

Klyngearkitekturer og beregninger 2008

Opgavestiller: Brian Vinter

Opgaveløser: Lars Ole Belhage

CPR: 090260-0795

E-Mail: belhage@midibel.com

Tlf: 36151730

Oversigt

Valg af assignments

Jeg har meget bevidst valgt de 3 opgaver ud fra nyheds og udbredelses principper.

PVM og MPI fordi begge metoder er nye for mig og fordi de er meget udbredt. Jeg har checket med – de få – erhvervskontakter som jeg ved bruger en eller anden form for HPC (Mentor Graphics, Nangate og TDC. De bruger gridframework SGE (Sun Grid Engine) og Platform LSF som begge har både PVM og MPI ”indbygget”.

TupleSpace's er valgt fordi både de og pyLinda er nye for mig (og der er jo noget sært velklingende over navnet Linda ;))

Shared Memory er fravalgt, dels fordi det er kendt stof for mig, dels fordi jeg anser det for en ”dead-end” - skalerbarheden er temmelig begrænset.

Remote Memory blev fravalgt, metoden er ny for mig – men vores muligheder for at afprøve det var begrænset til en ”hjemmelavet” implementation med RPC eller java-rmi. De er begge velkendte metoder for mig – og ikke nogen jeg ønsker frivilligt at bruge mere af mit liv på ;) !

Rækkefølge af assignments

Jeg har arbejdet på opgaverne samtidigt, men prosa beskrivelsen startede med MPI. Derfor kommer den først i sættet. Dernæst PVM og slutteligen pyLinda...

Kodestil og metoder

Jeg har valgt at bruge det udleverede materiale så ”råt” som muligt og prøvet bare at pakke det ind i paralleliserings-api'erne.

Dette betyder bl.a. kommentar-mængder, indryknings valg (Python rules!) og variabel navngivning er noget arbitrær (herligt at se Fortran-rødderne flere steder (iN, fY, ..)) - men på den anden side: der er så få linier kode at det bliver højt-læsning for burhøns hvis det skulle formatteres efter ”best practices”.

Jeg har derimod valgt at instrumentere rigeligt med tidsmålinger og tællere – som jeg har ladet blive i koden (selvom der sikkert kunne spares nogle procent ved at fjerne dem).

Jeg har også valgt at lade udkommenterede ”debug”-print og ”alternative” kald blive – det giver en ide om hvilke retninger jeg har forsøgt mig i...

Alle målinger, sekventielle som parallelle, er foretaget på udregningskernen og ikke på hele programmets køretid – lidt snyd, men mere pædagogisk.

Al kode og logfiler fra MiG er samlet i <http://lobnet.dk/klynger.tgz>

eAssignment-3

Udviklingsforløb:

Med udgangspunkt i c-versionen af SOR, omskrev jeg den først til en Rød-Sort udgave – og sammenlignede resultatet med sor.c

```
for (color = 0; color < 2; ++color)
{
    for(y=1;y<h-1;y++)
        for(x=1;x<w-1;x++)
            if (((x + y) & 1) ^ color)
            {
                old=trap_data[y*WIDTH+x];
                trap_data[y*WIDTH+x]=0.2*(trap_data[y*WIDTH+x]+trap_data[(y-1)*WIDTH+x]
+trap_data[(y+1)*WIDTH+x]+trap_data[y*WIDTH+x-1]+trap_data[y*WIDTH+x+1]);
                delta+=fabs(trap_data[y*WIDTH+x]-old);
            }
}
```

Med denne version som reference, udvikledes så en MPI version, som hvis den kaldes med antal processer=1 vil bruge den sekventielle version. Jeg startede med en version som brugte simple MPI_Send, MPI_Recv og MPI_ReduceAll (openMPI bufferer tilstrækkeligt til at der ikke opstod deadlocks). Denne version fulgte flowet i den indre løkke:

```
for hver farve
    Send borderdata i modsat farve
    Beregn indre celler i farven
    Modtag borderdata
    Beregn border celler
ReduceAll
```

Ud fra timing målinger på den simple version, kunne iagttages 2 problemer:

- MPI_Send tog relativt lang tid (indtil openMPI “overtog” kaldet og returnerede)
- MPI_Allreduce tog uforholdsmæssigt lang tid

For at afhjælpe MPI_Send ændrede jeg til non-blocking MPI_Isend/MPI_Irecv og indførte MPI_Test kald undervejs i beregningen af de indre celler – dette fjernede/flyttede ventetiden til modtagelsen af borderdata.

MPI_Allreduce er sværere at “forbedre” (men den burde jo også være optimeret i MPI-implementationen ;)) - den har ikke en non-blocking version. I og med at SOR itererer mange tusinde gange for asymptotisk at finde en løsning, ville min umiddelbare “løsning” være at checke stopbetingelsen sjældnere, f.x. hver 100. gang (dette vil gøre at man risikerer at iterere op til 99 gange for meget). Dog er denne metode dømt som “SNYD” (selvom man vel i den virkelige verden selvfølgelig ville gøre det ?).

Derfor har jeg lavet en implementation “i hånden” - der hvor MPI_Allreduce ville blive kaldt, sendes i stedet delta værdien til alle andre processer med MPI_Isend (broadcast kan ikke bruges idet ingen af MPI collective funktioner er non-blocking). Der indsættes MPI_Test i næste iterations send border – og lige før beregningen af de indre celler modtages delta-værdierne fra alle andre processer og stopbetingelsen kan checkes.

Overraskende var denne "reduce i hånden" væsentligt hurtigere end den indbyggede (pga overlap med border-kopi/send og nok også af andre årsager – idet den målbare tid som border-kopi/send tager ikke helt kan forklare forbedringen). Ved 16 Processer 500x500 er den samlede tid 98s med den indbyggede og 82s med den håndlavede (og 52s med SNYD check hver 100. gang)!

Næste forbedring jeg indførte var at bruge MPI_Testall / MPI_Waitany i stedet for at tvinge en given modtagelses rækkefølge – mindskede målbart ventetiden ved Recv, men ikke meget på den totale køretid. Det gjorde dog koden meget mindre læsbar (idet den ødelagde mit oprindelige layout – og jeg ikke liiiiige har brugt tid til at omstrukturere (det er jo ikke kunst vi laver ?))

Sidste ændring var at bruge "Persistent communication requests" - det gav en målbar hastigheds forbedring på 20-30% på den totale køretid. (og gør endda koden lidt "kønnere").

Jeg har også indført en option som laver en send/recv måling ved at sende en message rundt i ring af alle processerne (denne måling viser at en udveksling tager ca. 200 mikrosekunder).

Målinger:

Jeg har kørt den endelige version på MiG (og på min egen PS3, som er 30-40% hurtigere end MiG – compilerversioner ??)

Versionen er dog stadig instrumenteret med tidsmålinger – hvilket koster i underkanten af et mikrosek per måling – dvs nogle få sekunder totalt set – dvs ikke helt uden effekt på den totale køretid (dog < 10%)

Jeg har kørt på 3 forskellige problem størrelser 500x500 som anbefalet, 1000x1000 og 2000x2000.

De basale tal, Tabel 3.1:

Size	Iterations	1P-time	2P-time	2P-calc	4P-time	4P-calc	8P-time	8P-calc	9P-time	9P-calc	16P-time	16P-calc
500x500	46287	661	273	245	150	125	95	55	91	47	82	26
1000x1000	36943	2118	819	788	434	399	245	197	218	174	149	95
2000x2000	15764	3753	1352	1326	688	665	368	338	404	386	202	169

Alle tider i sekunder. "calc"-tiderne er tiden brugt i selve beregningsløkkerne.

Programmet måler en hel del tider: total tid, calc-tid, reduce-tid, send-tid, recv-tid, wait-tider, ... men kun total-tid og calc-tid er brugt her (de andre har dog været gode i optimeringsøjemed).

Målingen af message-ringen, Tabel 3.2:

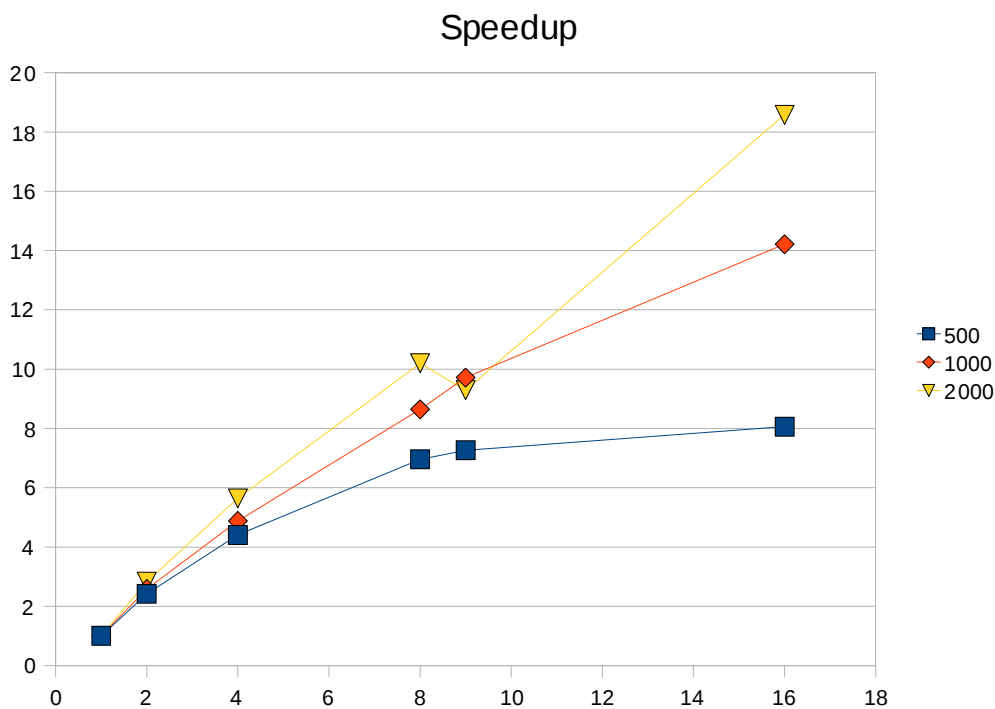
	2P	4P	8P	16P
Comm Time	419890	806244	1.61E+006	3.24E+006
Time/Comm	220	202	202	202

Tider i mikrosekunder.

