Michiel De Witte

Algoritmen en Datastructuren III

Academiejaar 2013-2014

Parallelle Algoritmen

Het Handelsreizigersprobleem

Implementatie:

De hoofdopbouw van mijn programma begint met het inlezen van de matrix in proces 0 en deze door te sturen naar de andere processen. Dan laat ik proces 0 de heuristieken draaien. Aangezien dit niet lang duurt en de 2-opt meestal snel goede bounding criteria geeft laat ik proces 0 dan samen met de andere processen verder rekenen als een proces voor de Branch & Bound. Dus alle processen gaan dus de Branch & Bound uitvoeren.

(Merk op dat het laatste proces het best de heuristieken draait aangezien bv: als je op branching niveau 20 kinderen hebt en 3 processen, dan zullen de eerste 2 processen elk 7 kinderen moeten verwerken en het laatste proces maar 6 en dus meer tijd vrij hebben om de heuristieken uit te voeren.)

Hierna kan het gebeuren dat sommige processen eerder klaar zijn dan andere. Dus kan er zich het geval voordoen dat proces *x* klaar is en het denkt de juiste oplossing te bezitten, maar een ander proces die nog niet klaar is nog een betere oplossing gaat vinden. Dus laat ik alle processen wachten op een MPI_Barrier. Hierna laat ik alle processen eventuele messages die nog niet ontvangen zijn ontvangen. Dus zal elk bericht ontvangen zijn en zullen alle processen die denken dat ze een optimale oplossing hebben effectief een optimale oplossing hebben.

Hierna stuurt elk proces zijn oplossing door naar proces 0, of als het proces 0 een optimale oplossing heeft print het dit af, anders print het de eerste ontvangen oplossing af.

Het Branch & Bound algoritme zelf begint door te luisteren als er betere grenzen zijn doorgegeven door andere processen (bv. in het begin wanneer proces 0 de heuristiek-benadering doorstuurt).

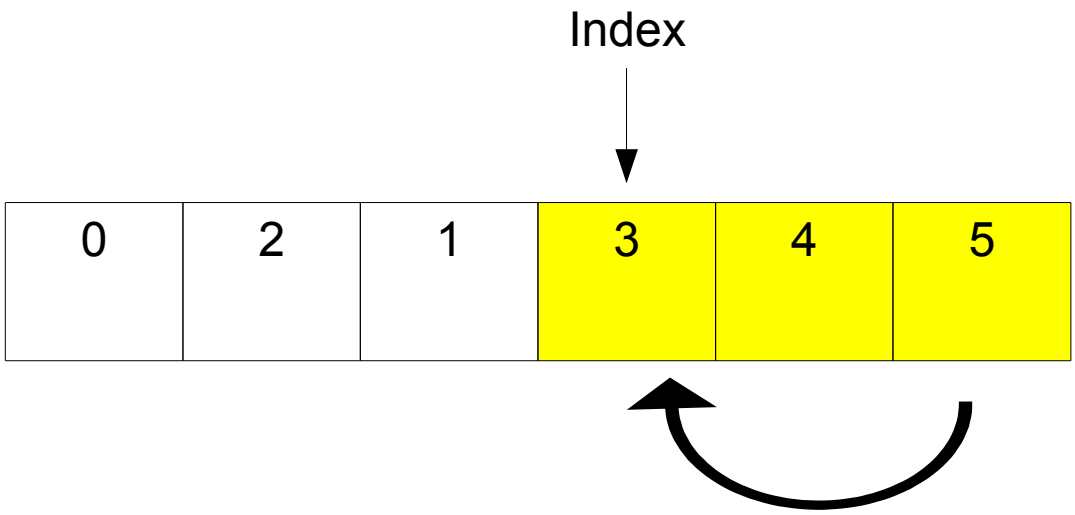
Daarna kijkt het als het algoritme in de laatste stad zit, indien dit zo is zal het kijken als het een mogelijke oplossing is door te kijken als het gewicht van deze route+ de route om terug naar het begin te gaan kleiner is dan de al reeds verkregen weight_best_route, indien dit zo is zal het deze beste oplossing bijhouden en doorsturen naar de andere processen.

