TEKST NR 282 1994

Parallelle algoritmer

Erwin Dan Nielsen
Jan Danielsen
Niels Bo Johansen

TEKSTER fra
IMFUFA ROSKILDE UNIVERSITETSCENTER
INSTITUT FOR STUDIET AF MATEMATIK OG FYSIK SAMT DERES
FUNKTIONER I UNDERVISNING, FORSKNING OG ANVENDELSER

IMFUFA, Roskilde Universitetscenter, Postbox 260, DK-4000 Roskilde

Parallelle algoritmer

Af: Erwin Dan Nielsen, Jan Danielsen og Niels Bo Johansen

IMFUFA tekst nr. 282/1994 90 sider ISSN 0106-6242

Abstract

Med udgangspunkt i traditionelle sekventielle algoritmer, som kan finde en grafs transitive afslutning, har vi konstrueret parallelle algoritmer. Disse algoritmer er velegnet til at blive implementeret på paralleldatamaten CM-200 (eller lignende paralleldatamater).

Vi har lagt vægt på at analysere algoritmernes kompleksitet, samt at opstille kriterier for, hvilke af algoritmerne der kan anvendes i en given situation.

Forord

Denne projektrapport er blevet til på 3. modul på overbygningsuddannelsen i datalogi ved Roskilde Universitetscenter.

Formålet ved projektarbejdet er at konstruere, implementere og vurdere parallelle algoritmer. Udgangspunktet for de parallelle algoritmer er traditionelle sekventielle algoritmer — ud fra disse sekventielle algoritmer udledes parallelle algoritmer.

Algoritmerne er blevet implementeret og afprøvet på paralleldatamaten CM-200. Vi er taknemlige over at kunne låne denne af UNI•C. Vi vil i denne forbindelse takke folkene på UNI•C, som har været os til stor hjælp. Specielt vil vi takke Malcolm Brown, Per Christian Hansen og Bjarne Stig Andersen.

Ligeledes vil vi takke Jesper Larsen (IMFUFA, Roskilde Universitetscenter) for at have kommenteret denne projektrapport.

Indhold

Indledning

"Mange bække små gør en stor å." Dette gamle ordsprog er tankegangen bag parallelprogrammering — hvorfor løse ét problem ad gangen, hvis flere kan løses på én gang.

Vi skal i denne projektrapport beskæftige os med paralleldatamater og parallelprogrammering. Men — lad os indlede med en advarsel! Læseren må ikke forvente, at alle svar vedrørende paralleldatamater og parallelprogrammering kan findes i denne projektrapport. I traditionel sekventiel programmering er udgangspunktet en overordnet algoritme, som i princippet kan implementeres på vilkårlige sekventielle datamater — i parallelprogrammering eksisterer denne frihed ikke. Algoritmerne skal konstrueres til en bestemt type paralleldatamat. Læseren vil derfor opleve, at vi i denne projektrapport er meget specifikke, og tilsyneladende knap så generelle. Vore algoritmer og programmer er tilpasset paralleldatamaten CM-200. Denne er placeret på UNI•C i Lyngby og skulle på nuværende tidspunkt være den hurtigste datamat i Danmark. Der er således tale om en "rigtig" paralleldatamat — ikke blot en samling sekventielle datamater, som til lejligheden er blevet sammenkoblet! Problemerne, som søges løst på CM-200, skal være formuleret i matrixlignende strukturer. Vi har derfor valgt at implementere et traditionelt grafproblem, som kan løses ved hjælp af manipulationer på matricer: At finde en grafs transitive afslutning.

Problemformulering og mål

Med udgangspunkt i traditionelle sekventielle algoritmer, som kan finde en grafs transitive afslutning, vil vi konstruere parallelle algoritmer. Disse algoritmer skal være velegnet til at blive implementeret på paralleldatamaten CM-200 (eller lignende paralleldatamater).

Vi vil lægge vægt på at analysere algoritmernes kompleksitet, samt at opstille kriterier for, hvilke af algoritmerne der skal anvendes i en given situation.

Vi vil implementere de parallelle algoritmer i programmeringssproget C^* og verificere algoritmernes kompleksitet empirisk.

Projektrapportens indhold

Projektrapporten indeholder følgende:

I kapitel 1 giver vi et overblik over paralleldatamater, samt en dybere præsentation paralleldatamaten CM-200. Specielt vil vi i dette kapitel finde frem til, hvorledes vi skal formulere vore problemer, således, at vi kan konstruere effektive algoritmer til CM-200.

I kapitel 2 opstiller vi en række algoritmer, som kan finde en grafs transitive afslutning. Ud fra traditionelle sekventielle algoritmer konstrueres parallelle algoritmer. For alle algoritmerne opstiller vi deres kompleksitet. Endvidere viser vi de konstruerede algoritmers egenskaber — specielt giver vi retningslinier for valget af algoritmer, når grafens størrelse n kendes.

I kapitel 3 verificerer vi algoritmernes kompleksitet, som blev fundet i kapitel 2.

I appendiks A beskriver vi den notation, som anvendes i denne fremstilling. Afsnit A.1 omhandler den grafteoretiske notation og definitionen på "en grafs transitive afslutning". Afsnit A.2 omhandler vor "pseudokode", som anvendes i forbindelse med algoritmerne.

I appendiks B berører vi kort nogle småproblemer, som ikke direkte indgår i vor problemformulering.

I appendiks C viser vi vore programmer i C^*.

I forbindelse med hvert kapitel (og i et enkelt appendiks) har vi en række henvisninger til og vurdering af andet litteratur.

Kapitel 1

Paralleldatamater

Behovet for regnekraft er altid større end den til en given tid største datamats kapacitet, således vil der altid være en efterspørgsel efter hurtigere datamater. For at efterkomme disse krav om hurtigere datamater, er der igennem de sidste år blevet satset stadig mere på udviklingen af paralleldatamater.

Vi vil fremover, når vi taler om paralleldatamater, anvende Aki's definition:

> *A parallel computer is one that consists of a collection of processing units, or processors, that cooperate to solve a problem by working simultaneously on different parts of that problem (Aki (1989) side xii).*

Formålet med dette kapitel er at finde frem til, hvorledes problemer skal formuleres for at opnå effektive algoritmer til de forskellige typer paralleldatamater.

Vi vil i afsnit 1.1 og afsnit 1.2 give et kort overblik over forskellige typer paralleldatamater. I afsnit 1.3 vil vi diskutere nogle kriterier for valg af algoritmer til de forskellige typer datamater.

I afsnit 1.4 vil vi beskrive CM-200 datamaten på UNI•C, hvorefter vi i afsnit 1.5 vil prøve at opstille nogle kriterier for udvælgelse af algoritmer til løsning af det problem, som vi har opstillet i indledningen.

