verschil is dat, in het onwaarschijnlijke geval dat dit algoritme een oplossing vindt, de variabele *stopSeq* op TRUE gezet wordt. Het volgende verschil is dat als $t = x - 1$ geldt, dus als alle mogelijkheden op *row* vastlopen, de variabele *stopSeq* op TRUE wordt gezet. Het laatste verschil is, logischerwijs, dat het algoritme stopt zodra *stopSeq* TRUE is. Aan het eind wordt ook nog de waarde *stoprow* doorgestuurd.

4.4.5 Only Valid

Om tijd de besparen bij het doorsturen van werk naar andere processoren, is het handig als de master-processor door heeft wat een valide plek is voor een koningin op de huidige rij. Anders kan het voorkomen dat een slave-processor een incorrecte plek krijgt om te doorzoeken. Deze processor heeft dat natuurlijk gelijk door; hij vraagt wederom om nieuw werk, en moet wachten tot de volgende synchronisatie. Om dit te voorkomen spreken we af om in de searchmatrix alleen de valide wegen op te slaan. Deze worden achterelkaar gezet in de rijen van de matrix, en worden afgesloten met de waarde -1. De allerlaatste waarde in een rij van de matrix is nog altijd beschikbaar om bij te houden waar we zijn in de rij van de zoekmatrix.
Het algoritme dat alleen de valide plaatsen uitzoekt,ziet er zo uit:

```
Algorithm *onlyValid(int *allsearch, int *bord, int row):
```

```
cnt <- 0;

for(t=0;t<x;t++) {
 if (validate(bord,row,allsearch[t])==TRUE) {
  returnarray[cnt] <- allsearch[t];
  cnt++;
  }
 }
 returnarray[cnt] <- -1;

return returnarray;
}

End Algorithm
```

Dankzij gebruik van dit algoritme kan de masterprocessor nagenoeg blindelings werk verdelen; het hoeft alleen de goede rij van voor af aan te doorlopen, totdat het een -1 tegen komt. Dan roept het het zogenaamde *move(int stoprow)*-algoritme aan, dat zoals in de vorige paragraaf op zoek gaat naar een ander pad. Van dit pad worden de tosearch-arrays weer in de searchmatrix opgeslagen, en vervolgens kan er weer een rij lang makkelijk werk verdeeld worden.

4.4.6 Clock-algoritme

Nu we de werkverdelingsstrategie hebben opgesteld kunnen we het clockalgoritme schrijven. Dit is het algoritme dat van tijd tot tijd wordt aangeroepen tijdens het zoeken, om alle processoren te synchroniseren. Tijdens deze synchronisatie wordt er hoofdzakelijk werk gestuurd van de master-processor naar de processoren die werk nodig hebben. Het clock-algoritme moet ook ervoor zorgen dat als alle processoren geen werk meer hebben, en er ook geen werk meer is om te verdelen, dat het zoekalgoritme termineert.
Het algoritme dat dit alles doet is redelijk complex, dus ik zal stap voor stap behandelen wat er precies geimplementeerd zal worden. In de variabele *stoprow* staat de rij waar de rundown-methode vast liep. We beginnen te zoeken op de rij voor de stoprow, en de rij waarop we zoeken kan later veranderd worden dankzij het werk verdelen. Het is dus handig om nog een variabele *currentrow* te gebruiken, om bij te houden op welke rij we aan het zoeken zijn. Met deze nieuwe variabele schrijven we het algoritme als volgt.
Aan het begin moet het algoritme synchroniseren; dit om de work-requests te mogen ontvangen. Daarna gaat het algoritme eerst kijken of het er door *alle* processoren werk wordt gevraagd, en of $currentrow < 0$; dan geldt dat elke processor geen werk heeft, en dat er geen werk meer is om te verdelen. Het algoritme moet dan stoppen met zoeken, en dus wordt het sendTerminate() algoritme aangeroepen. Na deze controle gaan we verder met het echte werk verdelen, als processoren erom vragen. De rij wrq wordt van 0 tot P doorlopen, en als $wrq[i]$ ==TRUE, moet er werk worden verstuurd.
Dit doen we door gebruik te maken van onze searchmatrix. Eerst incremeren we de teller van de searchmatrix, en zetten we de *current*-variabele naar dezelfde waarde. Deze variabele houdt dus ook bij waar we zijn in de houdige tosearch-array. In principe is het niet nodig, maar vanwege leesbaarheid van het algoritme is het wel gewenst. Vervolgens kijken we eerst even naar de waarde van *current*; als deze gelijk is aan x, zijn we het einde van de rij gepasseerd. We moeten dus volgens ons werkvedelingsalgoritme een ander pad vinden naar dezelfde rij (*currentrow*). Hetzelfde moet er gebeuren als $tosearch[current] = -1$, er zijn dan immers geen plaatsen meer op currentrow waar we een koningin legaal kwijt kunnen. Vervolgens stoppen we in tosearch de array die staat op searchmatrix[currentrow], minus het laatste element (dat was de teller). Nu beginnen aan het daadwerkelijk werk doorsturen. Eerst slaan we de *currentrow* op in een vaste plek in de rij *prCurrent*. Dit omdat als een andere processor hierna nog werk wil, *currentrow* kan veranderen terwijl we voor deze processor de huidige waarde willen doorsturen. Ook slaan we de huidige bordpostitie

