Ytelsesvurdering av ProtoMol på en PC-klynge

Åsmund Østvold
IDI NTNU

21. desember 2002

Sammendrag

Den viktigste grunnen til å parallellisere programmer har vært å få en hastighetsøkning i beregningstid. De fleste parallellprogrammene er programmert for og kjørt på superdatamaskiner. Et spørsmål nå er om man trenger superdatamaskiner eller om man kan klare seg med store PC-klynger. Programmet ProtoMolsom beregner molykelærdynamikk ble vurdert. Denne rapporten analyserer hastigheten til beregningen på en klynge av PC-er relativt til hastigheten til beregningen på en PC. Den relative hastighetsøkningen ble ikke funnet tilfredstillende. Spørsmålstillinger om mulige forbedringer av ProtoMol på ble reist.

Innhold

1 Introduksjon **1**

2 Programvarepakken ProtoMol **1**

3 Problemer og løsninger **2**
- 3.1 Prosjektets størrelse og forsinkelse 2
- 3.2 ProtoMol og MPI-2 2
- 3.3 Konvertering av ProtoMol fra MPI-2 til MPI-1 3
  - 3.3.1 Kompilering av ProtoMol med LAM MPI 3
  - 3.3.2 Kjøring av ProtoMol under LAM MPI 4
  - 3.3.3 Omskrivning av ProtoMol fra MPI-2 til MPI-1 standarden 4
  - 3.3.4 Problemer med kommandolinjeparametere i MPI . . . 5

4 Systembeskrivelse av ClustIS **6**

5 Parameterspesifikasjon **6**
- 5.1 Antagelser om ProtoMol 6
- 5.2 Nær optimale verdi for `cellsize` 7

6 Hastighetsmålinger av ProtoMol **8**
- 6.1 Definisjoner 8
- 6.2 Utføringen av tidsmålingene 9
- 6.3 Målinger 9
  - 6.3.1 Måling av Water 423 atomer 9
  - 6.3.2 Måling av BPTI 14281 atomer 11
  - 6.3.3 Måling av ApoA1 92224 atomer 12
  - 6.3.4 Oppsummering av måleresultater 13
  - 6.3.5 Riktighet av målingene 13
- 6.4 Resultat 14

7 Analyse av kommunikasjon og en mulig undersøkelse **15**
- 7.1 Kommunikasjon 15
  - 7.1.1 Benchmark av MPI_Allreduce på ClustIS 17
- 7.2 Mulig skaleringsundersøkelse 19

8 Konklusjon **19**

Etterord **20**

Referanser **22**

Apendiks **23**

A Plassering av målepunkter for tidtaking **23**

B ProtoMols kompilering på ClustIS **24**
B.1 MPICH 24
B.2 LAM MPI 25

1 Introduksjon

Det er ennå stor usikkerhet om hvilken relativ hastighetsøkning man kan oppnå ved å bruke klynger av PC-er i parallellprogrammering. Jeg har i denne rapporten testet og vurdert et program, PROTOMOL, for å se hvilken ytelsesforbedring det er mulig å få ved å kjøre dette programmet på en klynge av PC-er målt opp mot kjøring av det samme programmet på en enkel PC. PROTOMOL er et program som har vært brukt på stormaskin med bra resultater [4]. Dette programmet beregner og gjør simuleringer av molykelærdynamikk.

Jeg begynner med å gi et innblikk i programvarepakken PROTOMOL i kapittel 2. I kapittel 3 vil jeg beskrive prosessen frem til jeg kunne begynne å gjøre målinger på programmet PROTOMOL. ClustIS, PC-klyngen som alle målingene gjøres på, presenteres i kapittel 4. I kapittel 5 presenterer jeg mine antagelser om programmet PROTOMOL.

I kapittel 6 gjør jeg hastighetsmålinger av programmet PROTOMOL på klyngen. Jeg gjøre i kapittel 7 en analyse av den type kommunikasjon som PROTOMOL bruker mye. Kapittel 8 har en oppsummering av de målinger og slutninger jeg har gjort om PROTOMOL kjørt på klyngen.

2 Programvarepakken ProtoMol

Her vil jeg kort prøve å forklare litt om PROTOMOLs kompleksitet og presentere en del størrelser som belyser dette.

Programvarepakken PROTOMOL inneholder ikke bare programmet PROTOMOL som gjør beregningene, det inneholder i tillegg 3 andre programmer:

- analyzer
- compucell
- namd2protomol

Disse er ikke interessante for den vurderingen jeg gjør av denne programvarepakken.

Programmet PROTOMOL og disse tre programmene blir kompilert fra cirka 600 filer. Programmene er skrevet i C++. Alle klasser skal ha en headerfil og en implementasjonsfil. Hvis dette er tilfellet, er det i underkant av 300 klasser.

Den totale størrelsen på disse datafilene er cirka 3 MB. Størrelsen på programmet PROTOMOL i produksjonsutgave er 8.3 MB med statisk innlenket MPICH [1, 3] og 7.9 MB i seriell utgave.

Fra nå av mener jeg programmet `protomol` når jeg skriver PROTOMOL.

Det at PROTOMOL blir så stor i kompilert form, ca 8 MB, sier noe om hvor kompakt denne pakken er skrevet. PROTOMOL er skrevet med utstrakt bruk av C++ templates. Det er brukt nestede templates og dette har muliggjort en kompakt kode.

Under kjøring av et eksempel som har 92228 atomer er minnebruken oppe i cirka 100 MB per node.

3 Problemer og løsninger

Her presenteres problemer, løsninger og arbeidsmetoder som jeg møtte på under prosjektet.

3.1 Prosjektets størrelse og forsinkelse

Det var overraskende at det største problemet var å få tak i et program som jeg kunne gjøre målinger på. Det tok oppunder to måneder før jeg fikk programmet. Dette er en viktig årsak til at prosjektet fikk et mye mindre omfang enn planlagt og at jeg fikk utsatt innleveringen en måned senere enn den oppsatte offisielle innleveringsfristen. Dette er ikke den eneste grunnen, men en meget viktig grunn.

3.2 ProtoMol og MPI-2

Hvis Matthey [4] skulle ha parallellisert PROTOMOL for klynger, ville han ikke ha valgt å bruke funksjonalitet fra MPI-2 standarden fordi en del funksjonalitet i denne standarden har stor fordel av å kjøre på en maskin med delt minnearkitektur. En superdatamaskin med delt minnearkitektur vil ha større fordeler av deler av funksjonaliteten til MPI-2 standarden. Derfor tror jeg PROTOMOL ble utviklet i hovedsak for superdatamaskiner. Det er ikke noe i MPI-2 standarden som sier at den må kjøres på superdatamaskin. Når man ikke får støtte fra maskinvaren i det hele tatt, tar det mye lengre tid å utføre mange av funksjonene som er definert i MPI-2 standarden.

Det har ikke vært så stor etterspørsel etter MPI-2 funksjonaliteten til at noen hittil har laget en versjon som er av en slik kvalitet at et lønner seg å bruke den på klynger av PC-er.

Det første problemet jeg møtte på var at MPICH [2, 3], som er den implementasjonen som var installert på ClustIS, ikke hadde den funksjonaliteten som ProtoMol bruker fra MPI-2 standarden. De funksjonene som ProtoMol bruker er `MPI_Put()` og `MPI_Get()`. Disse funksjonene gjør det mulig å hente/sette data fra/til en annens minneområde uten at denne må ha et tilhørende funksjonskall til en slik operasjon (one-sided communication). Dette gjør at programmet slipper synkronisering mellom prosesser for slike operasjoner. Dette er en funksjonalitet som er grei når man som i superdatamaskiner har maskinvarestøtte helt eller delvis til dette. På klynger av PC-er som ikke har denne muligheten er effekten av å gjøre dette ikke så stor da man på ett "høyere nivå" i den kompilerte koden må ha et send/motta funksjonskall.