1.1 Flynn's taksionomi

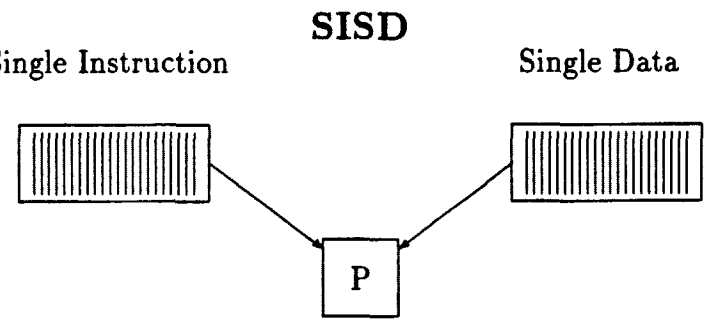
Flynn opstillede i 1966 en klassificering af datamater[1]. Taksionomien er opbygget med udgangspunkt i sammenhængen mellem datamatens behandling af instruktioner og data[2]. Figur 1.1 viser Flynn's taksionomi.

	Single Data	Multi Data
Single Instruction	SISD	SIMD
Multi Instruction	MISD	MIMD

Figur 1.1: Flynn's taksionomi.

Lidt mere detaljeret kan vi beskrive Flynn's taksionomi således:

- **SISD** datamater udfører en instruktionsstrøm på en datastrøm.



Figur 1.2: Single Instruction Single Data.

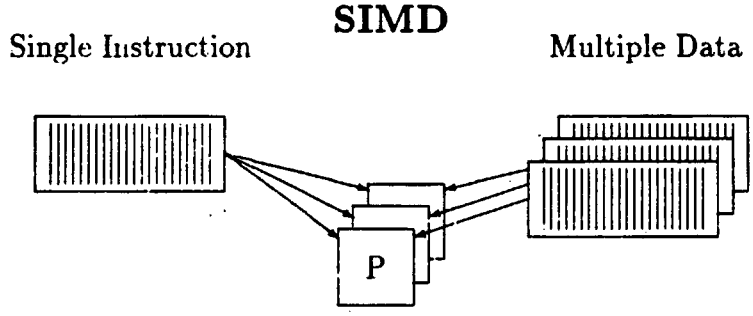
SISD datamater er således de almindelige sekventielle datamater som pc'er, arbejdsstationer etc. Det er vigtigt at forstå, at der

[1]Flynn's taksionomi opstilles i Hansen (1993), hvor han refererer til "M. Flynn, Very high speed computing systems, Proc. IEEE 54 (1966), 1901-1909". Det har ikke været os muligt at fremskaffe denne artikel, men også Tanenbaum (1990) beskriver taksionomien.

[2]Parallelitet i CPU'en tælles normalt ikke med.

til tiden T eksekveres én instruktion på ét dataelement[3] på en enkelt processor. Betingede sætninger kan nedsætte antallet af instruktioner på SISD datamater.

- **SIMD** datamater eksekverer én instruktions strøm, men udfører den på flere strømme af dataelementer samtidigt. De kan siges at arbejde synkront.



Figur 1.3: Single Instruction Multiple Data.

Af SIMD datamater kan nævnes CM-200 og MasPar MP-1. Denne type datamater eksekverer i princippet instruktions strømmen på samme måde som de sekventielle datamater, men her foregår instruktions strømmen på p processorer samtidigt, altså udføres instruktionen på p strømme af dataelementer.

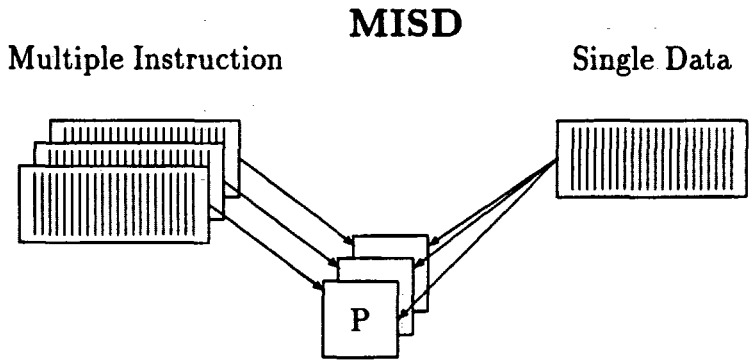
Dataelementerne skal være af samme type. På SIMD datamater kan det i visse tilfælde godt svare sig at undlade betingede sætninger. SIMD datamater er egnet til regulære problemer.

- **MISD** datamater udfører samtidigt flere instruktioner på et dataelement.

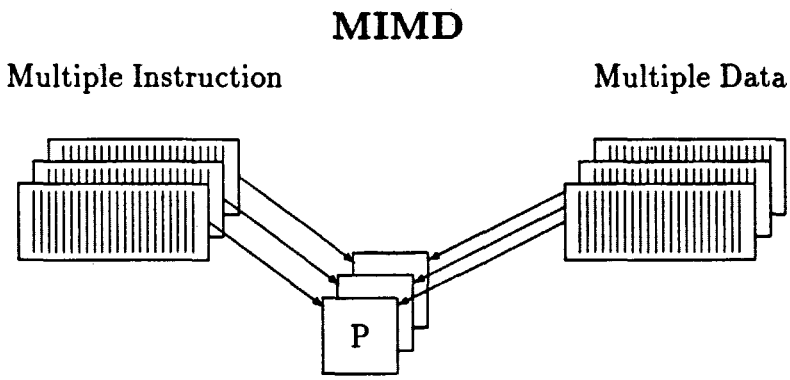
  Vi kender ingen eksempler på datamater af denne type, men de kunne eventuelt anvendes som kontrol-datamater i forbindelse med rumfart eller atomkraft. Kontrollen kunne foregå, ved at flere forskellige datamater regnede på samme data, hvis der så var overensstemmelse mellem resultaterne, kunne beregningerne siges at være "rigtige".

- **MIMD** datamater udfører samtidigt flere instruktioner på flere forskellige dataelementer. De kan siges at arbejde asynkront.

[3]Der anvendes i mange tilfælde mere end ét dataelement, eksempelvis additioner må nødvendigvis have mere end et dataelement for at kunne udføres. Det væsentlige er dog at dataelementerne læses sekventielt: I én strøm.



Figur 1.4: Multiple Instruction Single Data.



Figur 1.5: Multiple Instruction Multiple Data.

Eksempler på sådanne datamater er CRAY Y-MP og Allient FX. Der er blevet introduceret et nyt system — PVM (Parallel Virtual Machine), hvilket er en software pakke, der gør det muligt at sammenkoble flere arbejdsstationer, så de arbejder som en MIMD datamat.

