Practicum Ontwerp van Parallelle Programma's

Groep 10: Bas Kloet, Christian Luijten, Paul van Tilburg

19 maart 2004

Inhoudsopgave

1 Inleiding **2**

2 De veel-deeltjes simulatie **2**

2.1 Speedup van het ring-achtige systeem (opdracht 3.3) 2

2.2 Invloed van de grootte communicatie-eenheid (opdracht 3.4) 2

2.3 Het effect van proces-wisselingen (opdracht 3.5) 4

3 Extra opdracht: Creëer zelf een LAM/MPI cluster **4**

3.1 Testen van de cluster 6

3.2 Verbeteren van de prestaties 9

3.3 Conclusie 12

4 Conclusie **12**

Referenties **13**

A Tabellen meetgegevens **14**

A.1 Veel-deeltjes simulatie op de PAcluster 14

A.2 Veel-deeltjes simulatie op eigen cluster 14

A.3 CPU test op eigen cluster 14

A.4 Latency test op eigen cluster 15

B Geautomatiseerde test-scripts **16**

C VarQ code **18**

1 Inleiding

Bij het vak *Ontwerp van Parallelle Programma's* werd een praktische opdracht gegeven (zie [PROPP]). Deze opdracht diende te worden uitgevoerd in groepen van twee of drie studenten. Groepen van drie personen kregen een extra opdracht.

De opdracht bestond ten eerste uit het implementeren van enkele proef programma's; met behulp van barrière-synchronisatie en met synchronisatie door middel van messages passing. Ten tweede diende een bestaand programma voor een veel-deeltjes simulatie te worden onderworpen aan een timing test. De bevindingen hiervan worden besproken in hoofdstuk 2.

De extra opdracht voor onze groep bestond uit het opzetten van een eigen LAM/MPI cluster. Uitdagingen hierbij zijn het heterogeen karakter van de cluster, de hogere en verschillende latencies. Onze resultaten zijn te vinden in hoofdstuk 3.

2 De veel-deeltjes simulatie

De veel-deeltjes simulatie is een model voor het berekenen van de bewegingen en relaties tussen (veel) deeltjes. Toepassingsgebieden zijn zowel de nano fysica als de astronomie. Het model benodigt slechts de wetten van Newton en de Van der Waals krachten.

De simulatie wordt gebruikt voor het benchmarken van het LAM/MPI PAcluster van de faculteit Technische Informatica aan de Technische Universiteit Eindhoven. Bij de verschillende tests wordt een aantal variabelen constant gehouden. Ten eerste het aantal iteraties, van elk deeltje wordt 128 maal een nieuwe positie uitgerekend. Ten tweede is de Lennard-Jones potentiaal bij alle tests 3.

2.1 Speedup van het ring-achtige systeem (opdracht 3.3)

Te verwachten valt dat bij het toevoegen van meer processoren de simulatie sneller zal worden berekend. De mate waarin de berekening sneller wordt uitgevoerd heet de speedup.

Uitvoering: Er werden verschillende simulaties gestart met twee variabelen, ten eerste het aantal processen, ten tweede het aantal deeltjes.

Gebruikt werden simulaties met 256, 512, 1024, 2048, 4096 en 8192 deeltjes met 1, 2, 4, 8 en 16 processen. Zodoende werden 30 runs gemaakt, allen met 128 iteraties en een Lennard-Jones potentiaal van 3.

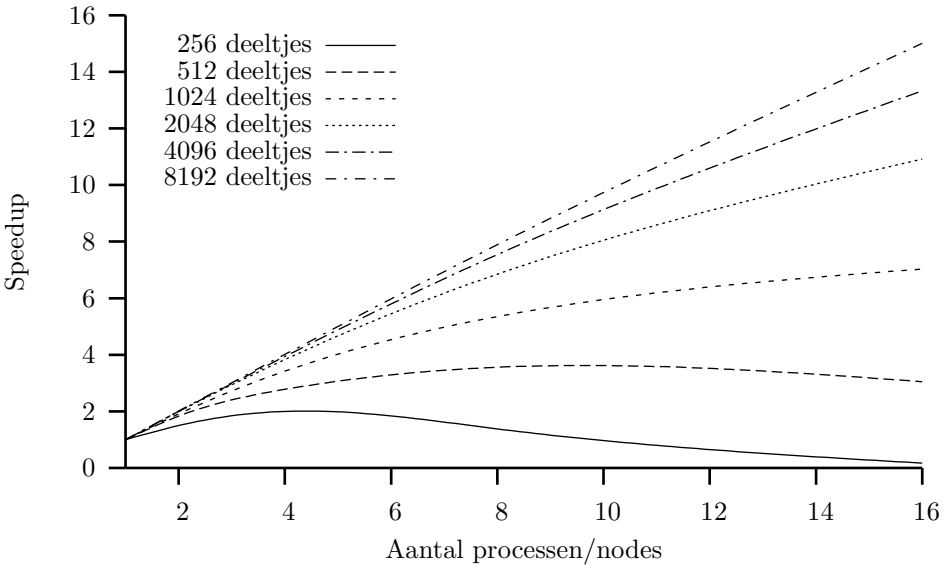
Grof-korrelige communicatie schaalt steeds beter naar mate er meer deeltjes te berekenen zijn. In Figuur 1 is te zien dat voor 256 en 512 deeltjes de speedup een maximum heeft. Te verwachten is dat deze ook bestaan voor meer deeltjes met meer processen en de speedup dus vanaf een zeker aantal processen zal afnemen en 0 zal benaderen.

Voor fijn-korrelige communicatie is het feit of er een speedup is sterk afhankelijk van het aantal deeltjes. Minder dan 2048 deeltjes levert zelfs een vertraging op, wat goed te zien is in Figuur 2.