Omregnet til speedup fås, Tabel 3.3:

Procs	500	1000	2000
1	1	1	1
2	2.42	2.59	2.83
4	4.41	4.88	5.64
8	6.96	8.64	10.2
9	7.26	9.72	9.29
16	8.06	14.21	18.58

og grafen:



Den superlineære speedup må skyldes cache-effekter (Cell/BE har ret lille cache størrelse). Som forventet giver større problemsizes bedre speedup – sålænge en process calc-tid er lang nok til at skjule kommunikationen.

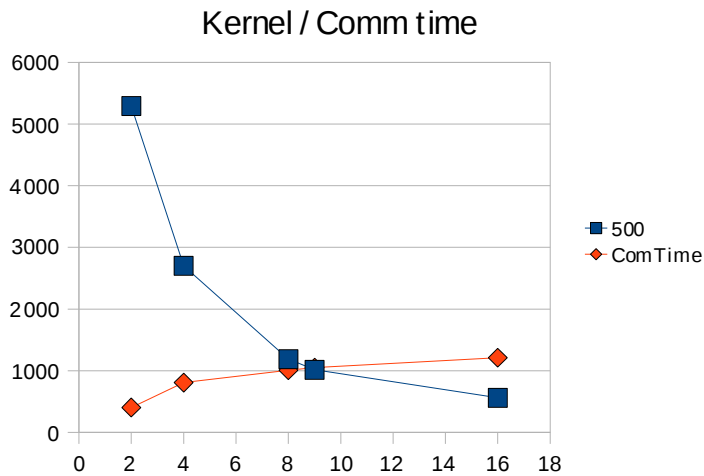
Hvis man kigger på de forskellige antal processer-konstellationer og antal af nabo-besked:

2: 1,1 + reduce ~ 2
 4: 2,2 + reduce ~ 4
 2,2
 8: 2,3,3,2 + reduce ~ 5
 2,3,3,2
 9: 2,3,2 + reduce ~ 5.2
 3,4,3
 2,3,2
 16: 2.3.3.2 + reduce ~ 6
 3,4,4,3
 3,4,4,3
 2,3,3,2

Tabel 3.4, tid brugt per udregning og tid brugt på “gæstimerede” nabo-besked (i mikrosek)

Procs	500	1000	2000	#Comm	ComTime
2	5293.06	10800.42	84115.71	2	404
4	2700.54	6631.84	42184.72	4	808
8	1188.24	5332.54	21441.26	5	1010
9	1015.4	4709.96	24486.17	5.2	1050.4
16	561.71	2571.53	10720.63	6	1212

Tid per udregning er den målte tid for alle udregninger delt med antal iterationer (500: 46301, 1000: 22696 og 2000: 15764)



Vi ser altså at ved 500x500 problemstørrelsen overskrider kommunikationstiden beregningstiden.

Men vi ser også ud fra tabel 3.1 at der stadig ved selv store problemstørrelser er forskel på beregningstid og totaltid – som er overhead i MPI api-kald, spildtid pga memorykopieringer (nødvendige pga non-blocking sends, unødvendige pga interne kopier i MPI og i Linux netværksstak) plus tids-instrumenteringen.

Amdahl:

Det bedste ønskescenarie er at fjerne netværkstiden. Hvis den var minimal (0) var der ingen grund til at lave latency-hiding, og dermed ingen grund til at bruge non-blocking kald og dermed igen ingen grund til unødige memorykopieringer. Det vil betyde at køretiderne bringes ned på calc-tiderne. Desuden vil det også tillade scaling til endnu flere noder (indtil der kun er få celler til hver processor).

Ikke afprøvede tiltag:

Jeg har ikke eksperimenteret med MPI-datatype, som fx kunne bruges til at “plukke” Røde/Sorte celler uden datakopi (i min kode – men jo nok i MPI-api, og så er man jo lige vidt).

Jeg har ikke eksperimenteret med manuel pack/unpack – men jeg kan ikke se nogen grund til at det skulle kunne vinde noget.

Til sidst har jeg heller ikke kigget på Buffered-Send/Recv og alternative modes. Buffering ville kunne simplificere koden en hel del – men nok ikke forbedre køretiden....

eAssignment-2

Udviklingsforløb:

Første trin var at modificere den udleverede c-kode til at modtage flere kommandolinie argumenter. Jeg “opfandt” et format for at overlevere en foldning – Up,Right,Down,Left... (á la turtlegraphics med GPS ;)). Med denne parameter kunne Prototein.c direkte bruges til at bearbejde delvist foldede pro(to)teiner – bedste del-/slut-foldning rapporteredes selv. i samme format.

Herefter var det en enkel sag at oversætte kommandoline kald til PVM. PVM er en “wrapper” om kommandolinien – så kald til PVM kunne overføres til “main(ac,av)” helt enkelt.

Næste beslutning var at vælge hvilken metode til fordele opgaverne – den simpleste (og den anbefalede) var at uddele sub-opgaver på en given dybde i søgetræet. (se nedenfor mht argumenter for ikke at være “smartere”)

Som nedenstående målinger viser, følte jeg ingen anledning til at forfølge ydeligere optimeringer...

Målinger:

Der er (mindst) 2 variable som kan undersøges – antal slaver og dybde. Jeg har lavet “indledende” undersøgelse af udfaldsrummet – hvilket har resulteret i 2 ortogonale måleserier (med globalt bedste punkt som midten): forskelligt antal slaver med 10 i dybden, forskellige dybder med 32 slaver...

Tabel 2.1: Køretid i millisec vs Antal “slaver” - og speedup ved dybde 10

Slaves	Time	Speedup
0	282633	1
1	285077	0.99
2	142590	1.98
4	71341	3.96
8	35725	7.91
16	17930	15.76
32	17006	16.62
64	21533	13.13

Tabel 2.2: Diverse data vs Dybde 32Slaver

Depth	Time	Speedup	Slaves	Avg-Sub-Time	Avg-Com-time	Max-Com-time	No-Solution
3			2				
4			5				
5			14				
6	21463	13.17	40	1.17E+007	164939	1188313	0
7	18050	15.66	110	4.45E+006	233104	1135171	0
8	18428	15.34	306	1.76E+006	204439	1767416	1
9	16681	16.94	834	625062	38790	562220	3
10	17006	16.62	2292	233861	27689	230111	17
11	19253	14.68	6213	98422	24258	126139	51
12	20221	13.98	16941	38123	11366	77624	197
13	23403	12.08	45755	16335	6945	32381	562
14	30166	9.37	124102	7783	4863	22095	1884
15	57818	4.89	334242	5517	4587	78982	5264

Tabel 2.2 viser målinger ved forskellige dybder i søgetræet. Kolonnerne er:

Depth – dybde i søgetræet hvor uddeling til slaver foregår

Time – køretid i millisek

Speedup – Speedup ift sekventiel version

Slaves – antal sub-problemer uddelt til slaver

Avg-Sub-Time – gennemsnitlig køretid for et sub-problem i microsec

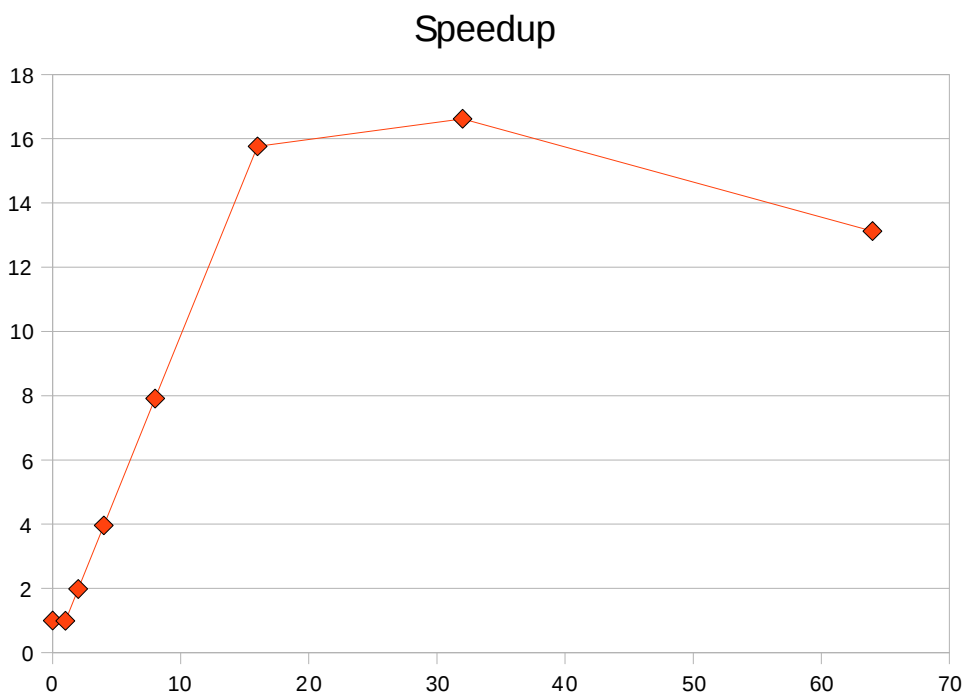
Avg-Com-Time – gennemsnitlig tid brugt på kommunikation master-slave i microsec