Da jeg fant ut at MPICH ikke støttet denne funksjonaliteten, tenkte jeg først på å finne en implementasjon som kunne installeres på ClustIS som støttet den nødvendige funksjonalitet fra MPI-2 standarden som ProtoMol brukte.

Jeg diskuterte denne løsningen med Matthey og han sa at LAM MPI som er en annen implementasjon av MPI hadde implementert `MPI_Put/Get` av MPI-2 standarden. Han sa også at han hadde gjort tester med LAM MPI og funnet ut at ProtoMol gikk saktere på en klynge enn den gjorde når den kjørte på en PC. Vi ble da enige om at det var mest interessant å få en versjon av ProtoMol som fungerte med MPI-1 standarden og som ble kompilert med MPICH da MPICH var den versjonen som implementerte MPI-1 og var installert på ClustIS. Problemet var nå at ProtoMol fortsatt brukte `MPI_Put/Get`. Konsekvensene av dette vil jeg ta opp nedenfor.

3.3 Konvertering av ProtoMol fra MPI-2 til MPI-1

Her beskriver jeg de vanskeligheter som jeg hadde når jeg prøvde å få konvertert programmet til MPI-1 standarden.

Matthey mente dette ikke skulle være så vanskelig å gjøre, 2 dagers arbeid mente han. Jeg hadde mine tvil, spesielt fordi dette er et stort program hvor jeg egentlig ikke fullt ut forstår hva slags metoder og løsningssystemer programmet bruker.

3.3.1 Kompilering av ProtoMol med LAM MPI

Før jeg kunne begynne å konvertere ProtoMol, mente jeg det ville være til stor hjelp å ha en kjørbar versjon av ProtoMol installert for å kunne sjekke om jeg hadde et program som produserte riktige resultater. Hvis jeg hadde

en kjørbar installasjon ProtoMol som kunne kjøre `MPI_Put/Get` ville det være mulig å gjøre omskrivningen gradvis.

Det var ikke noe problem å få LAM MPI installert på ClustIS. Det som viste seg vanskelig var å klare og kompilere ProtoMol. Det tok mye tid å få linke inn bibliotekene til LAM MPI i programmet. I ettertid sitter jeg igjen med mye kunnskap om linking av programmer. Jeg slet med å finne ut hvilke og hvordan jeg skulle bruke linkeflaggene til `gcc` for å få linket inn de riktige bibliotekene. Jeg har aldri lært skikkelig å linke inn eksterne biblioteker. Jeg lærte i denne prosessen hvordan man bruker linkeflaggene `-l` og `-L` til `gcc`. Jeg gjorde en del søk på Internett og leste `man` sidene til `gcc`. Med mine manglene forkunnskaper var det vanskelig å finne noen god måte å begynne og forstå hvordan linking skulle gjøres i praksis og hva som måtte spesifiseres. For eksempel fant jeg ut at `-lxx` sier at man linker inn `xx.lib` filen. Spesielt forvirret det meg at man hadde droppet endingen `.lib`. Det tok også en stund å finne ut av parameteren `-L` som gjør det mulig å legge til kataloger som gjennomsøkes for å finne bibliotek. Måten å spesifisere hva/hvordan man skal linke inn var ikke så intuitivt for meg som mange nettsider ville ha det til, men nå er det helt elementært etter at jeg har forstått tankegangen.

3.3.2 Kjøring av ProtoMol under LAM MPI

Det var et helt annen problem å kunne klare og kjøre ProtoMol med LAM MPI. En god stund trodde jeg at MPICH og LAM MPI hadde de samme systemene og da trodde jeg at bare jeg sørget for å bruke den riktige `mpirun` skulle det gå greit å kjøre ProtoMol under LAM MPI. Det viste seg at kjøresystemene var noe forskjellige og det tok *mye* tid å finne ut av dette. LAM MPI har en type deamon som må kjøre på alle nodene som man skal kjøre programmer på. Det å få deamon-ene til å kjøre på nodene og få disse til å fungere var ingen lett jobb. Med mye hjelp av Zoran Constantinescu-Fülöp fikk vi laget en løsning som fungerte for å kunne kjøre en parallell versjon av ProtoMol med LAM MPI på ClustIS.

Først da var jeg i stand til å begynne å se på hvilke forandringer som skulle til.

3.3.3 Omskrivning av ProtoMol fra MPI-2 til MPI-1 standarden

Den informasjonen jeg fikk fra Matthey var at den viktigste delen av omskrivningen var å skrive om noen funksjoner i C++ klassen Parallel.

Her er vel det stedet jeg stod mest fast, men jeg slet virkelig for å få til dette, men samme hvor mye jeg satt og leste og prøvde å forstå denne koden kom jeg ikke noen vei. Det er flere faktorer som jeg tror gjorde at jeg ikke

klarte å skrive om ProtoMol til å bruke MPI-1. Det første er at ProtoMol er et stort program. Det andre er at klassen Parallel ikke var godt kommentert. En siste viktig faktor er at jeg ikke har tilstrekkelig kunnskaper verken om de fysikalsk kjemiske prinsippene eller om algoritmer som brukes for å løse molykelærdynamiske problemer i ProtoMol.

Jeg ser i ettertid at jeg ville hatt nytte av enten å hatt en mer direkte dialog med Matthey eller hvert fall hatt en å jobbe sammen med på denne delen. Jeg ble ofte bare sittende å forsøke og prøve å forstå koden. Det å ikke ha noen å diskutere løsningsforsøk med gjorde at det ble meget vanskelig å komme videre for meg. Jeg gjorde en del forsøk på å skrive om koden, men det endte hele tiden opp i kode som ikke kjørte og noen ganger ikke kompilerte.

Heldigvis forbarmet Matthey seg over meg etter en del e-post sending og gjorde de forandringene som skulle til for å få ProtoMol til å kjøre MPI-1 implementasjon av standarden.

3.3.4 Problemer med kommandolinjeparametere i MPI

Med nytt mot gikk jeg i gang og trodde jeg kunne begynne å måle. Det jeg da oppdaget var rart. ProtoMol fungerte hvis jeg kompilerte og kjørte med LAM MPI. ProtoMol kompilerte, men jeg fikk ikke til å kjøre med MPICH. Programmet krasjet når jeg kjørte det på ClustIS. Jeg tok kontakt med Matthey og det fungerte hos han. Ved gjøre en del parallell debuging viste det seg at programmet krasjet når det prøvde å lese kommandolinjeparameterene. Ved å spørre på news-gruppen `comp.parallell.mpi` fikk jeg vite at det ikke er noen enighet om hvordan man skal håndtere parametere på kommandolinjen mellom de forskjellige MPI implementasjonene. Noen MPI implementasjoner bruker kommandolinjeparametere for å distribuere verdier programmet trenger for å kjøre. Det viste seg at MPICH bruker parametere på kommandolinjen for å distribuere informasjon som alle prosessene som kjøres trenger. Jeg fikk vite at LAM MPI bruker miljøvariabler for å distribuere denne inforasjonen. MPI standarden sier bare at etter eksekvering av funksjonen `MPI_Init(argc, argv)` så er `argc` og `argv` riktig i forhold til hva man selv har skrevet på kommandolinjen.