Indien het proces niet in een randstad zit, zal deze alle nog niet bezochte steden overlopen en kijken als je kan bounden in die knoop. Indien je kan bounden en boven de splitsdiepte zit zal de devide_depth_index geïncrementeerd worden met het aantal steden dat men op de devide_depth overslaat. Indien men op de splitsdiepte zit en kan bounden zal men de devide_depth_index met 1 incrementeren.

Als je niet kan bounden wordt er gekeken als je op de splits diepte zit, indien dit zo is wordt opnieuw de teller geïncrementeerd die het aantal kinderen bijhoudt op de splitsdiepte, en wordt er gekeken als het huidige proces moet verderwerken in deze tak door te kijken als de huidige devide_depth_index%size==procesnummer. Dus hier wordt er gebranched tussen de processen.

Als je hier als proces wel mag doorgaan of je zit niet op de splitsdiepte, dan swap je het element op de huidige index positie in je route_matrix met de stad waarnaar je toe gaat. Je incrementeert deze index positie en dan doe je een recursieve oproep naar het Branch & Bound algoritme. Hierna wordt de index terug verminderd als men uit de recursieve call gaat, en worden ook de steden terug-geswapt.

Dit terug-swappen van steden hoeft niet als men met maar een proces zit. Dit moet wel in dit Branch & Bound algoritme omdat elk proces de kinderen op de splitsdiepte in dezelfde volgorde moet doorlopen, en het terug-swappen is daar van cruciaal belang.



De broadcast functie wordt enkel opgeroepen als er in het proces zelf een betere oplossing is gevonden. Dit stelt dan ook een variabele in dat zegt dat een optimale oplossing is gevonden in dat proces. Dan stuurt het dit door naar alle andere processen.

De functie voor te luisteren kijkt eerst als er nieuwe berichten zijn met MPI_Iprobe, indien er zijn kijkt hij hoeveel en haalt ze allemaal op. Enkel indien het een betere oplossing is wordt deze aangepast in het proces en zal het proces bijhouden dat de optimale oplossing niet meer bij zichzelf zit.

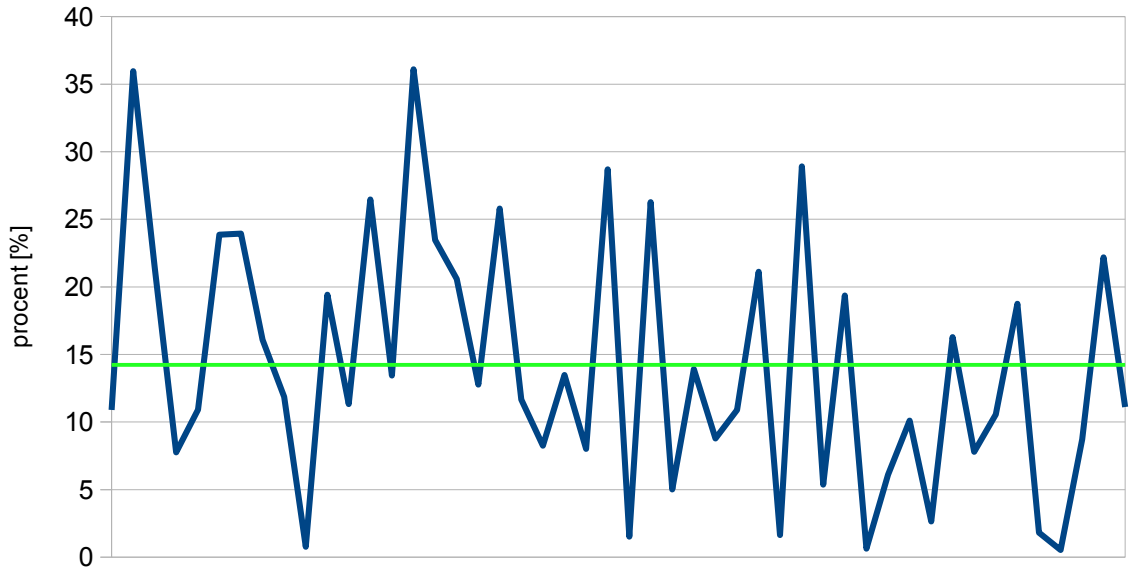
Heuristieken:

Het 2-opt algoritme werkt als volgt, eerst wordt er een willekeurig pad gegenereerd, hierna worden de toppen overlopen en de verbetering berekend als men deze swapt. Indien er een verbetering gevonden wordt zal deze swap worden uitgevoerd, dit wordt in een loop uitgevoerd tot er geen verbetering meer gevonden wordt.