Max-Com-Time – største tid brugt på kommunikation master-slave i microsec

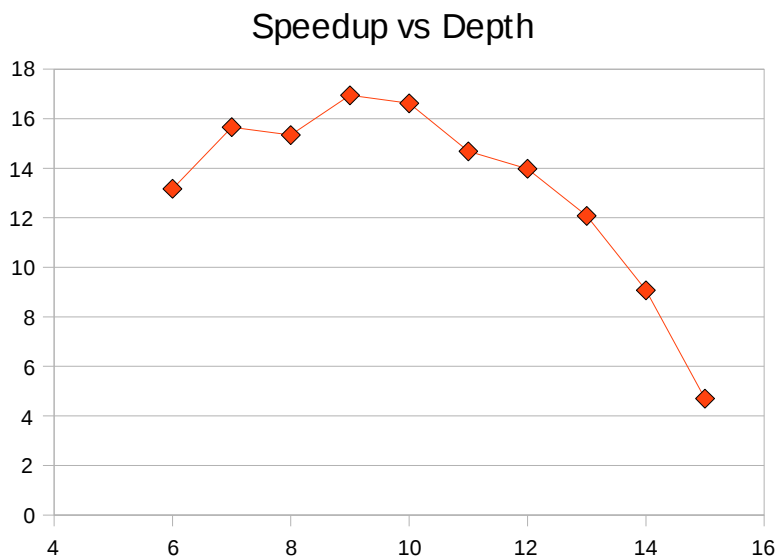
No-Solution – antal subproblemer som ikke har nogen løsning (“spærret inde”)

De 2 kommunikationstal er målt som forskellen i tiden slaven brugte på subproblemet og hvor lang tid masteren ventede på svar fra slaven. Da svaret “pvm_probes” er tallet en pessimistisk upper bound (specielt ved de små dybder) – Min_Com_time er 276us i alle forsøg – så den er ikke medtaget i tabellen.

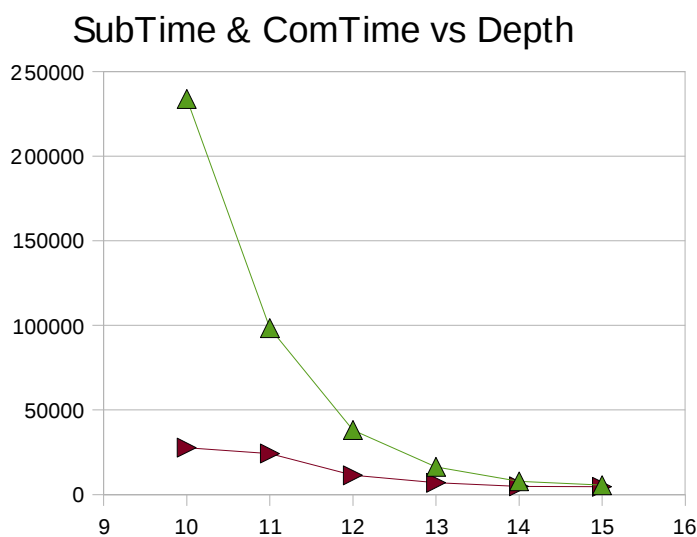
- med graferne:



Her ses et smukt lineært speedup hele vejen til 16 PS3'er og endda en smule glæde af "power-SMT" så vi får over-16 nodes-speedup ved 32 slaver... (muligvis er det ikke power-SMTs skyld, men derimod overlap på samme PS3 af idletid i den ene process med beregning i den anden)



Og på denne graf ses hvilken depth der skal vælges (ved 32/16-nodes) – I virkeligheden er det ikke dybden der er interessant, men derimod "halen" altså hvor stort subproblemet er. En hale på 12-13 stykker ser passende ud. Når halen bliver for kort bliver betydningen af kommunikationen vigtig, og når dybden bliver for lille bliver det mindre antal af subproblemer for lille til at loaden kan fordeles tilstrækkeligt.



Denne graf viser at ved en hale på mindre end 12 bliver kommunikationen sammenlignelig med beregningstiden.

Amdahl:

Hvad er smart at forbedre her:

Ram er ligegyldig, hele problemet kan snildt være i L1-cache (selv ved meget længere prototeiner).

CPU er mindre vigtig, ved en dybde på 10 og op, er der mange tusinder subproblemer, så der kan skaleres til tusinder af slaver

Netværk er interessant. Der er ingen mulighed for at lade slaven udføre beregninger i perioden "færdig med subproblem – få ny opgave" (se dog nedenfor). Da perioden måles i mindst 300us per subproblem er der en del idle slave-tid.

Ikke afprøvede tiltag:

Jeg har ikke leget med nogle af de sædvanlige "tricks" i backtracking: Lookahead (check for "spærret inde"), Alpha-Beta pruning (undersøg ikke subtræer som ikke kan opnå en bedre løsning end hidtil set), Hurtig scanning af potentielt gode løsninger (f.x. Lade prototeinet "glide" igennem en fladedækkende graf (Peano, Hilbert eller lign.)), Foldning fra midten i stedet for fra een ende af, ...

Disse optimeringer ville gøre koden hurtigere og øge spredning af køretid på subproblemer. Dette ville måske kunne have afsløret flaskehalse i løsningen...

Jeg har heller ikke kigget på måder at få bedre loadbalancering – f.x. ved at lade subproblemer spawn sub-subproblemer i stedet for at regne i bund.

Overlappende beregninger kunne ret nemt opnås ved altid at udlevere 2 eller flere opgaver til slaven (pvm_send flere gange til den samme slave – pvm_send er buffret og non-blocking). Minus er at det vil betyde at loadbalancering kan blive sværere, men så længe der er mange (>>) flere subproblemer end slaver burde det ikke være noget problem.

Efter redaktionens ophør, har jeg lige nået at implementere dette overlap og lige lave en kort test – uden forbedringer i resultatet:

```
PVM-Best is: -s10 -N8 -q2 -1 URDDL... num_slave_calls=2292  
totmilli=35731 milli=35719  
PVM-Best is: -s10 -N16 -q2 -1 URDDL... num_slave_calls=2292  
totmilli=17932 milli=17920  
PVM-Best is: -s10 -N32 -q2 -1 URDDL... num_slave_calls=2292  
totmilli=17579 milli=17564
```

eAssignment5

Udviklingsforløb:

Den direkte kørbare nbody.py er blevet tilrettet – mest med fjernelse af graphics og psycho, så den kan køre på de ikke-grafiske PS3.

Derefter har jeg tilføjet kommandolinie argumenter til at sætte antal processer (-N), antal bodies (-M) og antal iterationer (-I). Disse kommando linie parametre angives kun til “master-processen” (som laver udprint/grafisk visning). Alle slave processer startes uden parametre (og de får deres parametre fra Linda.universe).

Jeg har lavet det således at hver process har sit eget tuple space den skriver til og naboprocesen læser fra – sat op i en ring (ideen er at skrivning så er hurtig – så der efterfølgende kan laves overlappende udregninger – men dette er kun en forhåbning, der er ingen docs af pyLinda som antyder for eller imod blocking/buffered/non-blocking _out eller _in):

Flowet er:

```
Loop:
    skriv mine bodies til eget tuplespace
    udregn forces fra egen bodies
    for alle andre processer
        læs fra forrige process' tuplespace
        skriv fremmede bodies til eget tuplespace
        udregn forces fra fremmede bodies
    flyt bodies
```

Dvs at kommunikation overlappes med beregninger og at kommunikationen er “jævnt” fordelt henover loopet.

For at kunne skrive/læse bodies i et tuplespace kræver det at tuplespacet genkender typen. PyLinda er ret begrænset i hvilke typer man kan bruge, derfor konverterer jeg listen af bodies til en tekststreng. Dette gøre med python's marshal-modul. Marshall er også lidt begrænset, derfor må myBody object-listen først omformes til en simple tuple-liste.

```
def pck(d):
    p = []
    for b in d:
        p.append((b.iN, b.fMass, b.fXpos, b.fYpos, b.fXvel, b.fYvel))
    m = marshal.dumps(p)
    return m

def upck(ltp):
    d = []
    for p in marshal.loads(ltp):
        d.append(cBody(p[0], p[1], p[2], p[3], p[4], p[5]))
    return d
```

For at selecte de relevante tupler fra tuplespace bruges “filtrering”. Det viser sig at pyLinda arbejder bedst med filtrer baseret på integer-tuple-medlemmer, derfor angives alle data i tuplespacet med tal – lidt kryptisk, ift at bruge “pæne” tekststreng – men meget bedre (det virker).