I ProtoMol blir `MPI_Init(argc, argv)` kalt inne i klassen `Parallel` fordi ProtoMol også skal kunne kompileres og kjøres som et serielt program. Det som gjorde at dette ikke fungerte når jeg prøvde å kjøre ProtoMol parallelt med MPICH lenket inn var at parameterene `argc` og `argv` ikke ble referanseoverført. Når dette ble implementert, var jeg endelig i stand til å begynne å gjøre målinger.

4 Systembeskrivelse av ClustIS

ClustIS er klyngen som eksperimentet ble kjøre på. ClustIS er en klynge av PC-er koblet sammen med en 100 MBit swichet Ethernet. Klyngen består av 38 noder: 1 hovednode, 8 noder for testing av programmer og 29 noder til kjøring av eksperimenter. Det var på de siste 29 nodene ProtoMol ble kjørt på. Tabell 1 beskriver de viktigste dataene for disse 29 nodene.

Ant. noder	CPU type	klokke frekvens	RAM
12	AMD Athlon XP 1700+	1.46 GHz	1 GByte
8	AMD Athlon XP 1700+	1.46 GHz	2 GByte
8	AMD Athlon MP 1600+	1.40 GHz	1 GByte
1	AMD Dual Athlon MP 1600+	1.40 GHz	1 GByte

Tabell 1: Noder ProtoMol ble kjørt på.

Jeg fant ingen enkel måte selv å bestemme hvilke noder programmet skulle kjøres på. Jeg vil sannsynliggjøre i kapittel 6.2 at den lille forskjellen i maskinvare mellom nodene ikke hadde noen vesentlig innvirkning på måleresultatene som jeg er nødt til å ta hensyn til.

5 Parameterspesifikasjon

Her presenterer jeg først de antagelser som jeg gjort om ProtoMol og faktorer som påvirker størrelsen av beregningsproblemet. Jeg følger opp med hvordan jeg valgte parameteren som var mulig å variere uten forandre de fysikalske parameterene for eksperimentene.

5.1 Antagelser om ProtoMol

Dette avsnittet presenterer min forståelse av hvilke parametere som påvirker beregningskompleksiteten til algoritmene som brukes av ProtoMol og et forsøk på å belyse sammenhengen mellom disse.

Det er to parametere, antall atomer per CPU og `cellsize`, som ikke påvirker *selve* beregningsresultatene, men som har stor betydning når det gjelder beregningstiden i en parallellkjøring av programmet. Det eneste som kan skje når det gjelder beregningsresultatene er at det kan komme avrundingsfeil. Det er en sammenheng mellom antall atomer pr CPU og størrelsen på `cellsize`. Jeg har ikke arbeidet nok med molekylærdynamikk til å forstå sammenhengen mellom `cellsize` og antall atomer per CPU. Matthey har fortalt meg at hvis jeg har funnet en nær optimal verdi for `cellsize` ved et

eksperiment med 2 noder så kan jeg anta at denne verdien er nær optimal for alle størrelser av antall noder med dette problemet.

Parametere som gis av eksempelets natur og som samvirker med de to parameterene som er nevnt ovenfor:

- Antall atomer,
- `cutoff` (hvor langt en kraft har lov til å virke),
- Antall krefter (i et molekyl og mellom atomene),
- `boundaryConditions` som i mine data har verdien, vacuum eller periodic.

Disse parameterene er altså satt av eksperimentet og blir aldri justert i mine tidsmålinger.

5.2 Nær optimale verdi for `cellsize`

Jeg fikk vite at det var mulig å justere parameteren `cellsize`.

Her vil jeg i korte trekk forklare hvordan jeg bestemte den nær optimale verdien for `cellsize`. Jeg er blitt fortalt av Matthey at `cellsize`-verdien kan variere litt med cache-størrelsen på CPU-ene.

Enkel sagt er `cellsize` en parameter som sier hvor stor en "boks" i problemdomenet er. Slik jeg har forstått det, er det fordelaktig at denne boksen går opp i en heltallsfaktor av den totale utstrekningen av problemet. `cellsize` har også en relasjon til størrelsen `cutoff`. `cutoff` relaterer seg til krefter, og bestemmer på hvor lang avstand disse skal virke. Det vil være fordelaktig å ha en `cellsize` som er like stor som `cutoff`. Hvis man har flere krefter med forskjellige `cutoff`-verdier, bør man prøve å finne en enkel relasjon mellom `cellsize` og alle `cutoff`-verdiene. Hvis `cellsize` settes for stor og dermed inneholder for mange atomer, vil man få dårlig minnebruk med for mange cache-bommer(cache misses). `cellsize` har også en relasjon til antall atomer i en celle. Jeg fikk vite at det var fordelaktig å ha så mange atomer/molekyler per celle som mulig.

Innenfor disse rammene kan man justere størrelsen på `cellsize`. I tabell 2 viser en kjøringer med forskjellige verdier på `cellsize` for ApoA1 med 92224 atomer. `cutoff` er i dette eksperimentet gitt lik 10. Tabellen viser at man får et minimum for beregningstid når `cellsize` settes til 1/3 av `cutoff`.

Jeg tror at grunnen til at jeg fikk et minimum ved 3.33333 var at ProtoMol her ikke fikk så mange cache-bommer, men samtidig slapp ProtoMol å hente inn for mange celler.

`cellsize`	2.0	3.33333	5.0	6.0	10.0
atomer/celle	0.71	3.28	11.09	19.16	88.70
beregnings tid s	49.01	31.62	32.47	39.14	42.65

Tabell 2: Relasjonen mellom `cellsize`, antall atomer per celle og kjøretid for et eksperiment med 8 prosessorer, 92224 atomer, `cutoff` på 10 og problemområde i en kube med sider (108.861, 108.861, 77.758).

Jeg har gjort tilsvarende målinger for å bestemme nær optimale verdier for `cellsize` for de alle datasettene i rapporten. Jeg vil ikke presentere tilsvarende tabeller som den ovenfor for hvert eksperiment. De verdier jeg fikk for nær optimal størrelse på `cellsize` samsvarte med de verdier som Matthey hadde funnet.

6 Hastighetsmålinger av ProtoMol

Det vil bli presentert data for hastighetsøkning for forskjellige typer beregninger som ProtoMol kan gjøre. Problemene som ble målt er problemer som har en parallellisert løsningsmetode i ProtoMol. Dette er viktig da ikke alle metoder er parallellisert.

Nedenfor har jeg 3 datasett som hver har 2 `boundaryConditions`, vacuum og periodic. Det er gjort målinger av beregningshastighet på et varierende antall noder. Resultatene vil bli presentert i en figur som gir en relasjon mellom antall noder og hastighetsøkning. For noen av datasettene blir deler av resultatene vist i en tabell.

6.1 Definisjoner

I denne rapporten vil jeg bruke uttrykket hastighetsøkning(speedup) som seriell kjøretid delt på parallell kjørtid.

$$\text{Hastighetsøkning} = \frac{\text{Seriell kjøretid}}{\text{Parallell kjøretid}}$$

Et annet interessant mål er utnyttelsesgrad(efficiency). Utnyttelsesgraden er et mål på prosessorutnyttelse i et parallelt program relativt til et serielt program. Definisjonen er:

$$\text{Utnyttelsesgrad} = \frac{\text{hastighetsøkning}}{\text{antall prosessorer}}$$

Når jeg snakker om tid, mener jeg den tiden som faktisk har gått fra jeg startet målingen til jeg stoppet den uten å ta vekk noe fordi programmet

måtte vente på eller ble avbrutt av operativsystemet. Dette betyr at hvis det er mange forstyrrelser under beregningen som måles vil tiden som rapporteres også inneholde tid som er brukt til forstyrrelser.