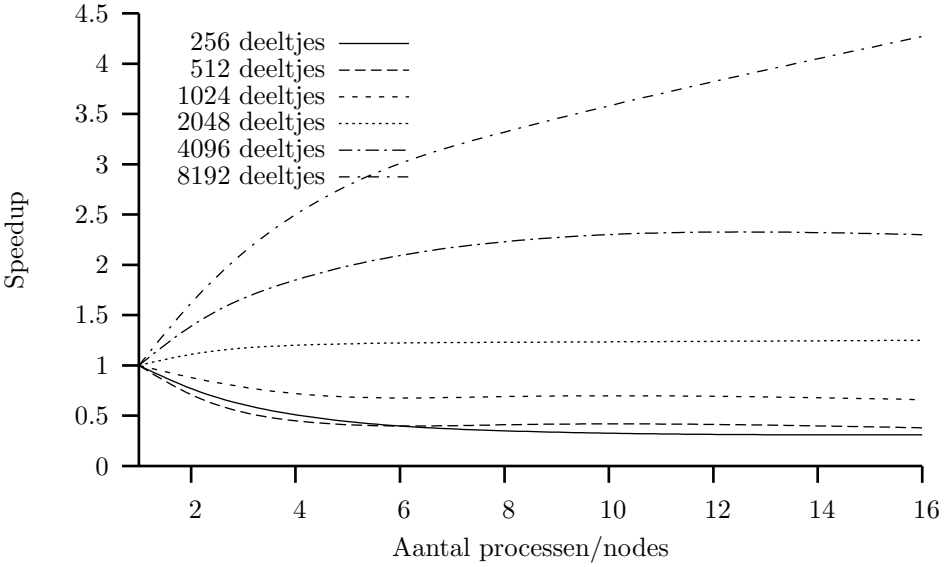
2.2 Invloed van de grootte communicatie-eenheid (opdracht 3.4)

Naast het variëren van het aantal processoren, kan ook de grootte van de communicatie-eenheden worden gevarieerd.

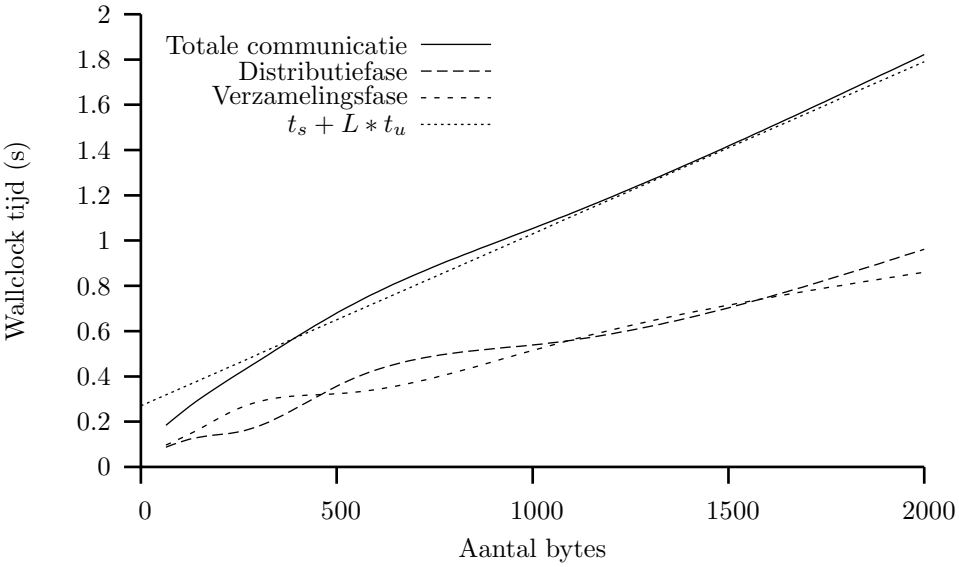
Op het college werd het volgende mogelijke timing model gegeven:



Figuur 1: Speedup bij grof-korrelige communicatie



Figuur 2: Speedup bij fijn-korrelige communicatie



Figuur 3: Tijdsduur van communicatie op een 2-node "ring"

$$\tau_c(L) = t_s + L \times t_u$$

Waarbij $\tau_c(L)$ de totale tijdsduur van communicatie is, t_s setup tijd, t_u tijd per communicatie-eenheid. L is de grootte van een eenheid.

Geverifieerd werd of dit model klopt en wat de waarden t_s en t_u zijn in dit geval.

Uitvoering: De grootte van de communicatie-eenheid L is recht evenredig met de workload Q in een systeem met 1 proces. Enkele runs op twee processoren met één rekenproces en een hoofdproces leverden een vrijwel rechte lijn op, zoals te zien in Figuur 3. Elk deeltje levert een communicatie van 8 bytes (2 integers van 4 bytes).

Voor de waarden $t_s = 0.27$ s en $t_u = 0.76$ ms onstaat een lijn die vrijwel overeenkomt met de gemeten communicatie.

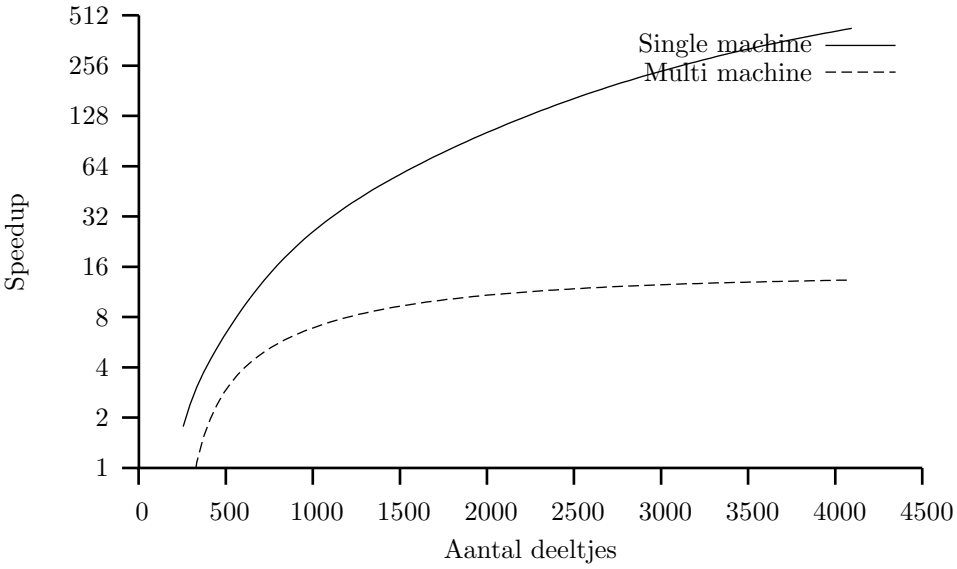
2.3 Het effect van proces-wisselingen (opdracht 3.5)