Der har ikke været mange “optimerings” runder udover lidt målinger af f.x. hastighed ved

forskellige tuple-størrelser – man kunne nøjes med at overføre fremmedes bodies Mass og Pos men det viste sig af være nærmest uden betydning (kan også verificeret med speed_server/client installeret sammen med pyLinda)

Målinger:

Jeg har målt med flere forskellige problem-størrelser (målt i antal bodies). Alle er lavet med kun 10 iterationer, idet køretiden ændrer sig helt lineært med antallet af iterationer (også verificeret med et par målinger – som dog ikke er medtaget her).

Som et eksperiment har jeg også målt lidt på en version, hvor der kun anvendes tuplespacet linda.universe – dvs at een enkelt node er flaskehalsen for al kommunikation.

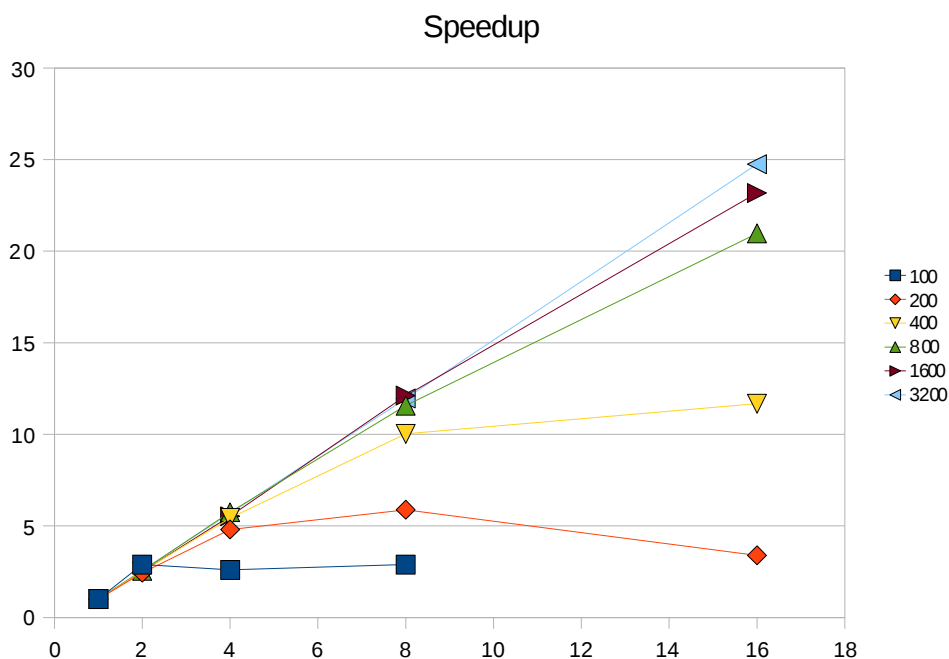
Køretider for forskellige problemstørrelser og antal processer, alle tider i sek. Tabel 5.1:

Size	1P	2P	4P	8P	16P	8P-universe	16P-universe
100		3.38	1.17	1.3			
200	13.88		5.68	2.89	2.36	4.09	3.56
400	54.74		21.81	10.08	5.46	4.69	6.58
800	223.37		87.53	38.86	19.31	10.65	20.48
1600	886.74			160.3	73.18	38.26	73.72
3200	3544.88				296.6	143.23	

Dette kan omskrives til en speedup tabel, Tabel 5.1:

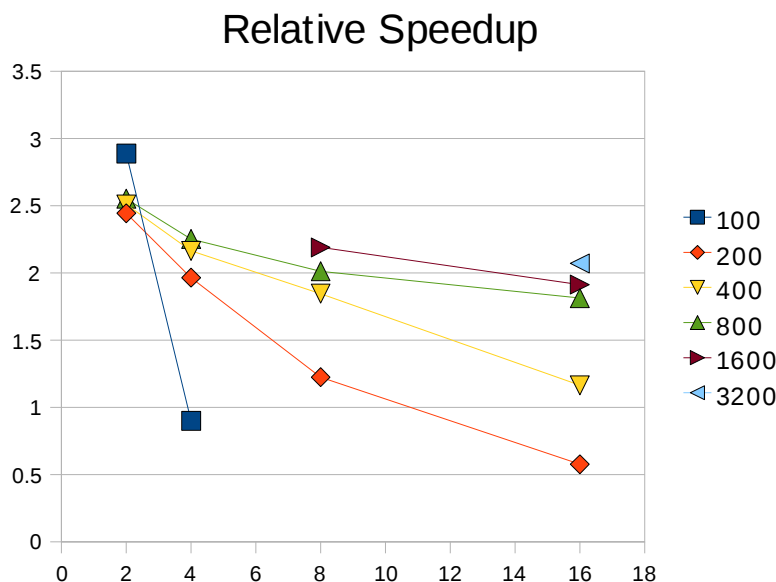
	100	200	400	800	1600	3200
1	1	1	1	1	1	1
2	2.89	2.44	2.51	2.55		
4	2.6	4.8	5.43	5.75	5.53	
8	2.89	5.88	10.03	11.57	12.12	11.95
16		3.39	11.67	20.97	23.18	24.75

med grafen:



Igen ses der en superlineær skalering ved de større problemer – og igen må det skyldes ram/cache. Python har et ret voldsomt memoryfootprint. (selvom 1P skaleringen i size følger meget fint 4-dobling af tid ved fordobling af størrelse ? Tabel 5.1 – se i øvrigt nedenfor under “Ikke afprøvede tiltag”).

Ved at betragte det relative speedup ved en fordobling af antal processer ser det lidt mere “normalt” ud.



Amdahl:

Ifølge de ekstra målinger med brug af linda.universe, kan man (måske ;) udlede at en flaskehals er tuplespace-serveren. Nu større tupler, nu længere tid... og da tuplerne er ret store (10-30k) ift ethernet pakkestørrelsen er det ikke ethernet latency der giver problemer – og ethernet båndbredden giver ikke begrænsningen her. Man kunne gætte på at det er pythons memoryhåndtering som skaber problemet...

En simplificeret formel for køretiden kunne være:

$$t = I \left(a \frac{M^2}{N} + bN \right) \Rightarrow \text{breakeven: } N = \sqrt{\frac{a}{b}} M$$

”t” er køretiden, I er antal iterationer, M er antal bodies, N er antal processer, ”a” regnetid per body per iteration og ”b” er kommunikationstiden (som består af netværkstid og tuplespace håndtering). Lidt høker-curvefitting viser at a=22us og b=16.5ms (millisec!) i de målte data ~ M skal være 27 gange større end N.

Hvis memoryflaskehalsen denne fjernes helt (altså at tuplespace operationer går uendeligt hurtigt) og vi sætter netværkstiden til (~250us latens + 250us transport (32Kbytes v/1Gbit) b=500us), kan vi nå til at M kun skal være 4 gange større end N.

Ikke afprøvede tiltag – og dog:

Under min famlen efter en rimelig forklaring på det store speedup hop fra sekventiel til bare 2 processer, er jeg faldet over denne python detaille:

Dobbelt forløkkerne á la:

```
for b in mydata:
    for r in mydata:
        if b != r: calculate(b, r)
```

er usædvanligt dyre!

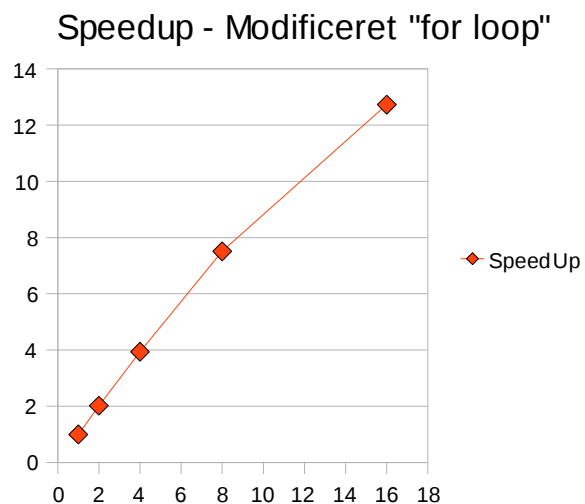
De kan omskrives en anelse:

```
ll = len(mydata)
for b in xrange(ll)
    for r in xrange(ll)
        if b != r: calculate(mydata[b], mydata[r])
```

Denne struktur giver en faktor 0.6 på køretiden – og mere spændende: fjerner det store spring!