6.2 Utføringen av tidsmålingene

Målingene, både for parallell- og seriellkjøringene, ble utført på dedikerte noder på ClustIS. Tiden jeg målte var bare den tiden programmet faktisk brukte til beregning. Altså ikke den tiden som programmet brukte til å lese inn data, initiere og distribuere datastrukturer. Tidtakingen ble stoppet før programmet genererte de ferdige resultatene. På denne måten ble bare den delen av programmet som skal kjøre *mange* ganger målt, altså den interessante delen, se appendiks A. Det å måle på denne måten mener jeg er interessant fordi alle forstyrrelsene fra for eksempel operativsystem, andre brukere på klyngen, osv. vil komme med. Slike forstyrrelser vil også være til stede i en praktisk anvendelse og være relevant tidsbruk når man vurderer effektiviteten av parallelliseringen.

Alle målinger er et snitt av fem kjøringer. Det var aldri mer en 1 – 2% variasjon i tidsmålingene mellom kjøringene i samme gruppe. Det er to kommentarer til dette resultatet: For det første er prosessorene litt forskjellige, jevnfør tabell 1. Prosessorene tildeles av køsystemet til programmet uten noen styringsmuligheter. For det andre er det en forstyrrelse fra operativsystemet, andre programmer på ClustIS, osv. Det er rimelig å anta at disse to faktorene er uavhengige av hverandre.

Som nevnt ovenfor var variasjonen 1 - 2% i hastighetsmålingene over fem målinger på sammen datasett. Dette kan tyde på at de typer forstyrrelser som er nevnt ovenfor påvirker resultatene i liten grad eller er konstante på ClustIS. Jeg tror at prosessortildelingen har liten innvirkning på resultatene og at støyen fra operativsystemet osv. er tilnærmet konstant.

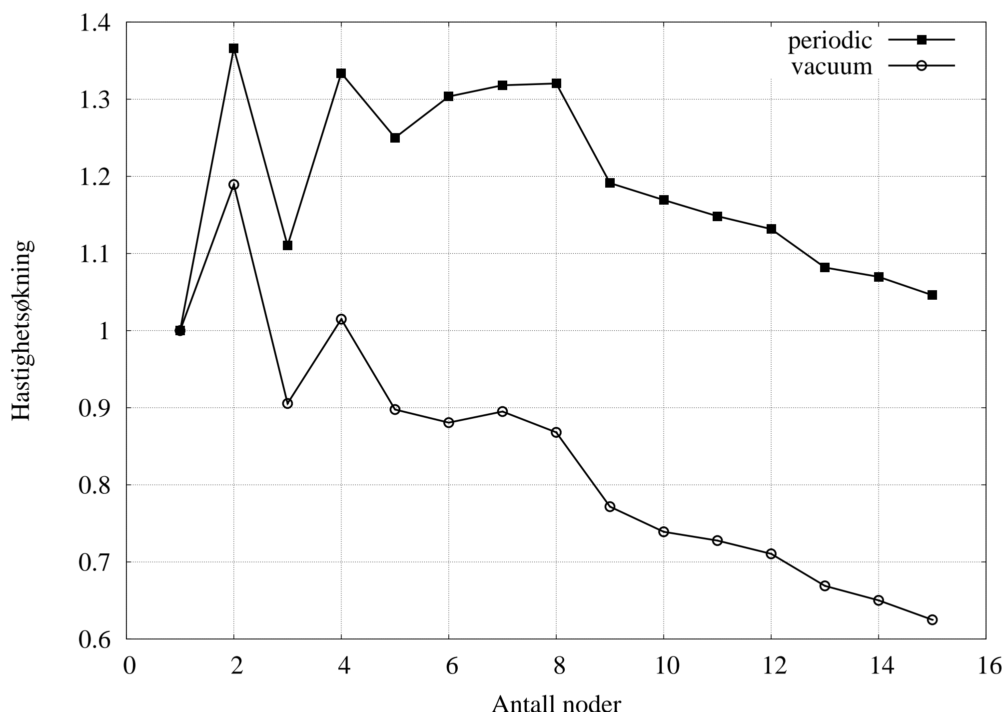
6.3 Målinger

For hvert av rådatasettene jeg har brukt er det utført beregninger under to forskjellige `bounderyConditions`, `periodic` og `vacuum`. Da det er en enkel relasjon mellom hastighetsøkning og utnyttelsesgrad har jeg bare delvis redegjort for utnyttelsesgraden nedenfor.

6.3.1 Måling av Water 423 atomer

Dette settet har færrest antall atomer. Jeg forventet ikke noen særlig hastighetsøkning da jeg regnet med at beregningsproblemet var for lite til at det i

noen særlig grad ville lønne seg å parallellisere løsningen.



Figur 1: Water 423 atomer. Hastighetsøkning som funksjon av antall noder.

Som figur 1 viser, var mine antagelser riktige. Man hadde en topp ved bruk av to noder og hvis man brukte flere en dette gikk det grovt sett seinere. Under vacuum-betingelsen er det så ille at det går saktere hvis man har flere noder enn bare med en node når antall noder er større enn to.

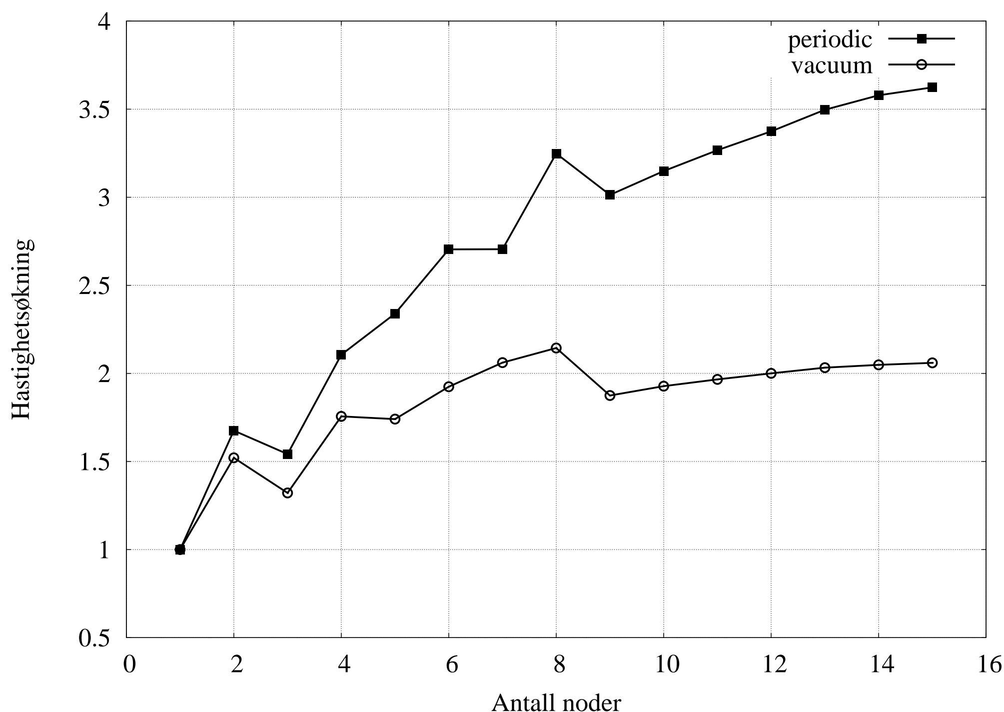
Tabell 3 viser forholdet mellom tid, hastighetsøking og utnyttelsesgrad for vacuum-betingelsen. Jeg har tatt med et begrenset antall analyser for å illustrere resultatene jeg fikk. Der hvor fet skrift er brukt, går det faktisk saktere i parallell- enn i seriellkjøring.