Om het effect van proces-wisselingen te bepalen is een run gedaan van 16 processen op één node. De resultaten (speedup) hiervan worden vergeleken met een run van 16 processen op 16 nodes zoals beschreve in sectie **??**. Zie Figuur **??** voor de vergelijking.

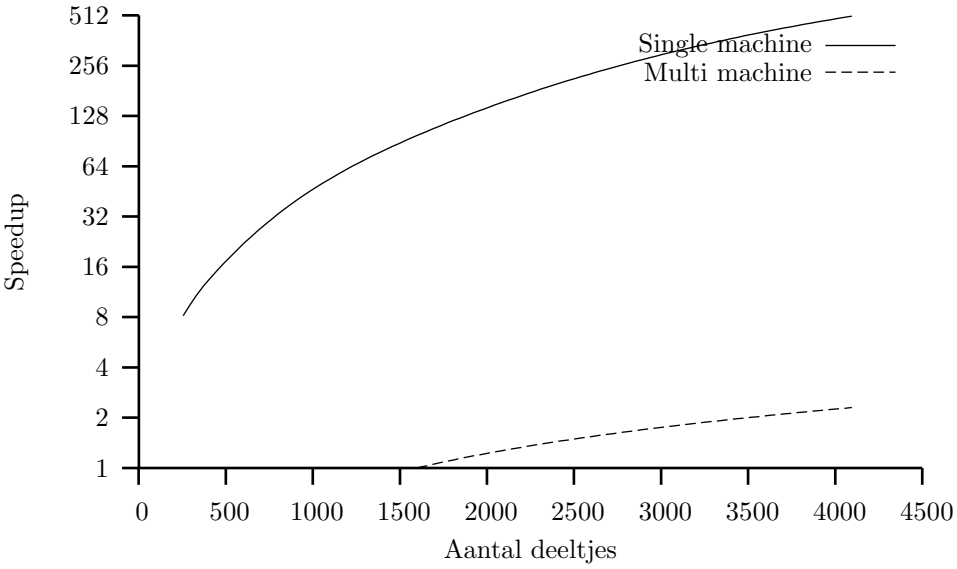
3 Extra opdracht: Creëer zelf een LAM/MPI cluster

Doordat onze groep uit drie personen bestaat, in plaats van twee, moest er een extra opdracht vervuld worden. Deze opdracht was om zelf een LAM/MPI cluster op te zetten.

Voor onze cluster hebben we zes machines gebruikt, met verschillende CPU snelheden, geheugen groottes en netwerkverbinding-en. Al deze machines draaien Debian GNU/Linux en hebben een



Figuur 4: FIXME



Figuur 5: FIXME

i386 architectuur. Alle andere relevante gegevens staan in tabel 1 en 2.

Nr.	Computer	CPU (MHz)	cache (KB)	RAM (MB)	Extra gegevens
0	Shenzou	PII 350	512	160	
1	Ranger	PIII 736	256	512	
2	Soyuz	PIII 930	256	512	
3	Darwin	PIII 733	256	256	
4	Vela	PIII 451	512	192	
5, 6	Target	PII 525	128	384	Dual CPU SMP

Tabel 1: Hardware gegevens.

Gem. latency	Darwin	Ranger	Shenzou	Soyuz	Vela	Target
Darwin	-	0.9	1.1	0.9	0.9	0.9
Ranger	0.9	-	0.4	0.1	0.1	0.1
Shenzou	1.1	0.4	-	0.4	0.5	0.4
Soyuz	0.9	0.1	0.4	-	0.2	0.2
Vela	0.9	0.1	0.5	0.2	-	0.2
Target	0.9	0.1	0.4	0.2	0.2	-

Tabel 2: Gemiddelde latency tussen de hosts (in ms).

Om van deze machines een cluster te maken, was het voldoende om op iedere machine LAM 7.0.4 te installeren. Om werkelijk van het cluster gebruik te kunnen maken, moet een gebruiker natuurlijk wachtwoordloos via RSA in kunnen loggen op alle machines van het cluster. Tevens moest in de home directory een .profile bestand aangemaakt worden. Hier hoeft niets in te staan, maar is puur omdat `lamboot` anders weigerde te werken. De 'Frequently Asked Questions' pagina van LAM/MPI [LAMFAQ] bevat alle informatie die nodig is voor het opzetten van een homogene of zelfs heterogene MPI cluster.

Nadat de cluster opgezet was, moest de prestatie getest worden. Voor alle tests zijn de parameters gebruikt als gegeven in sectie 2.

