Universitetet i Oslo
Institutt for informatikk

Optimalisering av parallelle Diffpack simuleringer

Håvard Wall

(haavardw@ifi.uio.no)

Hovedfagsoppgave

30. juli 2003

Innhold

Kapittel 1

Innledning

1.1 Bakgrunn

Det er alltid etterspørsel etter mer regnekraft. Man ønsker å løse større og vanskeligere problemer, eller løse eksisterende problemer raskere. Nye og raskere algoritmer eller maskinvare tar ofte lang tid å utvikle. I tillegg finnes det en rekke fysiske begrensninger for hvor rask en prosessor kan bli. For eksempel er signalhastigheten mellom de ulike komponentene i en prosessor begrenset av lysets hastighet.

Bruk av flere prosessorer samtidig, i parallell, kan være en raskere og billigere løsning å øke regnekraften på. Ved å bruke P prosessorer kan man potensielt utføre P ganger så mange regneoperasjoner på en gitt tid. Ofte vil man også kunne bruke P ganger så mye minne.

Et parallelt program er et program som bruker flere prosessorer samtidig for å løse én oppgave. Å skrive et parallelt program innebærer nye problemstillinger i forhold til et serielt program. Først og fremst må oppgaven deles opp og fordeles på de ulike prosessorene. En slik oppdeling inneholder vanligvis en rekke ulike kompromisser. For eksempel kan det ofte være mulig å redusere kommunikasjon mellom prosessorene dersom man isteden gjøre noe ekstra beregning på hver prosessor. Hvorvidt dette er lønnsomt vil være avhengig av forhold som kommunikasjonshastigheten mellom prosessorene og den enkelte prosessoren sin regnekraft.

1.2 Problemstilling

Denne oppgaven vil se på parallellisering av elementmetoden, en populær numerisk metode for å diskretisere og løse partielle differentiallikninger. Jeg tar utgangspunkt i en eksisterende parallell implementasjon i Diffpack og vil foreslå optimaliseringer i forhold til denne. Forbedringsforslagene er implementert og testet, og resultatene blir presentert. Vi vil se at disse optimaliseringsforslagene kan gi en vesentlig forbedring av den parallelle kjøretiden.

1.3 Kodeeksempler

Det blir gitt en rekke kodeeksempler i denne oppgaven. Disse er ment som skisser og vil som regel utelate en rekke detaljer som er nødvendig i en reell implementasjon. Eksemplene vil bygge på C/C++. Alle vektorer vil være av C type og vil derfor ha 0 som første indeks. Dette er gjort for å minimere kravet til forkunnskaper om Diffpack. De reelle implementasjonene vil derimot bygge på de ulike vektor-klassene i Diffpack der første element har indeks 1.

1.4 Oppgavens oppbygning

Kapittel 2 gir en kort overblikk over numerisk løsning med elementmetoden. Det blir her gjort rede for de viktigste beregningsmessige operasjonene som er tema gjennom senere i oppgaven. Jeg forutsetter allerede kjennskap til elementmetoden, og det blir ikke redegjort for alle detaljer. For en innføring i elementmetoden henvises leseren til annen litteratur (for eksempel [25]).

Kapittel 3 gir en innføring i grunnleggende begreper knyttet til parallellprogrammering. Jeg vil ta opp ulike aspekter ved parallell maskinvare og parallelle programeringsteknikker. Videre presenteres et lite analytisk rammeverktøy. Dette vil bli brukt til å evaluere ulike parallelle strategier senere i oppgaven. Til slutt presenteres MPI [10, 27]. MPI er et funksjonsgrensesnitt som blir brukt til kommunikasjon mellom ulike prosessorer i parallelle programmer.

Kapittel 4 diskuterer hvordan parallellisering av elementmetoden er gjort i Diffpack. Det blir her lagt vekt på å finne uttrykk for den parallelle kvaliteten av Diffpack sin implementasjon. Kapittel 5 kommer så med optimaliseringsforslag i forhold til Diffpack. Disse blir analysert og forsøkt sammenliknet med valgene gjort i Diffpack. Kapittel 6 utfører så noen tester av de nye optimaliseringsforslagene.

Kapittel 7 oppsummerer de viktigste resultatene og kommer med forslag til videre arbeid.

Kapittel 2

Numerisk simulering

En numerisk simulering tar utgangspunkt i en matematisk beskrivelse av et problem i form av en eller flere likninger. Disse likningene må diskretiseres og lineæriseres for å kunne løses på en datamaskin. Elementmetoden er en populær metode for diskretisering av partielle differentiallikninger. Diskretiseringen vil som regel resultere i et lineært likningssystem. Det eksisterer en rekke ulike metoder for å løse slike på en datamaskin.

Dette kapittelet går raskt igjennom elementmetoden og de viktigste egenskapene ved en lineær likningsløser. Jeg vil ikke ikke gå igjennom alle detaljer, eller være helt matematisk korrekt i de punktene som blir gjennomgått. Allerede kjennskap til elementmetoden og likningsløsere er en forutsetning, og dette kapittelet er kun ment som en gjenoppfriskning. Jeg vil legge vekt på å peke på de punktene som blir viktige for diskusjonen rundt parallellisering.

2.1 Elementmetoden

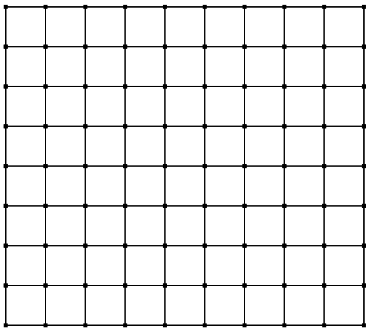
Utgangspunktet for elementmetoden er at man skal løse en likning over et underliggende domene, for eksempel et areal eller et volum. Dette domenet deles opp i en rekke mindre biter kalt *elementer*. Figur 2.1 på neste side viser en skisse av et rektangulært domene delt opp i kvadratiske elementer. Hvert element blir representert ved hjelp av en matematisk funksjon og et sett med kontrollpunkter. I figur 2.1 ligger kontrollpunktene i elementenes hjørner. Jeg vil videre i denne oppgaven referere til en slik representasjon av det underliggende domenet som et *grid*. Gridets kontrollpunkter vil videre bli kalt gridpunkter.

Et gridpunkt vil normalt assosieres med flere elementer. I figur 2.1 er de fleste gridpunktene assosiert med fire elementer. Dersom to elementer har ett eller flere felles gridpunkter, kalles disse for *naboelementer*.

Anta en likning som skal løses for en ukjent funksjon u. Den ukjente funksjonen blir i elementmetoden tilnærmet som en vektet sum av basisfunksjoner,

$$u \approx \hat{u} = \sum_{j=1}^{K} u_j N_j$$

der N_j angir en basisfunksjon, u_j den tilsvarende vekten og K er antall basisfunksjoner. Basisfunksjonene blir konstruert slik at de er forskjellig fra null kun



Figur 2.1: Rektangulært domene delt opp i kvadratiske elementer. Gridpunktene ligger i elementenes hjørner.

innenfor et lite område.

Man søker så å finne et estimat for u med minimal feil. Dette gjøres ved en eller annen form av vektete residualer[25], som innebærer at man må integrere et uttrykk med $\hat{u}$ over hele domenet. Dette vil normalt gi et lineært likningssystem med u_j som ukjente. Man velger vanligvis basisfunksjonene slik at de ukjente u_j tilsvarer den estimerte funksjonsverdien $\hat{u}$ i gridpunkt j.

Det at basisfunksjonene kun er lokalt forskjellig fra null, gjør det mulig å foreta integrasjonen på elementnivå, det vil si at man integrerer stykkevis for hvert element. Ved integrasjon på elementnivå sitter man igjen med et lite likningssystem for hvert element. Det globale likningssystemet kan da uttrykkes ved en form for summasjon av de elementvise likningssystemene (se avsnitt 2.3.4 i [25] for detaljer). Hvert element gir et bidrag til likningen for de gridpunktene den består av.

2.2 Løsning av lineære likningssystem

Det oppstår en såkalt "kobling" mellom to gridpunkter dersom de kan assosieres med et felles element. Ser vi på matrisen i det resulterende likningsystemet, betyr dette at det element på rad i og i kolonne j kun er forskjellig fra null dersom det er en kobling mellom gridpunktene assosiert med i og j. Normalt vil det totale antall gridpunkter være svært høyt, mens antall gridpunkter per element er lite. Dette gir en matrise som er *stor* og *glissen*.

Et stort og glissent likningsystem kan først og fremst utnyttes til å komprimere matrisen slik at koeffisientene med verdi null ikke lagres eksplisitt. Matriseoperasjoner kan da effektiviseres ved kun å traversere koeffisientene forskjellig fra null. Matriseformatet som blir brukt i denne oppgaven, er det såkalte *Compressed Row Storage*[2]. Figur 2.2 på neste side viser et eksempel på lagring med CRS. Elementene forskjellig fra null blir lagret i en én-dimensjonal tabell. I tillegg brukes to tabeller for å beskrive rad- og kolonne-numre. Figur 2.2 viser også matrise-vektor-produktet på en matrise lagret i CRS-format. Denne koden tar to ekstra parametere som angir et intervall med rader som skal være med i produktet. Jeg kommer til å utnytte denne funksjonaliteten senere i oppgaven. Et vanlig fullt matrise-vektor-produkt vil bruke 0 som `rowstart` og $n-1$ som

$$\begin{pmatrix} 4 & -1 & 0 & 0 \\ 0 & 4 & 0 & 1 \\ 0 & 0 & 4 & 0 \\ 0 & 1 & 0 & 4 \end{pmatrix}$$

vals	4	-1	4	1	4	1	4
col_ind	1	2	2	1	3	2	4
row_ptr	1	3	5	6	8		

```
1  void matvec_crs( int rowstart, int rowend, CRS *A, double *x, double *y )
2  {
3     for ( int i=rowstart; i <= rowend; i++ ) {
4        y[i] = 0;
5        for ( int k=A->row_ptr[i]; k < A->row_ptr[i+1]; k++ ) {
6           y[i] += A->vals[k] * x[ A->col_ind[k] ];
7        }
8     }
9  }
```

Figur 2.2: Eksempel på lagring av matrise i Compressed Row Storage. Tabellen vals lagrer her matrise-elementene forskjellig fra null, col_ind lagrer elementenes kolonnenummer mens verdi nummer i i row_ptr lagrer indeksen der første element i rad i befinner seg i vals. Se [2] for en mer detaljert beskrivelse av dette formatet. Nederst vises en skisse av matrise-vektor-produktet på en matrise lagret i CRS-format. Produktet blir her gjort fra og med rad rowstart til og med rowend.

`rowend` der n er vektorenes lengde.

Løsning av store og glisne likningssystemer blir så godt som alltid gjort ved hjelp av iterative metoder. Spesielt brukes en klasse algoritmer kalt Kyrlov underroms løsere (engelsk: Kyrlov subspace solvers). Disse metodene består stort sett av tre typer operasjoner[2, 5, 1]: Matrise-vektor-produkt, indreprodukt og vektoroppdateringer. Med en vektoroppdatering menes i denne oppgaven stort sett det å addere sammen to vektorer.

Konvergenshastigheten til en iterativ løser er som regel avhengig av kondisjonstallet til matrisen[26]. En metode for å redusere dette kondisjonstallet er å multiplisere likningssystemet med en såkalt prekondisjoneringsmatrise M^{-1}. Isteden for å løse systemet $Ax = b$ løser man da systemet $M^{-1}Ax = M^{-1}b$. M^{-1} må velges på en smart måte, slik at kondisjonstallet til $M^{-1}A$ blir lavere enn kondisjonstallet til A.

Kapittel 3

Parallellisering

Før jeg i neste kapittel ser på hvordan numerisk simulering med elementmetoden er parallellisert i Diffpack, vil jeg i dette kapittelet se generelt på noen punkter rundt parallellisering. Jeg vil se på parallell maskinvare, begreper rundt parallell programmering, definere et lite analytisk rammeverktøy og avslutte med å presentere MPI. MPI er et funksjonsgrensesnitt for parallell kommunikasjon som blir brukt i Diffpack.

Dette kapittelet gir kun et grovt overblikk over de viktigste punktene som er relevante for denne oppgaven. Se for eksempel [23] for en grundigere innføring i parallellisering.

3.1 Parallell maskinvare

Det finnes et stort spekter av ulik maskinvare brukt i parallelle beregninger. De maskinene som er brukt i denne oppgaven er såkalte MIMD-maskiner (Multiple Instructions Multiple Data). Dette betyr at de er sammensatt av flere uavhengige serielle prosessorer. Hver prosessor eksekverer egne instruksjoner og opererer på egne data uavhengig av de andre. Dette er i motsetning til SIMD-maskiner (Single Intruction Multiple Data) som består av spesielle prosessorer som alle må utføre de samme instruksjonene, men kan operere på ulike data.

Rask kommunikasjon er viktig for parallelle programmer. Man deler gjerne kommunikasjonshastighet opp i *forsinkelse* (engelsk: latency) og *båndbredde* (engelsk: bandwidth). Forsinkelse angir tiden det tar å klargjøre en melding for å bli sendt. Båndbredde angir maksimalt antall bit eller byte det er mulig å overføre per sekund, når man ser bort ifra forsinkelsen.

Minnet på parallelle maskiner kan være *delt* eller *distribuert*. Med delt minne deler prosessorer samme fysiske minne, mens med distribuert minne er minnet fysisk fordelt på ulike prosessorer. Med delt minne får man en båndbredde lik minnets båndbredde, som gjerne ligger nær prosessorens hastighet, og forsinkelsen er liten. Med distribuert minne må maskinene kommunisere gjennom en eller annen form for nettverk. Slik kommunikasjon har typisk mye høyere forsinkelse og lavere båndbredde enn ved bruk av delt minne. De maskinene som er brukt i denne oppgaven har hybride løsninger. Minnet er distribuert på små grupper av prosessorer, mens minnet er delt innenfor hver gruppe.

Nettverk brukt til kommunikasjon mellom prosessorene i en parallell maskin, kan organiseres med ulik topologi. En mulig topologi er å organisere prosessorene i en ring der hver prosessor kun kan kommunisere med 2 andre prosessorer. Tiden det tar å få sendt en melding mellom to prosessorer, vil da variere mellom ulike par av prosessorer. Hyperkube er et flerdimensjonalt nettverk med nøyaktig 2 prosessorer i hver dimensjon[23]. En hyperkube i $d+1$ dimensjoner defineres som en sammenkobling av to hyperkuber i d dimensjoner, der en hyperkube i én dimensjon er to prosessorer med en enkel sammenkobling. En tredje løsning er å koble maskinene i en stjerne. Prosessoren i sentrum kan da kommunisere med alle andre prosessorer. De andre prosessorene kan kun kommunisere direkte med prosessoren i midten. Av de maskinene jeg har brukt, er én koblet som en hyperkube, mens en annen har kommunikasjonskarakteristika som ligner på en blanding av en ring og en stjerne-kobling.

3.2 Parallellprogrammering

Kommunikasjon mellom prosessorer vil i denne oppgaven foregå med *meldingsutveksling*. Med meldingsutveksling blir data som skal kommuniseres (meldingen) eksplisitt sendt til en spesifisert prosessor gjennom et funksjonskall. Meldingsutveksling står i motsetning til programmering med "delt minne" der de ulike prosessorene (ihvertfall tilsynelatende) har et felles adresserom, og kommuniserer med å skrive og lese til spesielle delte minneområder. Programmering med tråder er et eksempel på programmering med delt minne. Både meldingsutveksling og programmering med delt minne kan gjøres uavhengig av om det fysiske minnet er delt eller distribuert.

Den vanligste fremgangsmåten når man parallelliserer numeriske simuleringer, er å skrive *ett* program som vil bli kjørt på alle prosessorene. Det blir startet en kopi av programmet på hver prosessor og hvert program får sitt eget adresserom (egne variabler). For å skille mellom prosessorene, gis hvert program et unikt prosessornummer. All parallell kode i denne oppgaven bruker denne fremgangsmåten. Alle parallelle kodeeksempler antas derfor å kjøre på flere prosessorer samtidig, men med lokale variabler. Alternativet til denne fremgangsmåten er å skrive flere programmer, gjerne ett for hver prosessor. Denne fremgangsmåten er vanligere innenfor såkalt distribuert programmering.

I en parallell simulering må arbeid fordeles på de ulike prosessorene. En vanlig teknikk for numeriske simuleringer er å dele opp det underliggende domenet i mindre subdomener eller partisjoner. Hver prosessor blir så ansvarlig for å beregne løsningen i én eller flere partisjoner. Arbeidet med å dele opp i partisjoner vil videre i denne oppgaven bli kalt for *partisjonering*. For enkelhets skyld vil jeg videre i denne oppgaven anta at hver prosessor kun er ansvarlig for én partisjon.

For å kunne minimere eksekveringstiden til et parallelt program, er det viktig å utnytte alle de tilgjengelige prosessorene så godt som mulig. Man må derfor fordele arbeidet (lasten) så likt som mulig mellom prosessorene. En konkret fordeling av arbeid kalles en *lastbalansering*. I denne oppgaven vil det være en nær sammenheng mellom en god partisjonering og en god lastbalansering. Vi vil se nærmere på dette i avsnitt 4.2.

3.3 Analyse

Et visst analytisk rammeverk er nødvendig for å kunne si noe om kvaliteten på en parallell implementasjon. Dette vil bli brukt i senere kapitler når vi ser på ulike alternative implementasjoner.

N vil angi antall gridpunkter i det globale problemet. Dette er også lik antall ukjente i likningssystemet. De ukjente blir i en parallell implementasjon blir fordelt på de ulike prosessorene. n vil angi antall gridpunkter eller ukjente forbundet med én prosessor. Normalt vil dette tallet variere fra prosessor til prosessor. Tiden til en parallell implementasjon vil da som regel være gitt av den prosessoren med flest ukjente. I de beregningene jeg vil foreta ser jeg bort fra dette, og antar n lik for alle prosessorer.

Symbolet T vil angi tid. Tiden til en kodebit vil være en funksjon av ulike parametere som antall gridpunkter og antall prosessorer. P vil angi antall prosessorer som er i bruk. Som regel er jeg interessert i tiden som funksjon av antall prosessorer i bruk. Dette angis med $T(P)$. $T(1)$ er et spesialtilfelle og angir tiden brukt med en seriell implementasjon, i motsetning til en parallell implementasjon med bare én prosessor.

I forhold til et serielt program vil et parallelt program som regel bruke tid på både ekstra beregning og kommunikasjon. Merarbeid (engelsk: overhead) er et mål på dette og defineres i denne oppgaven som $PT(P) - T(1)$. Jeg vil ikke bruke denne definisjonen direkte, men løst omtale merarbeid som ekstra arbeid forbundet med parallellisering. Jeg vil i kapittel 5 foreslå teknikker for å redusere det eksisterende merarbeidet i Diffpack.

Speedup sier noe om hvor mye raskere et program kjører for et gitt antall prosessorer. Dette defineres som $S = \frac{T(1)}{T(P)}$. Ideelt sett ønsker vi $S = P$. Når dette er tilfelle har vi en lineær speedup. Normalt har vi at $S < P$. Dersom $S > P$ har vi superlinær speedup. Dette vil normalt tyde på en suboptimal seriell implementasjon, eller effekter grunnet økt tilgjengelig minne.

Skalerbarhet er et annet viktig mål på kvaliteten til et parallelt program. Med at et program skalerer godt, mener jeg i denne oppgaven at det ved økende antall prosessorer er mulig å opprettholde en god parallel effektivitet. Normalt vil merarbeidet øke dersom antall prosessorer øker, slik at man må øke problemstørrelsen/antall gridpunkter for å kunne opprettholde en gitt effektivitet.

3.4 MPI

MPI er et standarisert[10, 27] funksjonsgrensesnitt for parallell kommunikasjon som bygger på meldingsutveksling. MPI er støttet på de fleste parallelle systemer. Bruk av MPI er derfor svært utbredt og portabelt. For å kunne være både portabilit og effektivt, inneholder MPI svært mange ulike funksjoner. Disse er delvis overlappende og på et relativt lavt abstraksjonsnivå. Jeg vil i dette avsnittet bare diskutere den funksjonaliteten som er aktuell for testene i denne oppgaven.

I de simuleringene jeg skal utføre, er det to former for kommunikasjon som er viktig. Prosessor-til-prosessor kommunikasjon og global reduksjon. En global reduksjon oppnås i MPI med funksjonskallet MPI_Allreduce. I mine simuleringer vil denne summere delresultater utregnet på hver enkelt prosessor, og

etterpå distribuere svaret til alle prosessorer. Dette brukes i det parallelle indreproduktet.

Prosessor-til-prosessor kommunikasjon går ut på å utveksle data mellom par av prosessorer. Dette blir blant annet brukt i matrise-vektor-produktet. MPI tilbyr to grunnleggende funksjoner for slik kommunikasjon: MPI_Send og MPI_Recv. I tillegg tilbyr MPI en rekke varianter av disse som gir ulike optimaliseringsmuligheter. Før jeg ser nærmere på optimaliseringsmulighetene, ser jeg først på de ulike aspektene ved en implementasjon av MPI.

3.4.1 Implementasjon av MPI

MPI-spesifikasjonen[10, 27] røper en del om hvordan MPI tenkes implementert. Beskrivelsen i dette avsnittet bygger i tillegg på MPI-implementasjonene MPICH[24, 13, 14] og LAM[7].

En enkel implementasjon av MPI for å sende og motta meldinger, kan tenkes å bestå av følgende elementer:

- Et buffer for midlertidig lagring av brukerens data til sending og mottak.
- En kø av konvolutter for å holde styr på meldinger som er klare til å bli sendt eller mottatt. En konvolutt må minst bestå av en entydig adresse til mottaker/avsender, en peker til hvor meldingens data ligger i bufferet, samt lengden på dataene.

I praksis vil konvoluttkøen som regel bestå av statisk allokert minne. Dette er altså en begrenset ressurs. Databufferet vil som regel også være statisk allokert, men brukeren har i MPI mulighet til å opprette ekstra bufferplass. Jeg vil ikke benytte meg av dette i denne oppgaven.

Jeg antar for enkelhets skyld at sending og mottak bruker separate buffere, og har separate køer av konvolutter.

Sending

Følgende kan tenkes å skje ved når en melding skal sendes: En ny konvolutt må opprettes og fylles ut, og meldingsdata må kopieres til bufferet. Videre trenger man en mekanisme til selve sendingen. Man har to hovedkategorier for slike mekanismer:

- Send konvolutt til mottaker, og vent på bekreftelse fra mottaker om at sending kan starte før meldingen blir sendt. Dette kalles håndhilsing eller rendezvous.
- Send konvolutt og data umiddelbart. Anta at mottaker er klar til å motta, eller er i stand til å mellomlagre meldingen. Dette kalles en ivrig protokoll (engelsk: eager).

Bruk av ivrig send har den ulempen at mottakers buffer eller konvoluttkø kan bli fullt. Håndhilsing har ikke dette problemet. På den annen side krever selve håndhilsingen ekstra kommunikasjon i forhold til ivrig send.

Mottak

Ved mottak av meldinger finnes det også to hovedteknikker:

Spørring (engelsk: polling) implementeres med en løkke som stadig sjekker etter nye meldinger.

Avbrudd går ut på å fange opp et signal når en melding først kommer. Et slikt signal blir som regel gitt av maskinvaren eller operativsystemet.

Det å sette opp en applikasjon til å håndtere avbrudd krever som regel mer arbeid enn spørring. På den annen side kan det gå mindre tid fra en melding faktisk har kommet, til applikasjonen får vite om det. Dette skyldes avbruddsmekanismen i operativsystemet, som vil avbryte eventuelle andre kjørende prosesser for å håndtere avbruddet[37]. Operativsystemet kan da velge å starte prosessen som mottok meldingen.

Så snart en melding er klar til å mottas, kan konvolutten puttes i mottattkøen og meldingen kopieres til bufferet.