MIMD datamater kan i princippet simulere de andre typer af datamater, men der distribueres oftest en større blok af instruktioner og en blok data til en processor. Det er således væsentligt, at programmøren tager højde for, at problemet skal opsplittes og bearbejdes på flere forskellige processorer, og at disse alle skal arbejde samtidigt og ikke vente på hinanden.

Hver enkelt processor kan betragtes som en SISD datamat, og tankegangen bag udviklingen af de enkelte dele af programmet svarer til tankegangen bag SISD datamater, men distribution af data og instruktioner skal varetages ud fra ønsket om, at alle processorer skal arbejde samtidigt.

Som det ses af Flynn's taksionomi findes der to grundlæggende forskellige typer paralleldatamater, nemlig SIMD og MIMD datamater.

SIMD datamater har ofte mange simple processorer, således skal data have samme struktur og det skal give mening at udføre samme instruktion på alle data. SIMD datamater beskrives ofte som "dataparallelle" datamater, eller datamater der er velegnet til finkornet parallelitet. SIMD datamater kan siges at være parallelle på instruktionsniveau, og de egner sig derfor bedst til regulære problemer.

MIMD datamater har ofte færre kraftige processorer. Derfor skal opgaverne opsplittes i større dele og distribueres til processorerne. Denne type af datamater er velegnet til grovkornet parallelitet. MIMD datamater kan siges at være parallelle på proces niveau og egner sig til et større spekter af problemer, når bare problemet kan splittes op i delprocesser.

Et konkret problem kan egne sig til både MIMD og SIMD datamater alt efter, hvordan det er muligt at opsplitte problemet. Nogle problemer kan løses lige effektivt på begge datamattyper, mens andre opgaver kun egner sig til en type datamat.

1.2 Paralleldatamater klassificeret på lager

Sekventielle datamater arbejder med en instruktion på et dataelement fra et lager, og da ingen andre end denne ene processor arbejder, så opstår der ingen problemer omkring lageranvendelse. På paralleldatamater er netop lagerstrukturen af afgørende betydning for, hvilke typer algoritmer der egner sig til at blive implementeret på datamaten.

Båndbredden på paralleldatamater er forholdsvis større end båndbredden på sekventielle datamater, da hver processor har mulighed for at kommunikere med lageret.

Hvis vi antager, at processorerne uafhængigt af hinanden kan læse fra og skrive til lageret og den overførte mængde data pr. tidsenhed mellem lageret og en enkelt processor betegnes med B, så vil båndbredden på paralleldatamaten være $p \times B$.

Processorerne kan enten dele et fælles lager eller de kan have deres eget lokale lager.

1.2.1 Fælles lager

Paralleldatamater med fælles lager kan i princippet læse fra eller skrive til en vilkårlig celle i lageret. Da vi arbejder med paralleldatamater kan der opstå situationer, hvor flere processorer samtidigt ønsker at arbejde i den samme lagercelle. Denne konflikt kan løses på fire måder[4], nemlig:

	Exclusive Read	Concurrent Read
Exclusive Write	EREW	CREW
Concurrent Write	ERCW	CRCW

Figur 1.6: Fire kombinationer af lageradministration på parallelle datamater.

- **EREW** en processor råder over datacellen både når den skriver, og når den læser.
- **ERCW** en processor kan læse datacellen, mens flere kan skrive i den.
- **CREW** flere processorer kan læse datacellen, men kun én har lov til at skrive.
- **CRCW** flere processorer kan læse datacellen, og flere kan skrive til den.

De fire forskellige måder at administrere fælles lager på giver hver især sine problemer.

At flere processorer kan læse i den samme datacelle samtidigt, er i mange tilfælde hensigtsmæssigt, og det skaber ingen teoretiske problemer. Hvis kun en enkelt processor kan læse datacellen vil der opstå

[4] Aki (1989) opstiller denne specificering af problemet på side 14 i sin bog.

flaskehalse omkring distribution af det samme dataelement til flere processorer.

Skal flere processorer derimod skrive til en celle kan der opstå store problemer. Hvem skal skrive hvis data ikke er ens? Der er flere strategier alt afhængigt af hvordan datamaten skal virke:

— Den processor, der returnerer den største værdi, kan skrive i cellen.
— Den processor, der returnerer den mindste værdi, kan skrive i cellen.
— En sum af alle processorernes værdier kan skrives i cellen.
— Den processor med det laveste nummer...
— En tilfældig processor...
— etc.

Som det ses er der næsten ubegrænsede muligheder, men det er nødvendigt at vælge en strategi. Afhængigt af hvilken strategi der vælges, så vil datamaten være velegnet til nogle opgaver, mens andre ikke løses særligt effektivt.

Datamater med fælles lager egner sig til opgaver hvor de enkelte processorer skal udveksle data, men som det ses er der en del komplikationer ved brug af fælles lager.

1.2.2 Lokalt lager

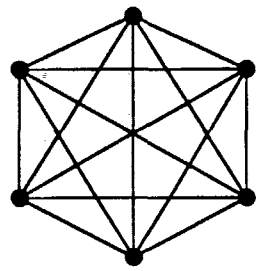
En anden mulighed er at lade hver processor råder over sit eget lager. På den måde opstår der ikke spørgsmål om, hvem der skriver hvad i hvilke dataceller, da det kun er en processor, der kan skrive i cellen. Processorerne skal dog stadig kunne udveksle data. Det gøres ved at sætte processorerne sammen i netværk.

1.2.2.1 Stjernenetværk

Det vil være optimalt at sætte processorerne sammen, som vist på figur 1.7, således, at der er en forbindelse fra en processor til alle andre. Det er dog kun muligt med meget små netværk, fordi der for større netværk opstår store tekniske problemer, da antallet af forbindelser $f(p)$ stiger efter formlen:

$$f(p) = \frac{p^2 - p}{2}. \tag{1.1}$$

Af formel 1.1 ses det at 8 processorer skal bruges 28 forbindelser, mens 128 processorer kræver 8128 forbindelser.



Figur 1.7: Et stjernenetværk kræver mange forbindelser og er derfor kun egnet til små netværk.

Det er altså nødvendigt for større netværk at forbinde processorerne på en anden måde. Der findes mange andre måder at forbinde netværk på; en af de mere hensigtsmæssige er af forbinde processorerne som en såkaldt hyperkube.

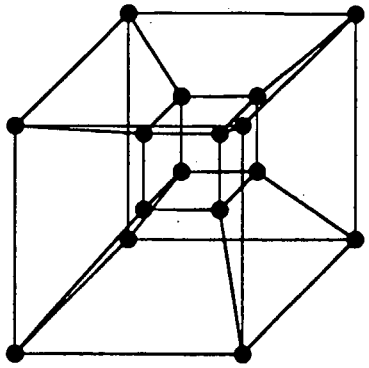
1.2.2.2 Hyperkube

Som det ses af figur 1.8 anvender en hyperkube langt færre forbindelser end et stjernenetværk. En hyperkube af dimension $d = \log_2(p)$ med d forbindelser fra hver processor har p processorer og antallet af forbindelser $f(p)$ er givet ved $f(p) = \frac{p \log_2(p)}{2}$, og stien fra en processor til en anden vil højest bestå af d kanter.