Jeg nåede lige at lave en enkelt måleserie med det modificerede program:

	800 SpeedUp	
1	141.14	1
2	69.83	2.02
4	35.81	3.94
8	18.79	7.51
16	11.09	12.73



Jeg er klar over at det som der gøres er at tage loopinvariant data ud af løkken (men der er ingen kollisioner af bodies i nogle af mine kørsler af den udleverede kode (sammenlign aldrig to floats for identity ! ;) , så mydata's length ændres ikke undervejs) - men det synes dog ikke at være hele forklaringen – men Monty Python var jo også lidt uforklarlige ind i mellem---

Appendix A – MPI

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

int micro_calls = 0;
double microsec_db()
{
    struct timeval t;
    ++micro_calls;
    gettimeofday(&t, 0);
    return t.tv_sec * 1000000.0 + t.tv_usec;
}

#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))

int splitProcs(int sx, int sy, int np, int * pinx, int * piny)
{
    int x, y, c = -1;
    for (x = 1; x <= np; ++x)
        for (y = 1; y <= np; ++y)
            if (x * y <= np)
                {
                    // int px = (sx + x - 1) / x;
                    // int py = (sy + y - 1) / y;
                    int cv = x * y + min(x,y);
                    if (cv > c)
                        {
                            *pinx = x;
                            *piny = y;
                            c = cv;
                        }
                }
    if ((sx != sy) && ((sx > sy) ^ (*pinx > *piny))) { int m = *pinx; *pinx = *piny; *piny = m; }
    return *pinx * *piny == np;
}

#define XY2I(x,y) (((x)+1)*(dy+2)+((y)+1))
#define CALC { ov = mat[XY2I(x,y)]; mat[XY2I(x,y)] = v = 0.2 * (mat[XY2I(x,y)] + mat[XY2I(x,(y-1))]  
+ mat[XY2I(x,(y+1))]) + mat[XY2I((x-1),y)]+mat[XY2I((x+1),y)]); delta += fabs(v - ov); }
#define isCOLOR(c) (((2 + x + xl + y + yl) & 1) ^ (c))

struct neibor {
    int n;
    double * obor;
    // MPI_Request oreq;
    // int oreqcmp;
    double * ibor;
    // MPI_Request ireq;
    // int ireqcmp;
};
typedef struct neibor neibor;

struct reduce {
    double odelt;
    // MPI_Request oreq;
    // int oreqcmp;
    double idelt;
    // MPI_Request ireq;
    // int ireqcmp;
};
typedef struct reduce reduce;

int main(int ac, char *av[])
{
    int numprocs, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;
```

```

MPI_Init(&ac, &av);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Get_processor_name(processor_name, &namelen);

char buf[8192];
int a;
*buf = 0;
for (a = 0; a < ac; ++a)
    sprintf(buf + strlen(buf), "%s%s", a>0? " ":"", av[a]);

int SIZE_X = (ac > 1 ? atoi(av[1]) : 500) - 2;
int SIZE_Y = (ac > 2 ? atoi(av[2]) : 500) - 2;
int check = ac > 3 ? atoi(av[3]) : 1;
int verbose = ac > 4 ? atoi(av[4]) : 0;
int explReduc = check < 0;
if (explReduc) check = -check;
if (check < 1) check = 1;

double epsilon = (SIZE_X + 2) * (SIZE_Y + 2) * 0.001;

int pinx, piny;
splitProcs(SIZE_X, SIZE_Y, numprocs, &pinx, &piny);
int xs = (rank % pinx);
int ys = rank / pinx;
int xl = SIZE_X * xs / pinx;
int yl = SIZE_Y * ys / piny;
int xh = SIZE_X * (xs + 1) / pinx; if (xh > SIZE_X) xh = SIZE_X;
int yh = SIZE_Y * (ys + 1) / piny; if (yh > SIZE_Y) yh = SIZE_Y;
neighbor nei[4];
nei[0].n = xl > 0 ? rank - 1 : -1;
nei[1].n = xh < SIZE_X ? rank + 1 : -1;
nei[2].n = yl > 0 ? rank - pinx : -1;
nei[3].n = yh < SIZE_Y ? rank + pinx : -1;

printf("Process %d on %s out of %d: '%s' %d,%d [%d,%d-%d,%d] me=%d,%d neighbours[%d,%d,%d,%d]\n",
rank, processor_name, numprocs, buf, pinx, piny, xl, yl, xh, yh, xs, ys, nei[0].n, nei[2].n,
nei[1].n, nei[3].n);

int dx = xh - xl;
int dy = yh - yl;
double *mat = (double*)malloc((dx+2) * (dy+2) * sizeof(double));
int n;
for (n = 0; n < 4; ++n)
{
    nei[n].obor = (double*)malloc(max(dx,dy) * sizeof(double));
    nei[n].ibor = (double*)malloc(max(dx,dy) * sizeof(double));
    // nei[n].oreqcmp = nei[n].ireqcmp = 1;
}
int x, y;
for (x = -1; x <= dx; ++x)
    for (y = -1; y <= dy; ++y)
        if (y == -1 && nei[2].n == -1) mat[XY2I(x,y)] = 40.0;
        else if ((x == -1 && nei[0].n == -1) ||
                (x == dx && nei[1].n == -1) ||
                (y == dy && nei[3].n == -1)) mat[XY2I(x,y)] = -273.15;
        else mat[XY2I(x,y)] = 0.0;

reduce * rdc = (reduce*)malloc(numprocs * sizeof(reduce));
for (n = 0; n < numprocs; ++n)
{
    rdc[n].odelt = rdc[n].idelt = 0.0;
    // rdc[n].oreqcmp = rdc[n].ireqcmp = n != rank;
}

// Pack Requests into arrays to be used with TestAny
int numnei = 0;
int nei2n[4];
int n2nei[4];
for (n = 0; n < 4; ++n)
    if (nei[n].n != -1)
    {
        nei2n[n] = numnei;
        n2nei[numnei] = n;
    }

```

```

    ++numnei;
}
int * rdc2n = (int*)malloc(numprocs * sizeof(int));
int * n2rdc = (int*)malloc((numprocs - 1) * sizeof(int));
for (n = 0; n < numprocs; ++n)
{
    rdc2n[n] = n < rank ? n : (n > 0 ? n - 1 : 0);
    n2rdc[rdc2n[n]] = n;
}
MPI_Status * stati = (MPI_Status*)malloc(max(numprocs - 1, numnei) * sizeof(MPI_Status));
MPI_Request * nei_oreq = (MPI_Request*)malloc(numnei * sizeof(MPI_Request));
MPI_Request * nei_ireq = (MPI_Request*)malloc(numnei * sizeof(MPI_Request));
MPI_Request * rdc_oreq = (MPI_Request*)malloc((numprocs - 1) * sizeof(MPI_Request));
MPI_Request * rdc_ireq = (MPI_Request*)malloc((numprocs - 1) * sizeof(MPI_Request));

int flag;
int index;
int allfinished_o = 1;
int allfinished_i = 1;
int allfinished_r = 1;

double t1, t0;
double t_send = 0.0;
double t_rcv = 0.0;
double t_center = 0.0;
double t_border = 0.0;
double t_reduce = 0.0;
double t_test = 0.0;
double t_wait = 0.0;
int num_reduce = 0;
int num_com = 0;
int num_test_o = 0;
int num_wait_o = 0;
int num_test_i = 0;
int num_wait_i = 0;
int explReducCheck = 0;
double delta;
int maxite = 1000000;
int ite = 0;
if (numprocs > 1)
{
    if (verbose & 8)
    {
        dx = 100;
        MPI_Send_init(nei[0].obor, dx, MPI_DOUBLE, (rank + 1) % numprocs, 1, MPI_COMM_WORLD,
&nei_oreq[0]);
        MPI_Recv_init(nei[0].ibor, dx, MPI_DOUBLE, (numprocs + rank - 1) % numprocs, 1,
MPI_COMM_WORLD, &nei_ireq[0]);
        if (!rank)
        {
            t0 = microsec_db();
            while (ite++ < 1000)
            {
                MPI_Start(&nei_oreq[0]);
                MPI_Wait(&nei_oreq[0], &status);
                MPI_Start(&nei_ireq[0]);
                MPI_Wait(&nei_ireq[0], &status);
            }
            printf("Circle time=%g ringlen=%d messlen=%d\n", microsec_db() - t0, numprocs, dx);
        }
        else
        {
            while (ite++ < 1000)
            {
                MPI_Start(&nei_ireq[0]);
                MPI_Wait(&nei_ireq[0], &status);
                MPI_Start(&nei_oreq[0]);
                MPI_Wait(&nei_oreq[0], &status);
            }
        }
        MPI_Finalize();
        return 0;
    }
}

for (n = 0; n < 4; ++n)

```

```

    if (nei[n].n != -1)
    {
        int d = n < 2 ? dy : dx;
        MPI_Send_init(nei[n].obor, d/2+1, MPI_DOUBLE, nei[n].n, 1, MPI_COMM_WORLD,
&nei_oreq[nei2n[n]]);
        MPI_Recv_init(nei[n].ibor, d/2+1, MPI_DOUBLE, nei[n].n, 1, MPI_COMM_WORLD,
&nei_ireq[nei2n[n]]);
    }
    if (explReduc)
        for (n = 0; n < numprocs; ++n)
            if (n != rank)
            {
                MPI_Send_init(&rdc[n].odelt, 1, MPI_DOUBLE, n, 2, MPI_COMM_WORLD, &rdc_oreq[rdc2n[n]]);
                MPI_Recv_init(&rdc[n].idel, 1, MPI_DOUBLE, n, 2, MPI_COMM_WORLD, &rdc_ireq[rdc2n[n]]);
            }

