Parallel clusteren in het Ising model

A. van Dam en G.J. Wolterink

Juni 2000

Samenvatting

Vele facetten van het Ising model zijn reeds onderzocht en er zijn dan ook vele algoritmen om dit model te simuleren. In dit verslag wordt bekeken hoe een cluster-algoritme (Swendsen-Wang) voor het twee-dimensionale Ising model geparallelliseerd kan worden. Met het verkregen programma wordt gekeken naar de maximale clustergrootte bij verschillende systeemtemperaturen, waarmee een benadering van de kritische temperatuur gevonden kan worden.

1 Modellering

Centraal staat het twee-dimensionale Ising model op een gelijkmatig rechthoekig rooster. Het Ising-model modelleert een atoomrooster, waarbij ieder atoom een + of − spin heeft ($S_{ij} = 1$ resp. $S_{ij} = 0$). Door het omdraaien van spins (het zogenaamde 'flippen') veranderen de magnetische eigenschappen van het gebied.

Bij de simulatie wordt een zogenaamd cluster-algoritme gebruikt. Deze klasse van algoritmen vormen eerst grote aaneengesloten gebieden van gelijke spins, die vervolgens met een bepaalde kans omgedraaid worden. Specifiek wordt hier het Swendsen-Wang algoritme gebruikt; dit algoritme is in 1987 voorgesteld door Swendsen en Wang [2].

1.1 Rooster

In het algemeen wordt het Ising model gediscretiseerd op een rechthoekig rooster met vierkante roostercellen. Zoals in figuur 1 is te zien, zijn de coördinaten nu iets anders geformuleerd. De elementen in twee opeenvolgende rijen zijn ten opzichte van elkaar een halve eenheid verschoven. Doordat het gehele rooster over een hoek van 45° is geroteerd, staan de roosterpunten weer rechthoekig ten opzichte van elkaar. Er is voor gekozen om de ten opzichte van elkaar verschoven kolommen als twee afzonderlijke kolommen te nummeren. Hetzelfde geldt natuurlijk voor de rijen en zo ontstaat de nummering zoals in de figuur. Het zo ontstane rooster is m rijen hoog en n kolommen breed. Voor het gemak worden m en n even genomen. Door de aangepaste oriëntatie heeft een gebied $(m \cdot n)/2$ roosterpunten. Bij het clusteren wordt gekeken naar de spin van de buurcellen. Een punt (i, j) heeft de buren $(i-1, j-1)$, $(i+1, j-1)$, $(i-1, j+1)$ en $(i+1, j+1)$.

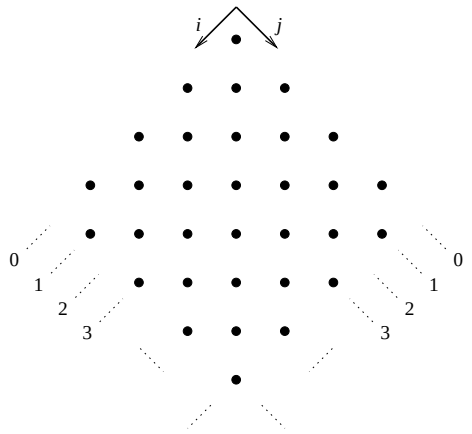
Om een 'oneindig' rooster te krijgen worden cyclische randen gebruikt. Een configuratie is een toestand van het rooster waarbij ieder roosterpunt een bepaalde spin heeft.

1.2 Cluster-algoritme: Swendsen-Wang

Het gebruikte cluster-algoritme werkt als volgt:

1. Begin met willekeurige start-configuratie en kies een systeemtemperatuur T.
2. Maak verbindingen tussen buurpunten met gelijke spin, met kans

$$P_{\text{add}} = 1 - e^{-2\beta J} \qquad (\beta = \frac{1}{k_B T}). \tag{1}$$



Figuur 1: Coordinaat-systeem in het ruit-rooster

3. Draai de spin van de zo ontstane clusters met kans 0.5 om (ieder cluster afzonderlijk).

4. Ga weer verder met stap 2.

In (1) geeft J het energieverschil aan wanneer een spin omgedraaid wordt. In de implementatie wordt een + spin met 1 aangeduid en een − spin met 0; de waarde van J is nu 1. k_B is de constante van Boltzmann.

2 Parallel Swendsen-Wang

2.1 Datadistributie

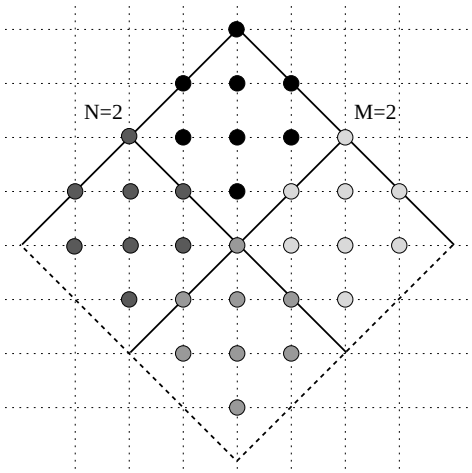
Een eerste stap in de richting van parallellisatie is het kiezen van een geschikte datadistributie. Aangezien voor de roosterpunten altijd naburige informatie nodig is, wordt het rooster in blokken over de processoren verdeeld. Het gehele rooster bestaat zo uit M processorrijen en N processorkolommen. De reeds in sectie 1.1 besproken ruitvorm is gekozen met het oog op de communicatie over de randen van een processorgebied. Er wordt nu een factor $\sqrt{2}$ bespaard op de communicatie. In figuur 2 is een verdeling van een rooster afgebeeld met $M = N = 2$ en $m = n = 4$.

2.2 Lokaal clusteren

Zodra stap 2 van het Swendsen-Wang algoritme (sectie 1.2) is voltooid, is voor ieder paar naburige roosterpunten bepaald of er al dan niet een verbinding tussen beide is gelegd; in het vervolg worden deze verbindingen 'bonds' genoemd. Alvorens naar het algoritme te kijken, wordt eerst een nieuwe indexering geïntroduceerd; in het vervolg zullen punten in het rooster niet meer door i en j geïdentificeerd worden, maar door een enkele index i. Werd bijvoorbeeld in het oude systeem een punt nog aangeduid met $(\tilde{i}, \tilde{j})$, nu wordt dit:

$$i = \tilde{i} \cdot \frac{n}{2} + \tilde{j}.$$

Hierna worden eerst lokaal de clusters van verbonden roosterpunten geïdentificeerd. Dit lokale clusteren geschiedt volgens algoritme 2.1. Hierin is G het rooster, $V[G]$ de verzameling roosterpunten en $E[G]$ de verzameling gemaakte bonds. Allereerst wordt van ieder roosterpunt een losstaand cluster gemaakt met MakeSet(v). Vervolgens worden alle gemaakte bonds langsgelopen en wordt met FindSet gekeken of de twee roosterpunten waartussen de bond ligt nog in



Figuur 2: Verdeling van het rooster in processor-blokken

Algoritme 2.1 lokaal clusteren

Algoritme MAKELOCALCLUSTERS
  for all $v \in V[G]$
    do MAKESET(v)
  for all $(u, v) \in E[G]$
    do if FINDSET(u) $\neq$ FINDSET(v)
      then UNION(u, v)

verschillende clusters zitten. Wanneer dit het geval is, worden met UNION(u, v) de clusters waar u en v in zitten tot één cluster samengevoegd.

Een cluster wordt als het ware 'geleid' door een *root*. Iedere roostercel heeft een lokaal indexnummer en zodoende wordt een cluster geïdentificeerd door het indexnummer van de root.

2.3 Globaal clusteren

Nu de clusters op de deelroosters per processor volledig gevormd zijn, moet er tussen de processoren gecommuniceerd worden, waarna clusters op verschillende processoren samengevoegd kunnen worden tot een *globaal cluster* wanneer de beide clusters over de rand met elkaar verbonden zijn. Hiervoor moeten echter eerst nog wat afspraken over notatie gemaakt worden. Ten eerste wordt een processor geïdentificeerd door een processorrij s en een processorkolom t. Soms wordt ook de processor-id gebruikt:

$$\text{pid} = s \cdot N + t.$$

Ten tweede moeten de clusternummers uniek zijn en daarom moeten de lokale clusternummers globaal gemaakt worden:

$$i_{\text{glob}} = \text{pid} \cdot \frac{m \cdot n}{2} + i_{\text{loc}}. \tag{2}$$

De identificatie van de randen geschiedt door in gedachten het rooster 45° tegen de wijzers van de klok in te draaien en vervolgens termen 'bovenrand' etc. te gebruiken.

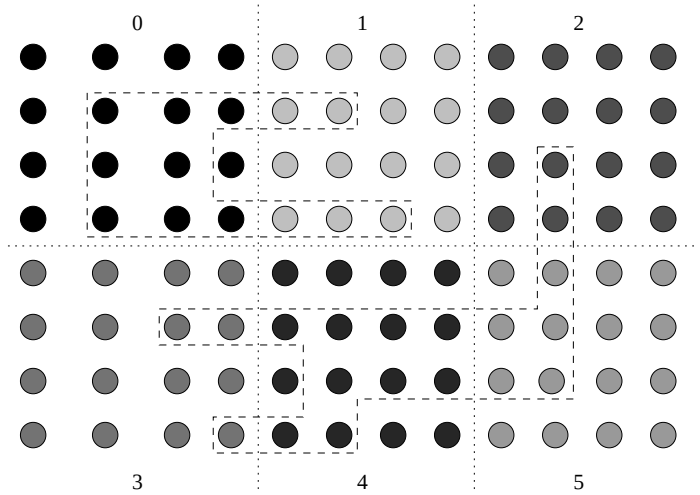
Het globale clusteren staat in algoritme 2.2. Voor het leggen van bonds over processorgrenzen heen wordt door de buurprocessoren eerst hun spins op linker- en bovenrand gestuurd. Vervolgens bepaalt iedere processor voor zijn rechter- en onderrand welke bonds er gelegd moeten worden met behulp van de kans in vergelijking (1). Tot slot verzendt iedere processor de roots van de clusters over de rechter- en onderrand naar de desbetreffende processoren.

Algoritme 2.2 globaal clusteren

```
Algoritme MakeGlobalClusters
    (0)  SendEdgeSpins to left and upper neighbour
    (1a) MakeEdgeBonds with right and lower neighbour
         SendClusterRoots to right and lower neighbour
         until nothing changed
    (1)      B_r := {clusterID(cell)|bonded cell at right edge}
             B_b := {clusterID(cell)|bonded cell at bottom edge}
             Put B_r in P(s,(t+1) mod N)
             Put B_b in P((s+1) mod M,t)
    (2)      for all received neighbourID's at left and top edge
                 change := false
                 if neighbourID < myID
                 then myID := neighbourID
                      change := true
                 else if neighbourID > myID
                 then Put myID in neighbourProc at neighbourID
                      change := true
             Broadcast change to P(*,*)
```

Nu heeft iedere processor genoeg gevens om zijn clusters langs de linker- en bovenrand te koppelen met buurclusters. Als regel wordt genomen dat het cluster met het laagste clusterID zijn nummer behoudt. Indien het nummer van het buurcluster kleiner is dan het eigen clusternummer, wordt lokaal het eigen clusternummer aangepast. In het andere geval wordt met een `put` direct het clusternummer bij de buurprocessor aangepast.