3.4.2 Prosessor-til-prosessor kommunikasjon

MPI tilbyr en rekke ulike variasjoner av MPI_Send og MPI_Recv for å kunne åpne for optimaliseringer. For eksempel kan man tenke seg systemer med en ekstra spesialisert prosessor som kan håndtere kommunikasjon. Et slikt system har mulighet til å foreta beregninger samtidig som kommunikasjon pågår. MPI er designet med tanke på at det skal være mulig å utnytte slike optimaliseringer.

De alternative funksjonene stiller visse krav til kontekst. MPI deler opp i fire potentielt forskjellige modi for en melding:

Standard modus innbefatter de vanlige kallene MPI_Send og MPI_Recv. I standard modus overlater man mest mulig kontroll til implementasjonen av MPI. Denne kan for eksempel velge å sende meldingen din med én gang (ivrig send), buffre den, eller vente på at mottaker er klar til å ta imot. Spesifikasjonen[10, kap3.4] foreslår at standard modus er det samme som klar modus for små meldinger, og synkron modus ellers. Disse blir forklart nedenfor.

Buffret modus garanterer at en sending ikke blokkerer. Enten blir meldingen sendt med en gang, eller så blir meldingen buffret og sendt på et senere tidspunkt. Det gjøres ingen garantier mot at tilgjengelig buffer kan brukes opp og oversvømmes.

Synkron (engelsk: synchronous) sending garanterer å blokkere inntil mottaker har begynt å *prosessere* meldingen. Dette innebærer en form for håndhilsing og garanterer at en passende MPI_Recv hos mottaker har blitt kalt.

Klar (engelsk: ready) modus er den mest aggressive. Bruk av klar sending kan bare brukes dersom mottaker er garantert klar til å motta, dvs at en passende MPI_Recv allerede er kalt. Denne modusen legger tydelig til rette for ivrig-protokollen. Sender kan potensielt hoppe over både buffring og håndhilsing.

I tillegg kan hver av de forskjellige modi være *blokkerende* eller *ikke-blokkerende*. Etter et blokkerende kall er du garantert å kunne gjenbruke de datastrukturer du har gitt til MPI. Ved sending innebærer dette at data enten har blitt buffret (kopiert), eller at meldingen er sendt. Ved mottak garanterer dette at den motatte meldingen er skrevet til datastrukturene.

Etter et ikke-blokkerende kall er du kun garantert at en passende konvolutt er generert når funksjonskallet returnerer. Du er ikke garantert noe om datastrukturenes tilstand. Ved en ikke-blokkerende sending vil MPI da kunne unngå buffring fullstendig ved at pekeren i konvolutten kan peke direkte på brukerens databuffer. Tilsvarende kan buffring unngås ved mottak dersom mottaket ble registrert *før* en forespørsel om sending.[1]

Til slutt tilbyr også MPI *persistent* kommunikasjon. Ved opprettelse av en persistent kommunikasjon, oppretter du en konvolutt. Denne konvolutten kan så gjenbrukes flere ganger. I tillegg er all persistent kommunikasjon implisitt ikke-blokkerende. Ved gjentatt bruk av en persistent kommunikasjon kan du da potensielt spare arbeidet ved opprettelse av en konvolutt, og muligens også buffring.

[1] Dette gjelder uavhengig om mottaket er blokkerende eller ikke-blokkerende.

Kapittel 4

Parallellisering av elementmetoden

Kapittel 2 gikk igjennom de viktigste punktene ved numerisk simulering med elementmetoden mens kapittel 3 gikk igjennom noen grunnleggende temaer for parallellprogrammering. Dette kapittelet ser på hvordan elementmetoden er parallellisert i Diffpack. Denne informasjonen bygger på to artikler[4, 5] og et kapittel i en bok om avanserte emner i Diffpack[6].

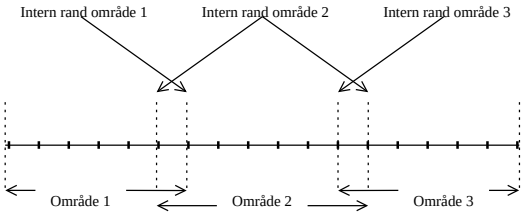
4.1 Innledning

Et viktig mål ved design av den parallelle delen av Diffpack, har vært å kunne beholde så mye som mulig av det eksisterende serielle, og godt utprøvde, klassebiblioteket. I tillegg til redusert utviklingstid, har dette gjort det svært enkelt for brukere å parallellisere allerede eksisterende serielle simuleringer.

Diffpack tilbyr parallellisering på to ulike måter. Den ene metoden blir kalt en simulator-parallell tilnærming. Denne tar i bruk såkalte domene-dekomposisjons-metoder[35] (engelsk: Domain Decomposition Methods) der man matematisk beskriver hvordan man iterativt kan bruke lokale serielle simuleringer parallelt på flere subdomener. Jeg vil ikke se på bruk av denne tilnærmingen.

Den andre metoden blir kalt for en lineæralgebra tilnærming. Denne går ut på å parallellisere de grunnleggende punktene i simuleringen hver for seg. Diskretiseringen blir parallellisert som et eget punkt, mens likningsløsningen deles opp i vektoroppdatering, indreprodukt og matrise-vektor-produkt. Hver del blir parallellisert ved å først kjøre den tilsvarende serielle koden parallelt på alle prosessorer. Deretter brukes parallell-spesifikk kode for å sikre et korrekt globalt resultat.

De neste avsnittene vil først diskutere hvordan Diffpack fordeler arbeid på prosessorene ved å partisjonere det underliggende domenet. Videre forklares i mer detalj hvordan Diffpack's lineæralgebra tilnærning er implementert. Neste kapittel vil se på hvordan denne tilnærmingen kan effektiviseres ved å løsne på kravet om bruk av eksisterende seriell kode.



Figur 4.1: Partisjonering av grid i en dimensjon med ett elements overlapp. Det er her brukt lineære elementer slik at hvert element inneholder 2 gridpunkter.

4.2 Partisjonering

I elementmetoden er det naturlig å partisjonere løsningsdomenet på elementnivå. Med dette menes at hver prosessor får ansvar for en viss mengde elementer og deres tilhørende gridpunkter. Det beregningsmessige arbeidet forbundet med ett element, vil normalt være tilnærmet likt for alle elementer. For å få en god lastbalansering må derfor antall elementer på hver prosessor fordeles så likt som mulig.

Naboelementer er elementer med ett eller flere felles gridpunkter (se avsnitt 2.1). Vi vil senere se (avsnitt 4.6) at dersom to naboelementer befinner seg på to ulike prosessorer, vil dette gi kommunikasjon under løsning av likningssystemet. Fordi kommunikasjon tar ekstra tid, ønsker vi å minimere antall slike tilfeller.

Vi har altså to ulike forhold å ta hensyn til når vi fordeler elementer på prosessorene: Antall elementer bør være så likt som mulig, og antall naboelementer som splittes over ulike prosessorer bør minimeres. Dette er et NP-komplett problem som må løses med en heuristisk metode for å kunne løses rask nok. Diffpack har valgt å bruke et eksternt bibliotek, Metis[20, 21, 22], til dette. Partisjoneringen i Metis tar tid $\mathcal{O}(N)$[20][1].

Partisjoneringene i Diffpack kan gjøres med eller uten overlapp. En partisjonering uten overlapp kjennetegnes ved at hvert element er representert på én og bare én prosessor. I en partisjonering med overlapp vil ett element derimot kunne være representert på flere prosessorer. En slik duplisering av elementer skjer langs grensene mellom de ulike prosessorene. Jeg vil komme tilbake til hvorfor det er nødvendig med overlappende partisjoneringer i avsnitt 4.4. Arbeidet med å beregne hvilke elementer som må legges til med overlapp, tar tid $\mathcal{O}(NP)$. Den totale tiden for partisjonering i Diffpack tar altså tid $\mathcal{O}(N + NP)$.

Punktene som ligger på en grense mellom to eller flere prosessorer vil videre referes til som en *intern rand*. Figur 4.1 viser en partisjonering i én dimensjon med ett elements overlapp, og peker på hvilke punkter som ligger på den interne randen.

[1] Egentlig $\mathcal{O}(K)$ der K er antall kanter i graf som representerer elementgridet.

4.3 Diskretisering

Som nevnt i avsnitt 2.1 er det mulig å gjøre integrasjonen i diskretiseringen på elementnivå. Hver prosessor utfører da den nødvendige integrasjonen på sine egne elementer. Hvert element får et lite likningssystem som må legges sammen med likningssystemene fra de andre elementene for å lage det globale likningssystemet. Fordi naboelementer gir bidrag til en felles likning i likningsystemet, vil to naboelementer som befinner seg på ulike prosessorer kreve kommunikasjon. Diffpack kjører bare ren seriell diskretisering på hver prosessor, og utfører *ikke* denne kommunikasjonen. Dette gir ufullstendige likninger for de gridpunktene som ligger på en intern rand. Diffpack passer istedet på å korrigere for dette under løsning av likningssystemet. Avsnitt 4.6 vil se nærmere på denne korreksjonen.

4.4 Prekondisjonering

Under løsning av det lineære likningssystemet, er det vanlig med bruk av en prekondisjonerer (se avsnitt 2.2). Det er ikke dokumentert nøyaktig hvordan parallell prekondisjonering blir gjort i Diffpack. [6] nevner at en mulig metode er å bruke en vanlig seriell prekondisjonerer lokalt på hver partisjon. Dette vil generelt gi inkonsistente verdier mellom de ulike prosessorene. En løsning på dette er å bli enige om en gjennomsnittsverdi for felles punkter. Dette krever kommunikasjon.

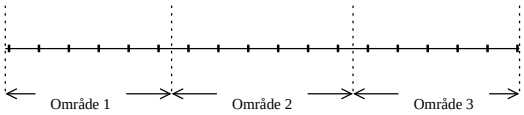
Parallell prekondisjonering i Diffpack har vist seg å bli ustabilt ved økende antall prosessorer. En løsning på dette er å legge til ekstra overlappende elementer langs grensen mellom prosessorene. Denne oppgaven vil derfor fokusere på tilfeller der partisjonene overlapper hverandre med ett element.

4.5 Dupliserte gridpunkter

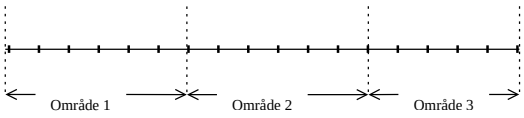
Naboelementer vil per definisjon ha ett eller flere felles gridpunkter. Dersom to naboelementer fordeles på to ulike prosessorer, blir deres felles gridpunkter representert på begge prosessorene. Et punkt som er representert på mer enn én prosessor vil videre bli kalt et duplisert gridpunkt. Som vi vil se i avsnitt 4.6, vil antall dupliserte gridpunkter angi hvor mye merarbeid og kommunikasjon vi vil få i en parallell løsning av likningssystemet. Jeg vil derfor her gi et estimat på dette antallet.

I det følgende antar jeg en partisjonering av et kartesisk domene. Som før vil P angi antall prosessorer, og N vil være det globale antall gridpunkter før partisjonering, dvs det antall gridpunkter man ville hatt i en seriell implementasjon. n vil være gjennomsnitlig antall gridpunkter per prosessor etter partisjonering.

I dette avsnittet er jeg spesielt interessert i verdien $\mathcal{N} = nP$ som gjenspeiler det antall gridpunkter som forekommer i en parallell beregning med P prosessorer. $\mathcal{N} - N$ vil gi antall globalt dupliserte gridpunkter.



Figur 4.2: Ikke-overlappende partisjon av grid i en dimensjon. Ingen punkter blir duplisert.



Figur 4.3: Grid i en dimensjon partisjonert med overlapp av grad 1. Et punkt blir duplisert på hver grense mellom to områder.

Gridpunkt-nivå

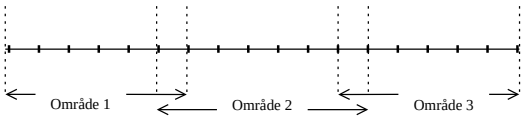
Anta et grid i én dimensjon med N gridpunkter. Vi skal fordele gridpunktene på P prosessorer. En ikke-overlappende partisjon på gridpunktnivå vil da gi $n = \frac{N}{P}$ gridpunkter til hver prosessor[2]. Summerer vi opp punktene på alle prosessorene, får vi da selvfølgelig $nP = \mathcal{N} = N$ globale gridpunkter. Denne situasjonen er vist i figur 4.2.

Velger vi istedet å partisjonere slik at grensen går *på* et gridpunkt, får vi situasjonen i figur 4.3. Gridpunktet som befinner seg på grensen (den interne randen) blir representert i to partisjoner, og er følgelig duplisert. Vi får $P - 1$ slike grenser og følgelig $P - 1$ dupliserte noder slik at $\mathcal{N} = N + P - 1$. En slik partisjonering kalles videre for en partisjonering med 1 grad av overlapp.

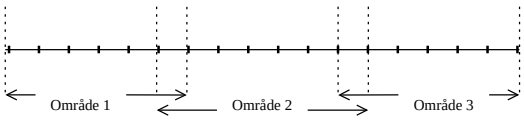
Vi kan også partisjonere med overlapp av grad 2. Vi får da situasjonen i figur 4.4. I denne situasjonen får vi 2 dupliserte noder per grense og $\mathcal{N}$ blir $N + 2(P - 1)$. Grad av overlapp gjenspeiler altså antall gridpunkter som dupliseres i hver grense. Setter vi graden av overlapp til l, får vi generelt $\mathcal{N} = N + l(P - 1)$ i for partisjoneringer i en dimensjon.

Antar vi kartesiske partisjoneringer, kan vi generalisere til grid i flere dimensjoner. Gitt en dimensjon d vil vi ha $\sqrt[d]{N}$ gridpunkter og $\sqrt[d]{P} - 1$ grenser

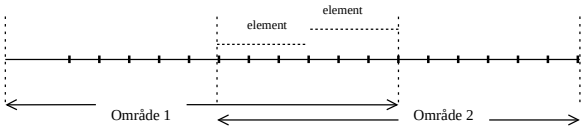
[2]Jeg antar alle divisjoner går opp.



Figur 4.4: Grid i en dimensjon partisjonert med overlapp av grad 2. To punkter blir duplisert på hver grense mellom to områder.



Figur 4.5: Grid i en dimensjon partisjonert med overlapp på elementnivå av grad 1. Ett element blir duplisert på hver grense mellom to områder. Her elementer av første orden slik at hvert element her inneholder 2 punkter.



Figur 4.6: Grid i en dimensjon partisjonert med overlapp på elementnivå av grad 2. To elementer blir duplisert på hver grense mellom to områder. Hvert element inneholder her 4 gridpunkter.

for hver dimensjon. Vi får da følgende utrykk:

$$\mathcal{N} = \left[\sqrt[d]{N} + l \left(\sqrt[d]{P} - 1 \right) \right]^d$$

Elementnivå

La oss nå igjen se på et grid i én dimensjon. Med første ordens elementer vil vi få ett element mellom to gridpunkter. Naboelementer vil dele gridpunkter mellom seg. Et grid med E elementer vil da inneholde $E+1$ gridpunkter. Grad av overlapp på elementnivå angir antall dupliserte *elementer* per grense. En ikke-overlappende partisjonering av elementer vil da tilsvare situasjonen for 1'te grads overlapp på gridpunkt-nivå (figur 4.3 på forrige side). Tilsvarende vil en første grads partisjonering av elementer (figur 4.5) være lik en 2. grads partisjonering på gridpunkt-nivå (figur 4.4).

Elementer kan generelt være av høyere grad og inneholde flere gridpunkter. Har vi elementer bestående av n_e gridpunkter, vil et grid med E elementer gi $En_e - (E-1) = E(n_e - 1) + 1$ gridpunkter. Antall dupliserte gridpunkter på hver grense med en grad av overlapp på elementnivå lik l, blir lik antall gridpunkter i et grid bestående av l elementer. Har vi n_e gridpunkter i hvert element får vi da $\mathcal{N} = N + (l(n_e - 1) + 1)(P - 1)$. Figur 4.6 viser et grid i en dimensjon med overlapp av grad 2 der ett element har 4 noder.

Igjen kan vi generalisere til høyere dimensjoner dersom vi antar kartesisk partisjonering og kartesiske elementer. For en dimensjon d får vi da i hver dimensjon $\sqrt[d]{N}$ gridpunkter, $\sqrt[d]{n_e}$ gridpunkter i hvert element og $\sqrt[d]{P}$ prosessorer. Det totale antall gridpunkter blir

$$\mathcal{N} = \left[\sqrt[d]{N} + [l\,(\sqrt[d]{n_e} - 1) + 1] \left(\sqrt[d]{P} - 1 \right) \right]^d$$

I denne oppgaven vil jeg kun bruke enkle elementer der $\sqrt[d]{n_e} = 2$. De fleste

N	P	$\frac{\mathcal{N}-N}{N}$
100^2	4	3%
100^2	8	6%
100^2	16	11%
100^2	32	16%
600^2	4	0%
600^2	8	1%
600^2	16	1%
600^2	32	3%
1000^2	4	0%
1000^2	8	0%
1000^2	16	1%
1000^2	32	1%

N	P	$\frac{\mathcal{N}-N}{N}$
10^3	4	28%
10^3	8	42%
10^3	16	54%
10^3	32	66%
60^3	4	5%
60^3	8	9%
60^3	16	13%
60^3	32	18%
100^3	4	3%
100^3	8	5%
100^3	16	8%
100^3	32	11%

Tabell 4.1: Estimat for andel duplierte gridpunkter per prosessor i to og tre dimensjoner ved bruk av ett elements overlapp.

tester vil bli gjort med overlapp $l = 1$. Vi får da en forenklet likning:

$$\mathcal{N} = \left[\sqrt[d]{N} + 2 \left(\sqrt[d]{P} - 1 \right) \right]^d \tag{4.1}$$

Likningene forteller oss at andel dupliserte gridpunkter øker med antall dimensjoner og antall prosessorer. Tabell 4.1 viser noen estimater av andel dupliserte gridpunkter for ulike problemstørrelser, og antall prosessorer ved bruk av ett elements overlapp. Vi ser her at andel dupliserte gridpunkter er betydelig for små problemstørrelser, selv med få prosessorer. Andel dupliserte gridpunkter øker også betydelig ved å gå fra to til tre dimensjoner. For moderate og store problemstørrelser i to dimensjoner er andel dupliserte gridpunkter beskjedent. For problemer i tre dimensjoner er derimot andelen betydelig, selv ved store problemstørrelser dersom man bruker av mange prosessorer. Det er bruk av mange prosessorer som er ønskelig for å få ned kjøretiden så mye som mulig på store simuleringer.

4.6 Løsning av likningssystem

Som nevnt i kapittel 2 blir likningssystemet som regel løst ved hjelp av iterative metoder. Disse innholder i hovedsak matrise-vektor-produkt, indreprodukt og vektoroppdateringer. Jeg vil her på hvordan disse operasjonene blir parallellisert i Diffpack.

I de neste avsnittene vil det globale likningssystemet bestå av N ukjente nummerert fra 1 til N. Disse blir angitt med settet $I = \{1 \ldots N\}$. Ukjent nummer i vil tilsvare løsningen i gridpunkt nr i. Videre i dette avsnittet antas prosessorene å være nummerert fra 1 til P. I_p angir det sett av ukjente som er tildelt prosessor nummer p.

N	P	S_v
100^2	4	3.84
100^2	8	7.45
100^2	16	14.24
100^2	32	26.78
600^2	4	3.97
600^2	8	7.90
600^2	16	15.68
600^2	32	31.03
1000^2	4	3.98
1000^2	8	7.94
1000^2	16	15.81
1000^2	32	31.41

N	P	S_v
10^3	4	2.87
10^3	8	4.63
10^3	16	7.22
10^3	32	10.83
60^3	4	3.77
60^3	8	7.25
60^3	16	13.80
60^3	32	25.94
100^3	4	3.86
100^3	8	7.54
100^3	16	14.63
100^3	32	28.16

Tabell 4.2: Estimert speedup for parallell vektoroppdatering, S_v, ved bruk av ett elements overlapp.

4.7 Vektoroppdatering

Vektoroppdateringer er trivielt parallelliserbare. Hver prosessor oppdaterer kun de ukjente den har fått tildelt. En vektoroppdatering tar tid proporsjonalt med antall gridpunkter. En parallell vektoroppdatering vil ta tid proporsjonalt med antall lokale gridpunkter. Antar vi en proporsjonalitetskonstant c_v, vil dermed en parallell vektoroppdatering ta tid

$$T_v(P) = c_v n \tag{4.2}$$

og vi får en speedup gitt ved

$$S_v(P) = \frac{T_v(1)}{T_v(P)} = \frac{c_v N}{c_v n} = \frac{NP}{\mathcal{N}}$$

Tabell 4.2 viser noen estimerte verdier for en slik speedup der $\mathcal{N}$ er beregnet ut fra likning 4.1. Det parallelle merarbeidet for en slik implementasjon vil være proporsjonalt med antall dupliserte gridpunkter, slik at avviket fra en ønskelig lineær speedup vil tilsvare andelen dupliserte gridpunkter gitt i tabell 4.1. Vi ser derfor, som i tabell 4.1, at avviket øker med økende antall prosessorer, men reduseres med økende problemstørrelse. Avviket er mer betydelig for problemer i tre dimensjoner enn i to dimensjoner.

4.8 Indreprodukt

I Diffpack beregnes først det lokale indreproduktet med en seriell funksjon. Denne vil inkludere alle de lokale gridpunktene. Av de lokale gridpunktene vil det være noen punkter som er duplisert på andre prosessorer. Før resultatene fra hver prosessor legges sammen, må man derfor først trekke fra de ekstra

bidragene som kommer fra dupliserte gridpunkter. La (x, y) betegne indreproduktet mellom vektorene x og y. Vi får da:

$$(x, y) = \sum_{i=1}^{N} x_i y_i = \sum_{p} \left(\sum_{i \in I_p} x_i y_i - \sum_{i \in \{I_p \bigcap (I \setminus I_p)\}} \frac{\alpha_i - 1}{\alpha_i} x_i y_i \right) \tag{4.3}$$

der α_i er lik punktet i sin *multiplisitet*. Med et punkts multiplisitet menes her antall partisjoner punktet i er representert i.

Denne korreksjonen forbundet med dupliserte gridpunkter vil kunne være relativt dyr. Fordyrende elementer er:

Ujevn fordeling av overlappende gridpunkter. Det globale indreproduktet kan ikke beregnes før alle prosesser er ferdige med sine lokale bidrag. Dersom en prosess har mer arbeid enn de andre, vil altså disse bli nødt til å vente på denne. Det er altså prosessen med mest arbeid som bestemmer den totale tiden. Denne forskjellen er lik forskjellen i antall gridpunkter per prosessor. Med bruk av Metis, ligger denne gjerne på noen få prosent[20].

Uheldig aksessering av minne. Gridpunktene som må korrigeres i likning 4.3 ligger generelt ikke sekventielt etter hverandre i minnet. Aksesseringsmønsteret blir således ikke forutsigbart, og man må forvente et betydelig antall ganger hvor en verdi må hentes utenfor det lokale hurtigminnet. Avhengig av hvor raskt det lokale hurtigminnet er i forhold til neste nivå med minne, og hvor tett de dupliserte gridpunktene faktisk ligger, kan dette bidra med en faktor på alt mellom noen få prosent til noen hundre prosent.

Indirekte indeksering. For å finne ut hvilke gridpunkt det skal korrigeres for, vil man måtte slå opp vektor-indeksene i et forhåndsberegnet sett av overlappende gridpunkter. Dette vil gjøre minneaksessen minst dobbelt så dyr, og bidra med en faktor på minst 2.

Overlappende gridpunkt korrigeres flere ganger. Dersom et gridpunkt er representert på to prosessorer, vil dette medføre *to* korreksjoner i indreproduktet, én korreksjon for hver prosessor gridpunktet er representert i. Dette vil bidra med en faktor på minst 2 (men neppe særlig mer).