En hyperkube af dimension 3 har 8 processorer og 12 forbindelser, mens en hyperkube af dimension 7 har 128 processorer og 448 forbindelser.

Denne konstruktion gør altså at kommunikationstiden mellem processorerne forøges med højest en faktor d i forhold til et stjernenetværk, men antallet af forbindelser falder markant.

Der er mange andre måder at forbinde processorer i et netværk. Det er dog givet, at uanset hvordan processorerne forbindes, kan der opstå flaskehalse i forbindelse med kommunikationen. Hyperkube datamater



Figur 1.8: En hyperkube med 16 processorer har 32 forbindelser og den længste sti bliver en afstand på 4 kanter.

arbejder altså bedst med problemer hvor kommunikationen mellem de enkelte processorer er minimal eller meget ensartet. Eksempelvis koster kommunikation mellem processorer, der er direkte forbundet, ikke mere i en hyperkube end i et stjernenetværk.

Det ses, at der er mange måder at administrere lageret på paralleldatamater. Hvordan det gøres er afhængigt af hvilken datamat vi ønsker at have, og det er svært at sige hvilke metoder der er bedst, da alle metoder har deres fordele og ulemper; valget må altså falde på hvad datamaten skal være god til.

1.3 Hvad kommer først: Paralleldatamat eller algoritme

Valget af algoritmer er direkte afhængigt af den datamat der ønskes anvendt. Problemet skal være af homogen karakter for at egne sig til SIMD datamater, hvorimod MIMD datamater kan løse flere slags opgaver, blot problemet kan fordeles nogenlunde ligeligt på alle processorer.

Når der vælges en paralleldatamat, er det nødvendigt at undersøge hvilke datastrukturer det er muligt at opstille for problemet — det skal undersøges hvilken type kommunikation problemet kræver, om det kan opstilles som matricer og så videre.

Det er for de fleste problemer muligt at opstille en algoritme, der kan implementeres på en given paralleldatamat. Derfor vælges ofte først, hvilken paralleldatamat der skal bruges, og derefter undersøges så hvilke algoritmer der egner sig til denne type paralleldatamat.

Da udbredelsen af paralleldatamater er meget lille sker det i praksis ofte, at der kun er adgang til en bestemt paralleldatamat, så problemet bliver oftest at finde algoritmer der egner sig til at blive implementeret på denne datamat.

1.4 Paralleldatamaten CM-200

CM-200 var den eneste paralleldatamat vi kunne få adgang til, så vi har valgt og udviklet algoritmer der egner sig til at blive implementeret på denne. Vi fandt det væsentligt, at algoritmerne blev testet på en "rigtig" paralleldatamat fremfor på en simulator eller PVM, da simulatorer mangler ægte samtidighed, og derfor ikke afspejler virkeligheden, og på PVM vil datakommunikation tage uforholdsmæssig lang tid, da den foregår på det eksisterende lokalnet.

CM-200 er en såkaldt "dataparallel datamat", det vil sige at det er en SIMD datamat med lokalt lager.

CM-200 indeholder følgende dele:

- En eller flere front-end datamater.
- Et I/O system.
- Et grafisk display system.
- En CM enhed.

1.4.1 Front-end'en

En CM-200 kan have op til 4 front-end'er. Front-end'en er en sekventiel datamat[5], hvorpå programmer administreres. Front-end'en har til formål at stille et udviklingsmiljø og en kommunikations forbindelse til CM enheden til rådighed for brugeren. Instruktioner, der kan udføres sekventielt, vil blive udført på front-end'en, mens instruktioner, der kan udføres parallelt, vil blive videresendt og udført på CM-200. Kommunikationen til CM-200 foregår via et kort, et såkaldt FEBI (Front End Bus Interface); en front-end kan have op til fire FEBI kort.

Fra front-end'en bliver data via FEBI sendt til nexusen, der distribuerer data/instruktioner med den rigtige clockfrekvens til sequenserne. En sequensers opgaver er at fortolke kommandoer og distribuere dem til processorerne indenfor en processorblok (se afsnit 1.4.4).

1.4.2 I/O systemet

For hver blok af processorer findes to output kanaler, en CM I/O Controller (CMIOC) og en framebuffer (FB). CMIOC'en har forbindelse til datavaulten, der er CM-200 datamatens parallelt adresserbare fastdisklager, via CM I/O bussen. Overførselshastigheden er ca. 50 Mbyte/s.

Datavaulten er opbygget af 42 disks på 256 Mbyte. Heraf kan de 32 kan adresseres parallelt, således at der gemmes en bit af et word på den samme adresse på hver sin disk. 7 disks anvendes til fejlkorrektion, og de sidste tre anvendes som sikkerhed ved nedbrud eller udskiftning af disks.

1.4.3 Det grafiske display system

Der findes et system bestående af framebufferen og en højopløsningsskærm, der gør det muligt at visualisere data med op til 40 Mbyte/s, således, at det er muligt at gengive databehandlingen, lige så hurtigt som den bliver beregnet. Systemet anvendes til at visualisere data, men også til debugging.