3.1 Testen van de cluster

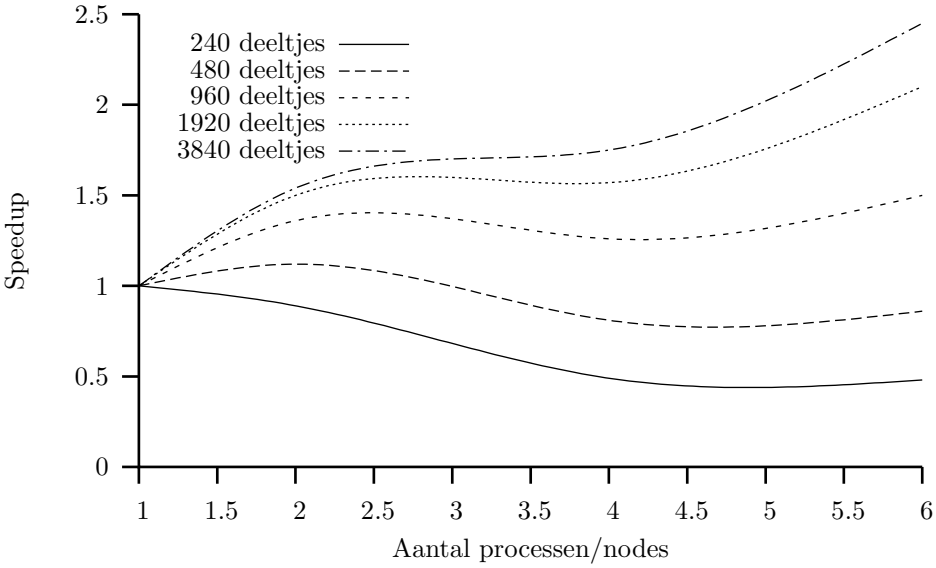
Bij het opzetten van de cluster hebben wij er specifiek voor gezorgd dat we op alle machines dezelfde binaries kunnen draaien, zodat code niet op iedere machine individueel gecompileerd hoeft te worden. Aangezien we op ons netwerk geen distributed filesystem hebben opgezet, moet de root node wel altijd het gecompileerde programma overzenden naar alle andere nodes. Dit kan gelukkig met een simpele command-line optie[1] van 'mpirun'.

Het feit dat de cluster bestaat uit machines met verschillende processor snelheden en latency ten opzichte van elkaar heeft natuurlijk een grote invloed op de snelheid van de berekeningen. Om de precieze invloed van deze twee factoren te bepalen, hebben we een aantal tests uitgevoerd. Bij al deze tests wordt Shenzou als de root node gebruikt, aangezien dit de traagste machine is en de root node meestal het minste rekenwerk hoeft te verzetten.

3.1.1 Algemene cluster test

Allereerst is er een test uitgevoerd van de totale cluster, om de algemene prestaties te meten. Hiervoor is gebruik gemaakt van de test van Assignment 3.3, met enige aanpassingen, namelijk het weglaten van de sequentiële test en alle tests met 8192 deeltjes.

[1] De command-line optie: -s n0



Figuur 6: Speedup bij grof-korrelig communicatie

Uitvoering: In deze test wordt gewerkt met een cluster van 1, 2, 4 en 6 nodes is aanpassing van N noodzakelijk. De gebruikte machines bij iedere clustergroote zijn af te leiden uit de tabel 1 op blz. 6. Bij de keuze van bijvoorbeeld $N = 256$ leidt dit al tot problemen omdat 256 niet door 6 deelbaar is. Er is gekozen voor de volgende waarden van N: 240, 480, 960, 1920 en 3840, die de waarden van de originele test beschreven in hoofdstuk 2.1 benaderen.

Resultaat: De resultaten zijn weergegeven in Figuur 6 en Figuur 7. De eigenlijk meetgegevens staan in Appendix A.2.

3.1.2 Latency test

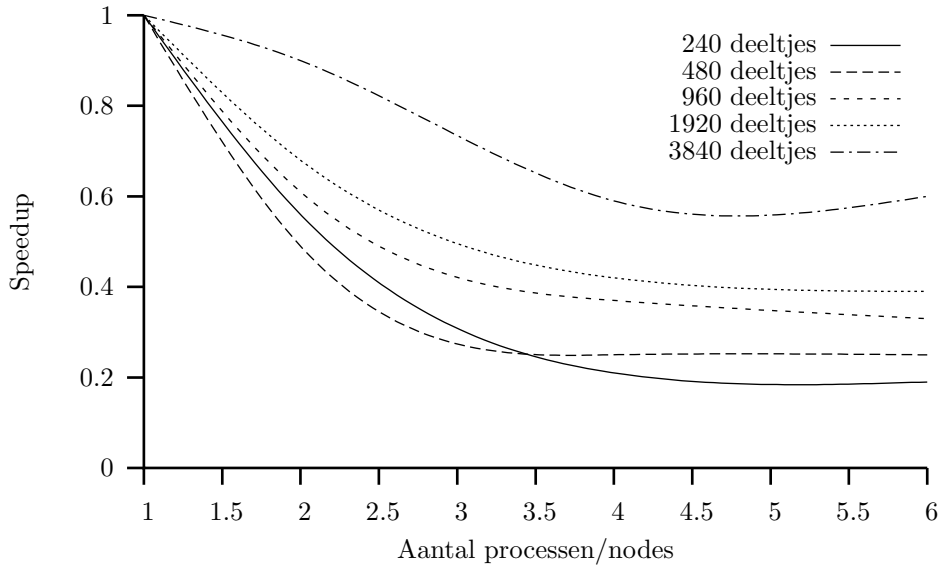
Zoals in tabel 2 te zien is, is de latency van Darwin naar de rest van de cluster ongeveer twee maal zo groot als tussen de andere nodes. Om het effect hiervan te meten, is een variatie op de algemene test van hoofdstuk 3.1.1 uitgevoerd:

Het aantal processen is gelijk aan het aantal machines in het deelcluster (4), maar dezelfde test wordt uitgevoerd op de volgende twee deelclusters:

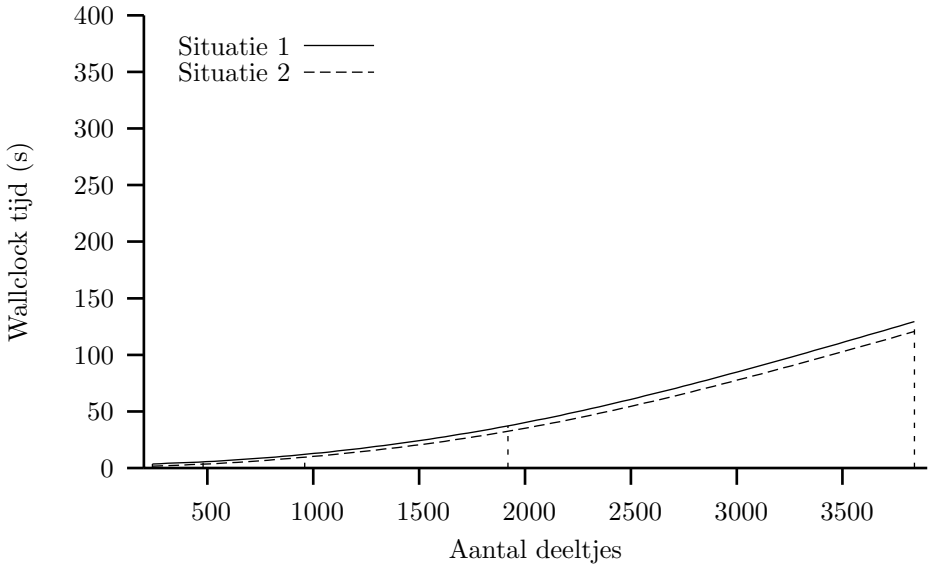
1. Shenzou + Soyuz + Vela + Ranger
2. Shenzou + Soyuz + Vela + Darwin

Omdat Ranger en Darwin qua hardware amper van elkaar verschillen, zal het gemeten verschil vrijwel alleen afhangen van het verschil in latency.

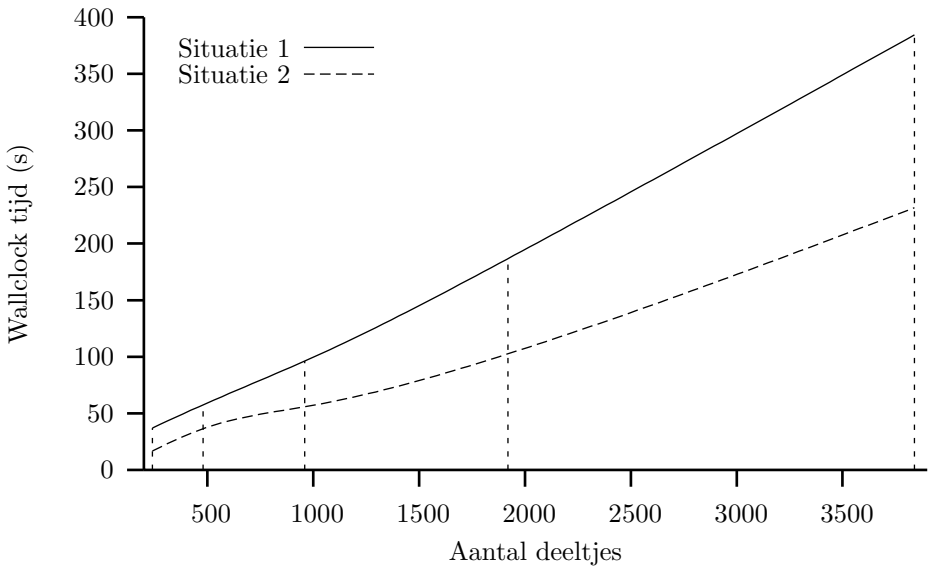
Hypothese: De hogere latency naar Darwin zal op de berekeningen met Course Grained communicatie vrijwel geen effect hebben[2], maar die met Fine Grained communicatie zullen significant langer duren, vooral bij veel deeltjes.



Figuur 7: Speedup bij fijn-korrelig communicatie



Figuur 8: Latency test – benodigde rekentijd bij grof-korrelige communicatie



Figuur 9: Latency test – benodigde rekentijd bij fijn korrelige communicatie

Resultaat:

3.1.3 CPU test

Om het effect van de CPU snelheid van de individuele nodes te testen, is het effect berekend van het toevoegen van een langzame node aan een snelle deelcluster. Hiervoor is dezelfde test uitgevoerd als bij hoofdstuk 3.1.1, maar in plaats van het verwisselen van twee machines, wordt nu een trage computer aan een snelle cluster toegevoegd.

Test clusters:

1. Shenzou + Soyuz + Ranger
2. Shenzou + Soyuz + Ranger + Vela

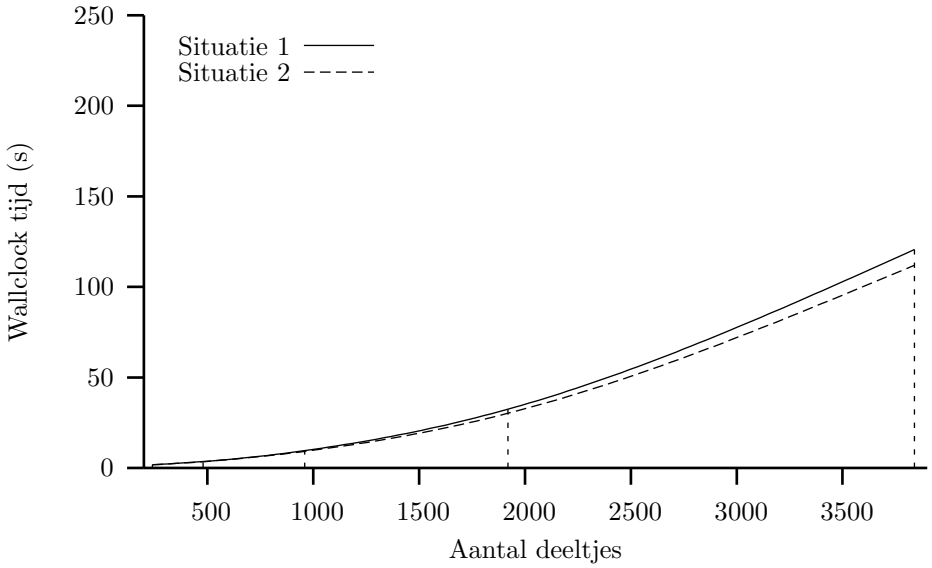
Hypothese: Aangezien ieder proces een gelijk deel van de berekening toebedeeld krijgt, zal het toevoegen van Vela eerder een negatief dan een positief effect op de prestatie van cluster hebben, doordat alle machines moeten wachten tot Vela klaar is met de berekening.