Antall prosessorer	1	4	8	12	15
Hastighetsøkning	1	1.01	**0.87**	**0.71**	**0.62**
Utnyttelsesgrad	1	0.25	**0.11**	**0.06**	**0.04**

Tabell 3: Selekterte data fra målingen av Water 423 atomer, vacuum.

6.3.2 Måling av BPTI 14281 atomer

Dette datasettet har 14281 atomer og er betydelig større enn i forrige settet med 423 atomer. Her forventet jeg at det skulle være mulig å få en hastighetsøking.



Figur 2: BPTI 14281 atomer. Hastighetsøkning som funksjon av antall noder.

Man ser fra figur 2 at hastighetsøkningen er bedre i disse målingene enn i figur 3, men de er ennå ikke tilfredstillende i forhold til de resultatene som er presentert i kapittel 5.9.2 [4].

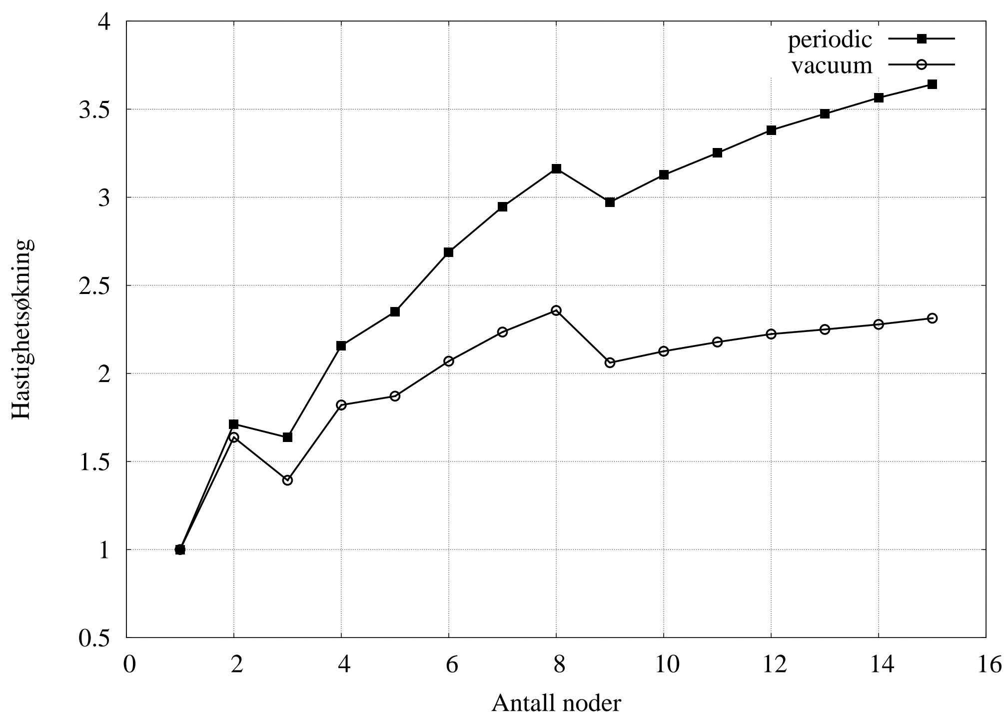
Tabell 4 inne holder resultater for periodic-målingen. Hvis man ser på resultatene for utnyttelsesgrad i tabell 4 se man at de er lave, men bedre enn i tabell 3. Det å ha en hastighetsøking på 3.62 ved bruk av 15 dedikerte PC-er er ikke en god utnyttelse.

Antall prosessorer	1	4	8	12	15
Hastighetsøkning	1	2.11	3.25	3.37	3.62
Utnyttelsesgrad	1	0.53	0.41	0.28	0.24

Tabell 4: Selekterte data fra målingen av BPTI 14281, periodic

6.3.3 Måling av ApoA1 92224 atomer

Dette er det siste og størst datasettet jeg regnet på. Etter å ha fullført de to første datasettene hadde jeg tro på at bare problemet ble stort nok ville det være mulig å få en forholdsvis bra hastighetsøking og utnyttelsesgrad.



Figur 3: ApoA1 92224 atomer. Hastighetsøkning som funksjon av antall noder.

Når man sammenligner figurene 3 og 2 og tabellene 4 og 3 ser en at det er minimal forskjell på hastighetsøkning på begge betingelsene på tross av at ApoA1 har over 6 ganger så mange atomer som BPTI. Ikke overraskende er utnyttelsesgraden tilnærmet lik utnyttelsesgraden for BPTI problemet selv om antallet atomer er over seks ganger større.

Antall prosessorer	1	4	8	12	15
Hastighetsøkning	1	2.16	3.16	3.38	3.64
Utnyttelsesgrad	1	0.54	0.40	0.28	0.24

Tabell 5: Selekterte data fra målingen av ApoA1 92224, periodic

6.3.4 Oppsummering av måleresultater

Det er ingen grunn til å bruke klynger av typen ClustIS til å beregne molekylærdynamikk for problemer av samme størrelsesorden som Water 423. Det er en klar forbedring av hastighetsøkning i de to største datasettene. Men det er overraskende at det ikke er noen forbedring i hastighetsøkning og utnyttelsesgrad i det siste eksempelet selv om antall atomer er betydelig større. Dette må bety at det er andre ting enn antall atomer som begrenser ProtoMols evne til å parallellisere på en klynge av den typen ClustIS er. En mulig årsak til den svake utnyttelsen av parallelliteten i ProtoMol er diskutert i 7.1.

6.3.5 Riktighet av målingene

Jeg var lenge i tvil om jeg gjorde målingene riktig fordi jeg fikk så veldig mye dårligere resultat enn det Matthey fikk i sin vurdering av ProtoMol i kapittel 5.9.2 [4]. Jeg var så heldig at han var her den 11.12.02 og vi fikk litt tid til å drøfte mine målinger.

Det som var meget interessant var at vi fant ut at vi var uenige i hvordan man skulle måle tid. Hvis jeg forsto han rett, mente han at man skulle bruke den tiden som prosessoren brukte og ikke den tiden som programmet brukte på å bli ferdig med beregningen. En prinsipiell forskjell mellom disse innfallsviklene er at i definisjonen av tid som jeg har brukt er "idle time" inkludert, men ikke i hans definisjon av tid. Det betyr at hans målinger er mer teoretisk interessante, mens mine målinger reflekterer den virkelighet man møter når anvender ProtoMol på et system i drift.

ProtoMol har noen innebygde funksjoner for å måle tiden. En stund trodde han at mine målinger var feil da det var en urimelig stor forskjell mellom resultatene i måling av total beregningstid. Mine verdier for beregningstiden var en stund cirka dobbelt så store som hans. Etter en del undersøkelser viste det seg at selv om han la sammen alle tidene som han mente skulle til for å få den totale tiden, altså kommunikasjon pluss beregningstid, fikk han ikke min tid. Disse tidene skulle være i samme størrelsesorden fordi idle time skulle være liten når kjøringene av programmet foregikk på dedikerte noder. Matthey flikket litt på ProtoMols interne tidtakingskode og jeg fikk inntrykk av at han forandret måten ProtoMol målte en av deltidene. Etter dette målte ProtoMol cirka de samme beregningstidene som jeg hadde fått. Forskjellen var da bare cirka et halvt sekund på en beregningstid på ca 26 sekunder. Denne forskjellen antar jeg er idle time for beregningen.