Den ytterste summasjonen i likning 4.3 vil kreve kommunikasjon i form av en global reduksjon av alle de lokale verdiene. Antar vi denne reduksjonen til å ta tid ρ og de dupliserte punktene til å være likt fordelt på prosessorene, får vi at parallell tid som går som:

$$T_I(P) = c_I n + d_I \left(\frac{\mathcal{N} - N}{P} \right) + \rho \tag{4.4}$$

der c_I er en proporsjonalitetskonstant forbundet med det lokale indreproduktet, og d_I er en proporsjonalitetskonstant forbundet med korreksjonen. Diskusjonen ovenfor antyder at $d_I / c_I > 4$.

Dersom vi antar ρ til å være neglisjerbar, får vi et estimat for parallell speedup for indreproduktet:

$$S_I(P) = \frac{T_I(1)}{T_I(P)} = \frac{c_I N}{c_I n + d_I \left(\frac{\mathcal{N} - N}{P} \right) + \rho} \approx \frac{NP}{\mathcal{N} + \frac{d_I}{c_I} (\mathcal{N} - N)}$$

N	P	$S_I(\frac{d_I}{c_I}=4)$	$S_I(\frac{d_I}{c_I}=10)$	N	P	$S_I(\frac{d_I}{c_I}=4)$	$S_I(\frac{d_I}{c_I}=10)$
100^2	4	3.33	2.77	10^3	4	1.34	0.75
100^2	8	5.83	4.40	10^3	8	1.72	0.89
100^2	16	9.89	6.78	10^3	16	2.26	1.11
100^2	32	16.20	10.18	10^3	32	2.97	1.42
600^2	4	3.87	3.73	60^3	4	3.08	2.41
600^2	8	7.54	7.05	60^3	8	5.27	3.74
600^2	16	14.54	13.10	60^3	16	8.89	5.80
600^2	32	27.67	23.81	60^3	32	14.76	8.96
1000^2	4	3.92	3.83	100^3	4	3.39	2.87
1000^2	8	7.72	7.40	100^3	8	6.13	4.78
1000^2	16	15.09	14.13	100^3	16	10.88	7.87
1000^2	32	29.26	26.54	100^3	32	19.03	12.81

Tabell 4.3: Estimert speedup for parallelt indreprodukt, S_I, ved bruk av ett elements overlapp.

Tabell 4.3 viser noen estimerte verdier for en slik speedup. Det er her brukt ett elements overlapp og likning 4.1 til å estimere antall dupliserte gridpunkter, $\mathcal{N} - N$.

$d_I/c_I = 4$ angir en øvre grense for hva vi kan oppnå av speedup for denne implementasjonen. Vi ser de samme tendensene som for parallell vektoroppdatering med tanke på avvik fra lineær speedup, men avvikene blir her noe større på grunn av faktoren d_I/c_I. Spesielt stor nedgang i speedup ser vi for problemene i tre dimensjoner. For $N = 100^3$ og $P = 32$ har vi for eksempel en nedgang fra speedup på 28 for vektoroppdatering, men kun 19 for indreproduktet. Dette er en nedgang på nesten 40%. Vi ser også en vesentlig forverring av speedup dersom vi øker d_I/c_I til 10. Dette sier oss at kvaliteten på denne implementasjonen er svært følsom for denne faktoren.

4.9 Matrise-vektor-produkt

Til beregning av et parallelt matrise-vektor-produkt, har Diffpack valgt samme fremgangsmåte som for det parallelle indreproduktet. Først utføres et vanlig serielt lokalt matrise-vektor-produkt som inkluderer alle lokale gridpunkter. Det lokale matrise-vektor-produktet vil da gi ufullstendige verdier for punkter som ligger på en intern rand. Dette fordi likningssystemet etter den parallelle diskretiseringen ble ufullstendig for disse punktene (se avsnitt 4.3). Etter det lokale matrise-vektor-produktet må man derfor korrigere disse verdiene. Hvordan denne korreksjonen blir gjort er avhengig av om vi har en overlappende eller ikke-overlappende partisjonering på elementnivå .

4.9.1 Ikke-overlappende partisjonering

Dersom vi har en ikke-overlappende partisjonering vil korrekt likning for et punkt på en intern rand være gitt ved å legge sammen likningene fra de pro-

sessorene som også har dette punktet[6]. Tilsvarende vil vi få en korrekt verdi for matrise-vektor-produktet ved å legge sammen resultatene fra alle prosessorer som har beregnet en verdi for dette punktet.

Det lokale matrise-vektor-produktet for en ikke-overlappende partisjonering kan da beskrives som

$$Ax|_{I_p} = \sum_{i \in I_p} (A_i, x) + \sum_{i \in \{I_p \bigcap (I \setminus I_p)\}} \left(\sum_{q \neq p}^{P} (A_i, x)_q \right) \tag{4.5}$$

der $Ax|_{I_p}$ betyr produktet Ax restriktert på punktene I_p.

Dette arbeidet vil være proporsjonalt med antall gridpunkter langs kantene og disse punktenes gjennomsnittlige multiplisitet. I tillegg kreves kommunikasjon av disse punktene. Vi antar at hver prosessor må kommunisere med Q andre prosessorer (har Q naboprosessorer) og at alle meldingene er like lange. Vi får da et estimat for parallell tid som

$$T_m(P) = c_m k n + d_m \frac{\mathcal{N} - N}{P} + e_m \pi(\frac{\mathcal{N} - N}{QP}) \tag{4.6}$$

der $c_m k$ er en konstant forbundet med det arbeidet for det lokale matrise-vektor-produktet (mer om dette senere), og d_m er en konstant forbundet med korreksjon for ett punkt. $\pi(x)$ er tiden det tar å sende og motta en melding med lengde x. Det må sendes Q slike meldinger. Avhengig av gjeldende plattform, *kan* disse overlappe hverandre i tid. e_m er en konstant som gjenspeiler dette og vil være mellom 1 og Q.

Når matrisen er lagret i CRS-format (se avsnitt 2.2) vil arbeidet med én rad i matrisen likne på arbeidet med et indreprodukt mellom to vektorer av lengde k, der k er antall verdier forskjellig fra null i den aktuelle raden. k vil som regel være nesten konstant. For bilineære elementer i to dimensjoner får vi en gjennomsnittsverdi nær $k = 9$. For trilineære elementer i tre dimensjoner får vi en gjennomsnittsverdi nær $k = 27$.

Fordi arbeidet forbundet med en rad i matrisen likner på et indreprodukt, antar jeg at c_m er nær konstanten c_I for indreproduktet. Man kan argumentere for størrelsen av d_m med de samme argumentene som for d_I i indreproduktet. Jeg antar derfor d_m nær d_I og får $d_m/c_m \approx d_I/c_I > 4$.

Dersom vi regner kommunikasjonskostnadene for neglisjerbare, får vi følgende uttrykk for parallell speedup:

$$\begin{aligned} S_m(P) = \frac{T_m(1)}{T_m(P)} &= \frac{c_m k N}{c_m k n + d_m \frac{\mathcal{N}-N}{P} + e_m \pi(\frac{\mathcal{N}-N}{QP})} \\ &\approx \frac{NP}{\mathcal{N} + \frac{d_m}{c_m k}(\mathcal{N} - N)} \end{aligned} \tag{4.7}$$

4.9.2 Overlappende partisjonering

I tilfellet med en overlappende partisjonering vil det alltid eksistere minst én prosessor der likningen for et gitt punkt er fullstendig[3]. Følgelig vil det lokale

[3]Dette garanteres internt i Diffpack.

N	P	$S_m(\frac{2f_m}{c_m}=4)$	$S_m(\frac{2f_m}{c_m}=10)$	N	P	$S_m(\frac{2f_m}{c_m}=4)$	$S_m(\frac{2f_m}{c_m}=10)$
100^2	4	3.78	3.68	10^3	4	2.70	2.48
100^2	8	7.27	7.02	10^3	8	4.31	3.91
100^2	16	13.78	13.14	10^3	16	6.65	5.94
100^2	32	25.60	24.02	10^3	32	9.86	8.69
600^2	4	3.96	3.94	60^3	4	3.73	3.67
600^2	8	7.87	7.82	60^3	8	7.15	7.00
600^2	16	15.59	15.46	60^3	16	13.55	13.21
600^2	32	30.78	30.40	60^3	32	25.38	24.59
1000^2	4	3.98	3.97	100^3	4	3.84	3.80
1000^2	8	7.92	7.89	100^3	8	7.47	7.38
1000^2	16	15.75	15.67	100^3	16	14.47	14.24
1000^2	32	31.26	31.03	100^3	32	27.79	27.24

Tabell 4.4: Estimert speedup, S_m, for parallelt matrise-vektor-produkt ved bruk av ett elements overlapp.

matrise-vektor-produktet på denne prosessoren også gi korrekt verdi for dette punktet. Denne kan da kommuniseres til de andre prosessorene.

Jeg antar som før at det lokale gridet på en prosessor vil være en kube i d dimensjoner med n punkter. Antall punkter langs en av kubens flater (en intern rand) vil da være $n^{\frac{d-1}{d}}$ og kuben vil ha $2d$ slike flater. For et slikt tilfelle vil hver prosessor altså måtte kommunisere $2dn^{\frac{d-1}{d}}$ verdier etter det lokale matrise-vektor-produktet. Et estimat for parallell kjøretid blir da:

$$T_m(P) = c_m kn + 2f_m dn^{\frac{d-1}{d}} + e_m \pi(n^{\frac{d-1}{d}}) \tag{4.8}$$

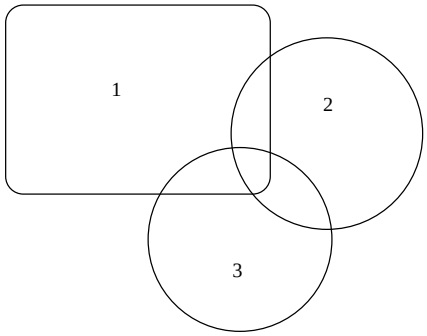
Det første og andre leddet er som for matrise-vektor-produktet uten overlapp. f_m er en konstant forbundet med det å erstatte en verdi i løsningsvektoren med en annen.

Dersom vi regner kommunikasjonskostnadene for neglisjerbare, får vi følgende uttrykk for speedup:

$$\begin{aligned} S_m(P) = \frac{T_m(1)}{T_m(P)} &= \frac{c_m kN}{c_m kn + 2f_m dn^{\frac{d-1}{d}} + e_m \pi(n^{\frac{d-1}{d}})} \\ &\approx \frac{NP}{\mathcal{N} + \frac{2f_m}{kc_m} d\mathcal{N}^{\frac{d-1}{d}} P^{(1-\frac{d-1}{d})}} \end{aligned} \tag{4.9}$$

Ser vi tilbake på argumentasjonen for c_I i indreproduktet, vil vi se at f_m vil lide av de samme effektene med unntak av det siste (korreksjonen skjer her normalt kun én gang per gridpunkt). Fjerner vi denne faktoren, kan vi gjette på $\frac{2f_m}{c_m} \approx \frac{d_I}{c_m} \approx \frac{d_m}{c_m} > 4$.

Tabell 4.4 viser noen estimater av speedup for en partisjon med ett elements overlapp. Vi ser her at matrise-vektor-produktet kan forventes vesentlig bedre verdier for speedup enn indreproduktet, og er rimelig nær lineær speedup. Dette skyldes verdien k som angir antall verdier forskjellig fra null per rad i matrisen. Ved økende k blir arbeidet med det lokale matrise-vektor-produktet



Figur 4.7: Eksempel på en relativt komplisert overlappende partisjonering i to dimensjoner.

større, men har ingen innvirkning på arbeidet forbundet med korreksjonen. Det parallelle merarbeidet blir derfor mindre merkbart. k er størst for problemer i tre dimensjoner, som var de problemene med dårligst speedup for indreproduktet. I forhold til indreproduktet, er den parallelle kvaliteten av matrise-vektor-produktet heller ikke spesielt følsomt for maskinavhengige forhold, representert med $\frac{f_m}{c_m}$.

4.9.3 Kommunikasjon

Kommunikasjonen i indreproduktet var svært enkel. Det eneste som ble brukt var MPI's funksjon for global reduksjon. Kommunikasjonen i matrise-vektor-produktet er derimot mer komplisert.

Figur 4.7 viser et eksempel på en overlappende partisjonering i to dimensjoner. Ta prosessorene med den rektangulære partisjonen (område 1) som eksempel. Etter det lokale matrise-vektor-produktet må denne prosessoren motta verdier for sin indre rand fra både område 2 og 3. Deler av denne randen ligger både innenfor område 2 og område 3. For å unngå unødvendig kommunikasjon trenger prosessoren bare å motta disse verdiene fra ett av de andre områdene, men det må være klargjort på forhånd hvilket område dette er. Prosessoren må også sende verdier til sine naboer. Til område 2 må han sende de punktene som tilsvarer den indre randen for område 2, og tilsvarende til område 3. I krysningen mellom den interne randen til område 2 og 3 har vi et punkt som må kommuniseres til begge områdene.

Generelt kan en prosessor måtte kommunisere med vilkårlig mange andre prosessorer. Og generelt kan et gridpunkt måtte kommuniseres til et vilkårlig antall andre prosessorer. Punktene som skal kommuniseres kan ha vilkårlige gridpunktnummere og således ligge med vilkårlige indekser i vektoren.

Diffpack løser dette ved å forhåndsberegne den nødvendige kommunikasjonen. Hver prosessor lager en liste over hvilke prosessorer den må kommunisere med. For hver av disse lages det en liste over hvilke gridpunkter den må sende og hvilke den må motta. I tillegg opprettes to kommunikasjonsbuffere, ett til mottak og ett til sending. Når en prosessor så skal sende sine verdier til en nabo, bruker han listen over gridpunkter til å plukke ut de riktige verdiene

```
int *neighbors;       // liste over prosessorer å kommunisere med
int **ids_send;       // for hver p i neighbors: liste over indekser å sende
int **ids_recv;       // for hver p i neighbors: liste over indekser å motta
double **buf_send; // buffere for å sende meldinger
double **buf_recv;  // buffere for å motta meldinger

void send( double *x )
{
   for p in neighbors
      int k = 0;
      for i in ids_send
         buf_send[p][k++] = x[i]; // kopier fra vektor
      MPI_Send( buf_send[p], p ); // send til buffer til p
}

void recv( double *x )
{
   for p in neighbors
      MPI_Recv( buf_recv[p], p ); // motta fra p i buffer
      int k = 0;
      for i in ids_recv
         x[i] = buf_recv[p][k++]; // kopier tilbake til vektor
}
```

Figur 4.8: Skisse i pseudo-kode av sending og mottak ved parallelt matrise-vektor-produkt i Diffpack.

fra vektoren og kopiere disse over i bufferet. Dette bufferet blir så kommunisert ved hjelp av MPI. Når en prosessor skal motta verdier, blir dette gjort ved å sende bufferet for mottak til MPI. Etter kommunikasjonen brukes så listen over hvilke punkter som ble mottatt til å kopiere (eller addere) disse til riktige plasser i vektoren. Figur 4.8 forsøker å oppsummere dette i pseudokode.

Denne kopieringen av verdier som skal sendes og mottas tar både ekstra tid og ekstra minne. I neste kapittel vil jeg komme med forslag til ny partisjonering som vil eliminere behovet for denne kopieringen.

Kapittel 5

Optimaliseringer

Forrige kapittel gikk igjennom hvordan parallelle simuleringer med elementmetoden er implementert i Diffpack. Diskretisering, vektoroppdatering, indreprodukt og matrise-vektor-produkt ble behandlet hver for seg. Diffpacks utgangspunkt var å bruke de vanlige serielle funksjonene for disse operasjonene lokalt på hver prosessor, for så å kommunisere og korrigere de globale resultatene. For å optimalisere dette har jeg måttet gå bort ifra bruken av de eksisterende serielle funksjonene, og lage egne parallelle versjoner av disse.

Diffpack brukte en partisjonerings-algoritme som tok tid $\mathcal{O}(N)$. I tillegg må det legges til overlapp som tar tid $\mathcal{O}(NP)$. Jeg vil istedet foreslå en partisjoneringsalgoritme som tar tid $\mathcal{O}(N \log N)$ og som bruker tid $\mathcal{O}(n)$ til å legge til overlapp. Denne partisjoneringen vil også kunne optimalisere kommunikasjonen i matrise-vektor-produktet.

5.1 Indreprodukt

Det serielle indreproduktet (figur 5.1) består av en enkel løkke med forutsigbar sekventiell minne-aksess og enkel aritmetikk. Dette er en enkel funksjon som er enkel å optimalisere med for eksempel ulike variasjoner av løkkeutrulling.

Det parallelle indreproduktet i Diffpack er skissert i figur 5.2. Denne inneholder et vanlig lokalt indreprodukt (linje 3) etterfulgt av en ekstra løkke der

```
1 double innerProd( double *x, double *y, int n )
2 {
3    double sum = 0;
4    for ( int i=0; i < n; i++ ) {
5       sum += x[i]*y[i];
6    }
7    return sum;
8 }
```

Figur 5.1: Enkel seriell implementasjon av indreproduktet.

```
1  int *D;        // sett av dupliserte punkter
2  int *m;        // punktenes multiplisitet
3  int D_size;    // antall punkter i D (og m)
4
5  double parallelInnerProd( double *x, double *y, int n )
6  {
7     double sum = innerProd( x, y, n ); // lokalt indreprodukt
8     for ( int i=0; i < D_size; i++ ) {
9        sum -= x[ D[i] ] * y[ D[i] ] * (m[i] - 1.0)/m[i];
10    }
11    MPI_Allreduce( sum ); // summerer del-resultater fra alle prosessorer
12    return sum;
13 }
```

Figur 5.2: Skisse av Diffpacks parallelle indreprodukt.

man må korrigere bidraget fra de dupliserte gridpunktene. Som vi så i avsnitt 4.8 vil denne løkken kunne være svært kostbar.

Jeg vil videre bruke notasjonen fra avsnitt 4.6 der det globale likningssystemet består av ett sett med ukjente $I = \{1 \dots N\}$, prosessorene er nummerert fra 1 til P og $I_p \subset I$ angir det sett av ukjente som er tildelt prosessor nummer p.

5.1.1 Eliminering av korreksjon

La oss nå anta at vi lager en funksjon

$$\boldsymbol{g}(i) : I \to \{1 \dots P\}\,, \quad \text{slik at} \quad i \in I_{\boldsymbol{g}(i)}$$

som tar et globalt gridpunkt-nummer og returnerer et prosessor-nummer der dette punktet er tilgjengelig. Vi kan da lage et parallelt indreprodukt uten behov for korreksjon (sammenlikn med likning 4.3 på side 22):

$$(x, y) = \sum_{i=1}^{N} x_i y_i = \sum_{p} \sum_{i \in G_p} x_i y_i\,, \quad G_p = \{i : \boldsymbol{g}(i) = p\} \tag{5.1}$$

der (x, y) som før betegner indreproduktet mellom vektorene x og y.

Funksjonen $\boldsymbol{g}$ har altså som hovedoppgave å fordele ansvaret for de dupliserte gridpunktene på ulike prosessorer. En annen måte å se dette på er at $\boldsymbol{g}$ lager en ny ikke-overlappende partisjonering på gridpunkt-nivå ut av den eksisterende partisjoneringen. Som resultat kan hver prosessor inkludere alle de lokale gridpunktene i G_p, som er dens andel av den ikke-overlappende partisjoneringen. Globalt blir da alle gridpunkter inkludert én og bare én gang slik at korreksjon er unødvendig.

Dersom man skulle implementere dette indreproduktet, ville man sannsynligvis endt opp med en kode nær den i figur 5.3. Her er G_p forhåndsberegnet for å kunne være så effektiv som mulig. I tillegg til å ha eliminert løkken med korreksjon, ser vi også at løkken vil gjøre færre iterasjoner enn det serielle indreproduktet som ble brukt i linje 3 i figur 5.2 (`G_size` $\leq$ `n`).

```
1  int *G;        // sett av punkter som skal inkluderes
2  int G_size;    // antall punkter i G
3
4  double parallellInnerProd( double *x, double *y )
5  {
6     double sum = 0;
7     for ( int i=0; i < G_size; i++ ) {
8        sum += x[ G[i] ] * y[ G[i] ];
9     }
10    MPI_Allreduce( sum );
11    return sum;
12 }
```

Figur 5.3: Skisse av parallelt indreprodukt uten korreksjon for dupliserte gridpunkter.

Ulempen med denne versjonen er bruk av indirekte indeksering, som også gir en uforutsigbar minneaksessering. Dette ødelegger for de fleste optimaliseringer. I Diffpacks vil i det minste det lokale indreproduktet være egnet for løkke-utrulling, selv om korreksjonen etterpå kan være kostbar. Man kan derfor ikke forvente at denne koden er raskere enn Diffpacks versjon.

5.1.2 Sortering av gridpunkter

I utgangspunktet vil dupliserte gridpunkter opptre på vilkårlige indekser i vektorene. Dette gjør det umulig å unngå løsninger som på en eller annen måte må bruke indirekte indeksering for å unngå korreksjon for dupliserte gridpunkter. For å få bukt med dette problemet må vi derfor eliminere den tilfeldige indekseringen.

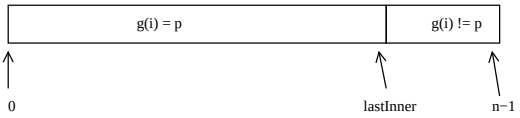
Hvilken indeks et gridpunkt får, blir i Diffpack bestemt under partisjoneringen. Der deles det globale gridet opp og man bygger et nytt lokalt grid. Et gridpunkt får et lokalt gridpunkt-nummer som også tilsvarer dets indeks i vektorene.

Her er det mulig å utnytte funksjonen $\boldsymbol{g}$. Hver prosessor kan sørge for å gi de globale gridpunktene der $\boldsymbol{g}$ returnerer dets eget prosessornummer, et lokalt gridpunktnummer, slik at de blir liggende etter hverandre i minnet. Figur 5.4 viser dette i pseudokode, og forsøker å illustrere hvordan gridpunktene blir liggende i vektoren.

Figur 5.5 viser hvordan det parallelle indreproduktet kan implementeres etter å ha sortert gridpunktene. Denne koden er på samme form som det serielle indreproduktet (figur 5.1) og kan optimaliseres med de samme teknikkene. Vi har her eliminert alt behov for korreksjon av dupliserte gridpunkter, redusert antall iterasjoner i det lokale indreproduktet samtidig som vi unngår indirekte indeksering.

Jeg har foreløpig basert meg på at det eksisterer en funksjon $\boldsymbol{g}$. Hvordan denne blir implementert blir først forklart i avsnitt 5.3, etter å ha gått igjennom optimaliseringsforslagene til matrise-vektor-produktet.

```
1  int p;  // lokalt prosessornummer
2  int n;  // antall lokale gridpunkter
3
4  int nextInnerIncluded = 0;
5  int nextInnerExcluded = n − 1;
6  int lastInner;
7
8  for i in I_p
9     if g(i) == p
10       assignLocalNumber(i, nextInnerIncluded++)
11    else
12       assignLocalNumber(i, nextInnerExcluded--)
13 lastInner = nextInnerIncluded-1;
```



Figur 5.4: Skisse i pseudo-kode av tildeling av lokale gridpunktnummere for å optimalisere indreproduktet. Nederst skisseres hvordan punktene blir liggende i vektorene.

```
1  int last_inner;  // fra figur 5.4
2
3  double parallelInnerProd( double *x, double *y )
4  {
5     double sum = 0;
6     for ( int i=0; i <= last_inner; i++ ) {
7        sum += x[i] * y[i];
8     }
9     MPI_Allreduce( sum );
10    return sum;
11 }
```

Figur 5.5: Skisse av parallelt indreprodukt etter sortering av gridpunktene.