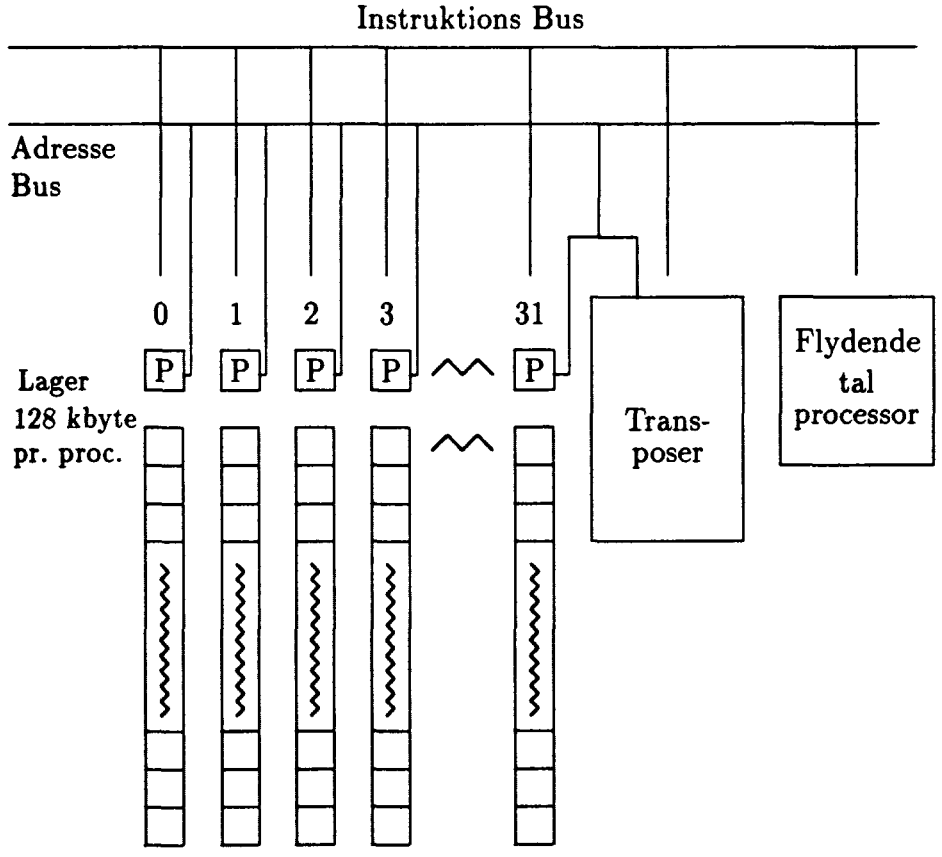
[5]På en CM-200 kan en front-end være en af flere VAX mikrodatamater eller en SUN-4 arbejdsstation.

1.4.4 CM enheden

Processorenheden En processorenhed består af 32 1-bit processorer med hver 128 kbyte lager, 1 transposer og 1 flydende tal processor. CM-200 på UNI•C har 256 processorenheder, og dermed 256 flydende tal processorer, 8192 1-bit processorer og 1 Gbyte lager.

Hvis processorenheden skal behandle 1-bit data, så behandler 1-bit processorerne disse data hver for sig, således, at der for hver instruktion udføres 32 beregninger parallelt for hver processorenhed.

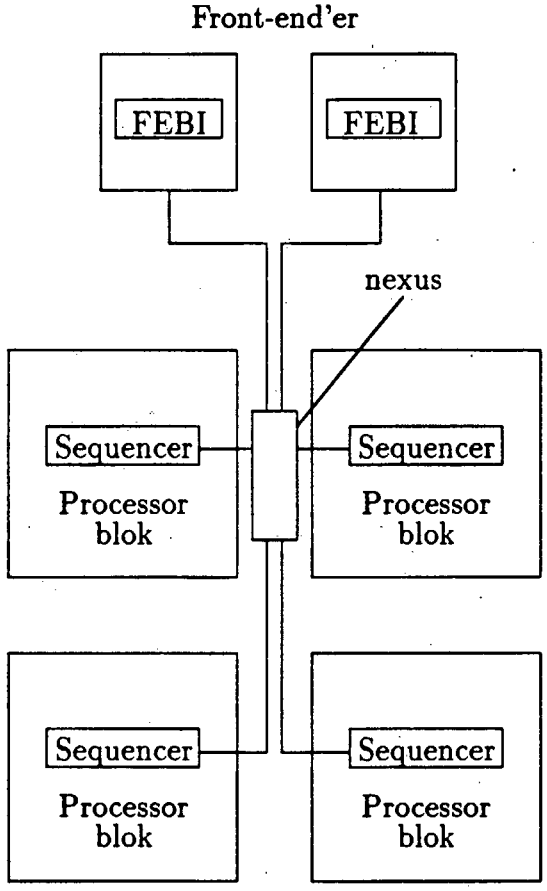
Skal processorenheden derimod arbejde med flydende tal gemmes data således at den n'te 1-bit processors lager indeholder den n'te bit i et 32 bit word, og transposeren omformer data således at flydende tal processoren kan modtage dem som word, se figur 1.9.



Figur 1.9: Processorerne på en processorenhed er fordelt således, at der for hver 32 1-bit processorer er en transposer og en flydende tal processor.

Processorblok Et fuldt udbygget CM-200 system vil bestå af 4 processorblokke af hver 16384[6] 1-bit processorer, således at der i det fuldt udbyggede system vil være 65536 1-bit processorer, 2048 flydende tal processorer og i alt 8 Gbyte lager.

I figure 1.10 er skitseret et CM system med 4 processorblokke. Denne konstruktion medfører dog ikke øget tidsforbrug ved kommunikation mellem processorerne, da processorblokkene er forbundet som en hyperkube. De enkelte processorblokke er også forbundet som hyperkuber, således at datamaten kan kører som 4 "uafhængige" datamater[7].



Figur 1.10: En CM-200 med 4 processorblokke har en nexus og 4 sequensere.

[6]CM-200 datamater med henholdsvis 8192, 16384, og 32768 processorer har 8192 processorer pr. processorblok, mens datamater med 65535 processorer har 16384 processorer pr. processorblok.

[7]De deler stadig Datavaulten, IO-systemet etc.

CM-200 på UNI•C har 8192 fysiske processorer. Mange problemer kræver flere end 8192 processorer. Imidlertid kan hver fysisk processor udføre rollen for mange (virtuelle) processorer.

For eksempel hvis problemet kræver 16384 processorer, så skal hver fysisk processor udføre arbejdet for to processorer. Dette sker selvfølgelig i to omgange. Hvis nu problemet kræver P processorer, hvor P ikke er et multiplum af 8192, så skal hver processor arbejde for $\lceil P/8192 \rceil$ processorer ($\lceil P/8192 \rceil$ er det mindste heltal, som er større end eller lig med $P/8192$). I dette tilfælde bliver nogle af processorerne nødt til at udføre "dummy-operationer", da CM-200 er SIMD datamat, hvor samme instruktion skal udføres på alle processorer samtidigt. For eksempel hvis problemet kræver 10000 processorer, så bliver problemet løst i $\lceil 10000/8192 \rceil = 2$ omgange. I første omgang er der ingen processorer som skal udføre "dummy-operationer", mens $2 \times 8192 - 10000 = 6384$ processorer skal udføre "dummy-operationer" i anden omgang.

1.4.5 Hvordan programmøren af CM-200 skal tænke

Vi vil give et kort eksempel på, hvordan programmøren skal tænke, når der skal udvikles programmer til paralleldatamaten CM-200. Eksemplet er matrixaddition (læseren kan eventuelt supplere læsningen med afsnit A.2, hvor vi har et lignende eksempel).

Vi vil addere to $n \times n$-matricer A og B. Resultatet af additionen er matricen C. Vi skal nu opfatte problemet på følgende måde: Matricerne A og B indeholder hver n^2 elementer. Derfor skal der udføres n^2 skalaradditioner. Disse skalaradditioner kan udføres uafhængigt af hinanden. Ved at anvende n^2 processorer, kan vi udføre n^2 skalaradditioner parallelt.