Jeg er klar over at dette er for dårlig dokumentert, alt skjedde på cirka en time. Når jeg likevel tar det med, er det fordi det er mulig vi har målt ganske

forskjellige ting. Jeg ble på et for sent tidspunkt klar over at dette kunne være et meget viktig punkt, men hadde da ikke mulighet til å undersøke eventuelle årsaker/konsekvenser.

Den opprinelige store forskjellen mellom Mattheys [4] og mine målinger kan ha sin årsak i flere faktorer:

- Det kan hende at det er innført en feil i det interne tidtakingssystemet da ProtoMol ble skrevet om for å kjøre på en klynge med MPI-1 standarden.
- Det kan være at operativsystemene gir forskjellige svar på de samme funksjonskallene fordi det er en del forskjeller mellom operativsystemet på ClustIS og den superdatamaskinen som han gjorde sine målinger på. Jeg vet ikke nok om dette til å gå i dybden.
- Hvis ikke noen av disse alternativene eller noen jeg ikke har oversikt over, må det ha noe med hvilke deltider som inkluderes i beregningstiden.

Dette gjør at jeg i ettertid er litt skeptisk til tolkningene av målingene som er gjort i hans doktoravhandling. Jeg spurte han om forskjellen mellom ProtoMols og mine resultater på ClustIS og om den flikkingen han var nødt til å gjøre for å få ProtoMols interne tidtakingssystem til å rapportere verdier som var samsvarende med mine hadde noen innflytelse på de dataene som han har rapportert i doktoravhandlingen [4]. Han svarte avkreftene på dette.

6.4 Resultat

Dette kapittelet vil være en diskusjon om de målinger som er presentert i denne rapporten i forhold til de målinger som er gjort av Matthey i kapittel 5.9.2 [4].

Før en begynner å diskutere disse målingene, er det viktig å nevne at det er en forskjell i definisjonen av tiden som måles. I doktorgranden måler man tiden som beregningen bruker på prosessoren. I min rapport er de målte tidene den tiden som beregningen brukte fra start til slutt(wall clock time).

Med unntak av det minste problemet(Water 423 atomer) har ProtoMol en utnyttelsesgrad som er i overkant av 0.5 i Mattheys målinger som var på en superdatamaskin. Mine målinger på ClustIS har klart dårligere resultater. Utnyttelsesgraden på ClustIS er oppe i cirka 0.4 ved 8 prosessorer, men synker til cirka 0.25 ved 15 prosessorer. Jeg kan ikke se noen grunn til at utnyttelsesgraden kommer til å stige hvis antallet prosessorer økes. I min

undersøkelse er det en betydelig forskjellen mellom betingelsene vacuum og periodic som ikke finnes i hans undersøkelse.

7 Analyse av kommunikasjon og en mulig undersøkelse

Dette kapittelet diskuterer hvordan ProtoMol i et eksempel kommuniserer og bruker tid. Jeg kommer også med et forslag til en måling som hadde vært interessant å gjøre.

7.1 Kommunikasjon

Her beskriver jeg mer inngående hvordan ProtoMol bruker tiden i parallelle kjøringer og sannsynliggjør hvorfor ProtoMol ikke skalerer på ClustIS.

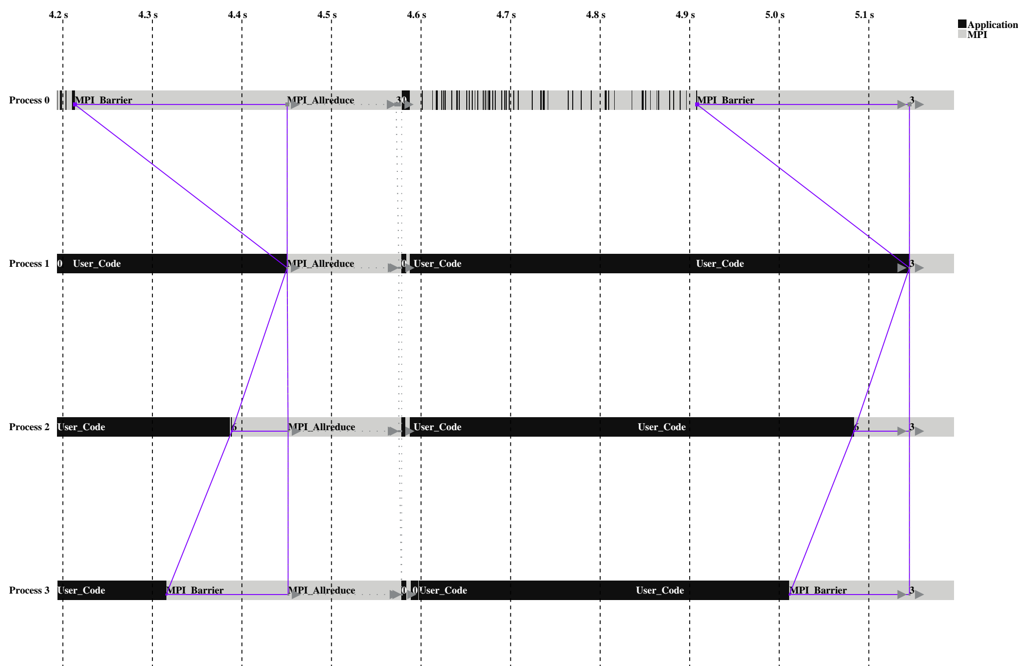
For å illustrere hvordan ProtoMol deler opp tiden mellom beregning av problemet og kommunikasjon av data som alle prosessene trenger, har jeg tatt med figur 4. Figuren er en skjermutskrift hentet fra programmet **vampir** som er et analyseverktøy produsert av Pallas[1]. Figuren illustrerer et utsnitt i tid av en kjøring av BPTI 14281 atomer, periodic med ProtoMol fra 4.2 sekunder som er helt til vestre i figuren til 5.1 sekunder som er helt til høyre. Denne kjøringen ble gjort på 4 noder på ClustIS. Disse nodene er illustert som fire brede horisontale linjer. Den øverste noden har en master-prosess som "styrer" beregningene til de andre nodene. Dette er et utsnitt som utgjør noe over en beregningsiterasjon. Hvis jeg hadde tatt med et større utsnitt i tid, ville dette mønsteret repetert seg omtrentlig på samme måte.

Det svarte området på prosessorstrekene er tiden som programmet bruker til å gjøre ikke-MPI-funksjoner, dette antar jeg er beregning av selve problemet.

Det er mulig å se ut fra figur 4 at nodene 2–4 ovenfra har i dette utsnittet en forskjellig belastning. I dette utsnittet må node 4 vente ca 0.13 sekunder før den kan gå ut av MPI_Barrier. ProtoMol utfører alltid en MPI_Barrier før MPI_Allreduce. Det hadde vært interessant å finne ut om det er nødvendig å gjøre en MPI_Barrier før MPI_Allreduce. Er det alltid nødvendig å gjøre dette? Hvis ikke, er det mulig å oppnå en ytelsesforbedring uten MPI_Barrier? Dette har jeg ikke hatt tid til å gå inn i.