t0 = microsec_db();
while (ite++ < maxite)
{
    double v, ov;
    delta = 0.0;
    int color;
    for (color = 0; color < 2; ++color)
    {
        double t3 = microsec_db();
        for (n = 0; n < 4; ++n)
            if (nei[n].n != -1)
            {
                if (explReducCheck && !allfinished_r)
                {
                    MPI_Testall(numprocs - 1, rdc_oreq, &flag, stati);
                    MPI_Testall(numprocs - 1, rdc_ireq, &allfinished_r, stati);
                }

                int x = 0, y = 0, d;
                if (!allfinished_o) { MPI_Wait(&nei_oreq[nei2n[n]], &status); ++num_wait_o; }
                switch (n)
                {
                    case 1: x = dx - 1; // NB: no break!
                    case 0:
                        for (y = 0; y < dy; ++y)
                            if (!isCOLOR(color))
                                nei[n].obor[y/2] = mat[XY2I(x,y)];
                        d = dy;
                        break;
                    case 3: y = dy - 1;
                    case 2:
                        for (x = 0; x < dx; ++x)
                            if (!isCOLOR(color))
                                nei[n].obor[x/2] = mat[XY2I(x,y)];
                        d = dx;
                        break;
                }
                MPI_Start(&nei_oreq[nei2n[n]]);
                MPI_Start(&nei_ireq[nei2n[n]]);
                ++num_com;
            }
        t_send += microsec_db() - t3;

        if (explReduc && explReducCheck)
        {
            t1 = microsec_db();
            delta = rdc[rank].idel;
            explReducCheck = 0;
            for (n = 0; n < numprocs; ++n)
                if (n != rank)
                {
                    if (!allfinished_r) MPI_Wait(&rdc_ireq[rdc2n[n]], &status);
                    delta += rdc[n].idel;
                }
            t_reduce += microsec_db() - t1;
            ++num_reduce;
            if (verbose & 2)
                printf("%s, line %d: process=%d delta=%g ite=%d epsilon=%g:\n", __FILE__, __LINE__,
rank, delta, ite, epsilon);

```

```

    if (delta < epsilon) goto END;
    delta = 0;
}

t1 = microsec_db();
allfinished_o = 0;
allfinished_i = 0;
for (x = 1; x < dx - 1; ++x)
{
    // if (!allfinished_i && (x == 1 || x == dx/2 || !(x & 0xf)))
    if (!allfinished_i && (x == 1 || x == dx/2 || x == dx/4 || x == 3*dx/4))
    {
        double t2 = microsec_db();
        if (!allfinished_o)
        {
            ++num_test_o;
            MPI_Testall(numnei, nei_oreq, &allfinished_o, stati);
        }
        if (1 || allfinished_o)
        {
            ++num_test_i;
            MPI_Testall(numnei, nei_ireq, &allfinished_i, stati);
        }
        t_test += microsec_db() - t2;
    }
    for (y = 1; y < dy - 1; ++y)
        if (isCOLOR(color))
            CALC;
}
t_center += microsec_db() - t1;

t3 = microsec_db();
int nn;
for (nn = 0; nn < numnei; ++nn)
{
    if (!allfinished_i) { t1 = microsec_db(); MPI_Waitany(numnei, nei_ireq, &index, &status);
    ++num_wait_i; t_wait += microsec_db() - t1; }
    else index = nn;
    // printf("%d.Waitany(%d, ..., %d, %d)\n", rank, nn, index, status);
    if (index != MPI_UNDEFINED)
    {
        n = n2nei[index];
        int x = -1, y = -1;
        // if (!nei[n].ireqcmp) { t1 = microsec_db(); MPI_Wait(&nei_ireq[nei2n[n]], &status); +
        ++num_wait_i; t_wait += microsec_db() - t1; }
        switch (n)
        {
            case 1: x = dx;
            case 0:
                for (y = 0; y < dy; ++y)
                    if (!isCOLOR(color))
                        mat[XY2I(x,y)] = nei[n].ibor[y/2];
                break;
            case 3: y = dy;
            case 2:
                for (x = 0; x < dx; ++x)
                    if (!isCOLOR(color))
                        mat[XY2I(x,y)] = nei[n].ibor[x/2];
                break;
        }
    }
}
t_recv += microsec_db() - t3;

t1 = microsec_db();
for (x = 0; x < dx; x += dx - 1)
    for (y = 0; y < dy; ++y)
        if (isCOLOR(color))
            CALC;
for (x = 1; x < dx - 1; ++x)
    for (y = 0; y < dy; y += dy - 1)
        if (isCOLOR(color))
            CALC;
t_border += microsec_db() - t1;
}

```

```

    if (!(ite % check))
    {
        if (explReduc)
        {
            explReducCheck = 1;
            rdc[rank].idelt = rdc[rank].odelt = delta;
            for (n = 0; n < numprocs; ++n)
                if (n != rank)
                {
                    rdc[n].odelt = delta;
                    MPI_Start(&rdc_oreq[rdc2n[n]]);
                    MPI_Start(&rdc_ireq[rdc2n[n]]);
                }
            allfinished_r = 0;
        }
        else
        {
            double nd;
            t1 = microsec_db();
            MPI_Allreduce(&delta, &nd, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
            t_reduce += microsec_db() - t1;
            ++num_reduce;
            delta = nd;
            if (verbose & 2)
                printf("%s, line %d: process=%d delta=%g ite=%d epsilon=%g:\n", __FILE__, __LINE__,
rank, delta, ite, epsilon);
            if (delta < epsilon) break;
        }
    }
}
}
else
{
    t0 = microsec_db();
    while (ite++ < maxite)
    {
        double v, ov;
        delta = 0.0;
        int color;
        for (color = 0; color < 2; ++color)
        {
            for (x = 0; x < dx; ++x)
                for (y = 0; y < dy; ++y)
                    if (isCOLOR(color))
                        CALC;
        }

        if (!(ite % check))
        {
            if (verbose & 2)
                printf("process=%d delta=%g ite=%d epsilon=%g:\n", rank, delta, ite, epsilon);
            if (delta < epsilon) break;
        }
    }
}
END:
t0 = microsec_db() - t0;
printf("process=%d delta=%g ite=%d epsilon=%g time=%g calc=%g reduce=%g/%d s/r=%g/%g rest=%g/%d te
st=%g/%d,%d wait=%g/%d,%d mc=%d\n", rank, delta, ite, epsilon, t0, t_center + t_border - t_test,
t_reduce, num_reduce, t_send, t_recv, t0 - t_center - t_border - t_reduce - t_send - t_recv,
num_com, t_test, num_test_o, num_test_i, t_wait, num_wait_o, num_wait_i, micro_calls);

if (verbose & 1)
{
    // sleep(1 + (numprocs - rank) * 3);
    printf("DUMPSTART: process=%d\n", rank);
    for (y = (nei[3].n != -1 ? dy - 1 : dy); y >= (nei[2].n != -1 ? 0 : -1); --y)
        for (x = (nei[0].n != -1 ? 0 : -1); x <= (nei[1].n != -1 ? dx - 1 : dx); ++x)
            printf("%03d, %03d: %3.3f %d\n", x1 + x + 1, y1 + y + 1, mat[XY2I(x,y)], isCOLOR(0));
    printf("DUMPEND: process=%d\n", rank);
}

MPI_Finalize();
return 0;

```

Appendix B – PVM

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pvm3.h"
/*
#include "StopWatch.h"
#include "graphicsScreen.h"
*/
#include <malloc.h>
#include <sys/time.h>

int verbose = 0;

double microsec_db()
{
    struct timeval t;
    gettimeofday(&t, 0);
    return t.tv_sec * 1000000.0 + t.tv_usec;
}

struct coordinate
{
    int x;
    int y;
};

int bestscore;
int winner_length;
struct coordinate* winner = 0;
const char* prototein = "HHHHPPPPPPPPPPPPHHHH";
int prototein_length;
int best_possible_score;
// Lob additions...
char *foldbuf = 0;
int spawndepth = 0;
const char * partialsolution = 0;
int num_slaves = 0;
int slave_ids[4096];
int num_slave_free = 0;
int slave_free[4096];
double slave_start[4096];
int num_slave_calls = 0;
int queue = 1;

#define MAP_WIDTH (prototein_length * 2 + 1)
#define MAP_HEIGHT (prototein_length * 2 + 1)
#define MAP_SIZE (MAP_WIDTH * MAP_HEIGHT)
#define MAP_COORDINATES(x, y) (((y)*MAP_WIDTH)+(x))

void fold(char* map, struct coordinate* place, int places_length);
void spawnJob(struct coordinate* place, int places_length);
void readResult();

inline void recurse(char* map, int newx, int newy, struct coordinate* place, int places_length)
{
    if (map[MAP_COORDINATES(newx, newy)]==' ')
    {
        map[MAP_COORDINATES(newx, newy)] = prototein[places_length];
        place[places_length].x = newx;
        place[places_length].y = newy;
        fold(map, place, places_length + 1);
        map[MAP_COORDINATES(newx, newy)] = ' ';
    }
}
/*
void drawmap(struct coordinate* place, int places_length)
{
    int i, x, y, color, lastx, lasty;

    gs_clear(WHITE);
    for(i = 0; i < places_length; i++)
```

```

    {
        x = place[i].x * 10;
        y = place[i].y * 10;
        color = prototein[i] == 'H' ? BLUE : RED;
        if (i != 0)
            gs_line(x,y, lastx, lasty, BLACK);
        gs_dot(x, y, 10, color);
        lastx = x;
        lasty = y;
    }
    gs_update();
}
*/

void mkFoldbuf(struct coordinate* place, int places_length)
{
    int i;
    for (i = 1; i < places_length; ++i)
        foldbuf[i-1] = place[i-1].x > place[i].x ? 'L' :
            place[i-1].x < place[i].x ? 'R' :
            place[i-1].y > place[i].y ? 'D' :
            place[i-1].y < place[i].y ? 'U' :
            '?';
    foldbuf[i-1] = 0;
}

int readFoldbuf(const char * buf, struct coordinate* place, char* map)
{
    int i = 1;
    while (*buf)
    {
        place[i].x = place[i-1].x; place[i].y = place[i-1].y;
        switch (*(buf++))
        {
            case 'L': --place[i].x; break;
            case 'R': ++place[i].x; break;
            case 'D': --place[i].y; break;
            case 'U': ++place[i].y; break;
            default: fprintf(stderr, "WHAT??\n"); break;
        }
        if (map) map[MAP_COORDINATES(place[i].x, place[i].y)] = prototein[i];
        ++i;
    }
    return i;
}

void calc_score(struct coordinate* place, char* map, int places_length)
{
    int score, i, x, y;
    score = 0;

    for(i = 0; i < prototein_length; i++)
    {
        x = place[i].x;
        y = place[i].y;

        if (map[MAP_COORDINATES(x, y)]=='H')
        {
            if (map[MAP_COORDINATES(x-1, y)]==' ') score--;
            if (map[MAP_COORDINATES(x+1, y)]==' ') score--;
            if (map[MAP_COORDINATES(x, y-1)]==' ') score--;
            if (map[MAP_COORDINATES(x, y+1)]==' ') score--;
        }
    }

    if(score>bestscore)
    {
        bestscore=score;
        winner_length = places_length;
        for(i = 0; i<places_length; i++)
            winner[i] = place[i];
        // printf("Bestscore is now: %i\n", bestscore);
        mkFoldbuf(place, places_length);
        // printf("foldbuf='%s'\n", foldbuf);
#ifdef GRAPHICS