Vi skal nu antage, at der netop er n^2 processorer til rådighed på CM-200 (hvis n^2 er større end antallet af fysiske processorer, så anvendes n^2 virtuelle processorer). De n^2 processorer kan nummereres entydigt ved taltuplen (i, j), hvor $i, j \in \{1, 2, \ldots, n\}$ — processor nummer (i, j) vil vi kalde $p(i, j)$.

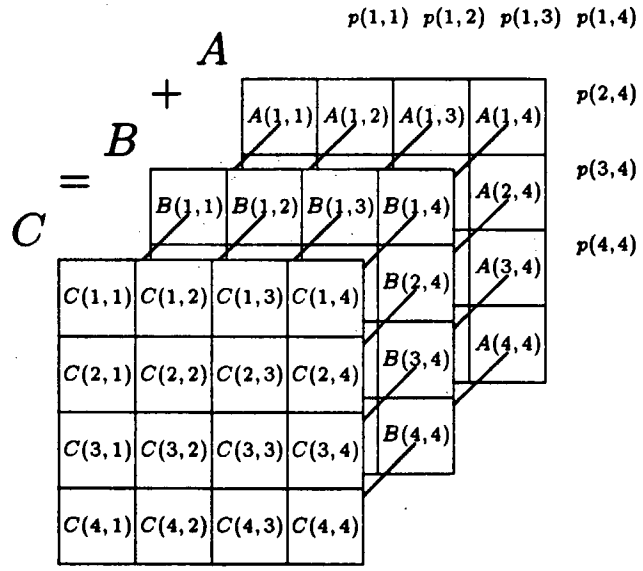
Processorerne i CM-200 har hver sit lokale lager. Hver af de n^2 processor skal have netop ét element fra hver af matricerne A og B i sit

lokale lager, således, at $p(i,j)$ kontrollerer elementerne a_{ij} og b_{ij}. Efter additionen skal $p(i,j)$ endvidere kontrollere elementet c_{ij}.

Matrixadditionen foregår således:

1. I $p(i,j)$ er oprettet tre variable i det lokale lager: $A(i,j)$, $B(i,j)$ og $C(i,j)$. Elementet a_{ij} og elementet b_{ij} opbevares henholdsvis i $A(i,j)$ og $B(i,j)$.
2. $p(i,j)$ adderer de to variable $A(i,j)$ og $B(i,j)$, og tildeler resultatet til variablen $C(i,j)$.

Vi ser, at i første punkt er alle elementerne i matricerne A og B fordelt på de n^2 processorer, og i det andet punkt bliver alle de n^2 skalaradditioner udført. Efter additionen besidder $p(i,j)$ elementet c_{ij} fra matricen C i variablen $C(i,j)$. Vi bemærker, at de n^2 processorer udfører præcis de samme operationer i den samme rækkefølge, men vi har $3n^2$ dataelementer. Dette er karakteristisk for SIMD datamater.



Figur 1.11: Eksempel på hvordan hver processor arbejder på hver sin del af problemet, når CM-200 adderer to 4 x 4 matricer.

CM-200 er som skabt til "brute-force" matrixaddition, det vil sige addition af tykke matricer. Dette er også karakteristisk for SIMD datamater, som jo egner sig til meget regulære problemer. Det kan ikke

betale sig at lave "smarte" algoritmer, som udnytter, at de pågældende matricer er tynde. Problemer, som søges løst på CM-200, skal derfor være formuleret som manipulationer på tykke matricer eller lignende strukturer. Hvis problemet ikke egner sig til at blive formuleret i disse strukturer, så er CM-200 ikke den rette paralleldatamat at anvende på dette problem.

I figur 1.11 er en matrixaddition af to 4×4-matricer skitseret.

1.5 Sammenfatning

CM-200 er en SIMD datamat med lokalt lager. I afsnit 1.1 og 1.2 så vi, at denne type datamat er bedst egnet til store regulære regneopgaver, hvor data er af samme struktur.

Vi har også set, at irregulær datakommunikation skal begrænses, da det er den type operationer der kan skabe flaskehalse. I afsnit 1.2 så vi derimod, at CM-200 er velegnet til ensartet kommunikation mellem processorerne.

Da vi fik adgang til at udvikle algoritmer på CM-200 på UNI•C, og da vi ud fra gennemgangen i dette kapitel må forvente, at de matrixbaserede algoritmer er de mest effektive algoritmer på CM-200, har vi derfor i gennemgangen af algoritmer i kapitel 2 koncentreret os om matrixbaserede algoritmer.

Det skal bemærkes at CM-200 kører med tidsdeling, det vil sige at flere processer kan kører samtidig. Hvis der kører mange processer stiger den tid CM-200 anvender til at administrere tidsdelingen. Hvis der ønskes meget hurtig eksekvering af de implementerede algoritmer, skal de altså afvikles uden tidsdeling, men når man som vi blot skal teste de implementerede algoritmers effektivitet, har tidsdelingen ingen betydning, se afsnit B.3.

REFERENCER

I afsnit 1.1 og 1.2 har vi brugt det upublicerede materiale fra et parallelprogrammerings kursus på UNI•C af Hansen (1993), herfra har vi også beskrivelsen af Flynn's taksionomi. Aki (1989) har i afsnit 1.2 af sin bog en udmærket gennemgang af forskellige typer paralleldatamater og lageranvendelsen på disse. McBryan introducerer i kapitel 2 i Carey's (1989) bog kort de forskellige arkitekturer, og hvilke datamater der findes eller er under konstruktion. Specielt vil vi henlede opmærksomheden på afsnit 2.3.3 i Carey's (1989) bog, hvor McBryan kort beskriver en CM-1 — en forgænger for CM-200. Hvis læseren ønsker en kort oversigt over paralleldatamater (og andre superdatamater), og ikke har nogen forhåndskendskab til området, vil vi henvise til Tanenbaum (1990) afsnit 8.2.1 side 488 - 530.

Afsnit 1.4 bygger først og fremmest på telefonsamtaler med Malcolm Brown, der er application engineer på CM-200 på UNI•C og udsendt af Thinking Machines Corporation. Brown har holdt et oplæg på parallel programmeringskurset på UNI•C, hvor han gennemgik CM-200. Materialet til denne gennemgang indgår også som en væsentlig del af denne fremstilling. Dernæst CM-200 usersguide, udgivet af Thinking Machines Corporation (1991a) og sidst en artikel af McBryan (1988/89), der er lidt mere uddybende end hans beskrivelse i Carey's bog.

Det skal bemærkes at CM-200 usersguide er en generel beskrivelse af CM-2 og CM-200 datamater; derfor er de fysiske karakteristika ikke i overensstemmelse med de faktiske forhold på CM-200 datamaten på UNI•C, da denne ikke er fuldt udbygget.