N	P	$\frac{T_I - T'_I}{T_I}$	
		$\frac{d_I}{c_I}$=4	$\frac{d_I}{c_I}$=10
100^2	4	16%	30%
100^2	8	27%	45%
100^2	16	38%	57%
100^2	32	49%	68%
600^2	4	3%	6%
600^2	8	5%	11%
600^2	16	9%	18%
600^2	32	13%	25%
1000^2	4	1%	4%
1000^2	8	3%	7%
1000^2	16	5%	11%
1000^2	32	8%	17%

N	P	$\frac{T_I - T'_I}{T_I}$	
		$\frac{d_I}{c_I}$=4	$\frac{d_I}{c_I}$=10
10^3	4	66%	81%
10^3	8	78%	88%
10^3	16	85%	93%
10^3	32	90%	95%
60^3	4	23%	39%
60^3	8	34%	53%
60^3	16	44%	63%
60^3	32	53%	71%
100^3	4	15%	28%
100^3	8	23%	40%
100^3	16	31%	50%
100^3	32	40%	59%

Tabell 5.1: Estimert gevinst for optimalisert versjon av indreproduktet.

5.1.3 Forbedringspotensiale

Et uttrykk for tiden til Diffpacks implementasjon av indreproduktet, $T_I(P)$, ble gitt i likning 4.4 på side 22. Antar vi en perfekt lastbalansering, vil det nye parallelle indreproduktet ta tid $T'_I(P) = c_I \frac{N}{P} + \rho$ der c_I er den samme proposjonalitetskonstanten som i likning 4.4 og ρ er tiden brukt på kommunikasjon. Dersom vi antar kommunikasjonskostnadene til å være neglisjerbare kan vi da estimere reduksjon av kjøretid:

$$\frac{T_I(P) - T'_I(P)}{T_I(P)} = 1 - \frac{c_I \frac{N}{P} + \rho}{c_I n + d_I \frac{\mathcal{N}-N}{P} + \rho} \approx 1 - \frac{N}{\mathcal{N} + \frac{d_I}{c_I}(\mathcal{N} - N)} \tag{5.2}$$

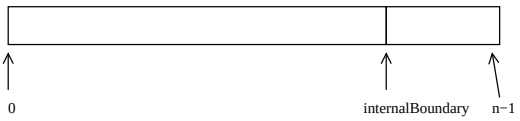
der d_I er en konstant forbundet med merarbeidet ved korreksjon (forklart i avsnitt 4.6).

Tabell 5.1 viser noen estimerte verdier for en slik relativ forbedring. Det er som før brukt ett elements overlapp, bi- og trilineære elementer i henholdsvis to og tre dimensjoner. Estimatet for nP bygger på likning 4.1. Vi ser her at vi kan forvente spesielt stor gevinst ved små problemstørrelser eller ved bruk av mange prosessorer. Gevinsten er betydelig større i tre dimensjoner der man får et høyt antall dupliserte gridpunkter (se tabell 4.1 på side 20).

5.2 Matrise-vektor-produkt

Vi husker fra avsnitt 4.9 at vi hadde to ulike tilfeller for det parallelle matrise-vektor-produktet, én for overlappende og én for ikke-overlappende partisjoneringer (på elementnivå). Begge tilfeller ble løst ved å først beregne et fullt lokalt matrise-vektor-produkt. Fordi den parallelle diskretiseringen ikke ble gjort fullstendig, gav dette ufullstendige svar for gridpunkter på den interne randen. For en overlappende partisjonering løste man dette ved å hente inn korrekte verdier fra en nabopartisjon. For en ikke-overlappende partisjonering

```
1  int internalBoundary; // indeks til første gridpunkt på intern rand
2
3  void matvec( CRS *A, double *x, double *y )
4  {
5     matvec_crs( 0, internalBoundary - 1, A, x, y ); // se figur 2.2
6     send( y );                                      // se figur 4.8
7     recv( y );                                      // se figur 4.8
8  }
```



Figur 5.6: Skisse av parallelt matrise-vektor-produkt for overlappende partisjonering der beregning av punkter på den interne randen er eliminert. Nederst skisseres hvordan punktene er antatt å ligge i vektorene.

måtte man for hvert punkt på den interne randen legge sammen delresultatene til alle prosessorene som også hadde dette punktet.

5.2.1 Eliminering av punkter på intern rand

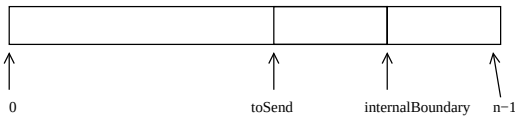
Fordi resultatene etter det lokale matrise-vektor-produktet likevel blir erstattet på den interne randen, er det mulig å hoppe over disse ved bruk av en overlappende partisjonering. Vi får da et tilfelle som ligner på optimaliseringen av indreproduktet. Første skritt er å identifisere punktene som kan utelates, og hvilke som må inkluderes. Indeksene til disse kan lagres i et sett, men dette vil gi en eller annen form for indirekte indeksering i matrise-vektor-produktet. For å eliminere denne indirekte indekseringen kan vi da igjen sørge for at vektorene allerede er sortert etter disse settene.

La oss anta at vi har klart å gi gridpunktene i det lokale gridet nummere slik at punktene som er på den interne randen kommer sist. Vi kan da implementere det parallelle matrise-vektor-produktet som skissert i figur 5.6. Denne antar at vi har implementert en funksjon `matvec_partial`, som gjør et delvis matrise-vektor-produkt mellom du angitte grensene. Jeg vil i avsnitt 5.3 komme tilbake til hvordan nummereringen av de lokale gridpunktene må gjøres i praksis.

5.2.2 Maskering av kommunikasjonskostnader

Med maskering av kommunikasjonskostnader mener jeg å gjøre nyttige beregninger samtidig som kommunikasjon pågår. På denne måten kan man utnytte tid man ellers ville brukt på å vente på svar fra mottaker. Kommunikasjonskostnadene blir således "maskert". I matrise-vektor-produktet er det mulig å maskere kommunikasjonskostnadene fordi det kun er punkter på den interne randen som er avhengig av kommunikasjon. Alle de andre punktene kan beregnes uavhengig av dette.

```
1  int toSend;              // index til første punkt som skal sendes
2  int internalBoundary; // index til første punkt på intern rand
3
4  matvec_masked( CRS *A, double *x, double *y )
5  {
6     matvec_crs( toSend,                  // beregn punkter som skal sendes
7                 internalBoundary - 1,
8                 A, x, y );
9     isend( y );                          // start ikke-blokkerende sending
10    irecv( y );                          // start ikke-blokkerende mottak
11    matvec_crs( 0, toSend - 1,           // beregn resten unntatt intern rand
12                A, x, y );
13    isend_finish( y );                   // avslutt ikke-blokkerende sending
14    irecv_finish( y );                   // avslutt ikke-blokkerende mottak
15                                         // og oppdater intern rand
16 }
```



Figur 5.7: Skisse av parallelt matrise-vektor-produkt for overlappende partisjonering med maskering av kommunikasjon og der beregning av punkter på den interne randen er eliminert. Nederst skisseres hvordan punktene er antatt å ligge i vektorene.

Fremgangsmåten blir følgende: Først gjør man et delvis matrise-vektor-produkt for de punktene som skal sendes til andre prosessorer. Dette er de punktene som er en del av en annen prosessor sin interne rand. Så starter man en ikke-blokkerende (se avsnitt 3.4) sending av disse verdiene og starter ikke-blokkerende mottak av verdiene som må mottas. Mens denne kommunikasjonen pågår, beregnes resten av punktene. Til slutt venter man til kommunikasjonen er ferdig, før man kan gjøre de nødvendige oppdateringer av punktene på den interne randen. Figur 5.7 viser hvordan dette kan implementeres. Funksjonen `matvec_crs` er her den samme som i figur 2.2. Funksjonene `isend`, `irecv`, `isend_finish` og `irecv_finish` har her samme funksjonalitet som funksjonene i figur 4.8, men er modifisert til å bruke ikke-blokkerende kommunikasjon. Jeg vil ikke gå igjennom detaljene av hvordan dette gjøres, og henviser til MPI-manualen[10] for detaljer rundt hvilke endringer som må gjøres.

Implementasjonen i figur 5.7 krever igjen at gridpunktene kommer i en bestemt rekkefølge i vektorene. Dette er av de samme effektivitetshensyn som tidligere. For å kunne være effektiv er det viktig å aksessere vektorerne sekventielt, og det er viktig å unngå indirekte indeksering.

Koden i figur 5.7 antar at vi har en overlappende partisjonering. Det er også mulig å lage en maskert versjon av matrise-vektor-produktet dersom vi har en ikke-overlappende partisjonering. Punktene som skal sendes og punktene som skal mottas vil være de samme, nemlig punktene på den interne randen. Første kall til `matvec_crs` må da gis med `internalBoundary` og $n - 1$ som start og stop parametere. Andre kall må gis med 0 og `internalBoundary-1`.

5.3 Sortering av gridpunkter

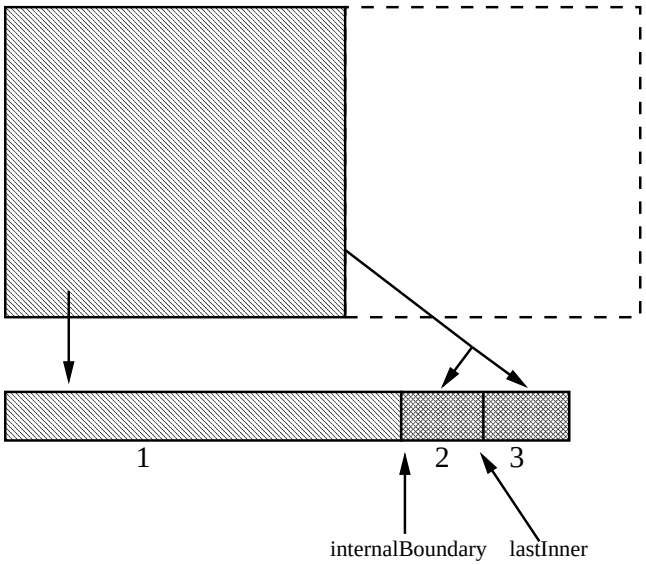
Optimaliseringsforslagene for indreproduktet og matrise-vektor-produktet bygget på at gridpunktene lå i spesielle grupperinger i vektorene. De ulike optimaliseringene hadde ulike krav. For én simulering kan vi derimot bare ha én sortering og vi må finne en sortering som tilfredstiller alle optimaliseringene.

La oss oppsummere de ulike kravene:

- Indreproduktet trengte å dele vektoren i to blokker: En blokk med punkter som skulle inkluderes i det lokale indreproduktet, og en blokk som kunne utelukkes. Punktene som kunne utelukkes ville bli inkludert av en annen prosessor.
- Matrise-vektor-produktet for en ikke-overlappende partisjonering av elementer trengte å skille ut punktene på den interne randen som en egen blokk.
- Matrise-vektor-produktet for en overlappende partisjonering av elementer trengte en blokk med punktene på den interne randen, og en blokk som inneholdt punktene som skulle sendes til en/flere andre prosessorer.

5.3.1 Ikke-overlappende partisjonering

For en ikke-overlappende partisjonering av elementer vil settet med dupliserte punkter da være lik settet med punkter på den interne randen. Resten av

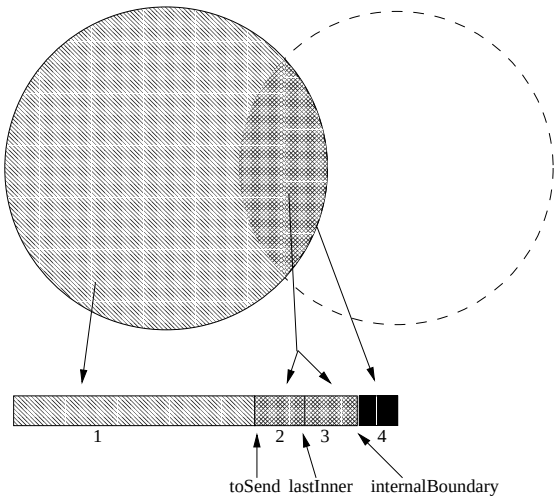


Figur 5.8: Sortering av gridpunkter for ikke-overlappende partisjon. 1: ikke-dupliserte punkter, 2: punkter på grense mot annen partisjon som inkluderes i indreproduktet, 3: punkter på grense mot annen partisjon som ignoreres i indreproduktet.

punktene vil ikke være dupliserte og må tas med i det lokale indreproduktet. Punktene på den interne randen kan deles i to blokker: en med punkter som kan utelukkes i det lokale indreproduktet, og en med punkter som må inkluderes. Hvis vi legger punktene som kan utelukkes sist, og de ikke-dupliserte punktene først, får vi situasjonen i figur 5.8. Denne vil tilfredstille kravene både til optimalisering av indreproduktet og matrise-vektor-produktet.

5.3.2 Overlappende partisjonering

For en overlappende partisjonering er vi garantert at for hvert punkt på den interne randen på én prosessor, vil det eksistere minst én annen prosessor der dette punktet er tilgjengelig uten å være på dens interne rand. La oss derfor legge disse punktene på slutten av vektoren og bestemme oss for at disse ikke trenger å være med i det lokale indreproduktet. Resten av punktene kan klassifiseres som dupliserte eller ikke-dupliserte. De ikke-dupliserte må være med i det lokale indreproduktet. Vi legger disse først i vektoren. Punktene som skal sendes til en/flere andre prosessorer under matrise-vektor-produktet vil være blant de dupliserte. Vi deler derfor de dupliserte punktene opp i to grupper: de som skal være med i det lokale indreproduktet, og de skal utelukkes. De som skal utelukkes legges etter de som må være med. Vi får da situasjonen i figur 5.9. Dette vil tilfredstille kravene til både indreproduktet og matrise-vektor-produktet. Legg merke til at punktene som er merket `toSend` generelt også vil kunne inneholde punkter som ikke kommer til å bli kommunisert. Dette vil som regel gjelde svært få punkter slik at det er lite å tjene på å sortere ut disse.



Figur 5.9: Sortering av gridpunkter for overlappende partisjon. 1: ikke-dupliserte punkter, 2: dupliserte gridpunkter som tas med i indreprodukt, 3: dupliserte gridpunkter som ignoreres i indreprodukt, 4: punkter på grense mot annen partisjon.

5.3.3 Fordeling av punkter til indreprodukt

I tilfellet med en ikke-overlappende partisjonering måtte vi dele punktene på den interne randen opp i de som skulle være med i det lokale indreproduktet, og de som kunne utelukkes. I tilfellet med en overlappende partisjonering måtte vi gjøre samme inndeling på de dupliserte gridpunktene (etter å ha trukket fra punktene på den interne randen). Denne inndelingen blir gjort slik at antall punkter som må inkluderes i indreproduktet, blir så likt som mulig mellom prosessorene.

Anta J_p er settet av punkter som må fordeles på prosessor p og la $J = J_1 \cup \ldots \cup J_P$. La m_p være antall punkter som det hittil er bestemt at prosessor p skal inkludere i sitt lokale indreprodukt. m_p vil initielt være lik antall ikke-dupliserte gridpunkter på prosessor p. Fordelingen skjer da slik at nye punkter hele tiden tildeles den prosessoren som hittil har færrest punkter i sitt indreprodukt. Algoritmen blir som følger:

for $j \in J$ **do**
  $K = \{p : j \in J_p\}$
  $\gamma, \mu = \infty$
  for $p \in K$ **do**
    if $m_p < \mu$ **then**
      $\mu = m_p$
      $\gamma = p$
    end if
  end for
  < prosessor γ tildeles punktet j >
  $m_\gamma = m_\gamma + 1$
end for

5.4 Partisjonering

Som nevnt i avsnitt 4.2 er det to forhold å ta hensyn til under partisjonering. Antall elementer bør være så likt som mulig på hver prosessor, og antall naboelementer som splittes på ulike prosessorer bør være så lavt som mulig. Dette siste kravet er det samme som at den interne randen bør være så liten som mulig, fordi det er er denne som krever kommunikasjon.

Diffpack bruker Metis[22] til partisjonering. Denne bruker tid $\mathcal{O}(N)$ og gir partisjoner med høy kvalitet[22] med tanke på de to kravene nevnt ovenfor. Problemet er derimot at Metis returnerer helt vilkårlige partisjoner. Med det mener jeg at det ikke er noen forhåndsbestemt struktur man kan utnytte. Dette gjør det svært plass- og tidkrevende og legge til overlapp, finne ut hvilke punkter som må kommuniseres og til hvilke prosessorer. Dette arbeidet tar tid $\mathcal{O}(NP)$. Det totale arbeidet forbundet med partisjonering blir altså $\mathcal{O}(N+NP)$ som skalerer svært dårlig. I tillegg har vi sett i avsnitt 4.9.3 at det kreves noe ekstra arbeid hver gang det skal kommuniseres i matrise-vektor-produktet. De riktige punktene må plukkes ut av vektorene og legges i egne meldingsbuffere.

5.4.1 Partisjonering langs én dimensjon

Jeg foreslår her en alternativ algoritme. Denne ser bort ifra kravet om å minimere den interne randen og velger å dele opp det globale gridet langs én dimensjon. Dette blir gjort etter følgende oppskrift (dette ligner på såkalt "recursive coordinate bisubsection"[11]):

1. For hvert element: Beregn elementets massesenter i x-retning.
2. Sorter elementene etter massesenterets x-verdi.
3. Lag en 1-D partisjonering av de sorterte elementene slik at antall elementer på hver prosessor blir så likt som mulig.

Første punkt kan gjøres i tid $\mathcal{O}(N)$. Sorteringen kan gjøres i tid $\mathcal{O}(N \log N)$ med for eksempel quicksort. Utregning av hvilke elementer hver enkelt prosessor får tildelt (tredje punkt), kan gjøres i tid $\mathcal{O}(1)$ med funksjonen i figur 5.10. Figur 5.11 forsøker å gi et eksempel på en partisjonering.

Ved å partisjonere på denne måten ønsker jeg å oppnå to egenskaper:

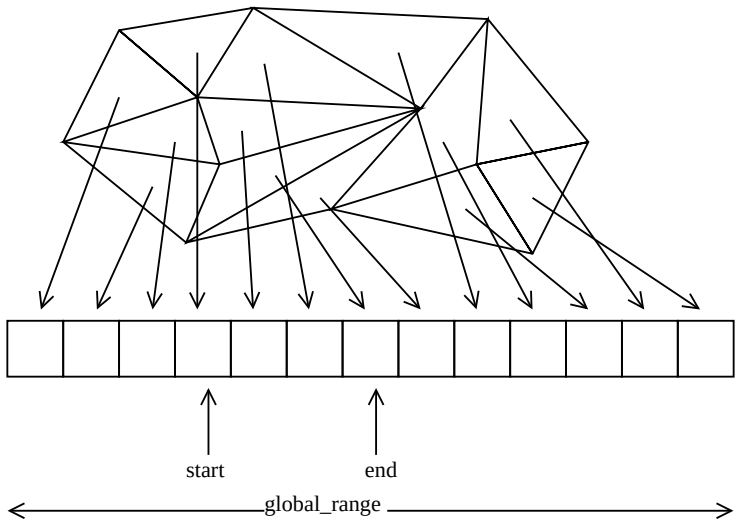
1. En prosessor må kommunisere med maksimalt to andre prosessorer.
2. Et gridpunkt må maksimalt kommuniseres mellom ett par av prosessorer.

Disse to egenskapene kan ikke garanteres for alle grid. Det kreves at det globale området i x-retning er "bredt nok" i forhold til antall prosessorer i bruk. Dersom ett av disse punktene ikke kan tilfredstilles, vil hele den videre simuleringen feile. Dette kan også oppdages under partisjoneringen, slik at den gamle metoden med Metis kan brukes isteden.

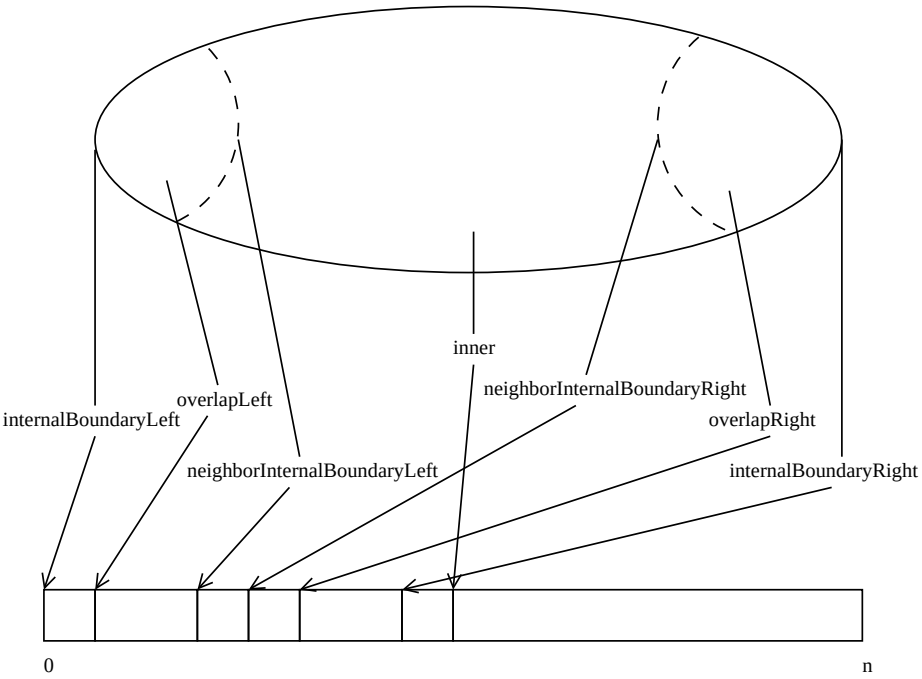
Det å legge til overlapp blir nå en relativt enkel jobb. Jeg har valgt å la hver prosessor legge til overlapp i kun én retning. Dette vil forskyve fordelingen av antall elementer noe ved at prosessoren i det ene endepunktet får noe færre elementer enn de andre. Dette tar tid $\mathcal{O}(n) \approx \mathcal{O}(\frac{N}{P})$.

```
1  void getLocalRange( int rank, int global_range,
2                      int &start, int &end ) const
3  {
4     int length = global_range / nprocs;
5     int num_extra = global_range % nprocs;
6
7     if ( rank < num_extra ) {
8        start = (length+1)*rank + 1;
9        end = (length+1)*(rank+1);
10    }
11    else {
12       start = length*rank + num_extra + 1;
13       end = start + length - 1;
14    }
15 }
```

Figur 5.10: Funksjon som beregner lokal andel av globalt grid. rank er her det lokale prosessornummeret, nprocs er det globale antall prosessorer. global_range er det globale antall elementer. start og end angir grensene for den lokale partisjonen.



Figur 5.11: Eksempel på globalt grid i to dimensjoner og hvordan elementene tenkes sortert etter massesenterets koordinat i x-retning. start og end avgrenser den lokale partisjonen for en prosessor.



Figur 5.12: Skisse av en lokal partisjon og hvordan gridpunktene lagres blokkvis i vektorene med partisjonering langs én dimensjon.

Det bør nevnes at Diffpack faktisk implementerer en liknende mulighet for partisjonering langs én dimensjon. Diffpack sin implementasjon fungerer imidlertid bare på strukturerte grid. Min implementasjon har ikke denne begrensningen.

5.4.2 Optimaliseringer

Figur 5.12 viser en generell lokal partisjon og hvordan gridpunktene lagres blokkvis i vektorene. Det skilles mellom ikke-dupliserte gridpunkter ("inner"), intern rand ("internalBoundary") og den lokale delen av nabo-partisjonenes interne rand ("neighborInternalBoundary"). Alle disse blokkene deles opp i høyre og venstre del.