```

```

        drawmap(place, places_length);
#endif
    }
}

void fold(char* map, struct coordinate* place, int places_length)
{
    if (places_length == spawndepth)
    {
        // mkFoldbuf(place, places_length);
        // printf("Prototein: %s %S %d\n", prototein, foldbuf, bestscore);
        if (num_slaves) spawnJob(place, places_length);
        return;
    }

    if (places_length == prototein_length)
    {
        calc_score(place, map, places_length);
        return;
    }

    if (places_length>3)
        recurse(map, place[places_length-1].x - 1, place[places_length-1].y, place,
places_length);

    if (places_length>2)
        recurse(map, place[places_length-1].x, place[places_length-1].y - 1, place,
places_length);

    if (places_length>1)
        recurse(map, place[places_length-1].x + 1, place[places_length-1].y, place,
places_length);

    recurse(map, place[places_length-1].x, place[places_length-1].y + 1, place, places_length);
}

int brian_main(int argc, char** argv)
{
    char buf[256];
    int i, j, places_length;

    // printf("Prototein: %s\n", prototein);
    prototein_length = strlen(prototein);
    char* map = (char*)malloc(sizeof(char) * MAP_SIZE);
    struct coordinate* place = (struct coordinate*)malloc(sizeof(struct coordinate) *
prototein_length);
    winner = (struct coordinate*)malloc(sizeof(struct coordinate) * prototein_length);
    foldbuf = (char*)malloc(sizeof(char) * prototein_length + 1);
    for(i = 0; i< MAP_SIZE; i++)
        map[i] = ' ';
    bestscore = -999999999;
    winner_length = 0;
    map[MAP_COORDINATES(prototein_length,prototein_length)] = prototein[0];
    place[0].x = prototein_length;
    place[0].y = prototein_length;
    places_length = 1;

    if (partialsolution) places_length = readFoldbuf(partialsolution, place, map);

    fold(map, place, places_length);
    while (num_slave_free < num_slaves * queue) readResult();
    /*
    sw_stop();

    sw_timeString(buf);
    printf("Time taken: %s\n",buf);
    */
    mkFoldbuf(winner, prototein_length);
    // printf("Best is: %d %s\n", bestscore, foldbuf);

    free(map);
    free(place);

#ifdef GRAPHICS
    drawmap(winner, prototein_length);
#endif
}

```

```

        sleep(5);
        gs_exit();
#endif
        return 0;
}

void err_exit(const char * msg)
{
    if (!msg) fprintf(stderr, "usage: lob2 -Pprototein -sspawndept -ppartialsolution -bbestscore
-Nnum_slaves\n");
    else fprintf(stderr, "ERROR: %s\n", msg);
    exit(2);
}

void readResult()
{
    int i;
    int len, tag, tid;
    char hostnam[256];
    // bufid = pvm_rcv(tids[i], -1);
    int bufid = pvm_rcv(-1, -1);
    int info = pvm_bufinfo(bufid, &len, &tag, &tid);
    int rv = pvm_upkstr((char*)foldbuf);
    int bs, micro;
    rv = pvm_upkint(&bs, 1, 1);
    rv = pvm_upkstr(hostnam);
    rv = pvm_upkint(&micro, 1, 1);
    if (bs > bestscore)
    {
        bestscore = bs;
        readFoldbuf(foldbuf, winner, 0);
    }
    for (i=0; i<num_slaves; i++)
        if (slave_ids[i] == tid)
        {
            ++slave_free[i];
            if (verbose)
                printf("from %s (%d): i=%d\tbufid=%d\tinfo=%d\trv=%d\tlen=%d\ttag=%d\tbs=%d\tfold=%s\tmicro=
%d\tcommicro=%d\n", hostnam, tid, i, bufid, info, rv, len, tag, bs, foldbuf, micro, (int)
(microsec_db() - slave_start[i]));
            break;
        }
    if (i >= num_slaves) fprintf(stderr, "ERROR: no used slave\n");
    ++num_slave_free;
}

void spawnJob(struct coordinate* place, int places_length)
{
    int i;
    if (!num_slave_free || pvm_probe(-1, -1)) readResult();
    ++num_slave_calls;

    for (i=0; i<num_slaves; i++)
        if (slave_free[i])
        {
            --slave_free[i];
            slave_start[i] = microsec_db();
            pvm_initsend(PvmDataDefault);
            int rv = pvm_pkint(&prototein_length, 1, 1);
            rv = pvm_pkstr((char*)prototein);
            mkFoldbuf(place, places_length);
            rv = pvm_pkstr(foldbuf);
            rv = pvm_pkint(&bestscore, 1, 1);
            pvm_send(slave_ids[i], 1);
            --num_slave_free;
            break;
        }
    if (i >= num_slaves) fprintf(stderr, "ERROR: no free slave\n");
}

void endJob()
{
    int i;
    for (i=0; i<num_slaves; i++)
    {

```

```

    pvm_initsend(PvmDataDefault);
    int e = -1;
    int rv = pvm_pkint(&e, 1, 1);
    pvm_send(slave_ids[i], 1);
}
}

#define min(a,b) ((a)<(b)?(a):(b))

int main(int ac, const char * av[])
{
    int a;
    for (a = 1; a < ac; ++a)
        if (*av[a] == '-')
            switch (av[a][1])
            {
                case 'P': prototein = av[a] + 2; break;
                case 's': spawndepth = atoi(av[a] + 2); break;
                case 'p': partialsolution = av[a] + 2; break;
                case 'b': bestscore = atoi(av[a] + 2); break;
                case 'N': num_slaves = atoi(av[a] + 2); /*num_slaves = min(num_slaves,
sizeof(slave_ids)/sizeof(int));*/ break;
                case 'v': verbose = atoi(av[a] + 2); break;
                case 'q': queue = atoi(av[a] + 2); break;
                default: err_exit(0); break;
            }
        else
            err_exit(0);

    if (num_slaves > 0)
    {
        char * avs[7];
        double t00 = microsec_db();
        // avs[0] = av[0];
        avs[0] = "-N-1";
        avs[1] = 0;
        int num = pvm_spawn("lob2_master", avs, 0, "", num_slaves, slave_ids);
        if (num != num_slaves) err_exit("Can't create slaves...");
        int i;
        for (i = 0; i < num; ++i) slave_free[i] = queue;
        num_slave_free = num * queue;
        double t0 = microsec_db();
        brian_main(0, 0);
        t0 = microsec_db() - t0;
        endJob();
        t00 = microsec_db() - t00;
        printf("PVM-Best is: -s%d -N%d -q%d %d %s num_slave_calls=%d totmilli=%d milli=%d\n",
spawndepth, num_slaves, queue, bestscore, foldbuf, num_slave_calls, (int)(t00 / 1000), (int)(t0 /
1000));
    }
    else if (num_slaves < 0)
    {
        char hostnam[256];
        gethostname(hostnam, 255);
        int ptid = pvm_parent();
        while (1)
        {
            // fprintf(stderr, "%s, line %d: listen at %s\n", __FILE__, __LINE__, hostnam);
            int bufid = pvm_rcv(ptid, -1);
            int tid;
            int info = pvm_bufinfo(bufid, NULL, NULL, &tid);
            int rv = pvm_upkint(&prototein_length, 1, 1);
            if (prototein_length <= 0) break;
            // prototein = (const char*)realloc((char *)prototein, prototein_length + 1);
            prototein = (const char*)malloc(prototein_length + 1);
            rv = pvm_upkstr((char *)prototein);
            // fprintf(stderr, "%s, line %d: proto=%s\n", __FILE__, __LINE__, prototein);
            // partialsolution = (const char*)realloc((char *)partialsolution, prototein_length + 1);
            partialsolution = (const char*)malloc(prototein_length + 1);
            rv = pvm_upkstr((char *)partialsolution);
            // fprintf(stderr, "%s, line %d: partial=%s\n", __FILE__, __LINE__, partialsolution);
            rv = pvm_upkint(&bestscore, 1, 1);
            // fprintf(stderr, "%s, line %d: score=%d\n", __FILE__, __LINE__, bestscore);

            // fprintf(stderr, "%s, line %d: brian is ready...\n", __FILE__, __LINE__);
        }
    }
}