Resultaat:

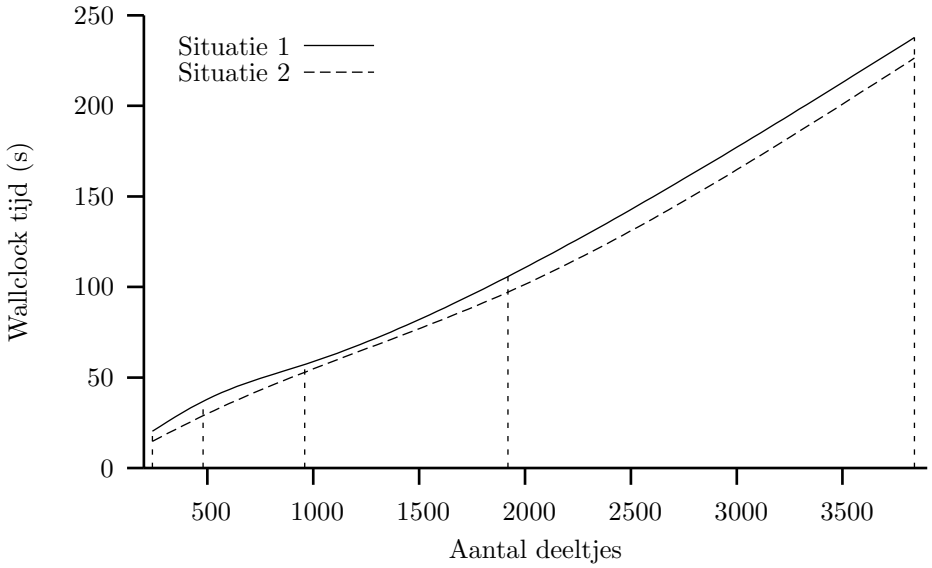
3.2 Verbeteren van de prestaties

Uit de voorgaande tests bleek wel dat de benutting van onze cluster verre van optimaal was. Een deel van deze problemen kan vermeden worden door een groot genoege korrel grootte te nemen, maar is er een probleem dat minder makkelijk op te lossen was. Dat was het probleem van de de grote verschillen in rekenkracht tussen de verschillende machines.

[2] Bij een klein aantal deeltjes is het effect waarschijnlijk nog wel significant..



Figuur 10: CPU test – benodigde rekentijd bij grof-korrelige communicatie



Figuur 11: CPU test – benodigde rekentijd bij fijn-korrelige communicatie

Zoals de CPU test als liet zien, heeft het toevoegen van een trage machine een negatief effect op de prestaties van het cluster, doordat alle andere nodes moeten wachten tot de traagste machine klaar is. De beste oplossing hiervoor zou zijn om de berekeningen over de nodes te verdelen op een manier die rekening houdt met de rekenkracht van een node. De belangrijkste problemen zijn:

1. Hoe de rekenkracht van een node te bepalen.
2. Hoe de bepaalde rekenkracht te gebruiken om de last te verdelen.

3.2.1 Rekenkracht bepalen

Er is een aantal mogelijkheden om de (relatieve) rekenkracht van een node te bepalen. Hieronder worden de mogelijkheden beschreven die door ons overwogen

De simpelste methode is om simpelweg te kijken naar de CPU snelheid van de machine. Dit heeft als voordeel dat het snel en simpel is, maar daar staat natuurlijk tegenover dat het erg onnauwkeurig is en geen rekening houdt met zaken als de CPU cache.

Een hogere mate van nauwkeurigheid kan bereikt worden door het doen van een algemene benchmark test op de gehele machine. Dit geeft een beter beeld van de werkelijke capaciteit van een node, maar houdt nog steeds geen rekening met het type berekening dat uitgevoerd moet worden. Tevens moet bij het toevoegen van een node de hele benchmark opnieuw worden uitgevoerd.

De beste oplossing is natuurlijk om de te berekenen functie te gebruiken bij het bepalen van de relatieve rekenkracht van een node, door deze als een specifieke benchmark te gebruiken. Deze oplossing is verreweg het nauwkeurigst, maar moet wel voor ieder type berekening opnieuw uitgevoerd worden.

3.2.2 Het verdelen van de rekenlast

Op het moment dat de relatieve rekenkracht van iedere node bepaald is, moet deze natuurlijk nog gebruikt worden om de rekenlast evenredig over elke node te verdelen. Hiervoor hebben wij een aantal mogelijkheden bekeken.

Allereerst kan er voor gekozen worden om de 'host file' aan te passen. Door een node meerdere malen in dezelfde host file te zetten, worden hier ook meerdere processen op uitgevoerd. Deze simpele oplossing heeft een aantal significante nadelen:

- Het is hoogst onnauwkeurig.
- Er worden meerdere processen gestart op dezelfde node, wat funest is voor de prestatie, zoals getoond in hoofdstuk 2.3.

Voor het beste resultaat, moet de geschatte/berekende rekenkracht van een node gebruikt worden om de iedere node een verschillende korrelgrootte te bedelen.

Wij hebben er voor gekozen om deze laatste oplossing verder uit te werken.

3.2.3 Een voorgestelde oplossing: VarQ

Het principe van onze oplossing is het toevoegen van een simpel stuk code aan het begin van ieder programma dat op de heterogene kluster wordt uitgevoerd.