Het globale clusteren gaat door totdat er bij geen enkele processor nog veranderingen in clusterID's zijn. De reden dat dit nodig is blijkt uit figuur 3.



Figuur 3: Globaal cluster over meerdere processoren

Wanneer in de eerste ronde het clusternummer van het gemeenschappelijke cluster op processor 5 verandert in dat van processor 2, is dit nog niet doorgegeven aan hetzelfde cluster op processor 3 en 4. De gedeelten van het gemeenschappelijke cluster hebben daar na de eerste ronde het nummer van het cluster op processor 3 aangenomen. Er zijn nog twee ronden nodig om het cluster dat op processor 2, 3, 4 en 5 zit overal hetzelfde globale clusternummer te geven.

Aan het einde van superstap (2) zendt een processor aan alle processoren of er bij hem cluster-ID's veranderd zijn of niet.

2.4 Flippen van globale clusters

De volgende stap in het Swendsen-Wang algoritme is het flippen van de gevormde clusters, met kans 0.5. In algoritme 2.3 wordt dit beschreven. Alle processoren hebben een lijst van de locale

Algoritme 2.3 Het swappen van globale clusters

```
Algoritme SwapGlobalClusters
        npts := m · n/2
   (0)  for all local clusters do
            if clusterID  div npts = pid and clusterID  mod npts = rootID
            then if random(1)< 0.5
                 then swap[clusterID] := 1
                 else swap[clusterID] := 0
   (1)  for all local clusters do
            get swap from P(clusterID  div npts)
   (2)  for all local clusters do
            if swap[clusterID] = 1
            then for all cells in cluster do
                 spin[cell] := −spin[cell]+1
```

clusters bijgehouden en die wordt in superstap (0) doorlopen. Indien de processor ook globaal de 'eigenaar' van het cluster is (als de globale root zich op die processor bevindt), wordt de swap-variabele gezet. Deze wordt in de tweede superstap door de processoren gelezen die ook een deel van dat cluster hebben. Dit wordt gedaan met een `get` omdat de globale eigenaar van een cluster niet kan weten op welke processoren er nog delen van het cluster aan hem zijn vastgegroeid. In de laatste superstap worden vervolgens door alle processoren lokaal de clusters doorlopen en eventueel geflipt, op basis van de in de vorige superstap verkregen beslissingsvariabele swap.

2.5 Kosten

De kosten voor het Swendsen-Wang algoritme in sectie 1.2 zijn $T = k_{\max}(T_{\text{cluster}} + T_{\text{flip}})$. $k_{\max}$ is het aantal iteraties in het Swendsen-Wang algoritme. Dit geldt voor zowel reken- als communicatiekosten.

De kosten van het clusteren zijn de som van het lokale en globale clusteren. Bij het lokale clusteren zijn de communicatiekosten sowieso 0, er worden ook nauwelijks significante floating-point berekeningen gedaan. Ook de routines MakeSet, FindSet en Union in algoritme 2.1 kosten gemiddeld constante tijd. Toch kost dit gedeelte wel tijd en wordt voor de rekenkosten volstaan met een afschatting $T = \mathcal{O}(m \cdot n)$.

Bij het globale clusteren gaan wel communicatiekosten meespelen; $T_{(0)} = (n/2 + m/2)g$. In superstap (1a) van algoritme 2.2 heeft alleen routine SendClusterRoots communicatiekosten: $T_{(1a)} = (n/2 + m/2)g$. De rekenkosten zijn $T_{(1a)} = (n/2 + m/2) \cdot 2 \cdot s \cdot 12$. Hierbij is s de kans dat twee naburige cellen gelijke spin hebben. De waarde 12 is een afschatting voor de kosten van één aanroep van de random-number-generator op een Cray T3E. Vervolgens wordt een lus gestart om de globale clusters bij te werken. De lengte van deze lus wordt afgeschat op $p^{d_{\min}/2}$, waarin $d_{\min}$ een kritische exponent van het globale cluster-label-proces is [1]. Binnen de lus zijn de communicatiekosten: $T_{(1)} = ((n/2+m/2) \cdot s \cdot P_{\text{add}} \cdot 2)g$, $T_{(2)} = ((n/2+m/2) \cdot 2 \cdot s \cdot P_{\text{add}} \cdot r + p)g$. Hierin geeft r de gemiddelde kans aan dat het clusterID op een processor kleiner is dan die op de buurprocessor (en dat die processor dus eigenaar wordt van het betreffende cluster). De rekenkosten in superstap (1) zijn $T_{(1)} = \mathcal{O}((n/2 + m/2) \cdot s \cdot P_{\text{add}})$. De rekenkosten in superstap (2) bestaan uit enkel optellingen van indices langs de rand en zijn dus verwaarloosbaar.

Tot slot de kosten van het swappen van clusters. Superstap (1) bevat communicatie: $T_{(1)} = (c_{\text{lok}} - c_{\text{own}}) \cdot g$. De overige twee bevatten alleen rekenstappen: $T_{(0)} = 12c_{\text{own}}$, $T_{(2)} = \mathcal{O}(m \cdot n/2)$. Hierbij geeft c_{lok} het aantal lokale clusters op een processor aan en c_{own} het aantal lokale clusters waarvan de processor globaal eigenaar is.

Voor de totale kostenfunctie moeten nu eerst afschattingen voor de gebruikte constanten worden gevonden. De waarde van s is afhankelijk van de iteratieslag. In het begin zijn de spins random verdeeld en is $s = 1/2$. Zodra de clusters groter worden, is de kans op gelijke spin binnen één cluster natuurlijk 1, op de rand is deze kans kleiner. Omdat echter al snel het aantal cellen binnen een cluster groter is dan het aantal cellen op de rand, wordt als bovengrens $s = 1$ gekozen. De waarde van r wordt afgeschat op 0.5. De waarden van c_{lok} en c_{own} zijn afhankelijk van systeemtemperatuur T, aantal iteraties $k_{\max}$ en de systeemgrootte. Verder onderzoek zou hiervoor een juiste afschatting moeten leveren. De totale kostenfunctie staat in vergelijking (3).

$$\begin{aligned} T_{\text{tot}} &= k_{\max}\Big(\mathcal{O}(m \cdot n) + (n+m)g + 12(n+m) + p^{d_{\min}/2}\left(0.9(n+m)g + pg + \mathcal{O}(n+m)\right) \\ &\quad + (c_{\text{lok}} - c_{\text{own}})g + 12c_{\text{own}} + \mathcal{O}(m \cdot n)\Big) \qquad (3) \\ &= \mathcal{O}\left(k_{\max} m \cdot n\right) + \mathcal{O}\left(k_{\max} p^{d_{\min}/2}\left(m+n\right)\right) g \end{aligned}$$

3 Implementatie

Het in sectie 2 beschreven algoritme is geimplementeerd in C++. Voor het parallel programmeren is gebruik gemaakt van het BulkSynchroonParallel-model (BSP). De programmacode staat in bijlage A.

bspising Op standaardwijze kunnen de besproken systeemparameters ingevoerd worden vanaf de commandoregel. Ook moet de systeemtemperatuur T ingevoerd worden; hierin is de constante van Boltzmann al verwerkt. In feite moet dus $1/\beta$ uit vergelijking (1) ingevoerd worden. Het echte algoritme begint bij de lus `k=1` tot `kmax`. Hierin wordt steeds één slag van het Swendsen-Wang algoritme uitgevoerd. Eerst worden in `isingLocalMonteCarloStep` de lokale bonds gelegd en daarmee de lokale clusters gevormd. Vervolgens worden in `isingGlobalMonteCarloStep` de globale clusters gevormd en geflipt.

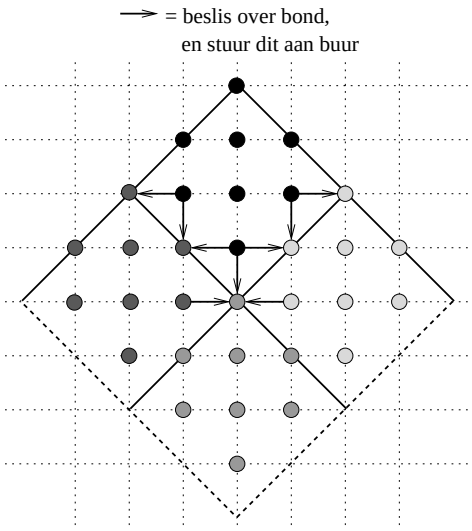
isingLocalMonteCarloStep Op basis van de spingegevens in de array `spins` kunnen de bonds gelegd worden. Methode `makeLocalBonds` doet dit door het rooster langs te lopen. Er worden alleen bonds gelegd naar beneden en naar rechts. Voor cel `i` wordt in `bbonds[i]`, respectievelijk `rbonds[i]` met een 1 of 0 aangegeven of er wel of niet een bond ligt.

Met deze gegevens kunnen de lokale clusters gevormd worden in `makeLocalClusters`. Deze methode gebruikt voor de opslag van de clusters drie arrays. In `root[i]` staat het celnummer van de root van het cluster waarin cel `i` zit. In `pred[i]` staat het celnummer van de cel die voor cel `i` in hetzelfde cluster als `i` zit. De volgende cel wordt in `succ[i]` opgeslagen.

De methode doorloopt het rooster en zoekt bij elke bond van iedere cel de roots van de twee verbonden cellen m.b.v. `findLocalRoot`. Vervolgens kan `mergeLocalCluster` met deze twee rootID's de twee betrokken clusters met elkaar verbinden. Laastgenoemde methode bepaalt de nieuwe root en werkt de twee dubbel gelinkte lijsten van beide clusters om naar één dubbel gelinkte lijst. Nadat alle lokale clusters gevormd zijn, wordt nog voor elke cel een directe link gelegd naar zijn root.

Tot slot maakt `makeGlobalClusterNumbers` de lokale clusternummers globaal m.b.v. vergelijking (2). Het globale nummer van de root met celnummer `i` wordt in `clusterID[i]` opgeslagen. Het lokale celnummer van de root van het `i`-de cluster wordt in `cluster_succ[i]` opgeslagen. De array `cluster_succ` wordt met een -1 afgesloten, zodat later eenvoudig alle clusters langsgelopen kunnen worden.

isingGlobalMonteCarloStep Als eerste worden de spins op de linker- en bovenrand naar de buren gecommuniceerd door `sendEdgeSpins`. Deze plaatst eerst alle spins op de linkerrand vanuit `spins` in `left_edge_spins`. Vervolgens kunnen deze array en het eerste gedeelte van `spins`-array (bovenrand) naar de buurprocessoren geput worden. In figuur 4 is te zien dat de middelste cel naar drie verschillende processoren verstuurd moet worden.



Figuur 4: Leggen van bonds over de randen

Iedere processor heeft nu de buur-spins langs zijn onder- en rechterrand van de buurprocessoren ontvangen en daarmee kan `makeEdgeBonds` bonds gaan leggen met de kans uit vergelijking (1). De te leggen bonds zijn in figuur 4 aangegeven met pijltjes. De laatste lege plekken in de al eerder besproken arrays `rbonds` en `bbonds` worden nu opgevuld. Op de onderrand worden bonds gelegd naar beneden en links. Omdat bij het lokaal bonds leggen op de bovenrand geen rbonds gelegd zijn, kunnen daar deze 'lbonds' van de onderrand in worden opgeslagen.