Aangezien deze heuristiek snel en efficiënt kan uitgevoerd worden zal dit 200 keer na elkaar uitgevoerd worden zodat het een aanvaardbaar dichte benadering kan vinden. Relatief tot het Branch & Bound algoritme is deze tijdskost te verwaarlozen.

De tweede heuristiek Kortste Pad Eerst, is eerder een filter om de meest gemakkelijke matrix te kunnen testen en zo te zien als het Branch & Bound algoritme wel deftig zijn werk doet (zie tests). Dit overloopt de nog niet bezochte steden en kiest er de dichtste van uit en zal dit blijven doen tot er geen steden meer zijn en dus een pad heeft.

De uitkomst van de 2-opt was telkens dezelfde als de oplossing van de matrix wat weer getuigd van de goeie benaderingen van 2-opt.
Hieronder de benadering van kortste pad eerst:



Tests:

Correctheid:

Om de correctheid van mijn algoritme aan te duiden heb ik bepaalde eigenschappen getest van het algoritme.
Bijvoorbeeld dat na uitvoering alle processen hetzelfde aantal toppen heeft moeten waarnemen op de splitsdiepte. Dit heb ik getest door verschillende random matrices door het algoritme te laten verwerken. Het is nuttig deze eigenschap te testen aangezien men boven de splitsdiepte kan bounden en dus niet tot op de splitsdiepte graakt. Dit werd in mijn code dan uitgerekend hoeveel kinderen op de splitsdiepte dan moeten worden bijgeteld.
Ook moet de volgorde van de kinderen hetzelfde zijn, als men ze tegenkomt. De koppels van {devide_depth_index,city} tussen de processen werden dus vergeleken,. Als devide_depth_index gelijk was moest city ook gelijk zijn.

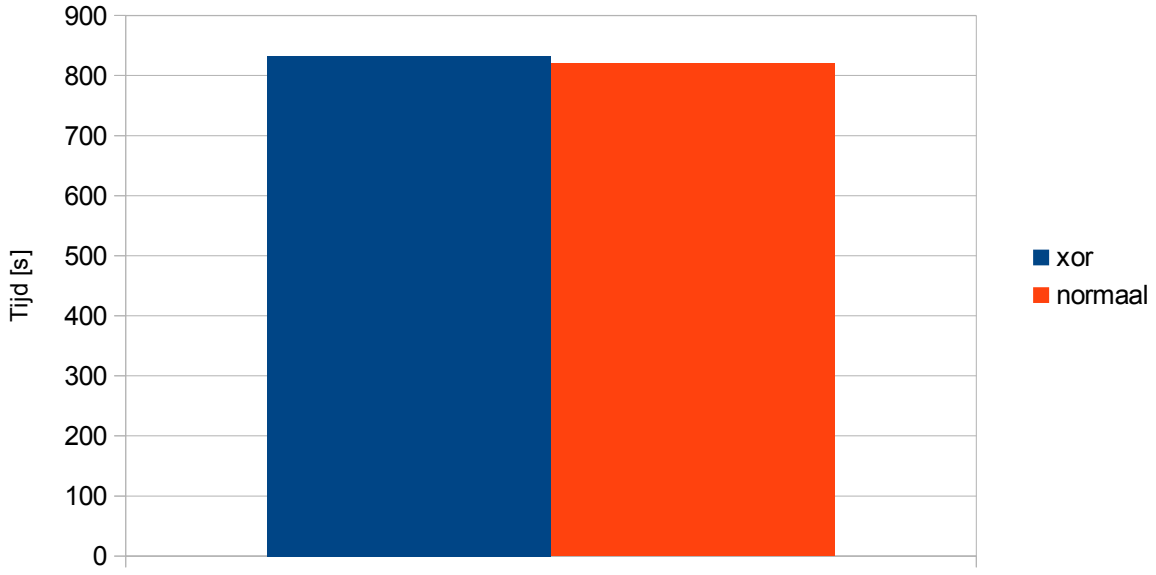
Om te kijken als mijn bounding wel goed werkt heb ik een voorbeeld gemaakt die direct het optimale pad zal geven bij het Kortste Pad Eerst algoritme en dus een goed bounding criteria. En waarbij alle branches die niet op het optimale pad liggen gebound zullen worden.
Hier is een makkelijk voorbeeld voor te maken: je maakt dat vanuit elke stad er maar 1 nog niet bezochte stad een route heeft met gewicht 1 en de andere routes een gewicht van oneindig. En van de laatste stad naar de eerste stad ook een gewicht 1 van route.
Het genereren van zo'n matrix wordt gedaan in matrix.py
Hierbij kan het algoritme een matrix van 100 nodes oplossen in onder 5 seconden.