Indreprodukt

I det lokale indreproduktet inkluderes nå de ikke-dupliserte punktene, samt alle dupliserte punkter til høyre. Ser vi på figur 5.12 er dette alle punktene fra og med "neighborInternalBoundaryRight". Dette vil gi litt færre punkter å beregne i partisjonen lengst til høyre, men vi beholder optimaliseringen der vi har eliminert korreksjon av dupliserte gridpunkter og indirekte indeksering.

```
1  int p_left;   // prosessor med venstre nabo-partisjon
2  int p_right; // prosessor med høyre nabo-partisjon
3
4  void send( double *x )
5  {
6     MPI_Send( x,
7               start=neighborInternalBoundaryLeft,
8               end=neighborInternalBoundaryRight-1,
9               p_left );
10    MPI_Send( x,
11              start=neighborInternalBoundaryRight,
12              end=overlapRight-1,
13              p_right );
14 }
15
16 void recv( double *x )
17 {
18    MPI_Send( x,
19              start=internalBoundaryRight,
20              end=overlapRight-1,
21              p_right );
22    MPI_Send( x,
23              start=internalBoundaryLeft,
24              end=overlapLeft-1,
25              p_left );
26 }
```

Figur 5.13: Skisse i pseudo-kode av sending og mottak ved parallelt matrise-vektor-produkt ved bruk av overlappende partisjonering langs én dimensjon.

Matrise-vektor-produkt

Matrise-vektor-produkt kan maskere kommunikasjonskostnader som i figur 5.7, men med litt andre parametere til det delvise matrise-vektor-produktet. Før kommunikasjonen beregnes punktene på den interne randen, dvs fra og med "internalBoundaryLeft" til og med "inner"-1. Samtidig som kommunikasjonen pågår beregnes så punktene i "inner".

Maskering av kommunikasjon i et matrise-vektor-produkt med overlappende partisjoner blir også som i figur 5.7. Før kommunikasjon beregnes nå punktene fra og med "overlapLeft" til og med "internalBoundaryRight"-1. Under kommunikasjonen beregnes så punktene i "inner". Denne metoden vil altså også eliminere den unødvendige beregningen av punkter på de interne rendene.

Kommunikasjon

Figur 5.13 skisserer hvordan sending og mottak blir gjort i matrise-vektor-produktet ved bruk av overlappende partisjonering. Sammenlikner vi med metoden Diffpack bruker i figur 4.8, ser vi at vi har eliminert all buffring. I tillegg har vi eliminert all bruk av tabeller som forteller hvor de mottatte grid-

punktene skal lagres.

Dette er mulig fordi punktene som skal kommuniseres ligger etter hverandre i egne blokker i vektorene, og fordi det er separate blokker for hver prosessor det må kommuniseres til. For at rekkefølgen på punktene som sendes skal passe nøyaktig til rekkefølgen mottakeren vil legge disse i sine vektorer på, er gridpunktene innenfor hver gruppe sortert på globalt gridpunktnummer. "neighborInternalBoundaryLeft" vil da være identisk lik "intenalBoundaryRight" for prosessoren til venstre. Tilsvarende for de andre gruppene.

Eliminasjon av buffring ved mottak av gridpunkter, er kun mulig dersom de mottatte punktene skal erstatte allerede eksisterende verdier. Dette er ikke tilfelle i matrise-vektor-produktet ved bruk at ikke-overlappende partisjoner. De mottatte punktene skal da adderes til allerede eksisterende verdier (se avsnitt 4.9). Det må da brukes et buffer for mottak, men vi har likevel eliminert behovet for bruk av tabellene for hvor de ulike mottatte punktene skal lagres. Tilsvarende kan vi ikke unngå buffring ved bruk av prekondisjonering. Etter lokal prekondisjonering tar man en global gjennomsnittsverdi av de punktene som er dupliserte (se avsnitt 2.2).

Kapittel 6

Tester

Vi har nå sett på hvordan elementmetoden er parallellisert i Diffpack (kapittel 4), og på noen optimaliseringsforslag utover dette (kapittel 5). Dette kapittelet vil forsøke å teste hvorvidt optimaliseringsforslagene gir noen gevinst i praksis. Jeg starter med å redegjøre for plattformene testene er utført på, og hvilke målings-teknikker som er brukt. Videre kommer målinger av kommunikasjonskostnader på de ulike plattformene før de ulike optimaliseringsforslagene blir målt og sammenliknet med Diffpack sin originale implementasjon.

All måling av tid blir oppgitt med enheten sekunder. Alle meldingslengder er oppgitt med lengden av datatypen double som enhet. Denne er på 8 byte.

6.1 Testede plattformer

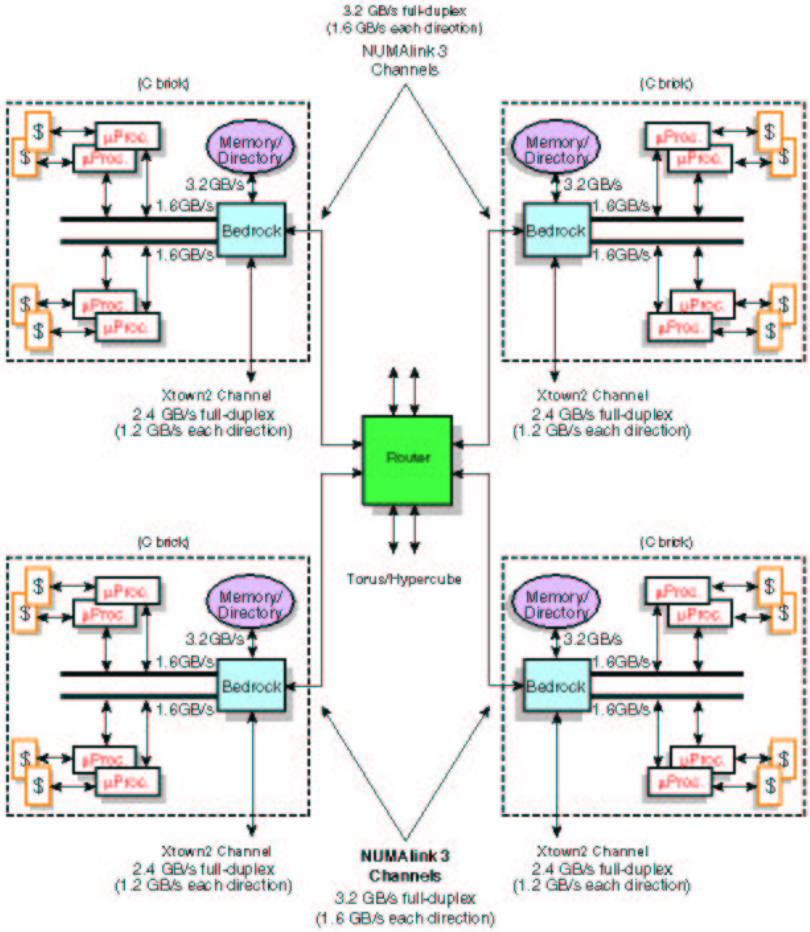
Jeg har hatt to parallelle plattformer til rådighet til testene i denne oppgaven. Tabell 6.1 oppsummerer noen av de tekniske spesifikasjonene ved disse.

"Diplopodus" er en såkalt "klynge av arbeidsstasjoner". Denne består av 16 vanlige pc'er. Hver pc består av 2 prosessorer som har 1GB delt minne. Pc'ene er koblet sammen med 100Mbit ethernet gjennom en switch. Dette gir en topologi som kan minne om en stjerne (se avsnitt 3.1).

Alle testene på diplopodus er gjort slik at det kun brukes én prosessor på hver maskin for testene, som bruker inntil 16 prosessorer. Ved bruk av over 16 prosessorer vil det være to prosessorer i bruk på hver maskin. Diplopodus er en maskin som er lite brukt. Jeg har derfor kunnet teste på denne maskinen uten at det har vært andre samtidige brukere. Dette vil hjelpe til med å redusere usikkerheten i målingene.

"Embla" er en såkalt supermaskin. Denne består av 512 prosessorer koblet sammen som en hyperkube. Grupper på 4 prosessorer deler på 4GB minne. Maksimale kommunikasjonshastigheter ligger teoretisk på størrelsesorden gigabit per sekund. Figur 6.1 viser en skisse av hvordan 16 prosessorer er koblet på embla.

Kjøring av programmer på embla skjer via et køsystem. Det er køsystemet som står for tildelingen av prosessorer. Jeg har derfor ikke hatt noen kontroll over hvilke prosessorer som er brukt i testene. Embla er en maskin som er svært mye brukt. Det har til enhver tid vært en rekke andre brukere av maskinen. Usikkerheten ved målingene på embla er derfor noe høyere enn på diplopodus.



Figur 6.1: Figur hentet fra SGI Origin 3000 Series Technical Configuration Owner's Guide[33]. Symbolet $ viser her til det lokalt hurtigminne (engelsk: cache). Kommunikasjonshastighetene er angitt som maksimalverdier.

	diplopodus	**embla**
Prosessor	Pentium III 1GHz	MIPS R14000
Antall prosessorer	32	512
1. nivå hurtigminne	16KB instruksjon 16KB data	32KB instruksjon 32KB data
2. nivå hurtigminne	256KB	8MB
Hovedminne	1GB per 2 prosessorer	4GB per 4 prosessorer
Nettverk	100Mbit ethernet koblet i switch	3.2Gbit hyperkube

Tabell 6.1: Oppsummering av noen tekniske data for de ulike parallelle maskinene.

```
1 double t = MPI_Wtime();
2 inner_prod(x, y, n);
3 t = MPI_Wtime() - t;
```

Figur 6.2: Enkel tidtakning av funksjonen inner_prod.

6.2 Tidtakning

Figur 6.2 viser en enkel måte å måle tiden brukt på en kodebit. Kodebiten er her funksjonen `inner_prod` som gjør et indreprodukt på to vektorer med 5000 elementer. `MPI_Wtime` er en funksjon som returnerer tiden siden programstart i sekunder.

Figur 6.3 viser noen målinger gjort med denne metoden. Vi ser her tydelig at en slik enkel måle-teknikk ikke er deterministisk. Den første målingen tar her lengst tid. Dette skyldes at vektorene i indreproduktet ikke er tilgjengelig i det lokale hurtigminnet første gang koden blir kjørt. Neste gang koden kjører vil disse vektorene (ihvertfall delvis) være tilstede, slik at kjøretiden går ned.

Ser vi bort ifra første målepunkt, ser tiden ut til å veksle mellom 5 og 6 mikrosekunder. Dette skyldes at funksjonen `MPI_Wtime` har en oppløsning på ett mikrosekund på de plattformene jeg har testet, og den ikke klarer å måle tidsrom som er kortere enn dette.

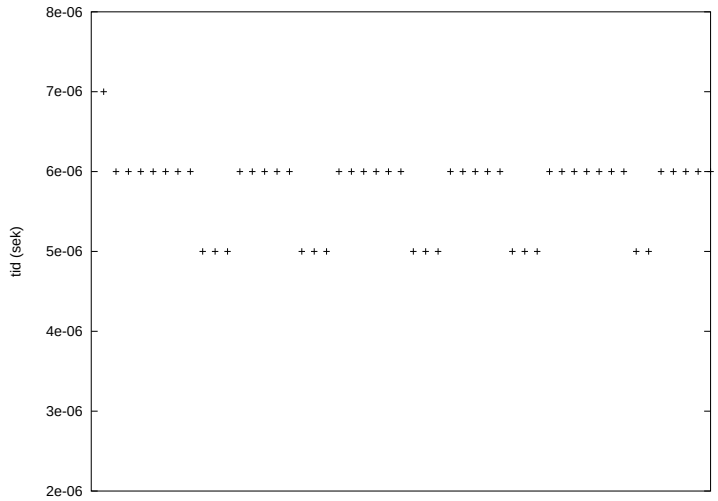
En annen viktig effekt som kan opptre ved tidsmåling av kode, er *avbrudd*. Denne effekten er ikke synlig i figur 6.2. Et avbrudd er et spesielt signal til prosessoren som avbryter det kjørende programmet og overlater kontrollen til operativsystemet. De fleste operativssystem bruker avbrudd til blant annet å implementere såkalt tidsdeling[37] slik at mange programmer kan kjøre samtidig på samme prosessor. Avbrudd er også viktig i implementasjon av kommunikasjon. For eksempel kan nettverkskortet initiere et avbrudd når en ny melding er klar til å mottas. Avbrudd gjør at en kodebit kan bli tilfeldig avbrutt, og det kan da ta vilkårlig lang tid før den får fortsette igjen[1].

6.2.1 Valg av klokke

Som en konsekvens av avbruddsmekanismen, snakker man om to ulike tider for en eksekvert kode: Klokketid (engelsk: wall-time) og prosessortid (engelsk: CPU-time). Prosessortiden er den tiden prosessen faktisk har vært aktivt kjørende i prosessoren. Denne kan som regel leses av med spesielle systemkall til operativsystemet. Klokketiden er det faktiske tidsrommet mellom kodens start til den er ferdig og leses av med for eksempel `MPI_Wtime`.

Når man skal sammenlikne ulike versjoner av en kode, kan det ofte være best å bruke prosessortiden. Dette for å eliminere eksterne forhold som tilfeldige avbrudd så langt det er mulig. Det meste av koden jeg skal måle inneholder derimot kommunikasjon. Kommunikasjon kan innebære at man må vente på svar fra motparten som ofte implementeres blant annet ved hjelp av avbrudd. Venting og avbrudd må altså inkluderes i tidtakningen. Det blir da galt å måle prosessortiden og man må ta utgangspunkt i klokketiden.

[1] Dette er ikke tilfelle for såkalte sanntidssystemer der operativssystemet gir visse garantier for hvor mye en prosess får kjøre.



Figur 6.3: 50 tidsmålinger av indreprodukt mellom to vektorer av lengde 5000. Målingene er her gjort med MPI_Wtime() som her har en oppløsning på 10^{-6} sekund. Samme kode er målt med en høyoppløslig klokke til 5,700 mikrosekunder.

Som vi så i figur 6.3 kan klokkens oppløsning være av betydning, spesielt for små kodebiter. Selve funksjonen som leser av tiden vil også bruke tid. Funksjonens returverdi vil være et tidspunkt en eller annen gang mellom funksjonens start og slutt. Dette kan også være av betydning for små kodebiter. En vanlig metode for å korrigere for dette, er å trekke fra tiden målt for en tom kodebit. En slik korreksjon kan imidlertid gi et underestimat av tiden[15].

Ved bruk av MPI er alltid funksjonen `MPI_Wtime` tilgjengelig. Denne har en oppløsning på ett mikrosekund på de plattformene jeg har testet. En prosessor med en klokkefrekvens på 1GHz vil på denne tiden kunne eksekvere størrelsesorden 1000 instruksjoner. En slik oppløsning blir derfor knapt for små kodebiter. I tillegg til `MPI_Wtime` har jeg derfor basert meg på en assemblerinstruksjon som returnerer antall klokkesykler siden oppstarten av en maskin. Koden til denne er gitt i appendiks B, linje 48-53. Denne funksjonen blir brukt direkte (inline) slik at funksjonskall blir unngått. Dette minimerer tiden brukt på selve tidtakningen samtidig som jeg får en svært høy oppløsning. Denne instruksjonen er kun tilgjengelig på Intel Pentium-kompatible maskiner (diplopodus). På embla har jeg måtte bruke `MPI_Wtime`.

6.2.2 Korreksjon av målt tid

Tiden målt på en tom kodebit med den høyoppløslige klokken er målt til 8 klokkesykler. Det er uvisst hvor mye av denne tiden som kan utnyttes av annen kode i relle målinger. 6 klokkesykler vil være en øvre terskel fordi to klokkesykler må brukes til eksekvering av selve klokken, mens 0 er en nedre terskel. Jeg antar en mellomverdi, og trekker derfor fra 3 klokkesykler fra alle tidsrom målt med denne klokken.

```
1 inner_prod(x, y, n); // cache warm-up
2 double t = MPI_Wtime();
3 for( int i=0; i < num_repititions; i++ ) {
4    inner_prod(x, y, n);
5 }
6 t = (MPI_Wtime() - t) / num_repititions;
```

Figur 6.4: Tidtakning med oppvarming av lokalt hurtigminne (engelsk: cache-warmup).

Ved sammenlikning av `MPI_Wtime` og den høyoppløslige klokken på små arbeidsmengder, har det vist seg at `MPI_Wtime` konsekvent returnerer tidslengder som er ett mikrosekund for høye[2]. Alle tidsrom målt med `MPI_Wtime` er derfor korrigert med dette.

Som sagt returnerer den høyoppløslige klokken, tiden målt i antall klokkesykler. Dette er blitt omregnet til sekunder. En slik omregning vil innebære en relativt stor usikkerhet. Fokus i denne oppgaven er å sammenlikne tidsrom, ikke selve tidsrommene. Omregningen til sekunder er gjort for å gjøre tabellene mer lesbare. All målt tid i tabellene er oppgitt i sekunder.

6.2.3 Teknikker for måling av tid

I arbeidet med å ta gode tidsmålinger, har jeg studert en rekke eksisterende verktøy beregnet på MPI[16, 12, 29, 8, 28, 32, 9, 19]. Felles for alle disse er at de er beregnet på å teste ytelsen på en parallell plattform for å kunne sammenlikne med andre plattformer. Mitt mål er derimot å kunne sammenlikne ulike kodebiter på samme plattform. Jeg legger derfor vekt på å få reproduserbare forskjeller, og ikke nødvendigvis korrekte enkeltverdier.

Metode 1: Middelverdi

Den vanligste metoden er å kjøre den aktuelle kodebiten mange ganger og måle den totale tiden. Tankegangen er her at lang kjøretid reduserer innvirkning av feil i klokken og at variasjoner i kjøretid blir jevnet ut. For å unngå effekten med ekstra lang tid for første måling, "varmer man" opp hurtigminnet ved å først gjøre en enkel kjøring av koden som ikke blir inkludert i totaltiden. Denne metoden er skissert i figur 6.2.3.

Det er mange svakheter med denne metoden. For små koder vil for-løkken kunne ha en betydelig innvirkning på totaltiden. Et annet problem for små koder er at eventuelle avbrudd vil få stor innvirkning på middelverdien.

Metode 2: Minimumsverdi

Bruk av middelverdi er en populær metode. Dette fordi man samtidig kan beregne et standardavvik som et mål på hvor stor feil det er målingen. Strengt tatt forutsetter bruken av standardavvik at vi måler på en normalfordeling[3, 36].

[2] Egentlig MPI_Wtick() for høye.

En normalfordeling kjennetegnes blant annet ved at det er størst sannsynlighet for å måle middelverdien, og at det er like stor sannsynlighet for å måle en verdi som er mindre enn middelverdien, som en som er større. Dette er ikke tilfellet når vi måler tiden på programkode.

For små arbeidsmengder er sannsynligheten for et avbrudd liten. Dersom det faktisk forekommer et avbrudd, vil det målte tidsintervallet øke. Noen effekter som kan redusere kjøretiden vil derimot ikke forekomme. Sannsynlighetsfordelingen for målt tid på en liten arbeidsmengde vil derfor ha et maksimum nær målt minimumsverdi og være lik middelverdien.

Sannsynligheten for avbrudd øker med økt arbeidsmengde. Vi ønsker likevel å minimere måling av avbrudd. Ved å bruke målt minimumsverdi minimerer man disse effektene. I mine målinger vil jeg derfor stort sett bruke minimumsverdier.

Metode 3: Medianverdi

En ulempe med å bruke minimumsverdien, er at den ser bort ifra problemet om usikkerhet i klokken. Figur 6.3 viser problemet med klokkens oppløsning ganske tydelig. I figur 6.3 er målt minimumsverdi lik 5 mikrosekunder, mens "korrekt" verdi målt med en høyoppløslig klokke, er 5,7 mikrosekunder. De fleste målingene på figur 6.3 ser likevel ut til å måle 6 mikrosekunder. Så lenge korrekt verdi er nærmere 6 enn 5, vil sannsynligheten for å måle 6 være størst. Tilsvarende vil medianverdien med høy sannsynlighet ligge nærmere korrekt verdi enn minimumsverdien. For så små tidsintervaller, er sannsynligheten for andre effekter, som avbrudd, svært liten og får ingen effekt på medianverdien.

Ingen av de verktøy jeg har sett på, bruker målt medianverdi. Jeg vil kun bruke disse verdiene til måling av små tidsintervaller.

Metode 4: Middelverdi med avskjæring

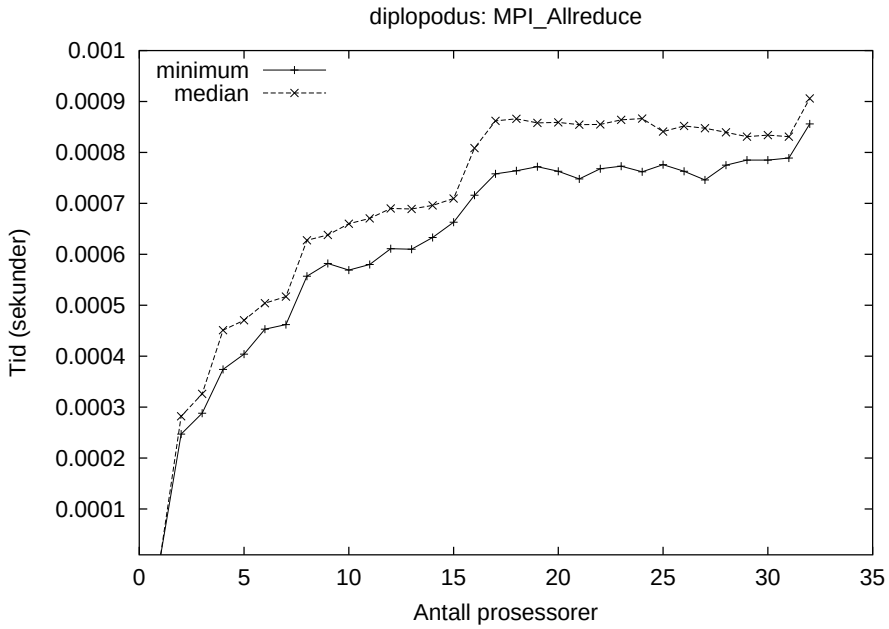
Blant de vektøy jeg har sett på er "middelverdi med avskjæring"[31, 30] den som kommer nærmest bruk av medianverdi. Man kaster da en viss andel av ekstremverdiene før middelverdi og standardavvik beregnes. Dette bør gi resultater svært lik medianverdien, men er mer kostbart å beregne. Jeg beregner derfor ikke denne verdien i noen av mine målinger.

6.3 Kommunikasjonskostnader

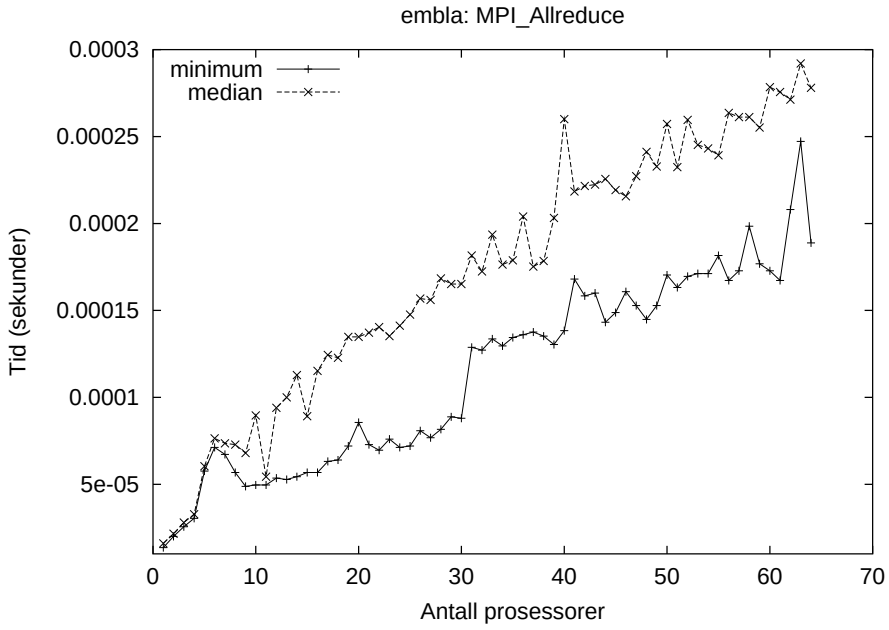
Når jeg i kapittel 4 og 5 har estimert verdier av speedup for parallell kode, har jeg alltid antatt kommunikasjonskostnadene å være neglisjerbare. Dette avsnittet forsøker å måle kommunikasjonskostnadene som er forbundet med indreproduktet og matrise-vektor-produktet for å senere kunne si noe om under hvilke forhold denne antagelsen kan gjelde.