op in een vaste plek in een matrix (*modBord*), vanwege dezelfde reden; de bordposities kunnen veranderen dankzij het werkverdelingsalgoritme. Het bord bijbehorend bij waar we op het moment zoeken is overigens opgeslagen in de variabele *globalbord*. De variabele *bord* wordt gebruikt bij het zoeken, en zal dus per processor verschillen. Voordat we dit alles doorsturen naar de processor, passen we het bord dat we willen doorsturen nog aan zodat de positie *currentrow* op dat bord gelijk is aan tosearch[current]. Immers, de werkbehoevende processor moest juist die deelboom doorzoeken. Ook is het handig om de slaveprocessor te laten weten dat het nieuw werk heeft, door zijn outofwork-variabele naar FALSE te zetten. Samengevat komt bovenstaande neer op de volgende pseudo-code.

```
Algorithm clock():

sync();

temp <- TRUE;
for (t=0;t<P;t++) {
 if (wrq[t]==FALSE)
  temp <- FALSE;
 }

if (temp==TRUE)
 if (currentrow<0)
  sendTerminate();

for (t=0;t<P;t++) {
 if (wrq[t]==TRUE) {
  teputten <- FALSE;
  searchmatrix[currentrow][x]++;
  current <- searchmatrix[currentrow][x];
  if(current==x)
   move(currentrow);
  if(tosearch[current]==-1)
   move(currentrow);
  tosearch <- searchmatrix[currentrow];
  prCurrent[t] <- currentrow;
  bsp_put(t,&prCurrent[t],&currentrow,0,SZINT);
  if (currentrow>-1) {
   modBord[t] <- globalbord;
   modBord[t][currentrow] <- tosearch[current];
   bsp_put(t,modBord[t],bord,0,x*SZINT);
   bsp_put(t,false,&outofwork,0,SZINT);
   }
  }
 wrq[t]=FALSE;
 }

sync();

End Algorithm
```