Behalve de aanwezigheid van een bond, moeten buurprocessoren ook van elkaar weten wat de roots van de aanliggende clusters zijn. Dit wordt gedaan door `sendClusterRoots` en is eenmalig nodig, omdat ondanks de globale clustering de lokale roots gelijk blijven, alleen de globale clusterID's zullen worden aangepast. Aangezien de uit te voeren simulaties vooral rond de kritische temperatuur plaatsvinden, waarbij er veel bonds worden gelegd, worden alle roots op de rand aan de rechter- en benedenbuur verstuurd zonder op bonds te controleren. Hierdoor kan ook een array in één put gedaan worden, hetgeen de communicatiekosten weer ten goede komt.

Nu alle benodigde gegevens op de randen bekend zijn, wordt een lus gestart die doorloopt tot er geen veranderingen meer zijn (d.m.v. `while(or(change))`).
In iedere slag worden de vernieuwde globale clusterID's door `sendClusterIds` over de randen naar beneden en rechts verstuurd. De ontvangende processoren moeten echter ook weten of er al dan niet bbonds en rbonds liggen op hun boven- en linkerrand. Ook moet bekend zijn *waar* op de rand de bonds liggen, omdat nu wel alleen data gestuurd worden wanneer er bonds zijn. De methode loopt nu eerst lokaal de `bbonds`- en `rbonds`-arrays op de rand langs, voor elke bond wordt in de arrays `bottom_edge_ids` en `right_edge_ids` het globale ID van het cluster langs de rand opgeslagen. Als lokatie van de bonds wordt de lokale cel-index gebruikt. Deze index is niet extreem groot, en dus worden twee van de voorste bits van iedere integer gebruikt om aan te geven of er al dan niet een rbonds, resp. bbond ligt. De lokaal opgeslagen data worden nu in een aantal puts verstuurd. Ook wordt aan de buren een enkele integer (`mr` en `nr`) verstuurd, die aangeeft hoeveel bonds er op de randen lagen, zodat deze de ontvangen array tot de juiste lengte kan doorlopen.

Nu heeft iedere processor *echt* alle benodigde data en kunnen de globale clusters bijgewerkt worden in de methode `updateGlobalClusters`. De ontvangen arrays met ID's en indices worden langsgelopen. Bij ieder element wordt uit de index-array de samengepakte informatie ontleed. Hiermee wordt het ontvangen clusterID van de buurcel vergeleken met het clusterID van de bijbehorende lokale cel. Op basis van deze nummers wordt of aan het lokale cluster een nieuw globaal clusternummer toegekend, of met een `put` het clusternummer bij de buurprocessor direct aangepast. Tot slot wordt een broadcast gedaan, of er een wijziging in de clustering is doorgevoerd.

Zodra de lus is beëindigd, kunnen de globale clusters met kans 0.5 geflipt worden. Dit wordt gedaan door de methode `swapGlobalCLusters`.
Nu komt de in het begin gemaakte array `cluster_succ` van pas. Iedere processor kan nu namelijk alle locale clusters eenvoudig langslopen. Wanneer een cluster nog steeds beheerd wordt door die processor, wordt een variabele met kans 0.5 op 0 of 1 gezet. Voor alle eigen clusters wordt dit opgeslagen in de array `pred`, die op dit moment toch niet meer in gebruik is.
In de volgende superstap loopt iedere processor nogmaals alle clusters langs. Nu wordt voor ieder cluster aan de 'beheerder'-processor met een `get` gevraagd of er voor of tegen flippen is gekozen.
In de laatste superstap loopt iedere processor de lokale clusters weer langs. Wanneer er voor flippen is gekozen, worden alle cellen van het betreffende cluster doorlopen en de spin daarvan geflipt.

Tot slot kan nog een aanmerking worden gemaakt bij het programma. Voor allerlei data (o.a. spins en bonds) hadden veel kleinere datatypen gebruikt kunnen worden, voor spins was zelfs 1 bit al genoeg geweest i.p.v. de huidige 32 of zelfs 64. Het gebruik van datatype `char` leverder echter problemen op bij de BSP-implementatie op de Cray T3E van de TU in Delft. Door het gebruik van deze grote datatypen is de lokale roostergrootte gelimiteerd tot 1700×1700.

4 Experimenten

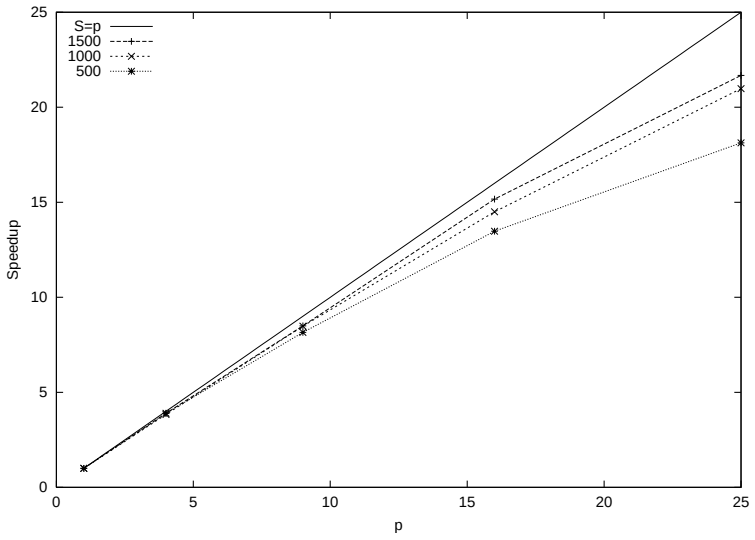
Aangezien de extra kosten die parallel programmeren met zich meebrengt voor communicatie alleen op de randen optreden, zijn de lokale (reken-) kosten waarschijnlijk dominant over de communicatiekosten. Om dit te onderzoeken zijn een aantal experimenten uitgevoerd, waarbij voor 50 Swendsen-Wang iteraties de totale tijd en de zogenaamde *relax-tijd* wordt gemeten. De relax-tijd is de tijd benodigd voor de `while`-lus, waarin de globale clusters gevormd worden.

Allereerst is naar de efficiëntie van het totale programma bij constante globale systeemgrootte gekeken, door de speedup voor verschillende waarden van p te bepalen. In figuur 5 is dit voor drie systeemgrootten gedaan. Vooral voor het grootste systeem blijkt het programma behoorlijk efficiënt te zijn: de speedup wijkt slechts weinig af van de ideale lijn $S = p$. De reden dat voor kleine systemen de speedup wat lager ligt, is dat er in verhouding meer gecommuniceerd moet worden over de randen. De verhouding (inwendige elementen):(randelementen) is namelijk kleiner. De efficiency (=speedup/p) van het programma blijkt iets te dalen voor stijgende p. Reden hiervoor is dat de relax-tijd groter wordt, door de factor $p^{d_{\min}/2}$ in vergelijking (3). Met andere woorden: het kost meer stappen om de globale clusters te vormen, omdat een cluster nu over meerdere processoren verdeeld kan zijn.

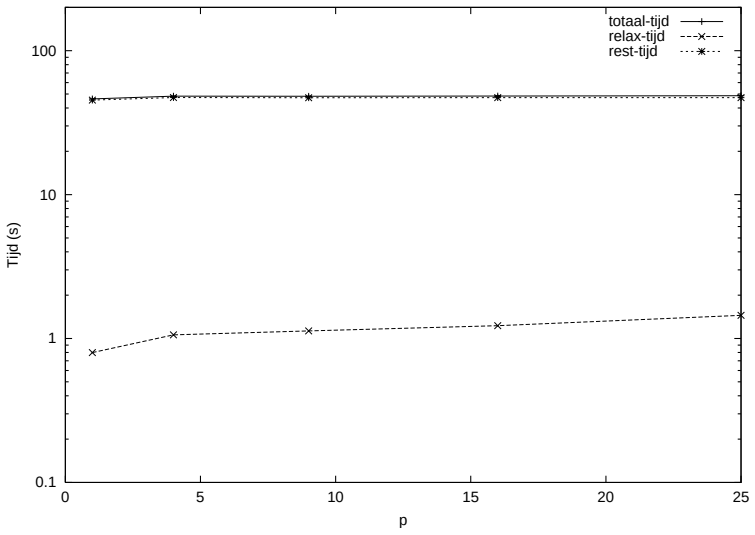
De invloed van het aantal processoren op de relax-tijd is nog verder onderzocht. Om de randbreedte geen invloed te geven wordt deze lokaal constant gehouden: $m = n = 1500$. Vervolgens worden voor variërende p de tijden gemeten. Deze staan in figuur 6. De relax-tijd lijkt niet veel toe te nemen, maar blijkt toch nog te stijgen van 0.799 s bij $p = 1$ naar 1.448 s bij $p = 25$. Toch is hier ook heel duidelijk te zien dat de relax-tijd slechts enkele procenten van de totale looptijd in beslag neemt.

5 Toepassing: maximale clustergrootte

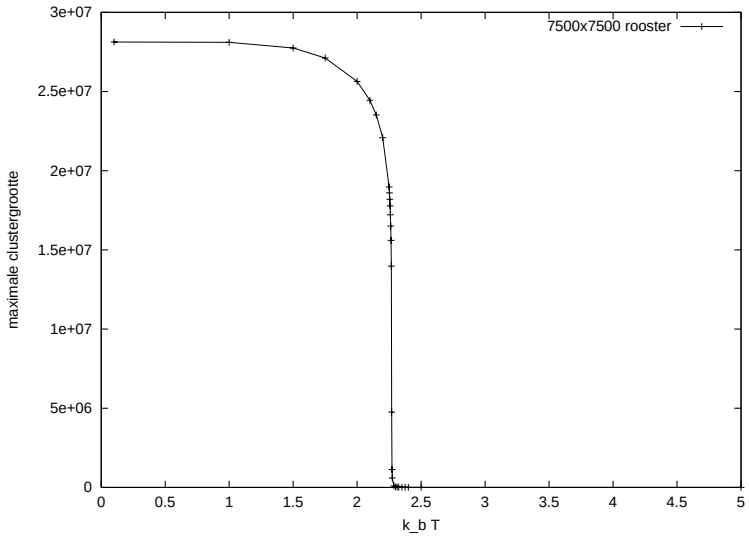
Om het programma niet alleen voor BSP-statistieken te gebruiken, is ook nog naar fysische eigenschappen van het gesimuleerde Ising-model gekeken. De maximale grootte van de globale clusters



Figuur 5: Speedup bij vaste systeemgrootten voor verschillende p.



Figuur 6: Opsplitsing van de looptijd bij vaste lokale systeemgrootte en variërende p.



Figuur 7: Maximale clustergrootte voor verschillende k_BT.

zegt iets over de huidige toestand van het systeem.