Hvis læsere af denne projektrapport ønsker at følge med i udviklingen indenfor paralleldatamater i Danmark, kan vi anbefale publikationen "Algoritmer, Beregninger og Computere" fra UNI•C. Publikationen er gratis, og kan rekvireres ved henvendelse til UNI•C. Også tidsskriftet KOMM UNI•C ATION fra UNI•C kan anbefales.

Kapitel 2

Algoritmer

Udgangspunktet for dette kapitel har været tre traditionelle sekventielle algoritmer, som kan finde en grafs transitive afslutning. Disse algoritmer har vi forsøgt at parallelisere. Imidlertid har vi måtte forkaste en af disse sekventielle algoritmer — den egner sig ikke til at blive formuleret som operationer på matrix-lignende strukturer. Den kan derfor ikke implementeres effektivt på CM-200.[1]

De to sekventielle algoritmer, som vi har kunnet parallelisere, bliver beskrevet detaljeret i dette kapitel.

Afsnit 2.1 omhandler matrixmetoden og afsnit 2.2 omhandler Floyds algoritme. I begge afsnit gennemgås den sekventielle algoritme, samt parallelle udgaver. For matrixmetoden har vi fire forskellige parallelle udgaver, mens vi har en for Floyds algoritme. De parallelle udgaver er effektive til tykke grafer (se afsnit 1.4.5). Hvis den pågældende graf er tynd, så skal der anvendes en anden type paralleldatamat end CM-200, samt en anden type algoritmer.

[1]Det er tale om Dijkstras algoritme — se Sedgewick (1988) side 457. Denne algoritme finder den transitive afslutning for en graf med n knuder ved at opdele problemet i n delproblemer, som kan løse uafhængigt af hinanden — algoritmen kunne udmærket paralleliseres til en MIMD paralleldatamat med n processorer. Dijkstras algoritme egner sig til tynde grafer.

For både de sekventielle algoritmer og for de parallelle algoritmer opstilles kompleksiteten. Kompleksiteten er dog ikke hele sandheden om de parallelle algoritmer — i afsnit 2.3 fortsætter vi analysen. I dette afsnit opstilles retningslinier for valget af algoritmer, når grafens størrelse n og antallet processorer p på paralleldatamaten er kendt.

Angående notation så bør læseren læse appendiks A før dette kapitel. I afsnit A.1 gennemgås den nødvendige grafteori, og i afsnit A.2 gennemgås den pseudokode, som vore algoritmer bliver angivet i.

I kompleksitetsberegningerne for de parallelle algoritmer ser vi kun på den parallelle kode — vi ser ikke på den sekventielle kode. Der vil altid være sekventiel kode i parallelle algoritmer. Den sekventielle kode kontrollerer antallet af iterationer i den parallelle kode. Imidlertid er kompleksiteten af den sekventielle kode i vore algoritmer mindre end eller lig med kompleksiteten af den parallelle kode — derfor kan læseren ikke finde nærmere beskrivelser af den sekventielle kode i vor fremstilling.

Det skal endvidere nævnes, at vi i kompleksitetsberegningerne for de forskellige algoritmer ikke medregner initialiseringer. Vi ser kun på de operationer, som indgår i beregningen af den transitive afslutning.

2.1 Matrixmetoden

I afsnit 2.1.1 beskrives en sekventiel algoritme for udregning af den transitive afslutning. Det er tale om den såkaldte matrixmetode.

I afsnit 2.1.2 paralleliseres den sekventielle udgave af matrixmetoden. Vi benytter *fire* forskellige parallelle algoritmer.

Vi vil indledningsvis gøre opmærksom på, at i denne fremstilling skal betegnelsen "matrixmultiplikation" opfattes anderledes end normalt — dette bliver præciseret i afsnit 2.1.1.

2.1.1 Den sekventielle matrixmetode

I afsnit A.1 er det omtalt, at relationen R er transitiv. Der gælder trivielt, at hvis de to kanter (v_i, v_k) og (v_k, v_j) tilhører E, så gælder, at $(v_i, v_j) \in R$.

I dette simple tilfælde er styrken af den stærkeste sti af typen v_i, v_j eller v_i, v_k, v_j fra v_i til v_j givet ved:

$$\tilde{s}_{ij} = \max_{k=1,2,\ldots,n} s_{ik}s_{kj}. \tag{2.1}$$

Vi ser således på alle stier fra v_i til v_j, hvor højest tre knuder indgår. Udregnes $\tilde{s}_{ij} = \max_{k=1,2,\ldots,n} s_{ik}s_{kj}$ for alle knuder v_i og v_j, så fås $n \times n$-matricen $\tilde{S}$. Denne matrix har følgende egenskaber: Elementet $\tilde{s}_{ij}$ er lig 0, hvis der ikke eksisterer en sti på højest tre knuder fra v_i til v_j. Ellers er $\tilde{s}_{ij}$ styrken af den stærkeste sti fra v_i til v_j, idet højest tre knuder indgår.

Formel 2.1 ligner den traditionelle formel for matrixmultiplikation. Forskellen er blot, at maksimumoperatoren anvendes fremfor addition. Matricen $\tilde{S}$ vil vi tillade os skrive som S^2, og vi vil anvende navnet "matrixmultiplikation" om den pågældende multiplikation.

Vi kan nu fortsætte med at betragte stier bestående af højest 4 knuder, 5 knuder, 6 knuder — og så videre. Vi vil generalisere på følgende måde: Det ij'te element i matricen S^m er styrken af den stærkeste sti fra v_i til v_j, der består af højest $m+1$ knuder. Da der er n knuder, så kan vi finde den stærkeste sti fra v_i til v_j ved at betragte stier, hvor alle n knuder indgår. S^{n-1} er med andre ord den transitive afslutning, det vil sige $S^{n-1} = S^*$. Matricen S^{n-1} er invariant overfor yderligere potensopløftninger:

$$S^{n-1} = S^n = S^{n+1} = \ldots = \lim_{n\to\infty} S^n.$$

I praksis forekommer det ofte, at $S^m = S^{m+1}$, hvor $m+1 \leq n-1$. I dette tilfælde gælder nødvendigvis, at $S^m = S^*$. Vi behøver derfor ikke altid at udføre $n-1$ matrixmultiplikationer for at udregne den transitive afslutning ved matrixmetoden. Endvidere kan det udnyttes[2], at $S^{2^m} = \underbrace{(\ldots(S^2)^2\ldots)^2}_{m \text{ gange}}$, således, at højest $m = \lceil \log_2(n-1) \rceil$ (m er det mindste heltal $\geq \log_2(n-1)$) matrixmultiplikationer behøves.