Dit is dus een mooi voorbeeld van waarom een goed gekozen heuristiek samen met Branch&Bound een exacte en snelle oplossing kan geven tot een moeilijk probleem.

Het omgekeerde geldt ook, en dus een matrix waarbij het gewicht tussen alle paden 1 is zorgt ervoor dat er nooit gebound kan worden. En dus een zeer slechte uitvoeringstijd geeft. Het geval met dimensie 13 geeft al een uitvoeringstijd van meer dan 6 minuten.

Random:

Een test om te optimaliseren was om te kijken als een swap met xor sneller ging dan een standaard swap met een temp variabele. Hier was het resultaat dat de normale swap een klein voordeel gaf, maar geen significant verschil.

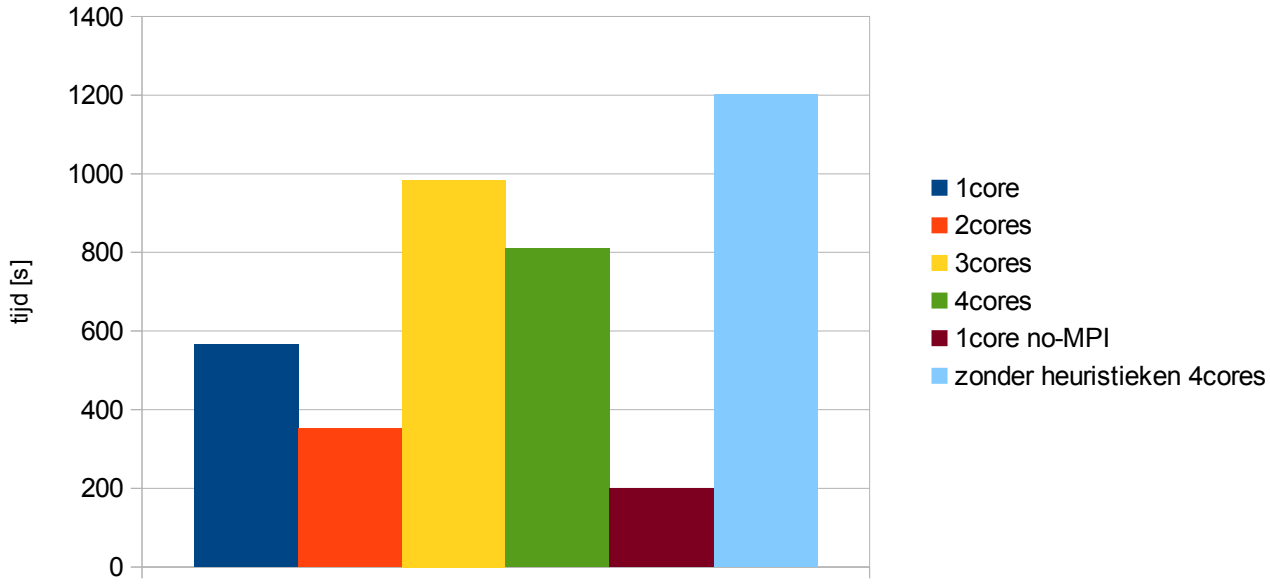


Iets wat ik niet verder geïmplementeerd heb bij mijn Kortste Pad Eerst algoritme is dat ik het algoritme kon laten lopen uit verschillende startsteden en ook een andere oplossing zou kunnen krijgen. Bijvoorbeeld wanneer de laatste stad een relatief enorme kost heeft, maar je deze route moet nemen aangezien het de laatste overblijvende is. Als je in de laatste stad begint zul je deze weg dus waarschijnlijk niet overlopen.