We zien hier een verwijzing naar het algoritme *move(int currentrow)*. Dat is het algoritme dat ervoor zorgt dat we eerst een rij teruggaan, en dan een ander pad zoeken naar een punt op *currentrow*. Dat

algoritme is opgesplitst in twee delen; *move* gaat omhoog in de boom, en *moveDown* omlaag:

```
Algorithm move(int rowtorecurto):

currentrow--;
searchmatrix[currentrow][x]++;
current <- searchmatrix[currentrow][x];

if (searchmatrix[currentrow][current]==-1)
 move(rowtorecurto);
else {
 if (current==x)
  move(rowtorecurto);
 else {
  globalbord[currentrow] <- searchmatrix[currentrow][current];
  moveDown(rowtorecurto);
  }
 }

End Algorithm

Algorithm moveDown(int rowtorecurto):

currentrow++;
searchmatrix[currentrow] <- onlyValid(genSearch(),globalbord,currentrow);
searchmatrix[currentrow][x] <- 0;
current=0;
if (searchmatrix[currentrow][0]==-1)
 move(rowtorecurto-1);
else {
 globalbord[currentrow] <- searchmatrix[currentrow][current];
 if (currentrow<rowtorecurto)
  moveDown(rowtorecurto);
 }

End Algorithm
```

Met deze twee algoritmes toegevoegd is onze implementatie voor een parallel algoritme compleet. We merken even nogmaals op dat de zoekstrategie nu compleet afhangt van de functie genSearch(). Voor de volledigheid geef ik nog de genSearch algoritmes voor normale DFS en de random DFS.
Normal DFS:

```
Algorithm *genSearch():

for (t=0;t<x;t++)
 toReturn[t]=t;

return toReturn;

End Algorithm
```

Random DFS:

```
Algorithm *genSearch():

return randomArray(x);

End Algorithm
```

Het algoritme van randomArray(int x) is voorheen al gegeven.

Hoofdstuk 5

Experimenten

5.1 Sequentiele resultaten

We gaan eerst de sequentiele algoritmes tegenelkaar uitspelen. Voor het meten van tijd gebruiken we hier het aantal clocks, daar hiermee de tijd preciezer kan worden gemeten dan door standaard timers. We bekijken hier problemen van 20, 25 en 30 koninginnen. Voor het DFS-algoritme meten we telkens maar 2 keer; als het goed is is het aantal clocks ongeveer hetzelfde. Voor random DFS gebruiken we 10 metingen, zodat we een redelijk beeld krijgen van gemiddelde oplostijden en de variatie daarvan. Deze tests zijn uitgevoerd op Grit; een multiprocessor computer van de Wiskunde-afdeling in utrecht.

$x = 20$:

Normal DFS	8670	8690								
Random DFS	0	0	10	40	40	20	0	10	720	10

$x = 25$:

Normal DFS	3250	3230								
Random DFS	150	1480	20	20	20	940	50	10	10	20

$x = 30$:

Normal DFS	–	–								
Random DFS	50	340	120	80	160	7921	10	140	840	70

Uit deze resultaten blijkt dat random-DFS veel sneller is dan normal-DFS. Het normale DFS algoritme was niet bij machte om het 30-koninginneprobleem binnen een redelijke tijd op te lossen (het duurde zowieso langer dan een kwartier, met als resultaat dat ik van de server werd gegooid). De gemiddelde oplostijd voor $x = 20$ van random-DFS is 85 clocks. Voor $x = 25$ is dat 272 en voor $x = 30$ $983,1$. Er lijkt geen direct verband te bestaan tussen de normale oplostijd en de gemiddelde random oplostijd.

5.2 Parallelle resultaten

De parallelle resultaten heb ik in drie secties opgedeeld. Het eerste deel behandelt het clock-probleem. Daarna wordt er kort gekeken naar het simpel geparalleliseerde algoritme. Het derde deel laat zien hoe het parallelle algoritme zoals gepresenteerd aan het eind van hoofdstuk 4 presteert.