```

```

    double t0 = microsec_db();
    brian_main(0, 0);
// fprintf(stderr, "%s, line %d: brian is finished %d %s\n", __FILE__, __LINE__, bestscore,
foldbuf);
    int micro = (int)(microsec_db() - t0);

    mkFoldbuf(winner, prototein_length);
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(bestscore > -prototein_length ? foldbuf : (char*)partialsolution);
    pvm_pkint(&bestscore, 1, 1);
    pvm_pkstr(hostnam);
    pvm_pkint(&micro, 1, 1);
    pvm_send(ptid, 1);

    free((char*)prototein);
    free((char*)partialsolution);
    free(winner);
    free(foldbuf);
}
pvm_exit();
}
else
{
    double t0 = microsec_db();
    brian_main(0, 0);
    t0 = microsec_db() - t0;
    printf("Best is: %d %s milli=%d\n", bestscore, foldbuf, (int)(t0 / 1000));
}
}
return 0;
}

```

Appendix C – pyLinda

```
#!/usr/bin/python

import sys
import random
import math
import marshal
import linda
import time

# from graphics import *

print "Getting commandline options..."

is_main = 0
num_slaves = 0
num_bodies = 500
num_ite = 100
for a in sys.argv:
    if a[:2] == "-N": num_slaves = int(a[2:])
    elif a[:2] == "-M": num_bodies = int(a[2:])
    elif a[:2] == "-I": num_ite = int(a[2:])
print num_slaves

if num_slaves == 1:
    print "Running baseline without linda..."
    is_main = 1
    n = 0
else:
    print "Connecting to linda..."
    linda.connect()
    if num_slaves > 0:
        is_main = 1
        print "Posting setup to Universe N=" + str(num_slaves) + " bodies=" + str(num_bodies) +
" ite=" + str(num_ite)
        for n in xrange(num_slaves):
            print "slave " + str(n)
            linda.universe._out((n, num_slaves, num_bodies, num_ite))

    print "Reading setup from Universe"
    n, num_slaves, num_bodies, num_ite = linda.universe._inp((int, int, int, int))
    nxt = (n + 1) % num_slaves
    prv = (num_slaves + n - 1) % num_slaves
    # print "Slave: " + str(n) + " of " + str(num_slaves)
    # print "Ring: " + str(prv) + "-" + str(n) + "-" + str(nxt)

    print str(n) + ": Creating TuplesSpace for Communicating"
    ts = linda.TupleSpace()
    print str(n) + ": Export TuplesSpace for Communicating"
    linda.universe._out((n, ts))
    # time.sleep(1)
    print str(n) + ": Import TuplesSpace for Communicating"
    prvts = linda.universe._inp((prv, linda.TupleSpace))[1]
    # time.sleep(1+n)

print str(n) + ": Creating myBodies"
class cBody:
    def __init__(self,n,fm,ix,iy,fx,fy):
        self.iN=n
        self.fMass=fm
        self.fXpos=ix
        self.fYpos=iy
        self.fXvel=fx
        self.fYvel=fy

fG=1.0
fMaxMass=5.0
fVmax=300000.0
iXmax=6000000000
iYmax=6000000000

def fForce(a, b, mydata, theirdata):
```

```

dx=abs(a.fXpos-b.fXpos)
dy=abs(a.fYpos-b.fYpos)
r=math.sqrt((dx*dx)+(dy*dy))
# print "fForce: " + str(dx) + "," + str(dy) + "-" + str(r)
if r==0:
    if a.iN < b.iN:
        a.fMass+=b.fMass
        a.fXvel+=b.fXvel
        a.fYvel+=b.fYvel
        print str(n) + ": Removing mydata"
        mydata.remove(r)
    else:
        b.fMass+=a.fMass
        b.fXvel+=a.fXvel
        b.fYvel+=a.fYvel
        print str(n) + ": Removing theirdata"
        theirdata.remove(b)
else:
    a.fXvel+=((fG*a.fMass*b.fMass)/(r*r))*((b.fXpos-a.fXpos)/r)/a.fMass
    a.fYvel+=((fG*a.fMass*b.fMass)/(r*r))*((b.fYpos-a.fYpos)/r)/a.fMass

def move(oGalaxy):
    for i in oGalaxy:
        if i.fXvel>fVmax: i.fXvel=fVmax
        if i.fXvel<-fVmax: i.fXvel=-fVmax
        if i.fYvel>fVmax: i.fYvel=fVmax
        if i.fYvel<-fVmax: i.fYvel=-fVmax
        i.fXpos+=i.fXvel
        i.fYpos+=i.fYvel
        if not 0<i.fXpos<iXmax:
            oGalaxy.remove(i)
        else:
            if not 0<i.fYpos<iYmax: oGalaxy.remove(i)

def random_body():
    return cBody(n, \
        random.random()*pow(10,random.randint(1,fMaxMass)), \
        random.randint(1,iXmax), \
        random.randint(1,iYmax), \
        random.random()*3-1.5, \
        random.random()*3-1.5)

def bigbang_body():
    fMass=random.random()*fMaxMass
    fXpos=iXmax/2+random.random() #0otherwise all will just merge into one
    fYpos=iYmax/2+random.random()
    angle = random.random()*math.pi*2
    fXvel=math.sin(angle)*fVmax
    fYvel=math.cos(angle)*fVmax
    return cBody(n,fMass,fXpos,fYpos,fXvel,fYvel)

mydata = []
for i in xrange((n + 1) * num_bodies // num_slaves - n * num_bodies // num_slaves):
    # print str(n) + ": Adding body " + str(i)
    mydata.append(bigbang_body())

def pck(d):
    p = []
    # for b in d[1:2]:
    for b in d:
        p.append((b.iN, b.fMass, b.fXpos, b.fYpos, b.fXvel, b.fYvel))
    m = marshal.dumps(p)
    # print str(n) + ": pck() of " + str(len(d)) + " bodies -> len=" + str(len(m))
    return m

def upck(ltp):
    d = []
    for p in marshal.loads(ltp):
        d.append(cBody(p[0], p[1], p[2], p[3], p[4], p[5]))
    # print str(n) + ": upck() len=" + str(len(ltp)) + " -> " + str(len(d)) + " bodies"
    return d

def zPlot(oGalaxy):
    global oScreen
    # oScreen.clear();

```

```

    for i in oGalaxy:
        v = int((i.fMass/fMaxMass)*(255*255*255))
        c = ((v>>16)%255, (v>>8)%255, v%255)
        c=(255,0,0)
        oScreen.plot((int(i.fXpos)/1000000,int(i.fYpos)/1000000), c)
    oScreen.update()

# oScreen=graphicsScreen(iXmax/1000000,iYmax/1000000)

start=time.time()
print str(n) + ": Running iterations"
# sys.exit(0)
ite = 0
while ite < num_ite:
    if is_main: print str(n) + ": Ite" + str(ite)
    # if num_slaves > 1: linda.universe._out((nxt, n, ite, pck(mydata)))
    # if num_slaves > 1: ts._out((nxt, n, ite, pck(mydata)))
    # print str(n) + ": A"
    # for r in mydata: print r.iN, r.fXpos, r.fYpos
    # if is_main:
        # oScreen.clear();
        # zPlot(mydata)
    for b in mydata:
        for r in mydata:
            if r != b: fForce(b, r, mydata, mydata)
    if num_slaves > 1:
        for dummy in xrange(num_slaves):
#             print str(n) + ": read theirdata " + str(dummy)
            # mm, m, i, pd = linda.universe._inp((n, int, ite, str))
            mm, m, i, pd = prvts._inp((n, int, ite, str))
            # print str(n) + ": B"
#             print str(n) + ": s=" + s
            if m != n:
                # linda.universe._out((nxt, m, ite, pd))
                ts._out((nxt, m, ite, pd))
                # print str(n) + ": C"
                theirdata = upck(pd)
                # if is_main: zPlot(theirdata)
                # for r in theirdata: print r.iN, r.fXpos, r.fYpos
                for b in mydata:
                    for r in theirdata: fForce(b, r, mydata, theirdata)
#             print str(n) + " end calculations"
    move(mydata)
    ite += 1

stop=time.time()
if is_main: print "Simulated "+str(num_bodies)+" bodies for "+str(num_ite)+" timesteps in
"+str(stop-start)+" seconds with "+str(num_slaves)+" processes"

```