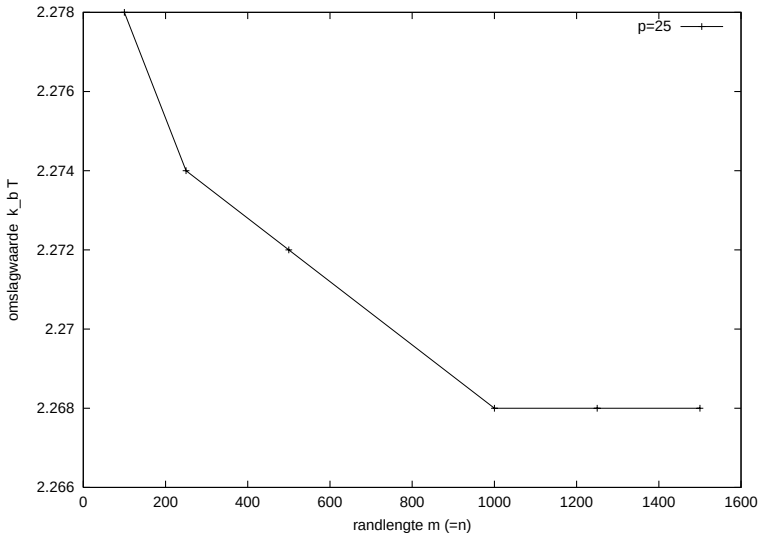
Wanneer de temperatuur T heel laag is, is de kans op clustering zeer groot; voor $T \to 0$ geldt dan $P_{\text{add}} \to 1$. Voor hoge temperatuur is dit juist andersom; $P_{\text{add}} \to 0$ en de clusters bestaan dan uit 1 cel. De vraag is, hoe de overgang tussen deze twee uitersten plaatsvindt. Aangezien reeds bekend is dat bij de kritische temperatuur een zeer scherpe val in clustergrootte optreedt, kan door lokaliseren van deze sprong, een schatting van de kritische temperatuur worden gevonden.

5.1 Implementatie

Voor het bepalen van de maximale clustergrootte van de globale clusters is een extra methode `determineGlobalClusterSizes` geschreven. Hierin loopt iedere processor zijn elementen langs en hoogt telkens een teller van het cluster van een element op. De lokale clustergrootten worden met een `bsp_send` naar de 'beheer-processor' gestuurd. Na een `bsp_sync()` kan iedere processor zijn messages-buffer uitlezen en zo de globale clustergrootte van zijn clusters bepalen. Elke processor houdt bij wat de grootte van zijn grootste cluster is en stuurt dit uiteindelijk naar processor 0. Deze bepaalt tenslotte wat globaal de maximale clustergrootte is.

5.2 Experimenten

Voor een systeem ter grootte 7500×7500 (28125000 roosterpunten), is voor verschillende waarden van k_BT de maximale clustergrootte bepaald. De resultaten staan in figuur 7. Duidelijk is de sprong te zien en de kritische waarde k_BT is af te schatten op 2.3. Om een grotere nauwkeurigheid te krijgen, wordt een systeemgrootte van 7500×7500 voor $p = 25$ gesimuleerd op een aantal waarden k_BT rond 2.3. Hiervan wordt de k_BT-waarde aan het begin van de sprong genomen. Dit wordt ook voor andere systeemgrootten gedaan. Deze waarden zijn in figuur 8 weergegeven (als functie van de lokale randbreedte m $(= n)$). Deze figuur heeft ook een beperkte mate van nauwkeurigheid, maar toch kan de schatting van k_BT hiermee al verfijnd worden naar een waarde 2.268.



Figuur 8: Convergentie van de omslagwaarde van k_BT naar de kritische waarde.

6 Conclusie

Tijdens dit project is, na vergelijking met een Metropolis-simulatie, nogmaals een bevestiging gevonden dat cluster-algoritmen sneller werken dan algoritmen als bijv. Metropolis bij het Ising model.

Wat het parallelle programma betreft kan worden opgemerkt dat het kiezen van een data verdeling in de praktijk anders kan uitpakken dan theoretisch verwacht. De gebruikte ruit-structuur zou een factor $\sqrt{2}$ winst bij de communicatie moeten opleveren. Deze wordt uiteindelijk wel gehaald, maar het vergt wel extra geheugenruimte voor administratie van index-gegevens. Verder blijkt het communicatiegedeelte qua kosten ondergeschikt aan het 'reken'-gedeelte. Reden hiervoor is de alleen op de randen gecommuniceerd wordt, terwijl het leggen van bonds en flippen van spins op het hele gebied moet gebeuren.

Verder bleken er problemen op te treden bij het versturen van `char`'s op een Cray T3E. Wanneer dit wel mogelijk was geweest, was een flinke geheugenbesparing in diverse arrays mogelijk geweest, waardoor nog grotere systemen gesimuleerd hadden kunnen worden. Eventueel kan dit ook nog verbeterd worden door in één datatype meerdere variabelen op te slaan door bepaalde bits voor bepaalde variabelen te gebruiken. Gedeeltelijk gebeurt dit al in het programma, maar dit kan eventueel nog verder uitgebreid worden. Hierbij ontstaan natuurlijk meer rekenkosten omdat de losse componenten uit één getal ontleed moeten worden.

Een belangrijke verbetering op het programma zou een echte parallelle random-number-generator zijn. De ontwikkeling hiervan is echter een project op zichzelf en dus is hier volstaan met een RNG met de processor-id als seed.

Tot slot de toepassing van het programma bij de bepaling van maximale clustergrootten. De verwachte val van clustergrootte bij de kritische temperatuur is duidelijk waargenomen en een redelijke schatting van $k_BT = 2.268$ kon hieruit afgeleid worden. In een meer fysisch georiënteerd onderzoek zou dit gedetailleerder en met meer aandacht voor de statistische fout onderzocht kunnen worden.

Referenties

[1] M.Flanigan en P.Tamayo, A parallel cluster labelling method for Monte Carlo dynamics, *Int. Jour. of Mod. Phys. C* 3, p.1235, 1992.

[2] R.H.Swendsen en J.S.Wang, Nonuniversal critical dynamics in Monte Carlo simulations, *Phys. Rev. Lett.* p.58:86, 1987.

A Listings

A.1 bspising.c

```
#include "bspising.h"
#include "bspising_mc_loc.h"
#include "bspising_mc_glob.h"

const int MAXSHIFT2 = (1<<(sizeof(int)*8-10)); /* used to separate swapsteps and max. clustersize */
int M, N;

void bspising(){
  int p, pid, m, n, mr, nr, l, k, kmax, npts, swapsteps, tagsz, glob_clust_max, swaptotal;
  int *rbonds, *bbonds, *spins, *left_edge_spins, *nb_right_edge_spins, *nb_bottom_edge_spins, *change,
      *pred, *succ, *root,
      *clusterID, *cluster_succ,
      *right_edge_ids, *bottom_edge_ids, *right_edge_ids_index, *bottom_edge_ids_index,
      *nb_left_edge_ids, *nb_top_edge_ids, *nb_left_edge_ids_index, *nb_top_edge_ids_index,
      *nb_left_edge_ids_root, *nb_top_edge_ids_root,
      *clusterSize;

  double time0, time1, time2, T, probSW, relaxtime, relaxtotal;
  double *meanLocalClusterSize;

  glob_clust_max = 0;
  tagsz = SZINT;
  relaxtotal = 0.0;
  swaptotal = 0;

  bsp_begin(M*N);
  pid= bsp_pid();            /* pid = processor number */
  p= bsp_nprocs();           /* p = number of processors */

  bsp_set_tag_size(&tagsz);
  bsp_push_reg(&M,SZINT);    /* M = number of proc.rows */
  bsp_push_reg(&N,SZINT);    /* N = number of proc.cols */
  bsp_push_reg(&m,SZINT);    /* m = local blockheight */
  bsp_push_reg(&n,SZINT);    /* n = local blockwidth */
  bsp_push_reg(&mr,SZINT);   /* mr = number of relevant elements at rightside */
  bsp_push_reg(&nr,SZINT);   /* nr = number of bonds at elements at bottomside */
  bsp_push_reg(&kmax,SZINT); /* kmax = max # of iterations */
  bsp_push_reg(&T,SZDBL);    /* T = system temperature (actucally k_B T) */
  bsp_sync();

  if (pid==0){
    printf("Please enter local blockheight m: ");
    scanf("%d", &m);
    printf("Please enter local blockwidth n: ");
    scanf("%d", &n);
    printf("Please enter max. nr. of iterations kmax: ");
    scanf("%d",&kmax);
    printf("Please enter system-temperature kb T: ");
    scanf("%lf",&T);

    if(m < 0 || n < 0 ){
      bsp_abort("Error in input: m and n must be positive");
    }

    for (k=0; k<p; k++){
        bsp_put(k,&M,&M,0,SZINT);
        bsp_put(k,&N,&N,0,SZINT);
        bsp_put(k,&m,&m,0,SZINT);
        bsp_put(k,&n,&n,0,SZINT);
        bsp_put(k,&kmax,&kmax,0,SZINT);
        bsp_put(k,&T,&T,0,SZDBL);
    }
    printf("Monte Carlo simulation of Ising model\n");
```

```
  printf("on a %d by %d (diamond-)grid\n",m,n);
  printf("using %d times %d processors\n",M,N);
}

bsp_sync();

npts = (m*n)>>1;                              /* number of points on one processor */
probSW = 1.0-exp(-2.0/T);                     /* probability of making a bond */

rbonds = vecalloci(npts);                     /* right-bond indicators */
bbonds = vecalloci(npts);                     /* bottom-bond indicators */
spins  = vecalloci(npts);                     /* up-spin (1) or down-spin (0) of a point */
pred   = vecalloci(npts);                     /* predecessors in list */
succ   = vecalloci(npts);                     /* successors in list */
root   = vecalloci(npts);                     /* root of a point's cluster */
clusterID = vecalloci(npts);                  /* global clusterID */
cluster_succ = vecalloci(npts+1);             /* index of next global cluster */
right_edge_ids = vecalloci(m>>1);             /* clusterID's at right edge */
bottom_edge_ids= vecalloci(n>>1);             /* clusterID's at bottom edge */
nb_left_edge_ids = vecalloci((m>>1)+1);       /* clusterID's at left-edge by neighbour */
nb_top_edge_ids = vecalloci(n>>1);            /* clusterID's at top-edge by neighbour */
right_edge_ids_index = vecalloci(m>>1);       /* index of point's at right-edge with a right-bond */
bottom_edge_ids_index = vecalloci(n>>1);      /* index of point's at bottom-edge with a bottom-bond */
nb_left_edge_ids_index = vecalloci((m>>1)+1); /* same as above for the neighbour */
nb_top_edge_ids_index = vecalloci(n>>1);      /* idem */
nb_left_edge_ids_root = vecalloci((m>>1)+1);  /* local rootid of global id  at left edge */
nb_top_edge_ids_root = vecalloci(n>>1);       /* idem */
left_edge_spins = vecalloci(m>>1);            /* my spins at left-edge */
nb_right_edge_spins = vecalloci((m>>1)+1);    /* neighbour spins at my right-edge */
nb_bottom_edge_spins = vecalloci((n>>1)+1);   /* neighbour spins at my bottom-edge */
change = vecalloci(p);                        /* indicates for each processor, whether a change in
                                                 global clustering was made */
clusterSize = vecalloci(npts);                /* the clustersizes of the local roots */
meanLocalClusterSize = vecallocd(p);          /* stores for each processor the mean clustersize */

bsp_push_reg(pred,npts*SZINT);
bsp_push_reg(nb_left_edge_ids,((m>>1)+1)*SZINT);
bsp_push_reg(nb_top_edge_ids,(n>>1)*SZINT);
bsp_push_reg(nb_left_edge_ids_index,((m>>1)+1)*SZINT);
bsp_push_reg(nb_top_edge_ids_index,(n>>1)*SZINT);
bsp_push_reg(nb_left_edge_ids_root,((m>>1)+1)*SZINT);
bsp_push_reg(nb_top_edge_ids_root,(n>>1)*SZINT);
bsp_push_reg(nb_right_edge_spins,((m>>1)+1)*SZINT);
bsp_push_reg(nb_bottom_edge_spins,((n>>1)+1)*SZINT);
bsp_push_reg(clusterID,npts*SZINT);
bsp_push_reg(clusterSize,npts*SZINT);
bsp_push_reg(change, p*SZINT);
bsp_push_reg(meanLocalClusterSize, p*SZDBL);

srandom(pid); /* processor-id is seed of RNG */

for(k = 0; k < npts; k++){
  spins[k] = randInt(2); /* assign a random spin (0 or 1) to each point */
}

bsp_sync();
if(pid == 0)
  printf("Starting Monte Carlo loop...\n");

time0=bsp_time();
time1 = time0;
  for(k = 1; k <= kmax; k++){
  change[0] = 1; /* dummy value; first iteration of while(change) should always succeed... */

  isingLocalMonteCarloStep(pid, M, N, m, n, probSW, rbonds, bbonds, pred, succ, root,
                           clusterID, cluster_succ, spins);
  swapsteps = isingGlobalMonteCarloStep(pid, M, N, m, n, &mr, &nr, probSW, rbonds, bbonds,
```

```
                                    pred, succ, root, clusterID, cluster_succ, right_edge_ids,
                                    bottom_edge_ids, right_edge_ids_index, bottom_edge_ids_index,
                                    nb_left_edge_ids, nb_top_edge_ids, nb_left_edge_ids_index,
                                    nb_top_edge_ids_index, nb_left_edge_ids_root,
                                    nb_top_edge_ids_root, left_edge_spins, nb_right_edge_spins,
                                    nb_bottom_edge_spins, change, meanLocalClusterSize,
                                    clusterSize, spins, &relaxtime
                                   );
  relaxtotal += relaxtime;
  if(swapsteps%MAXSHIFT2 > glob_clust_max) /* adjust global maximum of clustersize */
    glob_clust_max = swapsteps%MAXSHIFT2;
  swapsteps /= MAXSHIFT2;                  /* information on clustersize is no longer needed */
  swaptotal += swapsteps;                  /* swapsteps now only contains nr.of relax-iterationsteps */

  time2=bsp_time();
  if(pid==0) /* prints data to inform the impatient user */
    printf("Monte Carlo step #%d took %.6lf seconds, average %.6lf seconds, %d steps needed\n",
           k, (time2-time1), (time2-time0)/k, swapsteps);
  time1 = time2;
}

if(pid==0) /* prints data used during experiments */
  printf("%d %9.6f %9.6f %9.6f %d %d\n",
         M*N, time2-time0, timetotal, time2-time0-relaxtotal, swaptotal, m*n);
printBonds( pid, m, n, spins, rbonds, bbonds);              /* prints gnuplotdata into file */
printLocalClusters(pid, m, n, pred, succ, root, clusterID); /* idem */
if(pid==0) /* prints data used during clustersize-experiments */
  printf("%9.6f %d\n", T, glob_clust_max);

bsp_pop_reg(&M);
bsp_pop_reg(&N);
bsp_pop_reg(&m);
bsp_pop_reg(&n);
bsp_pop_reg(&mr);
bsp_pop_reg(&nr);
bsp_pop_reg(&kmax);
bsp_pop_reg(&T);
bsp_pop_reg(nb_left_edge_ids);
bsp_pop_reg(nb_top_edge_ids);
bsp_pop_reg(nb_left_edge_ids_index);
bsp_pop_reg(nb_top_edge_ids_index);
bsp_pop_reg(nb_left_edge_ids_root);
bsp_pop_reg(nb_top_edge_ids_root);
bsp_pop_reg(nb_right_edge_spins);
bsp_pop_reg(nb_bottom_edge_spins);
bsp_pop_reg(change);

free(rbonds);
free(bbonds);
free(spins);
free(pred);
free(succ);
free(root);
free(clusterID);
free(cluster_succ);
free(right_edge_ids);
free(bottom_edge_ids);
free(nb_left_edge_ids);
free(nb_top_edge_ids);
free(right_edge_ids_index);
free(bottom_edge_ids_index);
free(nb_left_edge_ids_index);
free(nb_top_edge_ids_index);
free(nb_left_edge_ids_root);
free(nb_top_edge_ids_root);
free(left_edge_spins);
free(nb_right_edge_spins);
```

```
  free(nb_bottom_edge_spins);
  free(change);
  free(clusterSize);
  free(meanLocalClusterSize);

  bsp_end();
} /* END bspising */

int main(int argc, char **argv){

  bsp_init(bspising, argc, argv);
  printf("Please enter number of processor rows M: ");
  scanf("%d",&M);
  printf("Please enter number of processor columns N: ");
  scanf("%d",&N);
  if (M*N > bsp_nprocs()){
    printf("Not enough processors available: %d wanted, %d available\n",
           M*N, bsp_nprocs());
    exit(1);
  }
  bspising();
  exit(0);
}
```