Tabell 6 er et forsøk på se på de globale totalsummene for bruk av tid i ProtoMol i beregningsfasen. Disse dataene er ikke veldig nøyaktige da jeg ikke fant noen måte å få **vampir** til å begynne å måle fra et bestemt

[1] `http://www.pallas.com/e/products/pmb/`



Figur 4: Utsnitt i tid av beregningsprosessen for BPTI 14281, periodic

Antall prosessorer	4		8	
Beregning	15.750	s	16.026	s
MPI_Allreduce	5.597	s	16.720	s
MPI_Barrier	4.369	s	3.376	s
MPI_Iprobe	2.571	s	1.350	s
MPI_Recv	0.165	s	0.340	s
MPI_Bsend	15.399	ms	36.885	ms

Tabell 6: Forandring i bruk av tid når man forandrer antallet prosesser for BPTI 14281, periodic

punkt i koden. Det eneste jeg kunne gjøre var å sette hvor langt etter starten som **vampir** skulle begynne å ta med data fra den genererte loggen. Jeg fikk heller ikke til å kjøre ProtoMol i køsystemet når biblioteker for **vampir** lenket med ProtoMol. Årsaken var at `vampir` sjekket en lisensvariabel som ikke ble eksportert til køsystemmiljøet. Det betyr at disse dataene ikke kan garanteres med samme nøyaktighet som andre målinger i rapporten. Det var heller ikke noen enkel måte å gjøre et snitt av 5 kjøringer. Resultatene i tabell 6 er derfor fra 1 kjøring.

Det er interessant å se fra tabell 6 at det er helt tydelig at den tiden som brukes til å kommunisere med MPI_Allreduce øker betraktelig med antallet noder. Det at tiden som ble brukt til MPI_Barrier minker tror jeg kan ha sammenheng med at det er mindre jobb per node når antall noder øker og dermed vanskeligere og komme i stor utakt. Dette er også en problemstiling som det hadde vært interessant å gå videre på.

7.1.1 Benchmark av MPI_Allreduce på ClustIS

Etter å ha gjort undersøkelsen som resulterte i tabell 6, mente jeg det ville være interessant å se hvilken maksimal ytelse det var mulig å få ut av MPI_Allreduce og i hvilket forhold tiden økte når vi gikk fra 4 til 8 noder. Med denne målingen kunne man muligens forklare den markante økningen i tid til MPI_Allreduce som ble vist i tabell 6.

MPI_Allreduce har rent teoretisk et tidsforbruk som er $O(\log_2 n)$ [6, s. 76] der n er antall noder fordi MPI_Allreduce bruker en butterfly-algoritme. Den relative økning i tidsbruken når man går fra 4 til 8 noder blir da:

$$\frac{\log_2(8)}{\log_2(4)} = 1.5$$

Det tabell 7 viser er resultater fra PBM som er en benchmark laget av Pallas som kan teste alle MPI funksjonene både i MPI-1 [1] og MPI-2 [5] standardene. Dataene i tabell 7 viser hvilken maksimal "hastighet" det er mulig å få ut av MPI_Allreduce på ClustIS. Dette er viktige data fordi det er den type MPI kommunikasjon som det er mye av i ProtoMol. Uten unntak er det slik at hvis man øker antallet noder fra 4 til 8 fikk benchmark-en en tidsøkning på cirka 50%. Det er interessant at MPI_Allreduce faktisk kan yte så optimalt på ClustIS.

Det er viktig at benchmark-verdiene er generert ved ideelle forhold for butterfly algoritmen. Både ved 4 og 8 noder er det balanserte grafer.

Det er viktig å huske at tidsforbruket til MPI_Allreduce vil øke kraftig hvis en eller flere av nodene får en "tyngre" arbeidsoppgave og kommer ut av synkronisering i forhold til resten av nodene, jevnfør figur 4.

bytes	4 noder snitt[μsec]	8 noder snitt[μsec]	$\frac{8\,\text{noder}}{2\,\text{noder}}$
0	0.22	0.21	0.95
4	293.59	447.84	1.53
8	295.83	452.38	1.53
16	299.52	457.98	1.53
32	311.63	475.38	1.53
64	334.91	512.19	1.53
128	373.75	573.30	1.53
256	457.42	700.24	1.53
512	626.71	958.46	1.53
1024	964.34	1470.31	1.53
2048	1468.09	2233.31	1.53
4096	2133.15	3232.66	1.52
8192	3616.31	5457.77	1.51
16384	6446.40	9692.94	1.50
32768	12266.51	18429.22	1.50
65536	24235.30	36183.22	1.49
131072	49025.03	73064.12	1.49
262144	97898.32	146138.70	1.49
524288	194688.93	290100.70	1.49
1048576	386752.84	576692.34	1.49
2097152	769336.50	1148787.80	1.49
4194304	1536710.90	2292240.58	1.49

Tabell 7: `MPI_Allreduce()` benchmark. Første kolonne er antall bytes som overføres, de to neste kolonnene er gjennomsnittstid av 1000 eksekveringer av funksjonen MPI_Allreduce, siste kolonne er forhold mellom tiden brukt for 8 og 4 noder.

Det er ikke så enkelt å få til lik belastning på nodene for slike problemer som ProtoMol løser. Hvis man skal kunne oppnå gode verdier for hastighetsøkning og samtidig forsette å bruke globale funksjoner som MPI_Allreduce, må man finne løsninger på dette problemet. Jeg forsto på Matthey at en annen algoritme brukes når ProtoMol kompileres og kjøres med MPI-2 funksjonene MPI_Put/Get. Disse funksjonene krever ikke en global synkron form for kommunikasjon. Dette fører til at prosessorene ikke behøver å vente så mye på hverandre. Om det er mulig å implementere denne type algoritme på en effektiv måte på ClustIS, har jeg ikke nok kunnskap til å diskutere her.

Det at ProtoMol skalerer bedre på stormaskin kan forklares på flere måter. En er at stormaskiner har et raskere nettverk. Dette vil bety at den ikke bruker så lang tid på å utføre MPI_Allreduce. Hvis jeg har forstått Matthey riktig, brukes ikke denne funksjonen i noen særlig grand når den kjøres på maskiner som har en versjon av MPI som støtter MPI-2 standarden. Hele algoritmen for ProtoMol blir litt annerledes parallellisert og mer asynkron for en superdatamaskin enn den må være når man har MPI-1 implementasjon.

7.2 Mulig skaleringsundersøkelse

Sack [7, s. 10] har undersøkt hvordan et program kan forbedre beregningstiden per problemenhet. Han øker problemstørrelsen, men holder antall prosesser konstant. Jeg mener at dette hadde vært en interessant undersøkelse for å se hvordan ProtoMol ville ha prestert i en slik testen. Det viste seg at det var ikke så lett å finne en klasse problemer hvor bare antallet atomer varierer fordi da det er mange parametere som varierer når man varierer antallet atomer. Jeg mener at det hadde vært en meget interessant undersøkelse å gjøre og jeg prøvde tidlig å finne noen som kunne hjelpe meg med å generere slike datasett. Det lot seg ikke gjøre.

Ut fra mine erfaringer med ProtoMol tror jeg at programmet ville ha hatt en kurve som ligner på den som presenteres på [7, s. 10], men jeg tror ikke kurven ville komme så langt ned eller falle så raskt. Dette er bare antagelser og må ikke taes for noe mer enn akkurat det.

8 Konklusjon

ProtoMol er et program som utfører molykelærdynamiske simuleringer. Her vil jeg konkludere om programmets evne til å kjøre effektivt på klyngen ClustIS.

De slutninger jeg gjør i denne konklusjonen er basert på den versjonen som jeg har hatt og ikke den versjonen som er blitt kjørt og testet i doktorgraden til Matthey [4]. En viktig forskjell på disse to versjonene er at den versjonen jeg har testet bruker ikke MPI_Put/Get, men bruker andre MPI kall. Siden det ikke var jeg som endte opp med å skrive om programmet, er jeg ikke sikker på om det er andre vesentlige forskjeller som kan innvirke på resultatene.