5.2.1 Clocktest

Tijdens het zoeken moet er van tijd tot tijd gesynchroniseerd worden. In deze sectie kijken we hoeveel tijd er tussen twee clocks moet zitten, willen we goede resultaten te behalen. We testen nu op teras, een van de nationale (Nederlandse) supercomputers, en op een 20-koninginnenprobleem. Het aantal processoren dat we gebruiken zijn 4, 8, 16. We gebruiken een normaal DFS algoritme, zodat enig tijdsverschil in de oplostijden alleen kan zijn veroorzaakt door het verschil in clock-intervallen. De resultaten zijn als volgt:

Number of Processors	Clock interval	Solve time
4	100	3
4	1000	2
4	10000	2
4	100000	1
4	10000000	1
8	100	2
8	1000	1
8	10000	1
8	10000000	1
16	100	2
16	1000	1
16	10000000	1

Uit deze gegevens kunnen we al een conclusie trekken: we zien dat heel vaak clocken (clockinterval van 100), slecht is. Verder zien we dat alleen clocken *na* de search minstens net zo goede resultaten geeft een goede clock-interval. Dit is een vreemd resultaat; maar misschien ook niet een juiste. De solve-tijden zijn in dit experiment zo klein dat we de verschillen niet goed zien. Om dit te onderzoeken, pakken we een groter probleem, en gebruiken we 14 processoren. We nemen $x = 29$. De resultaten:

Clock interval	Solve time
1000	38
10000	15
100000	13
10000000	14

Hieruit blijkt dat het goed kiezen van een clock-interval wel degelijk beter is dan het alleen clocken nadat alle processoren geen werk meer hebben. In het bepalen van een goede clockinterval is het noodzakelijk het aantal processoren mee te wegen. Hieronder volgt een tabel met het aantal processoren uitgezet tegen de tijd die nodig is om een clock uit te voeren.

P	Clock-time
16	0
17	1
25	2
32	3

Als we nu op een of andere manier konden bepalen hoeveel tijd een bepaalde hoeveelheid werk kost, konden we het percentage tijd besteed aan clocken *bij benadering* sturen. In onze algoritmes gebruiken we de variabele *counter* om de hoeveelheid werk weer te geven. We kunnen ons sequentieel programma zodanig manipuleren dat het bij de uitkomst ook de variabele *counter* weergeeft. Dan kunnen we de oplostijd tegenover de werkhoeveelheid uitzetten, door gewoon lukraak wat probleem groottes uit te kiezen en deze op te laten lossen.

Solve time	Work load
1	48683
3	199635
11	411608
12	397699
15	454213
39	1737188

Er valt gelijk iets op, namelijk de verlaging van de werkhoeveelheid terwijl de oplostijd omhoog is gegaan

(bij solve time 11/12). Kennelijk is de aanname dat de werkhoeveelheid evenredig is met het neerzetten van een koningin niet helemaal correct. Dit is een probleem waarvoor ik helaas geen tijd had om te corrigeren.
Men kan zich misschien afvragen wat het nut van deze waarnemingen zijn, maar door het sturen van het percentage aan clocks kan het algoritme op een geheel ander nivo geoptimaliseerd worden. Er kan onderzocht worden wat het beste clock-percentage is bij 17 processoren, of bij 32, enzovoorts. Hierdoor is het mogelijk om de beste clock-percentages voor willekeurige processor-aantallen te voorspellen, en kunnen we grote koninginnenproblemen zo efficient mogelijk oplossen.
Nog even een voorbeeldje hoe je het percentage ongeveer kunt bepalen. Stel we hebben 17 processoren. We zien dat de clock-time 1 bedraagt. Als we willen dat er 25 procent van de tijd wordt geclockt, moeten we de counter-interval rond 199635 zetten. Dit staat immers ongeveer voor 3 seconden tijd, en krijgen we dus een 1 op 3 verhouding van clocken vs werken; 25 procent.