A.2 bspising_mc_loc.c

```
#include "bspising_mc_loc.h"

void isingLocalMonteCarloStep(int pid, int M, int N, int m, int n, double probSW,
                              int *rbonds, int *bbonds, int *pred, int *succ, int *root,
                              int *clusterID, int *cluster_succ, int *spins
                             ){
  makeLocalBonds(m, n, probSW, spins, rbonds, bbonds);
  makeLocalClusters( m, n, rbonds, bbonds, pred, succ, root);
  makeGlobalClusterNumbers(pid, m, n, root, clusterID, cluster_succ);
}

/**
 * makeLocalBonds
 *  creates bonds between points with equal spin
 *  with a certain probability
 *  The walkthough of the grid is not straightforward
 *  because of edge-exceptions
 **/
void makeLocalBonds( int m, int n, double probSW, int *spins, int *rbonds, int *bbonds){
  int i, j, jh, mh, nh, limit, my_index;

  mh = m>>1; /* mh is half m (the real # elements in one column)*/
  nh = n>>1; /* nh is half n (the real # elements in one row)*/

  for( j = 0; j < nh; j++){          /* Loop columns of upper edge */
    if( spins[j] == spins[nh + j] ){ /* lower-right neighbour */
      if( drandom() < probSW )
        bbonds[j] = 1;
      else
        bbonds[j] = 0;
    }
    else
      bbonds[j] = 0;
  }

  limit = nh-1;
  for( j = 0; j < limit; j++){        /* Loop columns of lower edge */
    my_index = (m-1)*nh + j;
    if( spins[my_index] == spins[my_index - nh + 1] ){ /* upper-right neighbour */
      if( drandom() < probSW )
        rbonds[my_index] = 1;
```

```
      else
        rbonds[my_index] = 0;
    }
    else
      rbonds[my_index] = 0;
  }

  for( i = 1; i < mh; i++){        /* Loop over rows (split by even and odd) */
    for( j = 0; j < limit; j++){   /* Loop over columns of odd rows */
      my_index = ((i<<1)-1)*nh + j;
      if( spins[my_index] == spins[my_index - nh + 1] ){ /* upper-right neighbour */
        if( drandom() < probSW )
          rbonds[my_index] = 1;
        else
          rbonds[my_index] = 0;
      }
      else
        rbonds[my_index] = 0;

      if( spins[my_index] == spins[my_index + nh + 1] ){ /* lower-right neighbour */
        if( drandom() < probSW )
          bbonds[my_index] = 1;
        else
          bbonds[my_index] = 0;
      }
      else
        bbonds[my_index] = 0;
    }

    for( j = 0; j < nh; j++){      /* Loop over columns of even rows */
      my_index = (i<<1)*nh + j;
      if( spins[my_index] == spins[my_index - nh] ){ /* upper-right neighbour */
        if( drandom() < probSW )
          rbonds[my_index] = 1;
        else
          rbonds[my_index] = 0;
      }
      else
        rbonds[my_index] = 0;

      if( spins[my_index] == spins[my_index + nh] ){ /* lower-right neighbour */
        if( drandom() < probSW )
          bbonds[my_index] = 1;
        else
          bbonds[my_index] = 0;
      }
      else
        bbonds[my_index] = 0;
    }
  }
}

/**
 * makeLocalClusters
 *  identifies all the local clusters, by reading the rbonds and bbonds
 *  information is stored by making doubly-linked lists
 *  each cluster is identified by the array-index of its root
 **/
void makeLocalClusters( int m, int n, int *rbonds, int *bbonds, int *pred, int *succ, int *root){
  int i, j, npts, jh, mh, nh, limit, my_index, my_root, nb_root;
  npts  = (m*n)>>1; /* m rows and n/2 cols per row */
  mh = m>>1;        /* half m */
  nh = n>>1;        /* half n */

  /* (init: each gridcell is in its own cluster */
  for(i = 0; i < npts; i++){
    pred[i] = i;
```

```
    succ[i] = i;
    root[i] = i;
  }

  for( j = 0; j < nh; j++){          /* Loop over columns of upper edge */
    if(bbonds[j]){
      my_root = findLocalRoot(root, j);
      nb_root = findLocalRoot(root, j + nh); /* lower-right neighbour */

      mergeLocalCluster(my_root, nb_root, pred, succ, root);
    }
  }

  limit = nh-1;
  for( j = 0; j < limit; j++){        /* Loop over columns of lower edge */
    my_index = (m-1)*nh + j;
    if(rbonds[my_index]){
      my_root = findLocalRoot(root, my_index);
      nb_root = findLocalRoot(root, my_index - nh + 1); /* upper-right neighbour */

      mergeLocalCluster(my_root, nb_root, pred, succ, root);
    }
  }

  for( i = 1; i < mh; i++){        /* Loop over rows (split by even and odd) */
    for( j = 0; j < limit; j++){      /* Loop over columns of odd rows */
      my_index = ((i<<1)-1)*nh + j;
      if(rbonds[my_index]){
        my_root = findLocalRoot(root, my_index);
        nb_root = findLocalRoot(root, my_index - nh + 1); /* upper-right neighbour */

        mergeLocalCluster(my_root, nb_root, pred, succ, root);
      }

      if(bbonds[my_index]){
        my_root = findLocalRoot(root, my_index);
        nb_root = findLocalRoot(root, my_index + nh + 1); /* lower-right neighbour */

        mergeLocalCluster(my_root, nb_root, pred, succ, root);
      }
    }

    for( j = 0; j < nh; j++){      /* Loop over columns of even rows */
      my_index = (i<<1)*nh + j;
      if(rbonds[my_index]){
        my_root = findLocalRoot(root, my_index);
        nb_root = findLocalRoot(root, my_index - nh); /* upper-right neighbour */

        mergeLocalCluster(my_root, nb_root, pred, succ, root);
      }

      if(bbonds[my_index]){
        my_root = findLocalRoot(root, my_index);
        nb_root = findLocalRoot(root, my_index + nh); /* lower-right neighbour */

        mergeLocalCluster(my_root, nb_root, pred, succ, root);
      }
    }
  }

  /* Now, detect the direct root for each point */
  for(i = 0; i < npts; i++){
    findLocalRoot(root, i);
  }
}

/**
```

```
 * findLocalRoot
 *  takes an index of a point
 *  returns the index of the root that identifies
 *  the cluster in which the point lies
 **/
int findLocalRoot(int *root, int j){
  int i = j;
  while(root[i] != i){
    i = root[i];
  }
  root[j] = i;
  return i;
}

/**
 * mergeLocalCluster
 *  takes 2 roots identifying 2 clusters and the list-arrays
 *  merges these 2 clusters by making the right links in the lists
 **/
void mergeLocalCluster(int root1, int root2, int *pred, int *succ, int *root){
  int i;
  if( root2 < root1){ /* root1 will be the smaller one */
    i = root1;
    root1 = root2;
    root2 = i;
  }

  root[root2]= root1;
  succ[i = pred[root1]] = root2;
  succ[pred[root2]] = root1;
  pred[root1] = pred[root2];
  pred[root2] = i;
}

/**
 * makeGlobalClusterNumbers
 *  computes the global clusternr, so each cluster is uniquely identified
 *  stores it in a list-like structure
 **/
void makeGlobalClusterNumbers(int pid, int m, int n, int *root, int *clusterID, int *cluster_succ){
  int i, nrclusters, npts;
  nrclusters = 0;
  npts = m*n>>1;
  for(i = 0; i < npts; i++){
    if(root[i] == i){ /* if a cell is its own root, then it's a root of a cluster */
      cluster_succ[nrclusters++] = i;
      clusterID[i] = pid*npts + i;
    }
  }
  cluster_succ[nrclusters] = -1; /* Mark end of sequence of local clusterroots */
}
```