De målinger jeg har gjort på ProtoMol viser at ProtoMol ikke egner seg til å kjøre på klynger av den typen som ClustIS er. Jeg har sannsynliggjort at det kan være fordelingen av beregningsbelastningen mellom nodene som kan være grunnen at ProtoMol ikke klarer effektivt å nyttiggjøre seg den regnekraften den får tilgang til når ProtoMol kjøres i parallell.

Jeg har også belyst måter jeg tror det er mulig å få ProtoMol til å yte bedre på ClustIS. Uten å ta stilling til vanskelighetsgraden er to forskjellige forslag til implementasjoner:

- Å gjøre en bedre lastbalansering mellom nodene og dermed få en mer lik beregningstid mellom nodene.
- Prøve og lage en mer asynkron måte å utveksle data på som gjør at man ikke så ofte vil måtte vente på at andre prosesser skal bli ferdige.

For å kunne si noe mer om mulighetene til å implementere en eller begge av disse forslagene trengs det grundig kunnskap om mulige løsningsmetoder for den typen problemer som ProtoMol løser. Dette ligger utenfor målsetningen til denne oppgaven. I denne sammenheng er det interessant at den versjonen som Matthey brukte har MPI_Put/Get som ikke er globale operasjoner. Denne versjonen skalerer bedre. Er det mulig å omskrive algoritmene i denne versjonen til ikke å bruke MPI_Put/Get, men istedenfor bruke funksjoner som ikke har one-sided communication. På denne måten kan man muligens oppnå en større fleksibilitet i kommunikasjonen og derigjennom få større utnyttelsesgrad.

En viktig forskjell mellom denne rapporten og doktorgraden til Matthey [4] er måten tid er målt på. Dette er tidligere drøftet i punkt 6.3.5. Matthey måler brukt tid på prosessoren og jeg måler tiden som wall clock time. Dette gjør at målingene i denne rapporten og i doktorgraden ikke er direkte sammenlignbare.

Etterord

Jeg takker Anne C. Elster for veiledningen jeg fikk. Paul Sack ga meg en nytting innføring i dette feltet. Thierry Matthey hjalp meget mye med ProtoMol og gav inspirasjon til å se nærmere på tidsmålingsbegrepet. Jeg er Zoran

Constantinescu-Fülöp meget takknemlig for mange timers hjelp med kompilering og MPI-kjøreproblemer. Jeg vil gjerne takke Ketil og Eivind Østvold for å rette opp mitt meget dyslektiske førsteutkast av rapporten. Den endelige rapporten er selvfølgelig mitt ansvar mitt alene.

Referanser

[1] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical report, 1995.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, sep 1996.

[3] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[4] T. Matthey. *Framework Design, Parallelization and Force Computation In Molecular Dynamics.* PhD thesis, Department Of Informatics, University of Bergen, 2002.

[5] Message Passing Interface Forum. MPI-2.0: Extensions to the Message-Passing Interface. Technical report, 1997.

[6] Peter S. Pacheco. *Parallel programing with MPI*, chapter 5.6 Allreduce, page 76. Morgen Kaufmann Publishers, Inc., 1997.

[7] Paul Sack. Preformanc alysis of a phisics application on a pc cluster an a supercomputer. Tilgjenelig fra: `http://www.stud.ntnu.no/~sack/perf.pdf`.

A Plassering av målepunkter for tidtaking

Her har jeg tatt med den delen av `main()` som viser hvor jeg har plassert ut målingene.

```
//  5.  Run the simulation
int    my_rank  = 0;
double start    = 0.0;
double slutt    = 0.0;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

theSim.run();

MPI_Barrier(MPI_COMM_WORLD);
slutt = MPI_Wtime();
if(my_rank==0){
  printf(" %f", slutt - start );
}
```

`main()` inneholder mer, men dette er den interessante delen for å spesifisere hvordan jeg målte tid.

B ProtoMols kompilering på ClustIS

Her presenterer jeg de tilleggene jeg var nødt til å gjøre i `configure.in` filen for å få kompilert med MPICH og LAM MPI.

B.1 MPICH

Nedenfor har jeg tatt med det utsnitt fra `configure.in` filen som relaterer seg til kompilering av ProtoMol med MPICH på ClustIS.

```
#
# Linux x86 g++ flags MED MPICH
#
WANT_LINUX_GCC_MPICH_FLAGS=0
AC_MSG_CHECKING([if want preset Linux x86 AMD K6  related g++ flags])
AC_ARG_WITH(linux_gcc_mpich, [  --with-linux-gcc-mpich   Use g++ and MPICH ])
if test -n "$with_linux_gcc_mpich"; then
    if test "$with_linux_gcc_mpich" = "yes"; then
        WANT_LINUX_GCC_MPICH_FLAGS=1
    else
        WANT_LINUX_GCC_MPICH_FLAGS=0
    fi
fi
if test "$WANT_LINUX_GCC_MPICH_FLAGS" = 1; then
    AC_MSG_RESULT([yes])
    CC="mpicc"
    CFLAGS="-Wall -O9 -ffast-math -funroll-loops  -DVECTORTYPE_NVDOUBLE \
            -DTEMPLATE_IN_HEADER  -DHAVE_MPI -DMPI_NO_CPPBIND"
    CXX="mpiCC"
    CXXFLAGS="-Wall -O9 -ffast-math -funroll-loops -DVECTORTYPE_NVDOUBLE \
             -DTEMPLATE_IN_HEADER -DHAVE_MPI -DMPI_NO_CPPBIND"
    LIBS="-lm -lnsl"
    DEPFLAGS=""

    AC_SUBST(CC)
    AC_SUBST(CFLAGS)
    AC_SUBST(CXX)
    AC_SUBST(CXXFLAGS)
    AC_SUBST(LIBS)

else
```

```
    AC_MSG_RESULT([no])
fi
```

B.2 LAM MPI

Under viser jeg det utsnittet fra `configure.in` filen som jeg la til for å kunne kompilere med LAM MPI.

```
#
# Linux x86 g++ flags MED MPI(lam)
#
WANT_LINUX_GCC_LAMMPI_DEBUG_FLAGS=0
AC_MSG_CHECKING([if want preset Linux x86 AMD K6 debug related g++ flags])
AC_ARG_WITH(linux_gcc_lammpi_debug, [  --with-linux-gcc-lammpi      \
                                       Use g++ and LAM/MPI for compilation \
                                       along with debugging preset flags])
if test -n "$with_linux_gcc_lammpi"; then
    if test "$with_linux_gcc_lammpi" = "yes"; then
        WANT_LINUX_GCC_LAMMPI_FLAGS=1
    else
        WANT_LINUX_GCC_LAMMPI_FLAGS=0
    fi
fi
if test "$WANT_LINUX_GCC_LAMMPI_FLAGS" = 1; then
    AC_MSG_RESULT([yes])
    CC="gcc"
    CFLAGS="-Wall -O9 -I/opt/lam/include -ffast-math -funroll-loops
           -DVECTORTYPE_NVDOUBLE -DTEMPLATE_IN_HEADER  -DHAVE_MPI"
    CXX="g++"
    CXXFLAGS="-Wall -O9 -I/opt/lam/include -ffast-math -funroll-loops \
             -DVECTORTYPE_NVDOUBLE -DTEMPLATE_IN_HEADER  -DHAVE_MPI"
    LIBS="-lm -lnsl -L/opt/lam/lib -lmpi -llam"
    DEPFLAGS=""

    AC_SUBST(CC)
    AC_SUBST(CFLAGS)
    AC_SUBST(CXX)
    AC_SUBST(CXXFLAGS)
    AC_SUBST(LIBS)

else
```

```
    AC_MSG_RESULT([no])
fi
```