6.3.1 Indreproduktet

Til indreproduktet ble det brukt en global reduksjon som i MPI utføres av funksjonen `MPI_Allreduce`. Denne reduserer (her: summerer) delresultater fra hver enkelt prosessor til én verdi. Figur 6.5 viser målinger av `MPI_Allreduce`



Figur 6.5: MPI_Allreduce målt for varierende antall prosessorer på diplopodus. Hvert målepunkt er her målt 16 ganger.



Figur 6.6: MPI_Allreduce målt for varierende antall prosessorer på embla. Hvert målepunkt er her målt 32 ganger.

på diplopodus. Vi ser her at tiden flater ut ved økende antall prosessorer, og ender på noe under ett millisekund for 32 prosessorer. Tilsvarende målinger på embla (figur 6.6) antyder at tiden øker lineært. Likevel er tiden på embla vesentlig lavere enn tiden på diplopodus. For 32 prosessorer ser den ut til å være en tiendedel av tiden på diplopodus.

6.3.2 Matrise-vektor-produktet

I matrise-vektor-produktet blir det brukt kommunikasjon på formen prosessor-til-prosessor. Avsnitt 3.4 diskuterte de ulike funksjonene tilgjengelig i MPI for slik kommunikasjon. En vanlig metode for å si noe om kostnadene ved slik kommunikasjon, er den såkalte ping-pong testen. Denne går ut på å måle tiden det tar å sende en melding fram og tilbake mellom to prosessorer. Ved å variere meldingslengden og hvilke funksjoner som blir brukt til sending og mottak, kan man få hint om hvilke funksjoner som egner seg til kommunikasjon for den aktuelle plattformen.

Ping-pong testen har mange begrensninger. Sammenlikning av ulike funksjoner tar for eksempel ikke hensyn til at funksjonene er ment til ulik kontekst. Testen er også spesielt dårlig egnet på nettverksarkitekturer der kommunikasjonstiden kan variere mellom ulike par av prosessorer. Dette er spesielt tilfellet på embla, men også på diplopodus ved bruk av over 16 prosessorer.

Maskering av kommunikasjonskostnader

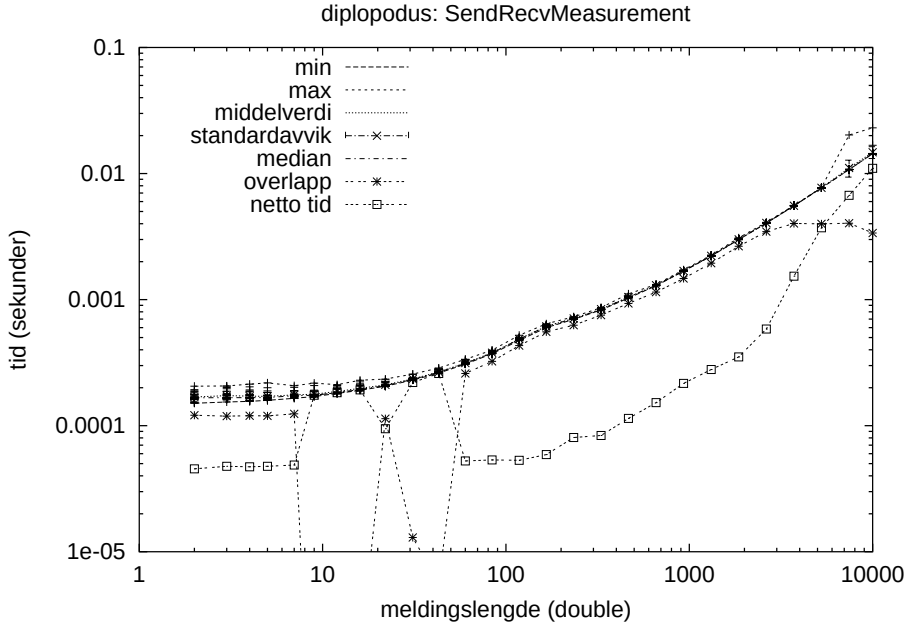
Det er til en viss grad mulig å teste hvorvidt det er mulig å maskere kommunikasjonskostnader ved å gjøre beregninger samtidig med kommunikasjon. Ut fra diskusjonen i avsnitt 3.4 må man regne med at ulike kommunikasjons-modi kan ha ulik grad av merarbeid. Dette får konsekvenser for hvor mye av kommunikasjonstiden som potensielt kan brukes til annen beregning.

Av de verktøy jeg har sett på, er det kun mpptest[16, 18] som gjør noe forsøk på å måle potentialet for samtidig kommunikasjon og beregning. I mpptest er ideen å gjenta ping-pong målinger med eksponensielt økende samtidig beregning. Stoppkriteriet, som definerer potensiale for samtidig beregning, er satt til når totaltiden er doblet i forhold til en ping-pong uten samtidig beregning.

Denne metoden inneholder svært store usikkerhetsmomenter. For det første vil den eksponentielle økningen av arbeidsmengden gi en usikkerhet i resultatverdien som også vokser eksponentielt. Et annet poeng er at enhver test gjort på denne måten vil gi et estimat for samtidig kommunikasjon som er minst like høy som den rene ping-pong-testen. I praksis er det ikke sikkert at noe av tiden er tilgjengelig til kommunikasjon, slik at dette resultatet kan være villedende.

Jeg foreslår istedet følgende metode for å estimere potensiale for maskering av kommunikasjonskostnader. Anta en allerede målt ping-pong verdi, t_{pp}. Anta videre en ny ping-pong-test der man utfører en lokal beregning mellom sending og mottak. En slik måling vil ta tid $t'_{pp} = t_{pp} + t_c - t_m$, der t_c er tiden brukt til lokal beregning og t_m er tiden der kommunikasjon og beregning foregår samtidig. Jeg definerer da potensialet for maskering lik den maksimale t_c slik at $t'_{pp} \leq t_{pp}$. Dette vil gi $t_c \leq t_m$.

For å bestemme t_c i praksis, bruker jeg en type binærsøk med varierende arbeidsmengde. Dette binærsøket er ytterligere dokumentert i appendiks A.



Figur 6.7: Ping-pong testen på diplopodus ved bruk av MPI_Send og MPI_Recv. Hvert målepunkt er målt 30 ganger.

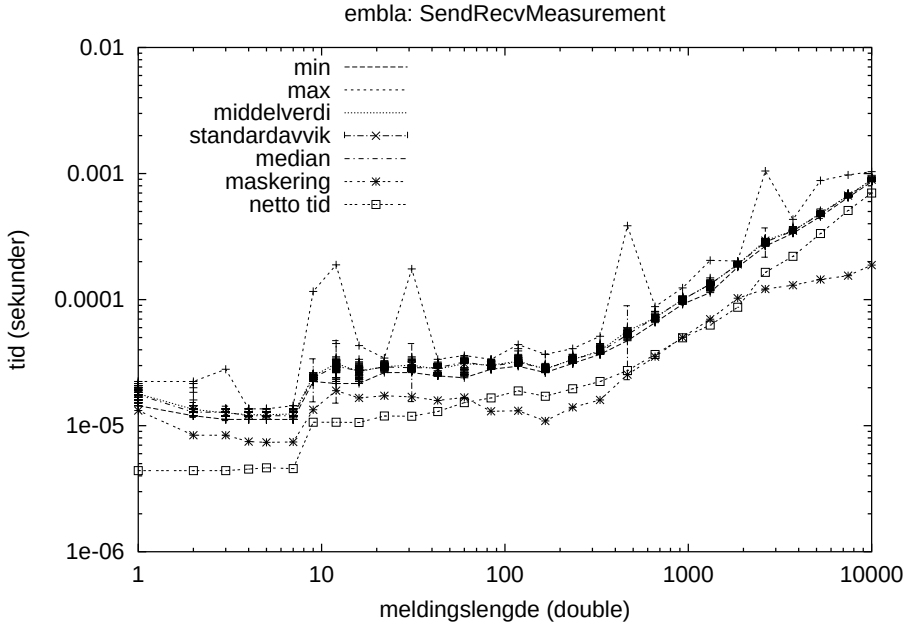
Denne metoden har en potensielt svært god oppløsning i resultatverdiene. Avhengig av kommunikasjonsprotokoll brukt i MPI vil denne metoden også kunne resultere i et potentiale for maskering lik null. Dette i motsetning til metoden brukt i mpptest. Praktisk bruk av resultatene må fortsatt gjøres med forsiktighet, da det er mange faktorer som kan spille inn på potentiale for maskering.

Strengt tatt vil målte verdier kun være gyldige for kommunikasjonsmønstre lik ping-pong testen. En mer nøyaktig metode ville vært å starte kommunikasjon samtidig på begge prosessorer. En slik samtidighet er dessverre vanskelig å garantere i praksis.

Målinger

Figur 6.7 og 6.8 viser ping-pong testen målt med MPI_Send og MPI_Recv på henholdsvis diplopodus og embla. Vi ser her at forsinkelsen (se avsnitt 3.1) er relativt høy, henholdsvis rundt 0.1 og 0.01 millisekund på diplopodus og embla. Vi ser også at kurvene er relativt flate for små meldingsstørrelser. Dette betyr at kostnaden per meldingsenhet er relativt dyr for små meldinger i forhold til lengre meldinger. Grensen for hva jeg kaller "små" meldinger, ser her ut til å ligge på en meldinglengde rundt 100. Dette forteller oss at det innenfor denne grensen ikke er kritisk for et programs parallelle effektivitet å redusere størrelsen på meldingene som utveksles.

For meldingslengder på over 100 blir kurvene vesentlig brattere. Ved hjelp av minste kvadraters metode, kan man estimere stigningstallet til å være $1.39 \cdot$



Figur 6.8: Ping-pong testen på embla ved bruk av MPI_Send og MPI_Recv. Hvert målepunkt er målt 100 ganger.

10^{-3} på diplopodus og $8.59 \cdot 10^{-8}$ på embla[3]. Dette tilsvarer en kommunikasjonshastighet på 186 MB per sekund på embla og 11,5 MB per sekund på diplopodus. Verdien for diplopodus stemmer godt overens med et nettverk på 100 Mbit (11.5 MB = 9.2 Mbit). Hastigheten for embla har riktig størrelsesorden, men er en del lavere enn hva man kunne forvente ut fra figur 6.1. Dette kan tyde på at prosessorene i denne testen kan ha ligget et stykke fra hverandre.

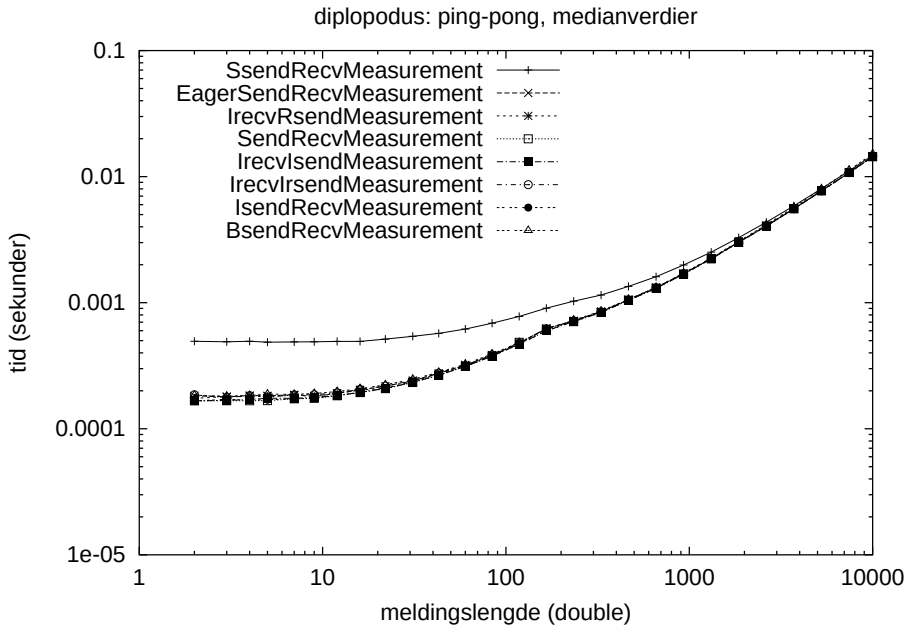
Vi ser også at for denne type kommunikasjon kan det være betydelig å hente på å maskere kommunikasjonskostnader med lokal beregning. Netto oppstartstid (median minus beregnet potentiale for samtidig lokal beregning) kommer ned i 10^4 klokkesykler (≈ 10 mikrosekunder) på diplopodus og noen få mikrosekunder på embla.

Medianverdier

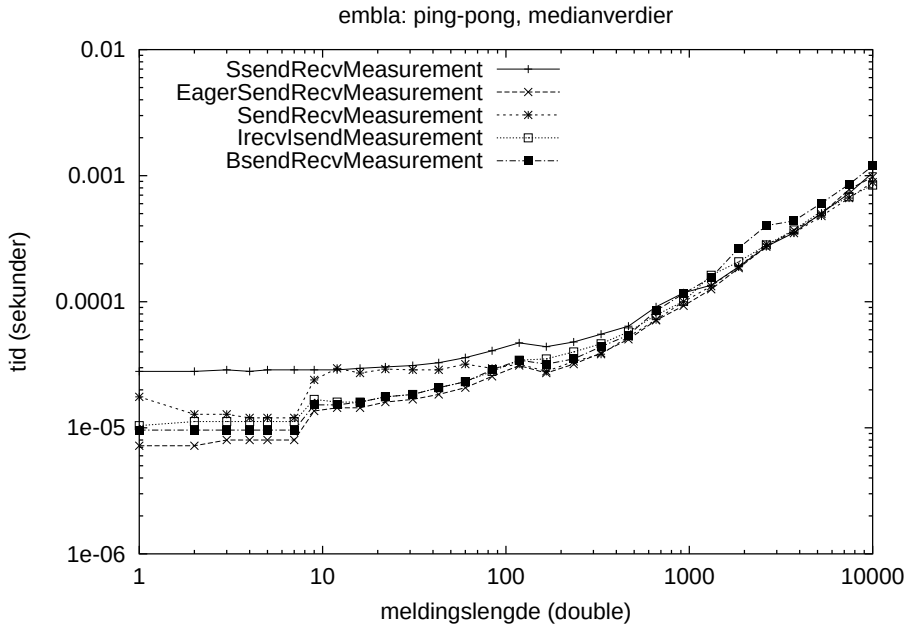
Figur 6.9 og 6.10 viser en sammenlikning av ping-pong testen for noen ulike funksjonskall. For det første legger vi her merke til testen SsendRecv. Denne bruker synkron modus ved hjelp av funksjonen MPI_Ssend, som ser ut til å en svært dyr modus på disse plattformene.

På embla ser vi at testen EagerSendRecv ser ut til å gi best ytelse. Denne testen bruker vanlig MPI_Send og MPI_Recv, men mottak er garantert startet hos mottaker før melding blir sent fra avsender. Denne forsøker således å fremprovosere en ivrig protokoll. Resultatet her tyder på at dette kan gi en viss gevinst.

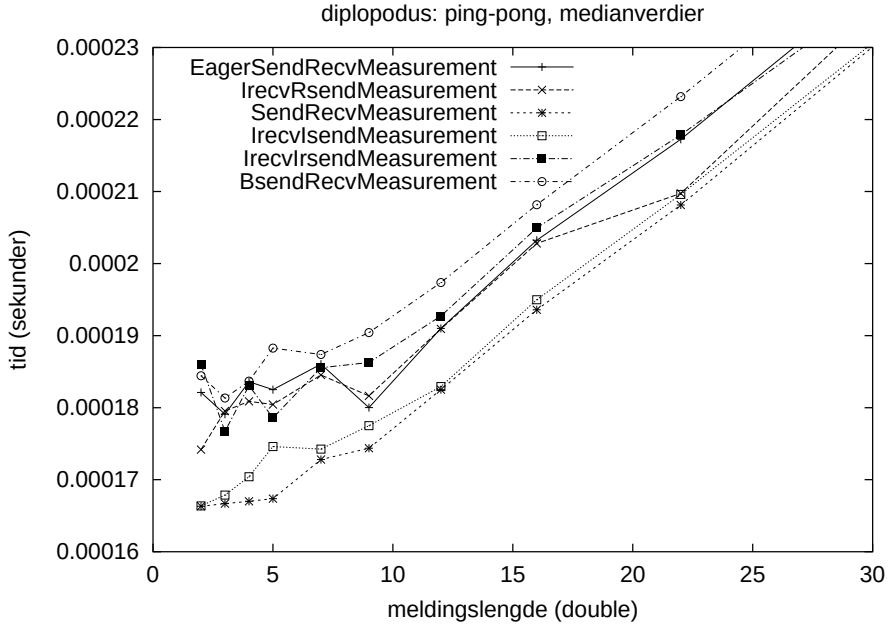
[3]Estimat av stigningstall bygger på minimumsverdier.



Figur 6.9: Ping-pong testen for ulike funksjonskall. Hvert målepunkt er målt 30 ganger.



Figur 6.10: Ping-pong testen for ulike funksjonskall. Hvert målepunkt er målt 100 ganger.



Figur 6.11: Ping-pong testen for ulike funksjonskall. Hvert målepunkt er målt 30 ganger.

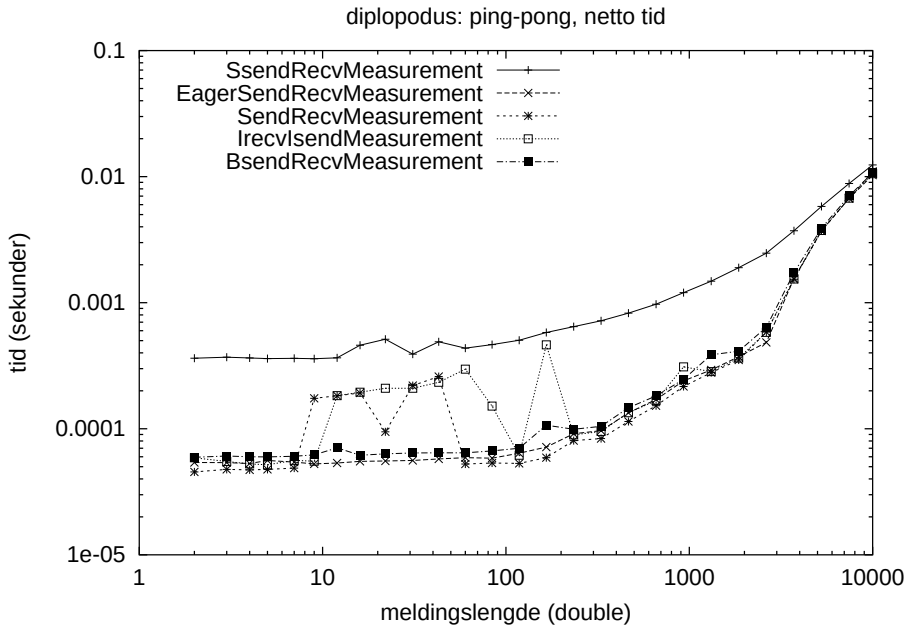
Resultatet er ikke så entydig på diplopodus. Figur 6.11 viser et mindre utklipp for små meldinger. Her ser det ut som om bruk av vanlig MPI_Send og MPI_Recv er det raskeste. Bruk av MPI_Irecv og MPI_Isend ligger på en god annenplass.

Netto tid

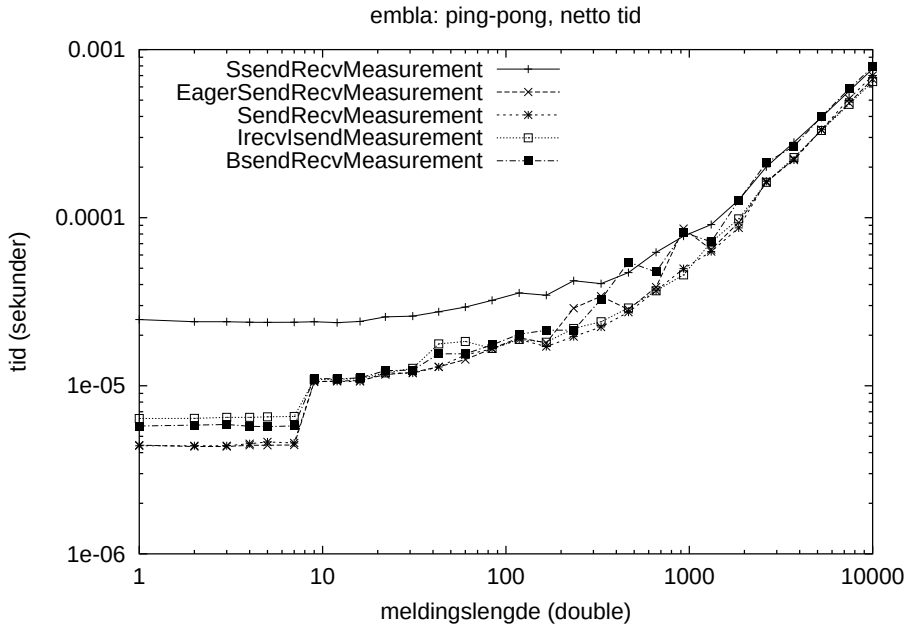
Figur 6.12 på neste side og 6.13 på neste side viser netto tid for ulike tester. Netto tid er medianverdien for ping-pong testen fratrukket estimert potensiale til maskering. Vi ser her at ivrig send ser ut til å være best, sammen med vanlig MPI_Send og MPI_Recv. For store meldinger er MPI_Irecv og MPI_Isend like bra. På diplopodus ser bruk av vanlig MPI_Send og MPI_Recv ut til å være best. Forskjellene ser derimot ut til å være beskjedne.

Konklusjon

Testene har antydet at det kan være en fordel å forsøke å fremprovosere en såkalt ivrig protokoll (se avsnitt 3.4.1). Ved implementasjon av matrise-vektorproduktet har jeg derfor organisert all kommunikasjon slik at mottak blir startet så tidlig som mulig, og alltid *før* sending. Dette er i motsetning til den originale implementasjonen av Diffpack.



Figur 6.12: Ping-pong testen for ulike funksjonskall. Hvert målepunkt er målt 30 ganger.



Figur 6.13: Ping-pong testen for ulike funksjonskall. Hvert målepunkt er målt 100 ganger.

6.4 Elementmetoden

I de videre tester av elementmetoden har jeg brukt Poissonlikningen i to og tre dimensjoner som testproblem. Poissonlikningen er gitt ved

$$\begin{aligned} -\nabla^2 u(x) &= f(x), \quad x \in \Omega \subset R^d, \\ u(x) &= g(x), \quad x \in \partial\Omega, \end{aligned}$$

der $\partial\Omega$ er randen til domenet Ω.

All problem-spesifikk kode er basert på programmet Poisson1 som følger med som eksempelprogram i Diffpack[25].

6.4.1 Strukturert grid

Jeg starter med noen tester på et strukturert grid. Jeg har brukt et kvadratisk domene med bilineære elementer i to dimensjoner, og et kubisk domene med trilineære elementer i tre dimensjoner. Dette tilsvarer antagelsene som ble lagt til grunn for de tidligere gitte estimatene for tid og speedup gitt i kapittel 4 og 5. Problemstørrelsene blir videre oppgitt i globalt antall elementer (E), i motsetning til antall gridpunkter (N) som ble brukt i estimatene. I disse testene vil derimot antall elementer være svært nær antall gridpunkter, slik at det er rimelig å sette $E \approx N$ for å sammenlikne med tidligere estimater.