5.2.2 Prestaties van Simpele Parallelisering

De simpele algoritmes zouden gegarandeerd minstens even snel zijn als hun sequentiele versie. Hier gaan we dat testen. Ook kijken we in het bijzonder naar wat er gebeurt met de random-DFS; deze zou beter moeten presteren dan normale DFS. Hier zijn wat testresultaten (getest op Teras).
Normal DFS:

x	P	Solve time
20	1	3
20	2	1
20	5	0
28	1	92
28	2	18
28	4	0
28	7	45
30	15	111

We zien dat parallelisatie zeker snelheidswinst geeft. Het sequentiele DFS-algoritme zou over $x = 30$ een behoorlijk tijdje doen, maar met behulp van 15 processoren wordt er relatief snel een oplossing gevonden. Bij $x = 28$ zien we in het tabel dat het opsplitsen van de eerste *tosearch*-array over de processoren soms voor gelukjes kan zorgen. Over 2 en 4 processoren waren de resultaten heel goed; er stonden snelle oplossingen in het begin van een van de deelrijen die een bepaalde processor moest onderzoeken. Bij $P = 7$ zijn al die snelle oplossingen blijkbaar niet meer in het begin van de deelrij te vinden, en duurt het oplossen ietwat langer; maar het is nog steeds sneller dan het sequentiele algoritme ($P = 1$).
Laten we nu kijken naar random-DFS, en of hier de snelheidswinsten net zo groot zijn.
Random DFS:

x	P	Avg. Solve time (5 runs)
60	1	62,2
60	3	14,8
60	6	1
60	10	0,2
70	1	23,4
70	7	0,8
70	10	1,8

We zien hier ook dat random-DFS veel sneller is dan de normale versie. Problemen rond $x = 30$ waren binnen een oogwenk opgelost, dus die heb ik uit bovenstaande tabel gelaten.
Deze 'simpele' parallelisatie zorgt al voor een aardige speedup. Men kan zich afvragen hoe veel sneller een Global Parallel Search zal zijn.

5.2.3 Prestaties van GPS

Aan het eind van hoofdstuk 4 hebben we een parallel algoritme opgesteld, welke we nu gaan testen op prestatie. We beginnen met het parallelle normale DFS algoritme, op Teras. We vergelijken de eerste twee resultaten met de oplostijd verkregen van het uitvoeren van hetzelfde algoritme met maar 1 processor.

x	P	Oplostijd	Clock-interval
20	1	20	nvt
20	2	2	50000
20	4	1	100000
25	1	6	nvt
25	2	1	50000
25	4	0	100000
29	4	27	50000
29	8	17	200000
29	16	9	500000

We zien dat het gebruik van meerdere processoren het algoritme aanzienlijk versnelt. Het 30 koninginnen probleem kon helaas niet binnen een redelijke tijd opgelost worden.
Laten we nu kijken naar het random DFS algoritme. Sequentieel liep deze al veel sneller dan de normale DFS, en we hopen dat dat parallel ook zo is.

x	P	Oplostijd	Clock-interval
30	1	0	nvt
30	1	0	nvt
30	1	0	nvt
50	1	2	nvt
50	2	0	50000
50	4	2	100000
50	4	0	100000
70	1	3	nvt
70	1	7	nvt
70	1	90	nvt
70	4	21	100000
70	4	1	100000
70	4	0	100000
100	4	15	100000
100	8	0	250000
100	16	164	600000
150	8	40	250000
150	16	2	750000
150	32	38	3000000

We zien dat ook parallel het random algoritme voor een grote snelheid zorgt. Er moet nog wel worden opgemerkt dat het algoritme soms een zeer slecht pad kiest; het duurt dan vele malen langer dan gemiddeld om een oplossing te vinden. Een voorbeeld zien we bij $x = 70$.
Door gebruik van meerdere processoren lopen we veel sneller, ook al hebben we een slecht pad gekozen. Een voorbeeld daarvan zien we bij $x = 100$ en $P = 4$, alsook bij $x = 150$ en $P = 8$.
Wat verder nog opvalt is dat het gebruik van teveel processoren voor een tegenvallende prestatie zorgt. Een reden hiervoor was al gegeven; de master processor krijgt het dan moeilijk met werk verdelen. Deze moet dan relatief vaak het *move*-algoritme aanroepen, wat tijd kost. Kiezen we het aantal processoren wat terughoudender, dan zien we zeer snelle resultaten ontstaan. Zie bijvoorbeeld bij $x = 100$ en $P = 8$ en $x = 150$ en $P = 16$. Ook is de ervaring dat het clock-interval het beste aan de hoge kant kan worden ingevoerd.