Deze code voert op iedere node van het kluster een benchmark uit door de belangrijkste functie van het programma (een aantal keer) uit te voeren en de benodigde tijd te meten. De gemeten tijd wordt doorgegeven aan de host, zodat deze de de korrel grootte voor iedere individuele node kan bepalen.

Om aan de hand van de gemeten tijd de variabele korrelgrootte te bepalen, moet een kleine berekening worden uitgevoerd.

Allereerst wordt de relatieve snelheid van een node bepaald, deze is 1/x, waarbij x de gemeten tijdsduur is. Door deze snelheid te delen door de som van alle snelheden wordt een genormaliseerde snelheid verkregen. De som van alle genormaliseerde snelheden is altijd 1, dus om de variabele korrelgrootte te bepalen hoeft de genormaliseerde snelheid alleen maar met N vermenigvuldigd te worden.

In tabel 3 is een voorbeeld uitgewerkt van deze berekening, met N=3600 en een gemeten tijdsduur per node van resp. 1, 2, 3 en 6 seconden.

Node	Tijdsduur	Snelheid	Genorm. snelheid	Korrelgrootte(Q)
1	3	1/3	1/6	600
2	1	1	1/2	1800
3	6	1/6	1/12	300
4	2	1/2	1/4	900

Tabel 3: Voorbeeldberekening variabele korrelgrootte.

De benodigde code voor deze berekening is toegevoegd als ‘proof of concept’ in appendix C.

3.3 Conclusie

4 Conclusie

Referenties

[LAMFAQ] The LAM team, *LAM FAQ*
`http://www.lam-mpi.org/faq/`

[PROPP] Lab course Design of Parallel programs (2IN13), *R.H. Mak*

[OPPDT] Lecture Notes in Parallel Programming (2R700), *Martin Rem*

A Tabellen meetgegevens

A.1 Veel-deeltjes simulatie op de PAcluster

- Grof-korrelige communicatie:

N	T_{seq}	T(1,N)	S(1,N)	T(2, N)	S(2,N)	T(4,N)	S(4,N)	T(8,N)	S(8,N)	T(16,N)	S(16,N)
256	2.1	1.80	1.00	1.20	1.50	0.90	2.00	1.30	1.38	1.05	0.17
512	8.3	6.40	1.00	3.50	1.83	2.30	2.78	1.80	3.56	2.10	3.05
1024	33.6	24.60	1.00	12.90	1.91	7.20	3.42	4.60	5.35	3.50	7.03
2048	141.3	97.20	1.00	48.90	1.99	25.40	3.83	14.20	6.85	8.90	10.92
4096	536.7	386.60	1.00	193.50	2.00	98.20	3.94	51.30	7.54	29.00	13.33
8192	2307	1546.00	1.00	768.00	2.01	386.00	4.01	196.00	7.89	103.00	15.01

- Fijn-korrelige communicatie:

N	T_{seq}	T(1,N)	S(1,N)	T(2, N)	S(2,N)	T(4,N)	S(4,N)	T(8,N)	S(8,N)	T(16,N)	S(16,N)
256	2.1	9.00	1.00	11.70	0.77	17.60	0.51	25.60	0.35	29.50	0.31
512	8.3	13.60	1.00	19.20	0.71	30.00	0.45	33.40	0.41	35.80	0.38
1024	33.6	34.70	1.00	39.50	0.88	48.10	0.72	50.20	0.69	52.80	0.66
2048	141.3	105.40	1.00	95.20	1.11	87.70	1.20	85.60	1.23	84.10	1.25
4096	536.7	388.00	1.00	279.00	1.39	210.00	1.85	174.00	2.23	169.00	2.30
8192	2307	1516.00	1.00	934.00	1.62	607.00	2.50	456.00	3.32	335.00	4.27

A.2 Veel-deeltjes simulatie op eigen cluster

- Grof-korrelige communicatie:

N	T(1,N)	S(1,N)	T(2,N)	S(2,N)	T(4,N)	S(4,N)	T(6,N)	S(6,N)
240	1.54	1.00	1.73	0.89	3.17	0.49	3.19	0.48
480	3.90	1.00	3.47	1.12	4.83	0.81	4.53	0.86
960	12.48	1.00	9.15	1.36	9.89	1.26	8.34	1.50
1920	45.27	1.00	30.21	1.50	28.09	1.57	21.60	2.10
3840	173.00	1.00	112.13	1.54	99.03	1.75	70.67	2.45

- Fijn-korrelige communicatie:

N	T(1,N)	S(1,N)	T(2, N)	S(2,N)	T(4,N)	S(4,N)	T(6,N)	S(6,N)
240	7.22	1.00	12.97	0.56	34.22	0.21	38.98	0.19
480	13.81	1.00	28.42	0.49	54.24	0.25	56.29	0.25
960	32.19	1.00	52.97	0.61	87.28	0.37	96.86	0.33
1920	64.42	1.00	96.66	0.68	157.07	0.42	168.35	0.39
3840	203.36	1.00	255.86	0.90	342.20	0.59	339.65	0.60

A.3 CPU test op eigen cluster

- Grof-korrelige communicatie:

N	Situatie 1	Situatie 2
240	1.73	1.79
480	3.47	3.56
960	9.16	9.60
1920	30.21	32.48
3840	112.01	120.65

- Fijn-korrelige communicatie:

N	Situatie 1	Situatie 2
240	14.72	20.30
480	28.88	36.88
960	52.92	57.23
1920	97.18	105.83
3840	226.39	237.70

A.4 Latency test op eigen cluster

- Grof-korrelige communicatie:

	N	Situatie 1	Situatie 2
1[em]	240	1.74	3.63
	480	3.42	5.47
	960	9.58	12.06
	1920	32.47	37.26
	3840	120.63	129.39

- Fijn-korrelige communicatie:

	N	Situatie 1	Situatie 2
1[em]	240	16.80	37.07
	480	36.44	57.56
	960	55.77	96.24
	1920	102.63	186.76
	3840	231.73	384.47