Metis

Partisjoneringen blir gjort av Metis[22] som kan gi vilkårlige partisjoner. Metis gir generelt ikke kvadratiske og kubiske partisjoner som lå til grunn i tidligere estimater. Tabell 6.2 og 6.3 viser noen data for de partisjonene som ble brukt i disse testene[4]. Ser vi på antall gridpunkter per prosessor (n), ser vi at disse er relativt godt fordelt. Med unntak av de ekstreme tilfellene med svært mange prosessorer på små grid, er variasjonene stort sett innenfor ett prosent. I denne forstand gir Metis altså gode partisjoner.

Antall andre prosessorer en prosessor må kommunisere med i matrise-vektor-produktet (antall naboprosessorer) er gitt ved Q i tabellene. Dette tallet varierer relativt mye for de fleste partisjonene. Dette kan bety at enkelte prosessorer må kommunisere relativt mye i forhold til de andre. Dette kan føre til en flaskehals ved at andre prosessorer må bli stående og vente på at denne prosessoren blir ferdig.

α og β angir andel gridpunkter som, med optimaliseringsforslagene fra forrige kapittel, kan elimineres for henholdsvis indreproduktet og matrise-vektor-produktet. Disse tallene varierer også en del innenfor hver enkelt partisjon. Dette kan igjen gi en flaskehals-effekt, slik at den reelle optimaliseringen vil ligge i den nederste delen av intervallet. α vil i gjennomsnitt være lik halvparten av andelen dupliserte gridpunkter på en prosessor. Sammenlikner vi derfor α med estimatene for andel dupliserte gridpunkter i tabell 4.1 på side 20, ser vi at Metis må gi en betydelig høyere andel dupliserte gridpunkter enn de partisjonene som ble lagt til grunn i estimatene. Metis er i denne forstand altså ikke optimal.

[4]Tallene er nær identiske for testene på diplopodus og embla. Tallene som vises i tabellene er hentet fra testene på embla.

E	P	n	Q	α	β
100^2	1	10201	-	-	-
100^2	4	$2688 \pm 0.6\%$	2-3	4-5%	4-5%
100^2	8	$1394 \pm 1.2\%$	2-5	5-12%	5-12%
100^2	16	$727 \pm 3.0\%$	2-7	7-18%	7-18%
100^2	32	$394 \pm 4.7\%$	2-7	10-27%	10-27%
600^2	1	361201	-	-	-
600^2	4	$91186 \pm 0.1\%$	2-3	0-1%	0-1%
600^2	8	$45870 \pm 0.3\%$	2-5	1-2%	1-2%
600^2	16	$23152 \pm 0.4\%$	2-6	1-3%	1-3%
600^2	32	$11736 \pm 0.9\%$	2-7	2-5%	2-5%
1000^2	1	1002001	-	-	-
1000^2	4	$251936 \pm 0.1\%$	2-3	0%	0%
1000^2	8	$126492 \pm 0.1\%$	2-5	0-1%	0-1%
1000^2	16	$63613 \pm 0.3\%$	2-7	0-2%	0-2%
1000^2	32	$32061 \pm 0.4\%$	2-7	1-3%	1-3%

Tabell 6.2: Data for tester på strukturert grid i 2 dimensjoner. P angir antall prosessorer, E det globale antall elementer før partisjonering, n antall gridpunkter per prosessor, Q antall nabo-prosesser, α andel gridpunkter som kan elimineres i indreproduktet, β andel punkter på den interne randen.

E	P	n	Q	α	β
10^3	1	1331	-	-	-
10^3	4	$495 \pm 0.3\%$	3	31-34%	29-33%
10^3	8	$305 \pm 0.7\%$	7	42-50%	41-50%
10^3	16	$188 \pm 8.8\%$	6-13	47-70%	47-67%
10^3	32	$126 \pm 11.1\%$	9-23	52-81%	51-81%
60^3	1	226981	-	-	-
60^3	4	$61437 \pm 0.0\%$	3	7%	7%
60^3	8	$31994 \pm 0.0\%$	7	11%	10-11%
60^3	16	$17154 \pm 0.1\%$	4-13	13-21%	13-21%
60^3	32	$9132 \pm 1.6\%$	5-18	16-30%	15-29%
100^3	1	1030301	-	-	-
100^3	4	$271135 \pm 0.0\%$	3	4-5%	4-5%
100^3	8	$139276 \pm 0.0\%$	4-7	6-9%	6-9%
100^3	16	$72363 \pm 0.3\%$	5-11	8-14%	8-14%
100^3	32	$37616 \pm 2.1\%$	5-17	10-20%	10-20%

Tabell 6.3: Data for tester på strukturert grid i 3 dimensjoner. P angir antall prosessorer, E det globale antall elementer før partisjonering, n antall gridpunkter per prosessor, Q antall nabo-prosesser, α andel gridpunkter som kan elimineres i indreproduktet, β andel punkter på den interne randen.

Indreprodukt

Tabell 6.4 til 6.7 viser målt tid og speedup for indreproduktet. De presenterte verdiene er her minimumsverdi etter 100 repititisjoner. Sammenlikner vi speedup for den originale implementasjonen av Diffpack med estimatene i tabell 4.3 på side 23, er speedup langt lavere enn forventet for de fleste problemstørrelsene. I estimatene antok jeg kommunikasjonskostnadene til å være neglisjerbare. En reduksjon på diplopodus ble målt til å kunne ta opp mot 0.7 millisekunder (figur 6.5). Dersom vi eksempelvis trekker 0.7 millisekunder fra den målte tiden for $P = 32, E = 1000^2$ på diplopodus, får vi en speedup lik 24.6 som passer rimelig godt med estimatene. Et unntak er for $E = 1000^2$ på embla. Her har vi et eksempel på superlineær speedup som skyldes bedre utnyttelse av lokalt hurtigminne ved å redusere lokalt minneforbruk i forhold til én prosessor.

Legg også merke til at det ikke finnes noen test for $P = 4, E = 1000^2$ på embla. Dette skyldes at dette tilfellet globalt bruker mer enn 4GB minne. Embla har maksimalt 4GB fysisk minne per 4 prosessorer. Denne testen ble derfor stående og flytte brukt minne til og fra harddisk, såkalt "trashing"[37]. Dette tok svært lang tid, og jeg valgte å avbryte disse testene.

Det er en vesentlig forbedret kjøretid med den optimaliserte versjonen for nesten alle problemer. Spesielt god effekt har vi for problemer i 3 dimensjoner, med over 50% forbedring for nesten alle problemer på embla, og mellom 28 og 43% for de fleste problemene på diplopodus. Sammenlikner vi den målte forbedringen med estimatene i tabell 4.3 på side 23, ser vi at estimatene stort sett er av samme størrelsesorden. Unntakene er for enkelte problemer på diplopodus der kommunikasjonskostnaden, og usikkerheten rundt kommunikasjonskostnaden, overskygger lokal beregning. Dette gjelder problemene med $E = 10^3$ og $E = 1000^2$ med 16 og 32 prosessorer.

Jevnt over er gevinsten med den optimaliserte versjonen nær dobbelt så høy på embla som på diplopodus. Dette kan tyde på at arbeidet forbundet med å korrigere for ett dupliserte gridpunkt, gitt som $\frac{d_I}{c_I}$ i avsnitt 4.8, er dobbelt så høyt på embla i forhold til diplopodus.

Matrise-vektor-produkt

Tabell 6.8 til 6.11 viser resultatene av målt tid for ulike versjoner av matrise-vektor-produktet. De presenterte verdiene er her minimumsverdi etter hundre repitisjoner. "Original" versjon viser til den opprinnelige implementasjonen i Diffpack. Denne bruker en vanlig seriell implementasjon av matrise-vektor-produktet lokalt på hver prosessor. Under arbeidet med ny parallell versjon, fant jeg at den serielle versjonen av Diffpacks matrise-vektor-produkt er subotimal. Figur 6.14 skisserer Diffpack sin implementasjon. Sammenliknet med min versjon (figur 2.2 på side 8) har denne en ekstra løkke der den fyller `y` med 0-verdier. Denne ekstra traverseringen av `y` gir svært utnyttelse av lokalt hurtigminne og resulterer i dårlig ytelse.

Versjonen kalt "reimplementert" i tabellene, er Diffpacks vanlige parallelle matrise-vektor-produkt, men der det lokale matrise-vektor-produktet er byttet ut med min nærmere optimale versjon. De parallelle optimaliseringsforslagene vil blir vurdert opp i mot den reimplementerte versjonen.

"Optimalisert" versjon er en implementasjon som eliminerer punkter på

E	P	original		optimalisert		gevinst
		T	S	T'	S'	$\frac{T-T'}{T}$
100^2	1	4.38e-05	-	-	-	-
100^2	4	1.60e-04	0.3	1.52e-04	0.3	4%
100^2	8	2.17e-04	0.2	1.47e-04	0.3	32%
100^2	16	1.34e-04	0.3	1.07e-04	0.4	19%
100^2	32	2.13e-04	0.2	1.60e-04	0.3	24%
600^2	1	7.54e-03	-	-	-	-
600^2	4	2.32e-03	3.2	2.12e-03	3.6	8%
600^2	8	1.41e-03	5.3	1.32e-03	5.7	6%
600^2	16	8.50e-04	8.9	7.07e-04	10.7	16%
600^2	32	7.84e-04	9.6	6.96e-04	10.8	11%
1000^2	1	2.09e-02	-	-	-	-
1000^2	4	5.72e-03	3.7	5.36e-03	3.9	6%
1000^2	8	3.16e-03	6.6	2.82e-03	7.4	10%
1000^2	16	1.89e-03	11.1	1.71e-03	12.3	9%
1000^2	32	1.55e-03	13.5	1.51e-03	13.8	2%

Tabell 6.4: Måling av indreproduktet på et strukturert grid i 2 dimensjoner på diplopodus.

E	P	original		optimalisert		gevinst
		T	S	T'	S'	$\frac{T-T'}{T}$
100^2	1	4.60e-05	-	-	-	-
100^2	4	5.50e-05	0.8	3.80e-05	1.2	30%
100^2	8	5.00e-05	0.9	4.20e-05	1.1	16%
100^2	16	6.40e-05	0.7	3.40e-05	1.4	46%
100^2	32	5.90e-05	0.8	3.50e-05	1.3	40%
600^2	1	2.27e-03	-	-	-	-
600^2	4	6.79e-04	3.3	6.14e-04	3.7	9%
600^2	8	3.83e-04	5.9	2.61e-04	8.7	31%
600^2	16	2.01e-04	11.3	1.36e-04	16.7	32%
600^2	32	1.58e-04	14.3	9.00e-05	25.2	43%
1000^2	1	2.29e-02	-	-	-	-
1000^2	8	9.81e-04	23.3	9.14e-04	25.0	6%
1000^2	16	4.44e-04	51.5	3.30e-04	69.3	25%
1000^2	32	2.79e-04	82.0	1.89e-04	121.0	32%

Tabell 6.5: Måling av indreproduktet på et strukturert grid i 2 dimensjoner på embla.

E	P	original		optimalisert		gevinst
		T	S	T'	S'	$\frac{T-T'}{T}$
10^3	1	6.31e-06	-	-	-	-
10^3	4	6.72e-05	0.1	5.89e-05	0.1	12%
10^3	8	1.00e-04	0.1	1.04e-04	0.1	-3%
10^3	16	1.49e-04	0.0	1.07e-04	0.1	28%
10^3	32	1.93e-04	0.0	1.95e-04	0.0	-1%
60^3	1	4.74e-03	-	-	-	-
60^3	4	2.06e-03	2.3	1.47e-03	3.2	28%
60^3	8	1.39e-03	3.4	8.63e-04	5.5	38%
60^3	16	9.70e-04	4.9	4.01e-04	11.8	58%
60^3	32	8.56e-04	5.5	4.50e-04	10.5	47%
100^3	1	2.15e-02	-	-	-	-
100^3	4	7.60e-03	2.8	5.53e-03	3.9	27%
100^3	8	4.53e-03	4.8	3.01e-03	7.1	33%
100^3	16	3.09e-03	7.0	1.74e-03	12.3	43%
100^3	32	2.51e-03	8.6	1.42e-03	15.1	43%

Tabell 6.6: Måling av indreproduktet på et strukturert grid i 3 dimensjoner på diplopodus.

E	P	original		optimalisert		gevinst
		T	S	T'	S'	$\frac{T-T'}{T}$
10^3	1	5.00e-06	-	-	-	-
10^3	4	3.00e-05	0.2	1.60e-05	0.3	46%
10^3	8	5.20e-05	0.1	2.60e-05	0.2	50%
10^3	16	6.20e-05	0.1	3.20e-05	0.2	48%
10^3	32	5.90e-05	0.1	3.30e-05	0.2	44%
60^3	1	1.45e-03	-	-	-	-
60^3	4	8.05e-04	1.8	3.08e-04	4.7	61%
60^3	8	4.58e-04	3.2	1.54e-04	9.4	66%
60^3	16	4.38e-04	3.3	1.07e-04	13.6	75%
60^3	32	2.86e-04	5.1	6.80e-05	21.3	76%
100^3	1	2.35e-02	-	-	-	-
100^3	4	5.06e-03	4.6	1.85e-03	12.7	63%
100^3	8	2.68e-03	8.8	8.67e-04	27.1	67%
100^3	16	1.55e-03	15.2	3.53e-04	66.6	77%
100^3	32	2.26e-03	10.4	3.62e-04	64.9	83%

Tabell 6.7: Måling av indreproduktet på et strukturert grid i 3 dimensjoner på embla.

```
1  void matvec_crs( CRS *A, double *x, double *y, int n )
2  {
3     for ( int i=0; i < n; i++ ) {
4        y[i] = 0;
5     }
6
7     for ( int i=rowstart; i <= rowend; i++ ) {
8        for ( int k=A->row_ptr[i]; k < A->row_ptr[i+1]; k++ ) {
9           y[i] += A->vals[k] * x[ A->col_ind[k] ];
10       }
11    }
12 }
```

Figur 6.14: Skisse av Diffpacks serielle matrise-vektor-produkt.

den interne randen. "Maskert" versjon eliminerer punktene på den interne randen, og prøver i tillegg å maskere kommunikasjonskostnadene med lokal beregning. Disse optimaliseringene er forklart i avsnitt 5.2.

La oss først se på hva vi kan forvente av gevinst med optimaliseringsforslagene. Ved å hoppe over punktene på den interne randen ("optimalisert" versjon), burde vi få en gevinst tilsvarende andel punkter på den interne randen. Denne andelen ble oppgitt i tabell 6.2 og 6.3 som β. Disse tallene ser ut til å stemme bra med alle testene på embla, og de fleste testene på diplopodus.

Ved å maskere kommunikasjon med lokal beregning skulle man i tillegg kunne fjerne noe av tiden brukt på kommunikasjon. Tiden brukt på prosessor-til-prosessor kommunikasjon ble målt i avsnitt 6.3.2. På diplopodus tok eksempelvis slik kommunikasjon omtrent 0.2 millisekunder for meldingslengder under 100, og omtrent ett millisekund for meldingslengder på 1000. Avhengig av hvor mange meldinger som må sendes, burde vi kunne maskere kommunikasjonstid av samme størrelsesorden. Ser vi på forskjellene mellom "optimalisert" og "maskert" versjon, ser dette ut til å stemme rimelig bra. De tilsvarende tallene for embla ligger på størrelsesorden mikrosekunder for meldingslengder opp til 100, og 10 mikrosekunder for meldingslengder på 1000. For embla ser den målte gevinsten ut til å faktisk ligge i overkant av hva man skulle forvente ut fra dette.

La oss så se litt på unntakene. Testene med 32 på diplopodus skiller seg noe ut. Ved bruk av 32 prosessorer på diplopodus hadde vi det tilfellet der to og to prosessorer deler minne på samme maskin. For små problemstørrelser ($E = 100^2$ i to dimensjoner, $E = 10^3$ i tre dimensjoner) gir bruk av den "optimaliserte" versjonen svært negativ effekt. Et forsøk på å forklare dette kan være at den lokale beregningen er så liten at det gir uheldig bruk av lokalt hurtigminne når matrise-vektor-produktet deles opp i to delvise produkter (se figur 5.7).

Bruk av ikke-blokkerende kommunikasjon for tilfellene med 32 prosessorer på diplopodus, ser derimot ut til å gi en svært positiv effekt. Jeg er ikke helt sikker på den riktige forklaringen på dette, men antar at årsaken ligger i hvordan to prosessorer på samme maskin kommuniserer med delt minne. Kanskje er det slik at prosessorene må kjempe om de samme ressursene dersom de kommuniserer samtidig, mens man unngår dette med ikke-blokkerende kom-

E	P	original		reimplementert		optimalisert		maskert		gevinst
		T	S	T'	S	T	S	T''	S	$\frac{T'-T''}{T'}$
100^2	1	8.10e+00	-	5.52e-03	-	-	-	-	-	-
100^2	4	3.47e+00	2.3	2.00e-03	2.8	2.81e-03	2.0	1.74e-03	3.2	13%
100^2	8	2.83e+00	2.9	2.60e-03	2.1	2.20e-03	2.5	2.24e-03	2.5	13%
100^2	16	7.49e-01	10.8	6.71e-04	8.2	5.88e-04	9.4	4.93e-04	11.2	26%
100^2	32	7.12e-01	11.4	3.98e-04	13.9	6.57e-04	8.4	5.12e-04	10.8	-28%
600^2	1	2.92e+02	-	1.96e-01	-	-	-	-	-	-
600^2	4	7.49e+01	3.9	5.11e-02	3.8	5.05e-02	3.9	5.02e-02	3.9	1%
600^2	8	3.75e+01	7.8	2.60e-02	7.5	2.57e-02	7.6	2.56e-02	7.7	1%
600^2	16	1.89e+01	15.5	1.31e-02	14.9	1.30e-02	15.0	1.29e-02	15.2	1%
600^2	32	1.01e+01	29.0	9.24e-03	21.2	9.04e-03	21.7	7.34e-03	26.7	20%
1000^2	1	7.74e+02	-	5.14e-01	-	-	-	-	-	-
1000^2	4	2.14e+02	3.6	1.39e-01	3.7	1.38e-01	3.7	1.38e-01	3.7	0%
1000^2	8	1.04e+02	7.5	7.03e-02	7.3	6.97e-02	7.4	6.98e-02	7.4	0%
1000^2	16	5.19e+01	14.9	3.52e-02	14.6	3.50e-02	14.7	3.47e-02	14.8	1%
1000^2	32	2.65e+01	29.2	2.32e-02	22.1	2.29e-02	22.4	1.79e-02	28.7	22%

Tabell 6.8: Måling av matrise-vektor-produktet på et strukturert grid i 2 dimensjoner på diplopodus.

E	P	original		reimplementert		optimalisert		maskert		gevinst
		T	S	T'	S	T	S	T''	S	$\frac{T'-T''}{T'}$
100^2	1	2.98e-03	-	1.62e-03	-	-	-	-	-	-
100^2	4	8.54e-04	3.5	5.06e-04	3.2	4.84e-04	3.4	4.93e-04	3.3	2%
100^2	8	5.42e-04	5.5	3.58e-04	4.5	3.40e-04	4.8	3.40e-04	4.8	5%
100^2	16	3.48e-04	8.6	2.18e-04	7.5	1.92e-04	8.5	1.91e-04	8.5	12%
100^2	32	2.06e-04	14.5	1.36e-04	12.0	1.17e-04	13.9	1.10e-04	14.8	19%
600^2	1	1.71e-01	-	1.22e-01	-	-	-	-	-	-
600^2	4	4.17e-02	4.1	3.00e-02	4.1	2.97e-02	4.1	2.74e-02	4.4	8%
600^2	8	1.77e-02	9.7	1.15e-02	10.6	1.06e-02	11.4	1.01e-02	12.0	11%
600^2	16	7.29e-03	23.5	4.25e-03	28.6	4.20e-03	29.0	4.09e-03	29.7	3%
600^2	32	3.68e-03	46.4	2.12e-03	57.5	2.05e-03	59.3	2.03e-03	59.9	3%
1000^2	1	4.72e-01	-	3.40e-01	-	-	-	-	-	-
1000^2	8	8.91e-02	5.3	4.90e-02	6.9	4.86e-02	7.0	4.81e-02	7.1	1%
1000^2	16	2.43e-02	19.4	1.47e-02	23.2	1.43e-02	23.7	1.41e-02	24.1	3%
1000^2	32	1.05e-02	44.9	5.64e-03	60.3	5.49e-03	61.9	5.47e-03	62.2	2%

Tabell 6.9: Måling av matrise-vektor-produktet på et strukturert grid i 2 dimensjoner på embla.

E	P	original		reimplementert		optimalisert		maskert		gevinst
		T	S	T'	S	T	S	T''	S	$\frac{T'-T''}{T'}$
10^3	1	2.39e+00	-	1.67e-03	-	-	-	-	-	-
10^3	4	7.48e-01	3.2	6.22e-04	2.7	4.53e-04	3.7	4.03e-04	4.1	35%
10^3	8	7.29e-01	3.3	6.54e-04	2.6	5.82e-04	2.9	4.56e-04	3.7	30%
10^3	16	7.46e-01	3.2	6.91e-04	2.4	6.33e-04	2.6	6.37e-04	2.6	7%
10^3	32	5.40e-01	4.4	5.11e-04	3.3	6.72e-04	2.5	8.22e-04	2.0	-60%
60^3	1	4.64e+02	-	3.05e-01	-	-	-	-	-	-
60^3	4	1.32e+02	3.5	8.92e-02	3.4	8.62e-02	3.5	8.24e-02	3.7	7%
60^3	8	6.78e+01	6.8	4.57e-02	6.7	4.23e-02	7.2	4.23e-02	7.2	7%
60^3	16	3.74e+01	12.4	2.54e-02	12.0	2.25e-02	13.5	2.24e-02	13.6	11%
60^3	32	2.37e+01	19.6	1.98e-02	15.4	1.68e-02	18.2	1.58e-02	19.3	20%
100^3	1	2.10e+03	-	1.38e+00	-	-	-	-	-	-
100^3	4	5.66e+02	3.7	3.81e-01	3.6	3.66e-01	3.8	3.68e-01	3.7	3%
100^3	8	3.00e+02	7.0	2.01e-01	6.9	1.93e-01	7.1	1.92e-01	7.2	4%
100^3	16	1.60e+02	13.1	1.09e-01	12.6	9.92e-02	13.9	1.01e-01	13.6	7%
100^3	32	1.04e+02	20.3	7.85e-02	17.6	6.67e-02	20.7	6.53e-02	21.1	16%

Tabell 6.10: Måling av matrise-vektor-produktet på et strukturert grid i 3 dimensjoner på diplopodus.

E	P	original		reimplementert		optimalisert		maskert		gevinst
		T	S	T'	S	T	S	T''	S	$\frac{T'-T''}{T'}$
10^3	1	8.96e-04	-	4.81e-04	-	-	-	-	-	-
10^3	4	3.94e-04	2.3	2.45e-04	2.0	2.05e-04	2.3	2.06e-04	2.3	15%
10^3	8	3.43e-04	2.6	2.50e-04	1.9	2.08e-04	2.3	2.10e-04	2.3	15%
10^3	16	3.13e-04	2.9	2.51e-04	1.9	1.84e-04	2.6	1.86e-04	2.6	25%
10^3	32	2.73e-04	3.3	2.38e-04	2.0	1.98e-04	2.4	1.98e-04	2.4	16%
60^3	1	2.82e-01	-	2.03e-01	-	-	-	-	-	-
60^3	4	1.71e-01	1.6	1.34e-01	1.5	1.27e-01	1.6	1.04e-01	2.0	22%
60^3	8	3.57e-02	7.9	2.36e-02	8.6	1.97e-02	10.3	1.95e-02	10.4	17%
60^3	16	1.80e-02	15.7	1.20e-02	16.9	1.14e-02	17.9	1.13e-02	18.0	5%
60^3	32	7.85e-03	36.0	4.42e-03	46.0	3.79e-03	53.7	3.73e-03	54.5	15%
100^3	1	1.30e+00	-	9.58e-01	-	-	-	-	-	-
100^3	4	6.90e-01	1.9	6.38e-01	1.5	4.93e-01	1.9	4.08e-01	2.3	36%
100^3	8	2.03e-01	6.4	1.57e-01	6.1	1.50e-01	6.4	1.29e-01	7.4	17%
100^3	16	1.02e-01	12.7	8.27e-02	11.6	7.62e-02	12.6	6.38e-02	15.0	22%
100^3	32	4.93e-02	26.3	3.41e-02	28.1	3.02e-02	31.7	2.72e-02	35.2	20%

Tabell 6.11: Måling av matrise-vektor-produktet på et strukturert grid i 3 dimensjoner på embla.