A.3 bspising_mc_glob.c

```
#include "bspising.h"
#include "bspising_mc_glob.h"
const int BONDSHIFT = (1<<(sizeof(int)*8-3));
const int MAXSHIFT = (1<<(sizeof(int)*8-10));
const int CLUSTERSHIFT = (1<<(sizeof(int)*8 - 25)); // Cray-specific

/**
 * isingGlobalMonteCarloStep
 *  communicate spins at left- and top-edges to neighbours
 *  create local-bond at edges (right and bottom)
 *  communicate with neighbour processors
 *  update global clusters until no change is detected
```

```
 *  determine maximum global clustersize
 *  swap global clusters with prob. 0.5
 **/
int isingGlobalMonteCarloStep(int pid, int M, int N, int m, int n, int *mr, int *nr, double probSW,
                              int *rbonds, int *bbonds, int *pred, int *succ, int *root, int *clusterID,
                              int *cluster_succ, int *right_edge_ids, int *bottom_edge_ids,
                              int *right_edge_ids_index, int *bottom_edge_ids_index,
                              int *nb_left_edge_ids, int *nb_top_edge_ids, int *nb_left_edge_ids_index,
                              int *nb_top_edge_ids_index, int *nb_left_edge_ids_root,
                              int *nb_top_edge_ids_root, int *left_edge_spins, int *nb_right_edge_spins,
                              int *nb_bottom_edge_spins, int *change, double *meanLocalClusterSize,
                              int *clusterSize, int *spins, double *relaxtime
                             ){
  int i, s, t;
  double time0, time1;

  s = pid/N;
  t = pid%N;
  i=0;

  sendEdgeSpins(s, t, M, N, m, n, left_edge_spins, nb_right_edge_spins, nb_bottom_edge_spins, spins);
  bsp_sync();

  makeEdgeBonds(m, n, probSW, spins, nb_right_edge_spins, nb_bottom_edge_spins, rbonds, bbonds);
  sendClusterRoots(s, t, M, N, m, n, mr, nr, rbonds, bbonds, root, clusterID, right_edge_ids,
                   bottom_edge_ids, right_edge_ids_index, bottom_edge_ids_index, nb_left_edge_ids,
                   nb_top_edge_ids, nb_left_edge_ids_index, nb_top_edge_ids_index,
                   nb_left_edge_ids_root, nb_top_edge_ids_root
                  );

  time0=bsp_time();
  while(or(M*N, change)){ /* while changes have been made to global clustering */
    sendClusterIds(s, t, M, N, m, n, mr, nr, rbonds, bbonds, root, clusterID, right_edge_ids,
                   bottom_edge_ids, right_edge_ids_index, bottom_edge_ids_index, nb_left_edge_ids,
                   nb_top_edge_ids, nb_left_edge_ids_index, nb_top_edge_ids_index
                  );
    bsp_sync();
    updateGlobalClusters(s, t, M, N, m, n, *mr, *nr, root, clusterID, nb_left_edge_ids,
                         nb_top_edge_ids, nb_left_edge_ids_index, nb_top_edge_ids_index,
                         nb_left_edge_ids_root, nb_top_edge_ids_root, change
                        );
    bsp_sync();
    i++;
  }
  time1=bsp_time();
  *relaxtime = time1-time0;
  i *= MAXSHIFT;
  i += determineGlobalClusterSizes(s, t, m, n, M, N, root, clusterID, cluster_succ, change,
                                   meanLocalClusterSize, clusterSize
                                  );

  swapGlobalClusters(pid, M, N, m, n, root, clusterID, pred, succ, cluster_succ, spins);
  return i;
}

/**
 * or
 *  takes array of integers
 *  returns 0 if all are 0, else returns 1 (=true)
 **/
int or(int n, int *bools){
  int i;
  for(i = 0; i < n; i++){
    if(bools[i]){
      return 1;
    }
  }
```

```
  return 0;
}

/**
 * determineGlobalClusterSize
 *  sums up the local clustersizes, sends it to the owner-processes
 *  these owners sum up the received local cluster sizes, to obtain global cluster sizes.
 */
int determineGlobalClusterSizes(int s, int t, int m, int n, int M, int N, int *root, int *clusterID,
                                int *cluster_succ, int *change, double *meanLocalClusterSize,
                                int *localSize
                               ){
  int i, c, npts, nmessages, nbytes, temp, status, globID, max, nclusters, p;
  double total;

  p = M * N;
  max = 0;
  nclusters = 0;
  total = 0.0;
  npts = (m*n)>>1;
  memset(localSize, 0, npts*SZINT); /* reset all the local sizes to 0 */
  for(i = 0; i < npts; i++){
    localSize[root[i]]++;           /* count local elements for local clustersizes */
  }

  temp = s*N+t;
  i = 0;
  while( (c=cluster_succ[i])!=-1){
    i++;
    if(clusterID[c]/npts == temp){/* if the global root is located at this processor */
      nclusters++;
      if(clusterID[c]%npts != c){ /* if cell c is not the global root */
        localSize[clusterID[c]%npts] += localSize[c]; /* add the amount to total clustersize */
      }
      total += localSize[c];
      if(localSize[clusterID[c]%npts] > max){
        max = localSize[clusterID[c]%npts];
      }
    }
    else{ /* else send the results to the processor, where the global root is located */
      bsp_send(clusterID[c]/npts, &clusterID[c], &localSize[c], SZINT);
    }
  }

  bsp_sync();

  bsp_qsize(&nmessages, &nbytes);

  for(i=0; i<nmessages; i++){ /* process all the received messages, sum up the received local
                                 clustersizes to obtain your global clustersize */
    bsp_get_tag(&status, &globID);
    bsp_move(&temp, SZINT);
    localSize[globID%npts] += temp;
    total += temp;
    if(localSize[globID%npts] > max){
      max = localSize[globID%npts];
    }
  }

  total += nclusters*CLUSTERSHIFT; // also store the nr of clusters, for communication

  /* for final global maximization, send all data to processor 0 */
  bsp_put(0, &total ,meanLocalClusterSize, (s*N+t)*SZDBL, SZDBL);
  bsp_put(0, &max, change, (s*N+t)*SZINT, SZINT); /* reuse integer-array change of length p */

  bsp_sync();
```

```
  max = 0;
  total = 0.0;
  nclusters = 0;

  if(s==0 && t==0){
    for( i=0; i<p; i++){
      total = total + ((int)(meanLocalClusterSize[i])%CLUSTERSHIFT);
      nclusters += (meanLocalClusterSize[i]/CLUSTERSHIFT);
      if(change[i] > max){
        max = change[i];
      }
    }
    total /= nclusters; /* total now contains the mean (no longer used, max. is much more interesting) */

  }
  return max;
}

/**
 * sendEdgeSpins
 *  sends the local spins at left and top edge to left and upper neighbour
 *  left edge-spins are written in local array first to put this edge at once
 **/
void sendEdgeSpins(int s, int t, int M, int N, int m, int n, int *left_edge_spins,
                   int *nb_right_edge_spins, int *nb_bottom_edge_spins, int *spins
                  ){
  int i, mh;

  mh = m>>1;
  for(i = 0; i < mh; i++){  /* store spins at left edge in one array for bulk communication */
    left_edge_spins[i] = spins[i*n];
  }
  bsp_put(s*N + (t-1+N)%N, left_edge_spins, nb_right_edge_spins, 0, mh*SZINT);
  bsp_put(((s-1+M)%M)*N + t, &spins[0], nb_bottom_edge_spins, 0, (n>>1)*SZINT);
  bsp_put(((s-1+M)%M)*N + (t-1+N)%N, &spins[0], nb_right_edge_spins, mh*SZINT, SZINT);
  bsp_put(((s-1+M)%M)*N + (t-1+N)%N, &spins[0], nb_bottom_edge_spins, (n>>1)*SZINT, SZINT);
}

/**
 * makeEdgeBonds
 *  compares own spins with neighbour spins and creates a bond
 *  with certain probability if spins are equal
 *  this is done at right and bottom edge
 **/
void makeEdgeBonds(int m, int n, double probSW, int *spins, int *nb_right_edge_spins,
                   int *nb_bottom_edge_spins, int *rbonds, int *bbonds
                  ){
  int i, mh, nh, tmp;

  mh = m>>1;
  nh = n>>1;

  for(i = 1; i <= mh; i++){
    if(spins[i*n-1] == nb_right_edge_spins[i-1]){ /* rbond at right-edge */
      if(drandom() < probSW){
        rbonds[i*n-1] = 1;
      }
      else
        rbonds[i*n-1] = 0;
    }
    else
      rbonds[i*n-1] = 0;

    if(spins[i*n-1] == nb_right_edge_spins[i]){ /* bbond at right-edge*/
      if(drandom() < probSW){
        bbonds[i*n-1] = 1;
      }
```

```
      else
        bbonds[i*n-1] = 0;
    }
    else
      bbonds[i*n-1] = 0;
  }

  tmp = (m-1)*nh;
  for(i = 0; i < nh; i++){
    if(spins[tmp+i] == nb_bottom_edge_spins[i]){ /* lbonds at bottom edge */
      if(drandom() < probSW){
        rbonds[i] = 1; /* leftbonds at bottom edge are stored
                          in memory of rightbonds at top edge */
      }
      else
        rbonds[i] = 0;
    }
    else
      rbonds[i] = 0;

    if(spins[tmp+i] == nb_bottom_edge_spins[i+1]){ /* bbonds at bottom edge */
      if(drandom() < probSW){
        bbonds[tmp+i] = 1;
      }
      else
        bbonds[tmp+i] = 0;
    }
    else
      bbonds[tmp+i] = 0;
  }
}

/**
 * sendClusterRoots
 *  communicate all local clusternumbers at edge to neighbours
 **/
void sendClusterRoots(int s, int t, int M, int N, int m, int n, int *mr, int *nr, int *rbonds,
                      int *bbonds, int *root, int *clusterID, int *right_edge_ids,
                      int *bottom_edge_ids, int *right_edge_ids_index, int *bottom_edge_ids_index,