Hoofdstuk 6

Conclusie

Gegeven het x-koninginnenprobleem heb ik geprobeerd een zo snel mogelijk parallel algoritme te bedenken, dat een oplossing van het probleem kan vinden. Dit is redelijk goed gelukt, in de vorm van een parallel random DFS algoritme.
Aan het begin van het project, dacht ik, na wat sequentiele test programma's te schrijven, dat ik waarschijnlijk met moeite een 100 koninginnenprobleem parallel aankon; sequentieel duurde een probleem van 30 koninginnen al een eeuwigheid om op te lossen. Nu blijkt het echter geen probleem te zijn; met 8 processoren lost random DFS het meestal binnen tien seconden op. Soms zelfs binnen een luttele seconde! Ik was hier aangenaam verrast over. Ook een probleem van 150 kan het programma nog goed aan; maakt me nieuwsgierig wat voor probleem het aankan als Teras er een nachtje ongestoord aan kon werken...
Natuurlijk zijn er nog wel mogelijkheden tot verbetering. Zoals in het stukje over de clock-tijdsintervallen te lezen is, is het vinden van zo'n interval een interessant probleem. Als we dit realtime kunnen laten afhangen van het aantal processoren, en de huidige grootte van deelbomen, kunnen we mogelijk betere resultaten krijgen dan dat we nu hebben.
Ook loont het waarschijnlijk om te analyseren wat theoretisch, of probabilistisch, de beste plaats zou zijn voor een koningin, gegeven een deel van een bord en een rij. Als we dit zouden weten, kunnen we een heuristieke functie opstellen; deze functie zou een voorkeur kunnen geven voor bepaalde kolom-posities, waarop het waarschijnlijker is een oplossing te vinden. Maar vanwege de ongeordend ogende oplossingen die het programma gaf, lijkt me dat het random-algoritme niet ver zal onderdoen aan zo'n heuristiek algoritme.
Verder kunnen er nog andere zoekstrategieen worden geprobeerd. Ik had er zelf nog een bedacht, maar ben er helaas niet aan toegekomen het te implementeren. Dit algoritme dat ik maar even *Random Traceback* noem, genereert om te beginnen een random gevuld bord van grootte x. Hier staan natuurlijk geheid koninginnen op die elkaar diagonaal kunnen slaan. Daarom scannen we van de bovenste rij naar beneden, opzoek naar de eerste koningin die een andere koningin *boven* haar eigen rij slaat. Als er een wordt gevonden, laten we zeggen op positie i, dan worden er een reeks random transposities uitgevoerd op de koninginnen van i tot en met x. Dit gaat zo door totdat i voldoende dicht bij x ligt, om een random DFS-algoritme te starten. Dit random DFS-algoritme blijft vanaf dat punt doorzoeken, ook al was er in de deelboom van i geen oplossing; we gebruiken het random genereren van kolomposities dus als een soort *rundown*-algoritme.
Al met al lijkt me dit een geslaagd project, eentje waar ik zeer waarschijnlijk van tijd tot tijd nog even aan zal werken. In de appendices staan de uiteindelijke implementaties van de algoritmes nog ter inzage. Voor opmerkingen en vragen, kan een ieder me bereiken op onderstaand emailadres; ik zie ze graag tegemoet!

Albert-Jan Yzelman

ajy777@gmail.com

Hoofdstuk 7

Bronvermeldingen

Parallel Scientific Computation, Rob H. Bisseling. Oxford University Press
Introduction to Parallel Computing, second edition, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar. Addison-Wesley

`http://mathworld.wolfram.com/QueensProblem.html`