E	P	Partisjonering
100^2	4	0.16
100^2	8	0.27
100^2	16	0.52
100^2	32	1.34
600^2	4	7.37
600^2	8	11.56
600^2	16	21.66
600^2	32	49.26
1000^2	4	19.92
1000^2	8	31.67
1000^2	16	57.63
1000^2	32	140.18

E	P	Partisjonering
10^3	4	0.03
10^3	8	0.04
10^3	16	0.05
10^3	32	0.14
60^3	4	9.90
60^3	8	12.82
60^3	16	19.20
60^3	32	36.66
100^3	4	47.55
100^3	8	61.28
100^3	16	90.83
100^3	32	181.60

Tabell 6.12: Tid i sekunder brukt på partisjonering av strukturert grid ved hjelp av Metis i original implementasjon av parallell Diffpack.

munikasjon.

Partisjonering

Tabell 6.12 viser tiden brukt på partisjonering med Metis i Diffpack[5]. Vi ser her at tiden vokser proporsjonalt med antall prosessorer i bruk. Tiden blir svært betydelig når vi sammenlikner med tiden brukt på for eksempel et matrise-vektor-produkt.

Tabell 6.13 og 6.14 viser tiden brukt på partisjonering langs én dimensjon som foreslått i avsnitt 5.4. En slik partisjonering måtte oppfylle visse krav (se avsnitt 5.4) som at et gridpunkt ikke måtte kommuniseres mellom mer enn ett par av prosessorer. Dette stiller krav til at det globalet gridet må ha nok antall elementer i x-retning i forhold til antall prosessorer, for å kunne fungere. Antall ulike tester er derfor redusert til de tilfellene der dette er oppfylt.

Vi ser her at tiden er vesentlig mindre enn den brukt med Metis i Diffpack. Tiden er likevel fortsatt betydelig sammenliknet med de andre operasjonene, som matrise-vektor-produktet. Tabell 6.13 viser også tiden bruk på indreproduktet og matrise-vektor-produktet på denne typen partisjoner. Disse er implementert med optimaliseringsforslagene angitt i avsnitt 5.4. Disse innebar eliminering av buffring for matrise-vektor-produktet i tillegg til de tidligere foreslåtte optimaliseringene.

Sammenlikner vi tiden brukt på dette indreproduktet med det optimaliserte indreproduktet i tabell 6.4 til 6.7, ser vi tidene er rimelig like. For en del problemstørrelser er dette indreproduktet faktisk noe raskere, og for en del problemstørrelser er det noe tregere. Dette er sannsynligvis bestemt av hvilken lastbalansering som er best.

La oss så sammenlikne tiden brukt på matrise-vektor-produktet med den maskerte versjonen i tabell 6.8 til 6.11. Ser vi på problemstørrelsene i to dimensjoner, er dette matrise-vektor-produktet raskere enn den maskerte versjonen

[5]Tidene er rimelig like for embla og diplopodus. Her vises tidene for diplopodus

E	P	Partisjonering	Indreprodukt	Matrise-vektor
1000^2	4	6.32	5.33e-03	1.27e-01
1000^2	8	5.67	2.91e-03	6.85e-02
1000^2	16	5.28	1.58e-03	3.44e-02
1000^2	32	5.97	1.43e-03	2.24e-02
100^3	4	12.25	5.63e-03	3.51e-01
100^3	8	11.20	3.20e-03	1.97e-01
100^3	16	10.64	1.95e-03	1.15e-01

Tabell 6.13: Målinger gjort for partisjonering langs én dimensjon på strukturert grid på diplodpodus.

E	P	Partisjonering	Indreprodukt	Matrise-vektor
1000^2	4	5.70	1.64e-03	9.43e-02
1000^2	8	5.02	6.97e-04	4.46e-02
1000^2	16	4.74	3.27e-04	1.31e-02
1000^2	32	4.55	2.70e-04	6.12e-03
100^3	4	13.26	1.93e-03	3.64e-01
100^3	8	11.42	9.19e-04	1.23e-01
100^3	16	11.02	4.92e-04	6.48e-02

Tabell 6.14: Målinger gjort for partisjonering langs én dimensjon på strukturert grid på embla.

på partisjoner laget av Metis. Denne gevinsten skyldes sannsynligvis at vi har eliminert buffringen under kommunikasjon. Ved bruk av mange prosessorer ($P = 32$ i to dimensjoner), får vi derimot noe dårligere tider. Dette skyldes sannsynligvis at antall gridpunkter som må kommuniseres blir svært høyt med denne partisjoneringen. Det blir således vanskeligere å maskere kommunikasjon med lokal beregning. Dette problemet er forsterket i tre dimensjoner.

6.4.2 Ustrukturert grid

Testene på strukturert grid var tatt med for å best mulig kunne sammenlikne med estimatene gjort i tidligere kapitler. I relle simuleringer er det derimot ofte bruk av elementmetoden på ustrukturert grid som er mest interessant.

Tabell 6.15 og 6.16 viser derfor noen tester av optimaliseringsforslagene på ustrukturerte grid. Versjonen merket "Org." er her den originale implementasjonen i Diffpack. Versjonen merket "Opt." er de implementerte optimaliseringsforslagene på partisjoner gitt av Metis. For matrise-vektor-produktet er dette versjonen som både eliminerer punktene på den interne randen, og prøver å maskere kommunikasjonskostnadene med lokal beregning. Versjonen merket "1D" er implementasjonen som baserer seg på partisjonering langs én dimensjon.

Testene i to dimensjoner er gjort på et 120 graders sirkelsegment med bikvadratiske elementer[6]. Testene i tre dimensjoner er gjort på et sfære-segment

[6]Gridgenerering gjort i Diffpack med "P=PreproStdGeom | DISK_WITH_HOLE a=0.5 b=0.8

E	P	Indreprodukt			Matrise-vektor-produkt			Partisjonering		
		Orig.	Opt.	1D	Orig.	Opt.	1D	Orig.	Opt.	1D
1000^2	8	3.30e-03	2.94e-03	2.94e-03	1.71e-01	1.14e-01	1.07e-01	16.64	18.01	1.94
1000^2	16	2.08e-03	1.73e-03	1.76e-03	8.60e-02	5.77e-02	5.40e-02	24.43	26.72	1.71
1000^2	32	1.78e-03	1.53e-03	1.36e-03	5.36e-02	3.83e-02	3.62e-02	46.92	51.54	1.88
100^3	4	7.62e-03	5.53e-03	5.69e-03	5.76e-01	3.69e-01	6.84e-01	52.18	53.78	14.67
100^3	8	4.62e-03	3.01e-03	3.15e-03	3.00e-01	1.96e-01	2.00e-01	67.24	69.61	13.56
100^3	16	2.98e-03	1.80e-03	1.95e-03	1.63e-01	1.01e-01	1.13e-01	96.93	100.77	12.94

Tabell 6.15: Tester på diplopodus av alle de gjennomgåtte optimaliseringsforslagene på ustrukturert grid.

E	P	Indreprodukt			Matrise-vektor-produkt			Partisjonering		
		Orig.	Opt.	1D	Orig.	Opt.	1D	Orig.	Opt.	1D
1000^2	8	1.96e-03	1.22e-03	6.97e-04	2.54e-01	2.27e-01	4.47e-02	19.30	20.01	5.03
1000^2	16	7.92e-04	3.49e-04	3.27e-04	9.20e-02	3.61e-02	1.31e-02	24.31	25.95	4.75
1000^2	32	3.19e-04	2.02e-04	2.70e-04	1.88e-02	9.92e-03	6.13e-03	25.79	28.83	4.55
100^3	4	5.06e-03	1.85e-03	1.93e-03	6.90e-01	4.08e-01	3.65e-01	59.37	43.42	13.27
100^3	8	2.68e-03	8.67e-04	9.19e-04	2.02e-01	1.29e-01	1.24e-01	46.20	48.97	11.42
100^3	16	1.54e-03	3.53e-04	4.92e-04	1.02e-01	6.38e-02	6.48e-02	57.23	60.21	11.03

Tabell 6.16: Tester på embla av alle de gjennomgåtte optimaliseringsforslagene på ustrukturert grid.

med bilineære elementer[7]. Tiden er her som før oppgitt i sekunder.

Resultatene er stort sett som for testene gjort på strukturert grid. Optimaliseringsforslagene på partisjoneringer gjort med Metis, gir vesentlig gevinst i forhold til den originale implementasjonen i Diffpack. Det å partisjonere langs én dimensjon tar vesentlig kortere tid enn partisjonering med bruk av Metis. De fleste testene viser også at både indreproduktet og matrise-vektor-produktet kan holdes omtrent like effektivt ved partisjonering langs én dimensjon, som optimaliseringene gjort på partisjoneringer fra Metis.

degrees=120 | d=2 e=ElmB9n2D [1000,1000] [1,1]"

[7] Gridgenerering gjort i Diffpack med "P=PreproStdGeom | SPHERE_WITH_HOLE a=2.5 b=6 theta=45 phi=45 | d=3 e=ElmB8n3D [100,100,100] [1,1,1]"

Kapittel 7

Konklusjon

7.1 Resultater

Jeg har i denne oppgaven sett på den Diffpack sin lineæralgebra tilnærming til parallellisering av elementmetoden. Det beregningsmessige arbeidet bestod her av blant annet av partisjonering, indreprodukt og matrise-vektor-produkt. Denne oppgaven har foreslått metoder som kan optimalisere disse funksjonene.

Indreproduktet ble optimalisert ved å fjerne duplisert beregning av dupliserte gridpunkter. Dette ble gjort ved å fordele ansvaret for disse punktene på ulike prosessorer. Tester viste at tiden brukt på indreproduktet gikk betraktelig ned med denne implementasjonen.

Matrise-vektor-produktet ble optimalisert på to måter. Den første gikk på å eliminere unødvendig beregning for punkter på den interne randen i partisjonene. I de fleste av testene på grid i to dimensjoner, reduserte dette kjøretiden med et par prosent. På tester i tre dimensjoner reduserte dette kjøretiden med nær 20 prosent for store grid og bruk av mange prosessorer. Den andre optimaliseringen bestod i å maskere kommunikasjonskostnadene med lokal beregning. Tester viste at disse kostnadene til en stor del faktisk lot seg maskere.

Partisjoneringen ble forenklet til å partisjonere kun langs én dimensjon. Dette ble betydelig raskere enn partisjonering ved hjelp av Metis gjort i Diffpack, og åpnet i tillegg for å eliminere buffring i kommunikasjonen forbundet med matrise-vektor-produktet. Denne måten å partisjonere på kunne derimot kun brukes på partisjoner av en viss størrelse, og kunne gi uheldig høyt kommunikasjonsbehov.

Jeg har også sett på de ulike funksjonskallene som er tilgjengelig i MPI for prosessor-til-prosessor kommunikasjon. Tester viste her at det kan være en fordel å starte mottak av meldinger så tidlig som mulig for å fremprovosere en såkalt ivrig protokoll.

7.2 Videre arbeid

7.2.1 Partisjonering

I mitt forslag til partisjonering langs én dimensjon, må det for hvert globale element beregnes et massesenter i x-retning. Dette er det mulig å parallellisere slik at arbeidet vil reduseres fra å ta tid $\mathcal{O}(N)$ til $\mathcal{O}(\frac{N}{P})$. Etterpå må elementene sorteres. Det eksisterer en rekke parallelle algoritmer for sortering, og dette burde kunne få den tiden ned fra tid $\mathcal{O}(N \log N)$ til tid $\mathcal{O}(\frac{N}{P} \log N)$.

Mitt forslag til partisjonering langs én dimensjon, åpner for flere optimaliseringer enn de jeg har rukket å implementere. En av disse muligheten i MPI til å utnytte topologi. MPI har funksjonskall som åpner for at prosessorene kan stokke om på prosessornumre for at prosessor-til-prosessor kommunikasjon skal bli så effektivt som mulig. En partisjonering langs én dimensjon er spesielt godt egnet til dette. De aller fleste parallelle topologier kan effektivt simulere en topologi langs en linje.

Implementasjonen av MPI på de plattformene jeg har testet, gjør ikke bruk av optimaliseringer som for eksempel persistent kommunikasjon[34]. Andre implementasjoner, som for eksempel HP sin implementasjon[17], gjør derimot dette. Dette ville være relativt enkelt å implementere dersom man partisjonerer langs én dimensjon.

7.2.2 Prekondisjonering

Den parallelle implementasjonen av Diffpack består av flere deler enn de jeg har forsøkt å optimalisere i denne oppgaven. Den viktigste av disse, er sannsynligvis prekondisjonering.

Gjeldende implementasjon av Diffpack bruker en vanlig seriell prekondisjonerer lokalt på hvert subdomene i parallelle simuleringer. Dette har blant annet den ulempen at prekondisjoneringen blir dårligere med økende antall prosessorer. Jeg vil tro det kunne være gunstig å prøve å implementere en parallell prekondisjonerer istedet. Et forslag kunne være å bruke en likningsløser på et grovt globalt grid som prekondisjonerer. Slike prekondisjonerere er blant annet ofte brukt i såkalte domene-dekomposisjons metoder[35].

Tillegg A

Maskering av kommunikasjonskostnader

I beregningen av potentiale for maskering av kommunikasjonskostnader, ønsker man å bestemme en maksimal tid for lokal beregning, t_c, slik at målt tid for en ping-pong-test med lokal beregning t'_{pp} ikke overstiger tiden for en ping-pong-test uten lokal beregning t_{pp}. Som diskutert i avsnitt 6.3.2 antas $t'_{pp} = t_{pp} + t_c - t_m$ der t_m er tiden der lokal beregning foregår samtidig med kommunikasjon. En slik maksimalverdi vil gi $t_c = t_m$.

Binærsøket som brukes til denne beregningen er listet i figur A.1 på neste side. Measurement::measure(x,y) er her en metode som utfører en ping-pong-test med meldingslengde x. Mellom sending og mottak blir det utført en beregning med størrelse y.

En test "feiler" dersom den totale tiden for en ping-pong test overstiger threshold (tilsvarer t_{pp}). Da tiden for en test vil kunne variere, blant annet avhengig av eksterne forhold (se diskusjon i kapittel 6), vil en test gjentas max_fail ganger selv om den skulle feile. Dersom en test feiler fler enn max_feil ganger antas den lokale beregningsmengden å være for stor og blir derfor redusert i neste test. Omvendt vi beregningsmengden øke i neste test dersom en test var vellykket.

Potentiale for maskering beregnes kun på én prosessor, det er derfor kun denne som avgjør når man skal terminere. Metoden overlap_done synkroniserer prosessorene etter hver test. Dens siste argument indikerer hvorvidt beregningen er ferdig eller ikke. Dersom dette argumentet er 0, vil alle de involverte prosessorene gjøre ytterligere en ping-pong-test.

Metoden vil returnere det estimerte potentiale for samtidig lokal beregning og kommunikasjon, angitt i arbeidsmengde. Arbeidsmengden må så oversettes til tid. For å redusere kjøretiden, er dette estimert på følgende måte:

Anta tiden $t_c(l)$ for en arbeidsmengde l. Bestem initielt tiden for en liten arbeidsmengde ϵ, $t_c(\epsilon)$. Jeg kan da estimere $\frac{\partial t_c(l)}{\partial l} \approx \frac{t_c(\epsilon)}{\epsilon}$ dersom t_c antas å være en lineær funksjon.

I ettertid har jeg angret meg for dette valget å bestemme beregningstiden på. Jeg burde istedet ha bestemt tiden $t_c(l)$ med egne målinger for en aktuell l. Dette ville ha redusert usikkerheten i tidsbestemmelsen.

```
1  int MPIExplore::MeasureSuite :: estimate_overlap(Measurement &sm,
2                                    int x, double threshold,
3                                    int low, int high, int guess)
4  {
5        const int max_fail = this->repititions;
6        bool success = false;
7        int failed = 0;
8
9        while (!success && failed < max_fail) {
10             overlap_done(m, x, 0);
11             if (m.measure(x, guess) <= threshold) success = true;
12             else failed++;
13       }
14
15       if (success) {
16             if (guess >= high) {
17                   overlap_done(m, x, 1);
18                   return guess;
19             }
20             low   = guess;
21             guess = (guess + high)/2 + 1;
22       } else {
23             high  = guess;
24             guess = (guess + low)/2;
25             if (guess <= low) {
26                   overlap_done(m, x, 1);
27                   return guess;
28             }
29       }
30
31       return estimate_overlap(m, x, threshold, low, high, guess);
32 }
```

Figur A.1: Binærsøk for å estimere potentiale for overlapp

Tillegg B

Høyoppløslig tidtakning

```
1  class MPIExplore::Timer {
2  private:
3          unsigned long long starttick;
4          unsigned long long endtick;
5          double error;
6
7  public:
8          Timer()
9          {
10                 set_error();
11         }
12
13         inline void start()
14         {
15                 starttick = tick();
16         }
17
18         inline void stop()
19         {
20                 endtick = tick();
21         }
22
23         // Returns number of clock cycles between start() and stop()
24         // in million cycles.
25         double result() const
26         {
27                 unsigned long long result;
28
29                 // ticks will overflow
30                 if (endtick < starttick) {
31                         result = (unsigned long long)(-1);
32                         result -= starttick;
33                         result += endtick;
34                 } else {
35                         result = endtick - starttick;
36                 }
37                 return (result/1.0e6 - error);
38         }
```

```
39
40          inline double get_error() const
41          {
42                  return error;
43          }
44
45  private:
46          // Returns number of clock ticks since last reboot.
47          // This will only work on Intel Pentium or later processors.
48          inline unsigned long long int tick()
49          {
50                  unsigned long long int t;
51                  __asm__ volatile (".byte 0x0f, 0x31" : "=A" (t));
52                  return t;
53          }
54
55          // There is an overhead involved calling start and stop
56          // The time consumed of this function calling theese functions
57          // is an upper bound of the time measurement. A lower bound
58          // would be to subtract the value of a call to start immediately
59          // followed by a call to stop, minus 2 cycles to the actual clock.
60          // The error will be set to be half the upper-lower bound.
61          void set_error()
62          {
63                  error = 999999;
64
65                  // Find best value ouf of ten
66                  for(int i=0; i<10; i++) {
67                          start();
68                          stop();
69                          double tmp = (endtick-starttick-2)/2e6;
70
71                          if (tmp < error && tmp > 0) {
72                                  error = tmp;
73                          }
74                  }
75
76                  if (error < 0) { // try again
77                          set_error();
78                  }
79          }
80  };
```

Bibliografi

[1] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, og Henk van der Vorst, redaktører. *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM, 2000.

[2] R. Barret, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine og H. Van der Vorst. *Templates for the Solution of Linear systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1994.

[3] Gouri K. Bhattacharyya og Richard A. Johnson. *Statistical Concepts and Methods*. John Wiley & Sons, 1977.

[4] Are Magnus Bruaset, Xing Cai, Hans Petter Langtangen og Aslak Tveito. Numerical solution of pdes on parallel computers utilizing sequential simulators. I *Scientific Computing in Object-Oriented Parallel Environment*, bind 1343 av *Lecture Notes in Computer Science*, side 161–168. Springer-Verlag, 1997.

[5] Xing Cai. Two object-oriented approaches to the parallelization of Diffpack. I *HiPer'99*, 1999.

[6] X. Cai, E. Acklam, H. P. Langtangen og A. Tveito. Parallel computing. I H. P. Langtangen og A. Tveito, redaktører, *Advanced Topics in Computational Partial Differential Equations*. Springer, 2003.

[7] Ohio Supercomputing Center og University of Notre Dame. Lam (Local Area Multicomputer). http://www.lam-mpi.org.

[8] The PARKBENCH committee. Parkbench (parallel kernels and benchmarks). http://www.netlib.org/parkbench/.

[9] Jack Dongarra, Jim Bunch, Cleve Moler og Pete Stewart. Linpack. http://www.netlib.org/linpack/.

[10] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):159–416, 1994.

[11] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[12] John L. Fustafson og Q. O. Snell. HINT: A new way to measure computer performance. Rapport, Supercomputing 1994, 1994.

[13] William Gropp, Ewing Lusk, Nathan Doss og Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[14] William Gropp og Ewing Lusk. MPICH working note: The second-generation ADI for MPICH implementation of MPI, 1996. ftp://info.mcs.anl.gov/pub/mpi/workingnote/nextgen.ps.

[15] William Gropp og Ewing Lusk. Tuning MPI programs for peak performance. http://www-unix.mcs.anl.gov/mpi/tutorial/perf/, 1996.

[16] William Gropp og Ewing Lusk. MPPTEST - measuring MPI performance. http://www-unix.mcs.anl.gov/mpi/mpptest/.

[17] Hewlett Packard. *HP MPI Users Guide*, 5. utgave, 2000.

[18] Roger Hockney og Michael Berry. Public international benchmarks for parallel computers. Rapport, PARKBENCH Committee, 1994.

[19] Haoqiang Jin, Michael Frumkin og Jerry Yan. NAS parallel benchmarks. http://www.nas.nasa.gov/NAS/NPB/, 1997.

[20] George Karypis og Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. Rapport, University of Minnesota, Departement of Computer Science, 1995.

[21] George Karypis og Vipin Kumar. A coarse-grain parallel formulation of multilevel k-way graph-partitioning algorithm. I *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[22] George Karypis og Vipin Kumar. Multilevel k-way hypergraph partitioning. Rapport, University of Minnesota, Departement of Computer Science, 1998.

[23] Vipin Kumar, Ananth Grama, Anshul Gupta og George Karypis. *Introduction to Parallel Computing, Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[24] Argonne National Laboratory. MPICH – a portable MPI implementation. http://www-unix.mcs.anl.gov/mpi/mpich/.

[25] Hans Petter Langtangen. *Computational Partial Diffeential Equations – Numerical Methods and Diffpack Programming*. Lecture Notes in Computational Science and Engineering. Springer, 1999.

[26] Steven J. Leon. *Linear Algebra with Applications*. Prentice Hall, 6. utgave, 2002.

[27] Message Passing Interface Forum. MPI2: A message passing interface standard. *High Performance Computing Applications*, 12(1-2):1–299, 1998.

[28] Philip J. Mucci. Llcbench. http://icl.cs.utk.edu/projects/llcbench/.

[29] Pallas. PMB - Pallas MPI Benchmarks. http://www.pallas.com/pages/pmb.htm.

[30] Ralf H. Reussner. SKaLib: SKaMPI as a library. Rapport, Departement of Informatics, University of Karlsruhe, Germany, juli 1999.

[31] Ralf H. Reussner. SKaMPI: The Special Karlsruher MPI-Benchmark. Rapport, University of Karlsruhe, Germany, 2002.

[32] Ralf H. Reussner. Special Karlsruher MPI-benchmark. http://liinwww.ira.uka.de/~skampi/.

[33] Silicon Graphics. *SGI Origin 3000 Series Technical Configuration Owner's Guide*, 2. utgave, 2001. http://techpubs.sgi.com, document number 007-4311-002.

[34] Silicon Graphics. *Message Passing Toolkit: MPI Programmer's Manual*, 8. utgave, 2003. http://techpubs.sgi.com, document number 007-3687-008.

[35] Barry Smith, Petter Bjørnstad og William Gropp. *Domain Decomposition – Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[36] G. L. Squires. *Practical Physics*. Cambridge University Press, 3. utgave, 1998.

[37] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 2. utgave, 1997.

Register