                      int *nb_left_edge_ids, int *nb_top_edge_ids, int *nb_left_edge_ids_index,
                      int *nb_top_edge_ids_index, int *nb_left_edge_ids_root,
                      int *nb_top_edge_ids_root
                     ){
  int i, jr, jb, mh, nh, tmp;

  mh = m>>1;
  nh = n>>1;
  jr = 0;
  jb = 0;

  /* store roots at right and bottom edge in two arrays for bulk communication */
  for(i = 1; i <= mh; i++){
    right_edge_ids[i-1] = root[i*n-1];
  }

  tmp = (m-1)*nh;
  for(i = 0; i < nh; i++){
    bottom_edge_ids[i] = root[i+tmp];
  }


  bsp_put(s*N + (t+1)%N, right_edge_ids, nb_left_edge_ids_root, 0, SZINT*mh);
  bsp_put(((s+1)%M)*N + t, bottom_edge_ids, nb_top_edge_ids_root, 0, SZINT*nh);

  if(bbonds[m*nh-1]){
    bsp_put(((s+1)%M)*N + (t+1)%N, &root[m*nh-1], nb_left_edge_ids_root, mh*SZINT, SZINT);
  }
```

```
}

/**
 * sendClusterIds
 *  check for bonds to neighbours
 *  communicate global clusternumbers to neighbours
 **/
void sendClusterIds(int s, int t, int M, int N, int m, int n, int *mr, int *nr, int *rbonds,
                    int *bbonds, int *root, int *clusterID, int *right_edge_ids, int *bottom_edge_ids,
                    int *right_edge_ids_index, int *bottom_edge_ids_index, int *nb_left_edge_ids,
                    int *nb_top_edge_ids, int *nb_left_edge_ids_index, int *nb_top_edge_ids_index
                   ){
  int i, jr, jb, mh, nh, tmp;

  mh = m>>1;
  nh = n>>1;
  jr = 0; /* counter for # rbonds */
  jb = 0; /* counter for # bbonds */

  for(i = 1; i < mh; i++){ /* do not take the last one, special check follows */
    if(rbonds[i*n-1] || bbonds[i*n-1]){   /* bond at right-edge */
      right_edge_ids[jr] = clusterID[root[i*n-1]];
      /* in the 1st and 2nd bit of .._index is stored whether this point has a rbond resp. bbond */
      right_edge_ids_index[jr] = i*n-1 + ((rbonds[i*n-1]<<1)+bbonds[i*n-1] )* BONDSHIFT;
      jr++;
    }
  }
  i = mh;
  if(rbonds[i*n-1]){ /* only take rbond at last element of right-edge */
    right_edge_ids[jr] = clusterID[root[i*n-1]];
    right_edge_ids_index[jr] = i*n-1 + ((rbonds[i*n-1]<<1)+bbonds[i*n-1] )* BONDSHIFT;
    jr++;
  }

  tmp = (m-1)*nh;
  for(i = 0; i < nh-1; i++){
    if(rbonds[i] || bbonds[i+tmp]){ /* lbonds at bottom edge ; leftbonds at bottom edge are stored
                                       in memory of rightbonds at top edge */
      bottom_edge_ids[jb] = clusterID[root[i+tmp]];
      bottom_edge_ids_index[jb] = i + tmp + ((rbonds[i]<<1)+bbonds[i+tmp] )* BONDSHIFT;
      jb++;
    }
  }

  i = nh-1;
  if(rbonds[i]){ /* only take lbond at last element of bottom-edge */
    bottom_edge_ids[jb] = clusterID[root[i+tmp]];
    bottom_edge_ids_index[jb] = i + tmp + ((rbonds[i]<<1)+bbonds[i+tmp] )* BONDSHIFT;
    jb++;
  }

  bsp_put(s*N + (t+1)%N, right_edge_ids, nb_left_edge_ids, 0, SZINT*jr);
  bsp_put(((s+1)%M)*N + t, bottom_edge_ids, nb_top_edge_ids, 0, SZINT*jb);
  bsp_put(s*N + (t+1)%N, right_edge_ids_index, nb_left_edge_ids_index, 0, SZINT*jr);
  bsp_put(((s+1)%M)*N + t, bottom_edge_ids_index, nb_top_edge_ids_index, 0, SZINT*jb);
  bsp_put(s*N + (t+1)%N, &jr, mr, 0, SZINT);
  bsp_put(((s+1)%M)*N + t, &jb, nr, 0, SZINT);

  if(bbonds[m*nh-1]){ /* send bbond at lower corner of diamond; index is known, no need to send */
    bsp_put(((s+1)%M)*N + (t+1)%N, &clusterID[root[m*nh-1]], nb_left_edge_ids, mh*SZINT, SZINT);
  }
  else{ /* no bbond; let neighbour know by value -1; */
    jr=-1;
    bsp_put(((s+1)%M)*N + (t+1)%N, &jr, nb_left_edge_ids, mh*SZINT, SZINT);
  }

}
```

```
/**
 * updateGlobalClusters
 *  loop over clusterID's of neighbours with bond
 *  compare them wiht own clusterID's
 *  'merge' clusters by updating own- or neighbourclusterID's
 *  choose smallest ID as ID for merged-cluster
 **/
void updateGlobalClusters(int s, int t, int M, int N, int m, int n, int mr, int nr, int *root,
                          int *clusterID, int *nb_left_edge_ids, int *nb_top_edge_ids,
                          int *nb_left_edge_ids_index, int *nb_top_edge_ids_index,
                          int *nb_left_edge_ids_root, int *nb_top_edge_ids_root, int *change
                         ){

  int i, index, mh, nh, npts, rb, bb, ch;

  mh = m>>1;
  nh = n>>1;
  npts = m*nh;
  ch = 0;

  for( i=0; i < mr; i++){  /* bonds at rightedge */
    /* now split up the received index in three parts: rbond, bbond and real index. */
    rb = nb_left_edge_ids_index[i]/BONDSHIFT;
    bb = rb%2;
    rb>>=1;
    index = nb_left_edge_ids_index[i]%BONDSHIFT;

    if(rb){
      if( nb_left_edge_ids[i] < clusterID[root[index-n+1]]){
        clusterID[root[index-n+1]] = nb_left_edge_ids[i];
        ch = 1;
      }
      else if( nb_left_edge_ids[i] > clusterID[root[index-n+1]]){
        bsp_put(s*N + (t-1+N)%N, &clusterID[root[index-n+1]], clusterID,
                nb_left_edge_ids_root[index/n]*SZINT, SZINT);
        ch = 1;
      }
    }

    if(bb && index < npts-1){
      if( nb_left_edge_ids[i] < clusterID[root[index+1]]){
        clusterID[root[index+1]] = nb_left_edge_ids[i];
        ch = 1;
      }
      else if( nb_left_edge_ids[i] > clusterID[root[index+1]]){
        bsp_put(s*N + (t-1+N)%N, &clusterID[root[index+1]], clusterID,
                nb_left_edge_ids_root[index/n]*SZINT, SZINT);
        ch = 1;
      }
    }
  }
  for(i = 0; i < nr; i++){  /* bonds at bottomedge */
    rb = nb_top_edge_ids_index[i]/BONDSHIFT;
    bb = rb%2;
    rb>>=1;
    index = nb_top_edge_ids_index[i]%BONDSHIFT;

    if(rb){ /* leftbonds */
      if( nb_top_edge_ids[i] < clusterID[root[index-(m-1)*nh]]){
        clusterID[root[index-(m-1)*nh]] = nb_top_edge_ids[i];
        ch = 1;
      }
      else if( nb_top_edge_ids[i] > clusterID[root[index-(m-1)*nh]]){
        bsp_put(((s-1+M)%M)*N+t, &clusterID[root[index-(m-1)*nh]], clusterID,
                nb_top_edge_ids_root[index%nh]*SZINT, SZINT);
        ch = 1;
```

```
      }
    }

    if(bb && index < npts-1){ /* bottombonds */
      if( nb_top_edge_ids[i] < clusterID[root[index-(m-1)*nh+1]]){
        clusterID[root[index-(m-1)*nh+1]] = nb_top_edge_ids[i];
        ch = 1;
      }
      else if( nb_top_edge_ids[i] > clusterID[root[index-(m-1)*nh+1]]){
        bsp_put(((s-1+M)%M)*N+t, &clusterID[root[index-(m-1)*nh+1]], clusterID,
                nb_top_edge_ids_root[index%nh]*SZINT, SZINT);
        ch = 1;
      }
    }
  }
  if(nb_left_edge_ids[mh] != -1){ /* check for bbond at top corner */
    if(nb_left_edge_ids[mh] < clusterID[root[0]]){
      clusterID[root[0]] = nb_left_edge_ids[mh];
      ch = 1;
    }
    else if(nb_left_edge_ids[mh] > clusterID[root[0]]){
      bsp_put(((s-1+M)%M)*N + (t-1+N)%N, &clusterID[root[0]], clusterID,
              nb_left_edge_ids_root[mh]*SZINT, SZINT);
      ch = 1;
    }
  }

  change[s*N+t] = ch;
  for(i = 0; i < M*N; i++){ /* broadcast change to all processors */
    bsp_put(i, &change[s*N+t], change, (s*N+t)*SZINT, SZINT);
  }
}

/**
 * swapGlobalClusters
 *  loop over local clusters and for each, determine whether to swap or not
 *  for all clusters owned by others, get swap-variable from 'owner-processor'
 *  swap clusters locally
 **/
void swapGlobalClusters(int pid, int M, int N, int m, int n, int *root, int *clusterID,
                        int *pred, int *succ, int *cluster_succ, int *spins
                       ){
  int i, j, npts, remote_pid, newspin;

  npts = m*(n>>1);
  i = 0;
  while(cluster_succ[i] != -1){ /* for all clusters that are under a proc's supervision */
    if(clusterID[cluster_succ[i]]/npts == pid && clusterID[cluster_succ[i]]%npts == cluster_succ[i]){
      if(drandom() < 0.5)      /* determine whether or not to swap the global cluster */
        pred[cluster_succ[i]] = 1;
      else
        pred[cluster_succ[i]] = 0;
    }
    i++;
  }

  bsp_sync();
  i = 0;
  while(cluster_succ[i] != -1){ /* for all clusters at this proc */
    remote_pid = clusterID[cluster_succ[i]]/npts;
    if(remote_pid != pid){
      bsp_get(remote_pid, pred, (clusterID[cluster_succ[i]]%npts)*SZINT, &pred[cluster_succ[i]], SZINT);
      /* put swapIndicator (0 or 1) in *pred, because *pred is not used for other things now */
    }
    i++;
  }
  bsp_sync();
```

```
  i = 0; j = 0;
  while(cluster_succ[i] != -1){ /* for all clusters at this proc */
    if(((clusterID[cluster_succ[i]]/npts==pid)&&pred[clusterID[cluster_succ[i]]%npts])
       || ((clusterID[cluster_succ[i]]/npts!=pid)&&pred[cluster_succ[i]])){
      newspin = -spins[cluster_succ[i]] + 1; /* just inverting the spin: 0 <-> 1 */

      j = succ[cluster_succ[i]];
      spins[cluster_succ[i]] = newspin;
      while(j != cluster_succ[i]){
        spins[j] = newspin;
        j = succ[j]; /* traverse through members of this cluster */
      }
    }
    i++;
  }
}
```