B Geautomatiseerde test-scripts

run.sh

```
#!/bin/sh
#
# Executes MPS binaries for testing purposes with variable
# number of processes, used CPUs, load and type of communication.

# Constants:
E=3
K=128

# File paths and parameters
BINCGC=./mpsringcgc
BINFGC=./mpsring
BINSEQ=./mpsseq
MPIRUN=mpirun
MPIARGS="-v -ssi rpi tcp -s n0 C"
INFILE=mps_A
ININPT=equi
OUTDIR=out
export RUNID=$$

for N in 240 480 960 1920 3840; do

  echo ">>> Run for $N particles:"

  # Prepare input
  echo Using input equi file for: $ININPT.$N
  ln -s -f $ININPT.$N $INFILE

  # Sequential run:
  (echo $N; echo $K; echo $E) | $BINSEQ \
   > $OUTDIR/run$RUNID-seq-$N.out

  # Parallel runs:
  for P in 1 2 4 6; do
    Q=$(( $N / $P ))
    echo -n "... using $P processes: "

    # Coarse-grained communication version:
    echo -n "CGC "
    (echo $P; echo $Q; echo $K; echo $E) | $MPIRUN $MPIARGS $BINCGC \
      > $OUTDIR/run$RUNID-cgc-$N-$P.out

    # Fine-grained communication version:
    echo -n "FGC"
    (echo $P; echo $Q; echo $K; echo $E) | $MPIRUN $MPIARGS $BINFGC \
      > $OUTDIR/run$RUNID-fgc-$N-$P.out
    echo "."

  done #P

done #N

# Generate results summary file:
./results.sh > $OUTDIR/run$RUNID-results.out
```

run-cpu.sh

Dit script verschilt van het script in de vorige sectie door het feit dat het aantal gebruikte processen (en ook nodes) constant is gehouden tijdens de test. Er is dus geen iteratie over P.

```
#!/bin/sh
#
# Executes MPS binaries for testing purposes with variable
# number of processes, used CPUs, load and type of communication.
#
# CPU test: use 4 nodes (ring size is 3 processes)

# Constants:
P=3
E=3
K=128

# File paths:
BINCGC=./mpsringcgc
BINFGC=./mpsring
MPIRUN=mpirun
MPIARGS="-v -ssi rpi tcp -s n0 C"
INFILE=mps_A
ININPT=equi
OUTDIR=out/cpu
export RUNID=$$

for N in 240 480 960 1920 3840; do

  echo ">>> Run for $N particles:"

  # Prepare input:
  echo Using input equi file for: $ININPT.$N
  ln -s -f $ININPT.$N $INFILE

  # Parallel run:
  Q=$(( $N / $P ))
  echo -n "... using $P processes: "

  # Coarse-grained communication version:
  echo -n "CGC "
  (echo $P; echo $Q; echo $K; echo $E) | $MPIRUN $MPIARGS $BINCGC \
    > $OUTDIR/run$RUNID-cgc-$N.out

  # Fine-grained communication version:
  echo -n "FGC"
  (echo $P; echo $Q; echo $K; echo $E) | $MPIRUN $MPIARGS $BINFGC \
    > $OUTDIR/run$RUNID-fgc-$N.out
  echo "."

done #N

# Generate results summary file:
./results-cpu.sh > $OUTDIR/run$RUNID-results.out
```

run-latency.sh

Voor de latency-test is hetzelfde script gebruikt als in sectie **??** met alle voorkomens van "cpu" vervangen door "latency".

C VarQ code

De hieronder gegeven functie *varQ* kan in MPI programma's gebruikt worden om een gebalanceerdere Q te vinden per node afhankelijk van zijn rekenkracht in plaatst van N gelijkmatig over P te verdelen:

calcQ.h

```
/* Wrapper function for the time-consuming functionality to be timed by the
 * nodes. */
typedef void (*Mainfunc) (void *user_data);

/* Calculates the distribution of N * Q workload over P nodes based on
 * calculating power of each node, such that fast nodes get a larger Q
 * and slow node a smaller Q.
*/
extern int** calQ(int N, int P, Mainfunc mainf);
```

calcQ.c

```
#include <sys/time.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

// Precision in calculating the variable Q, higher value => more accuracy
#define PRECISION 10

extern int** calQ(int N, int P, Mainfunc mainf) {
  float total_ntime = 0;        // Total time, after normalisation
  unsigned long duration;       // End time of a process
  unsigned long durations[P]; // Array of durations per node
  int* varQ = malloc(P * sizeof(int) + 1); // Calculated new Q value per node

  struct timeval tv1, tv2;
  int rank = MPI::COMM_WORLD.Get_rank();

  varQ[0] = 0; // Set a value for the root node;

  if (rank == 0) {
    // Receive timing info from all nodes:
    for (int x = 1; x <= P; x++) {
      MPI::COMM_WORLD.Recv(&duration, 1, MPI::LONG, P, P);

      // Save duration:
      durations[x - 1] = duration;
      total_ntime += 1 / (float)durations[x - 1];
    }
  }
```

```
30    else {
        // Time PRECISION executions of running Mainfunc mainf:
        get_time_of_day(&tv1,NULL);

        for (int i = 0; i < PRECISION; i++) {
          Mainfunc();
        }

        get_time_of_day(&tv2, NULL);

40      // Calculate and send duration to host:
        duration = (tv2.tv_sec - tv1.tv_sec) * 1000000 +
                   tv2.tv_usec - tv1.tv_usec;

        MPI::COMM_WORLD.Send(&duration, 1, MPI::LONG, 0, P);
      }

      // Calculate new Q per host:

      double float cfactor = N / total_ntime; // Constant factor
50
      for (int x = 0; x < P; x++) {
        varQ[x + 1] = cfactor / durations[x]; // varQ[0] is the root node
      }

      return &varQ;
    }
```