Tijdsmetingen:

De tijdsmetingen heb ik gedaan op de voorbeeld-bestanden op minerva en werden gedaan via MPI_Wtime.

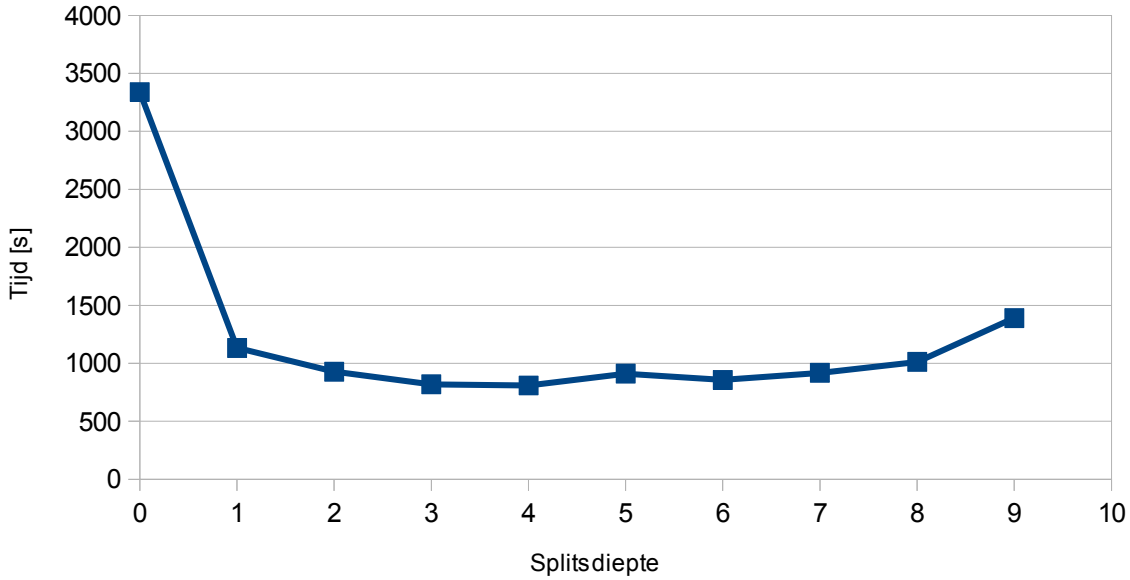
Hieronder de tijdsmetingen van het standaard-algoritme:



De balken staan in volgorde van links naar rechts zoals ze opgesomd zijn van boven naar onder in de legende.

Hieruit merk je dat de uitkomsten redelijk random lijken. Een groot deel hiervan is te wijten aan de overhead van OpenMPI. Bij 2 cores is de overhead van de berichten nog niet overdadig aangezien je maar naar 1 proces moet sturen en maar van 1 proces berichten zult krijgen. Bij 3 cores zie je al dat deze overhead ervoor zorgt dat je eigenlijk meer op OpenMPI wacht dan rekent. Des te meer cores des te meer voordeel je krijgt door OpenMPI te gebruiken. Je kunt hier duidelijk zien dat als je geen OpenMPI gebruikt de uitvoeringstijd minimaal is aangezien dit op te weinig cores is getest. Als je 4 cores met en zonder heuristieken vergelijkt zie je dat je ongeveer 30% winst hebt met heuristieken.

Hieronder de tijdsmetingen van de splitsdiepte:



Hierbij kan je zien dat de splitsdiepte een grote rol speelt. Logischerwijs als je de splitsdiepte instelt als 0 zal het algoritme nooit splitsen, en dus zal de branching niet gebruikt worden en zal elk proces de 'volledige' boom uitvoeren. (volledige bounding werkt wel nog). Je ziet dan dat een splitsdiepte van 3-4 ideaal is en dat dit dan naarmate dat de splitsdiepte groter wordt het ook terug langer duurt. Dit komt doordat elk proces hetgeen boven de splitsdiepte uitvoert, en als men de splitsdiepte vergroot, vergroot ook hetgeen erboven en dus is er voor elk proces meer rekenwerk.

De resultaten en grafieken in dit verslag kunt u vinden in het bestand results.ods. Het gebruikte testscript test.sh.