A.4 printers.c

Deze listing bevat niet besproken code, die wel gebruikt is tijdens het debuggen van het programma. Het print de gemaakte bonds op een processor (ook over de randen) en print de gemaakte clusters, samen met hun clusterID. Het geheel wordt in gnuplot formaat opgeslagen. In bijlage B is een voorbeeld te zien.

```
#include "printers.h"
#include "alloc.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/**
 * reverse
 *  reverses a string s
 *  n is the length of the string
 **/
void reverse(char s[], int n){
  int c, i, j;

  for(i = 0; i < n; i++, n--){
    c = s[i];
    s[i] = s[n];
    s[n] = c;
  }
}

/**
 * itoa
 *  returns the string-expression for the given integer n
 **/
int itoa(int n, char s[]){
  int i, sign;

  if((sign = n) < 0) /* record sign */
    n = -n;

  i = 0;
  do {  /* generate digits in reverse order */
    s[i++] = n%10 +'0'; /* get nest digit */
  } while ((n/=10) > 0); /* delete it */
  if(sign < 0)
    s[i++] = '-';
  s[i] = '\0';
  reverse(s, i-1);
  return i;
}
```

```
/**
 * printBonds
 *  loops through all local points and prints rbonds and bbonds, if present
 **/
void printBonds( int pid, int m, int n, int *spins, int *rbonds, int *bbonds){
  int i, j, mh, nh, index, npts, sl;
  char *filebonds = (char*)(calloc(5, sizeof(char)));
  char *filescript = (char*)(calloc(9, sizeof(char)));
  char *pstr = (char*)(calloc(3, sizeof(char)));
  FILE *fp, *fp2;

  /* filebonds is to contain the bonds,
     filescript is an overall script which will eventually print bonds and clusters at once. */
  sl = itoa(pid, pstr); /* returns stringlength */
  filebonds[0] = 'b';
  filebonds[1] = '_'; /* Quick 'n Dirty */
  filescript[0] = 'i';
  filescript[1] = 's';
  filescript[2] = 'i';
  filescript[3] = 'n';
  filescript[4] = 'g';
  filescript[5] = '_'; /* Quick 'n Dirty */
  for(i = 0; i < sl; i++){
    filebonds[2+i] = pstr[i];
    filescript[6+i] = pstr[i];
  }

  npts = (m*n)>>1;
  mh = m>>1;
  nh = n>>1;

  fp =fopen(filebonds,"w");
  fp2 = fopen(filescript,"w");

  fprintf(fp2, "set nolabel\n");

  for(index = 0; index < npts; index++){
    i = index/nh;
    j = index%nh; /* lokale j*/
    j<<=1;
    if(i%2 == 1)
      j++;
    fprintf(fp2, "set label %d \"%d\" at %9.6f,%9.6f\n",
                  index+1, spins[index], ((j-i)>>1)+0.15,(-(j+i)>>1)-0.15);
                  /* swap rows and columns for easy plot */
  }
  fprintf(fp2, "plot '%s' using 2:1 with lines", filebonds);
  fclose(fp2);

  for(index = 0; index < npts; index++){
    i = index/nh;
    j = index%nh; /* lokale j*/
    j<<=1;
    if(i%2 == 1)
      j++;
    if(rbonds[index] == 1){
      fprintf(fp, "%d %d\n", -(j+i)>>1, ((j-i)>>1)+1);
    }
    fprintf(fp, "%d %d\n", -(j+i)>>1, (j-i)>>1);
    if(bbonds[index] == 1){
      fprintf(fp, "%d %d\n", -(((j+i)>>1)+1), (j-i)>>1);
    }

    fprintf(fp, "\n");
  }
  fclose(fp);
}
```

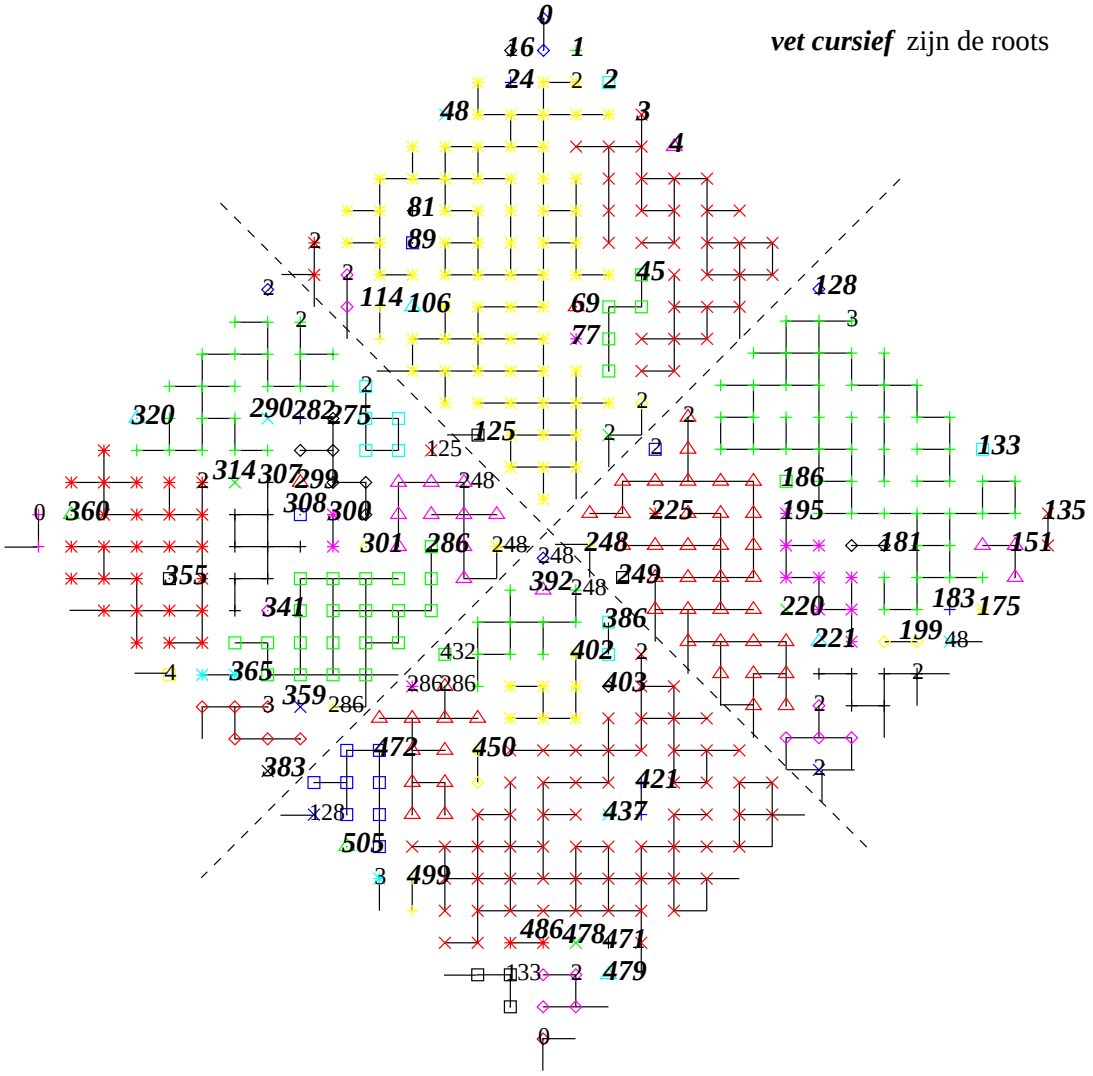
```
/**
 * printLocalClusters
 *  loops through local clusters and calls the print routine for each cluster
 **/
void printLocalClusters(int pid, int m, int n, int *pred, int *succ, int *root, int *clusterID){
  int i, j, nh, ci, npts, sl, index;
  FILE *fp;
  char *filescript = (char*)(calloc(9, sizeof(char)));
  char *pstr = (char*)(calloc(3, sizeof(char)));
  char *filename;

  sl = itoa(pid, pstr); /* returns stringlength */
  filescript[0] = 'i';
  filescript[1] = 's';
  filescript[2] = 'i';
  filescript[3] = 'n';
  filescript[4] = 'g';
  filescript[5] = '_'; /* Quick 'n Dirty */
  for(i = 0; i < sl; i++){
    filescript[6+i] = pstr[i];
  }

  fp = fopen(filescript,"a");

  npts = (m*n)>>1;
  ci = 0;
  nh = n>>1;

  for(index = 0; index < npts; index++){
    if(root[index] == index){
      filename = printCluster(pid, m, n, index, ci, succ);
      fprintf(fp, ", '%s' using 2:1 notitle with points", filename);
      ci++;
    }
  }
  for(index = 0; index < npts; index++){
    if(root[index] == index){
      i = index/nh;
      j = index%nh; /* local j*/
      j<<=1;
      if(i%2 == 1)
        j++;
      fprintf(fp, "\n set label %d \"%d\" at %9.6f,%9.6f\n",
                   npts+index+1, clusterID[index], ((j-i)>>1)-0.15, (-(j+i)>>1)+0.15);
                   /* swap rows and columns for easy plot */

    }
  }
  fclose(fp);
}

/**
 * printLocalClusters
 *  traverses through cells of a cluster and prints it in file.
 *  returns the filename in which this cluster was written.
 **/
char* printCluster(int pid, int m, int n, int root_nr, int cluster_nr, int *succ){
  int i, j, index, nh, sl, s2l;
  FILE *fp;
  char *filename = (char*)(calloc(10, sizeof(char)));
  char *pstr = (char*)(calloc(3, sizeof(char)));
  char *nr = (char*)(calloc(4, sizeof(char)));

  nh = n>>1;
  sl = itoa(cluster_nr, nr);
  s2l = itoa(pid, pstr);
```

```
  filename[0] = 'c';
  filename[1] = 'l'; /* Quick 'n Dirty */
  for(i = 0; i < sl; i++)
    filename[2+i] = nr[i];

  filename[2+sl] = '_'; /* Quick 'n Dirty */
  for(i = 0; i < s2l; i++)
    filename[3+sl+i] = pstr[i];

  fp = fopen(filename, "w");
  i = root_nr/nh;
  j = root_nr%nh; /* local j*/
  j<<=1;
  if(i%2 == 1)
    j++;
  fprintf(fp, "%d %d\n", -(j+i)>>1, (j-i)>>1);

  index = succ[root_nr];

  while(index != root_nr){
    i = index/nh;
    j = index%nh; /* local j*/
    j<<=1;
    if(i%2 == 1)
      j++;
    fprintf(fp, "%d %d\n", -(j+i)>>1, (j-i)>>1);
    index = succ[index];
  }
  fclose(fp);
  return filename;
}
```

B Cluster-voorbeeld

Met de in appendix A.4 besproken print-routines zijn voor 2×2 processoren en een lokale systeemgrootte van 16×16, 10 iteraties en $k_B T = 2.268$ plaatjes gemaakt van de clustering. In figuur 9 zijn de gebieden van de 4 processoren samengebracht.

De globale roots van de clusters zijn ***vet cursief*** gedrukt. Nu is te zien hoe bijvoorbeeld het cluster met globale ID=2 gegroeid is over alle vier de processoren.



Figuur 9: De globale clustering, verdeeld over de 4 processorgebieden