[2]Tilsvarende den almindelige matrixmultiplikation, så er denne matrixmultiplikation associativ: Hvis A, B og C er matricer, så gælder: $ABC = (AB)C = A(BC)$. Eksempelvis kan vi skrive produktet $SSSSSSSS$ på følgende måde: $((SS)(SS))((SS)(SS))$.

Idet ϵ kun er lidt større end den største afrundingsfejl, som kan opstå ved regning på flydende tal, så får vi følgende algoritme:

Algoritme 2.1: Den sekventielle matrixmetode

1. /* Initialisering. */
   $\forall i,j \in \{1,2,\ldots,n\}: A(i,j) \leftarrow s_{ij}$.
   $m \leftarrow 0$.

2. /* $(S^{2^m})^2 = S^{2^{m+1}}$ udregnes. Matrixmetodens */
   /* centrale punkt: Matrixmultiplikationen. */
   $m \leftarrow m+1$.
   $\forall i,j \in \{1,2,\ldots,n\}: B(i,j) \leftarrow \max_{k=1,2,\ldots,n} A(i,k) \times A(k,j)$.

3. /* Stopbetingelsen: */
   /* Hvis $S^{2^m} = S^{2^{m+1}}$, så er $S^{2^m} = S^*$. */
   Hvis der for alle $i,j \in \{1,2,\ldots,n\}$ gælder,
   at $\mid A(i,j) - B(i,j) \mid < \epsilon$, så fortsæt med punkt 5.

4. /* Den transitive afslutning er endnu ikke fundet. */
   $\forall i,j \in \{1,2,\ldots,n\}: A(i,j) \leftarrow B(i,j)$.
   Fortsæt med punkt 2.

5. /* Den transitive afslutning er fundet. */
   Elementerne s^*_{ij} i den transitive afslutning er givet ved $A(i,j)$.

For matrixmultiplikationen i punkt 2 skal n^3 skalarmultiplikationer udføres, og vi skal anvende maksimumoperatoren $(n-1)n^2$ gange. Dette giver en samlet kompleksitet på $(O(n^3) + O((n-1)n^2))$. I punkt 3 benytter vi, at hvis der gælder $S^{2^m} = S^{2^{m+1}}$, så er S^{2^m} den transitive afslutning. Da der er n^2 elementer i hver af matricerne, så kan vi i n^2 tidsenheder afgøre om $S^{2^m} = S^{2^{m+1}}$. Tildelingen i punkt 4 kræver også n^2 tidsenheder. Den samlede kompleksitet af punkt 3 og 4 bliver da $O(n^2)$. Idet vi højest skal udføre $\lceil \log_2(n-1) \rceil$ matrixmultiplikationer, så bliver kompleksiteten af den sekventielle matrixmetode:

$$(O(n^3) + O((n-1)n^2) + O(n^2)) \times O(\log(n)) = O(n^3 \log(n)).$$

2.1.2 De parallelle matrixmetoder

De parallelle matrixmetoder er baseret på, at formlen

$$c_{ij} = \max_{k=1,2,\ldots,n} a_{ik} b_{kj}, \tag{2.2}$$

for alle $i, j \in \{1, 2, \ldots, n\}$ bliver paralleliseret. Dette svarer til, at matrixmultiplikationen i punkt 2 i algoritme 2.1 bliver paralleliseret.

Vi vil opstille forskellige parallelle algoritmer for matrixmultiplikation. I afsnit 2.1.2.1 beskrives kube matrixmultiplikation. Med kube matrixmultiplikation som grundlag opfinder vi i afsnit 2.1.2.2 en ny algoritme — denne kalder vi plan matrixmultiplikation. I afsnit 2.1.2.3 beskrives Cannons matrixmultiplikation. I afsnit 2.1.2.4 er permutation matrixmultiplikation — en algoritme, som vi selv har opfundet. De forskellige matrixmetoder bliver navngivet efter hvilken matrixmultiplikation der anvendes. Matrixmetoden, som er baseret på kube matrixmultiplikation, bliver derfor kaldt kube matrixmetoden. Og så videre.

Det skal nævnes, at i denne fremstilling beskriver vi kun multiplikation af $n \times n$-matricer, da det grafteoretiske problem kræver dette. De forskellige parallelle algoritmer for matrixmultiplikation, som bliver beskrevet i de efterfølgende afsnit, kan middelbart generaliseres.

2.1.2.1 Kube matrixmetoden

Vi vil i dette afsnit parallelisere den sekventielle algoritme 2.1. Først vil vi opstille en parallel algoritme for multiplikation af $n \times n$-matricer, idet formel 2.2 bliver paralleliseret. Det er kube matrixmultiplikation.

En matrixmultiplikation af to $n \times n$-matricer kan opnås ved at anvende n^3 processorer. De n^3 processorer nummereres ved taltuplen (i, j, k), hvor $i, j, k \in \{1, 2, \ldots, n\}$.

Lad os nu udregne matrixproduktet $C = AB$. Elementerne i $n \times n$-matricerne A og B fordeles ud på de n^3 processorer. Den parallelle variabel $A(i,j,k)$ skal sættes lig værdien af a_{ij}, og den parallelle variabel $B(i,j,k)$ skal sættes lig værdien af b_{jk}. Imidlertid er matricerne A og B i vort tilfælde identiske. Dette kan udnyttes. Antag, at vi har foretaget følgende initialisering:

$$\forall i,j,k \in \{1,2,\ldots,n\}: A(i,j,k) \leftarrow a_{ij}.$$

Ved hjælp at et koordinatskifte kan vi "transponere" matricen B:

$$\forall i,j,k \in \{1,2,\ldots,n\}: B(i,j,k) \leftarrow A(j,k,i).$$

Koordinatskiftet tager højest $\lceil \log_2(n^3) \rceil = \lceil 3 \times \log_2(n) \rceil$ tidsenheder på en hyperkube paralleldatamat, da kommunikationen mellem to vilkårlige processorer højest tager $\lceil \log_2(n^3) \rceil$ tidsenheder. Kompleksiteten af koordinatskiftet er derfor $O(\log(n))$.

Efter denne tildeling gælder:

$$A(i,j,k) = a_{ij} \text{ for alle } k \in \{1,2,\ldots,n\}$$
$$B(i,j,k) = b_{jk} \text{ for alle } i \in \{1,2,\ldots,n\},$$

hvor matricerne A og B er identiske.

Matrixmultiplikationen udføres på følgende måde: Processor nummer (i,j,k) udfører skalarmultiplikationen $A(i,j,k) \times B(i,j,k)$ og gemmer resultatet i variablen $C(i,j,k)$. Efter denne tildeling gælder: $C(i,j,k) = a_{ij}b_{jk}$ for alle $i,j,k \in \{1,2,\ldots,n\}$.

Da vi er i besiddelse af n^3 processorer, så kan de n^3 skalarmultiplikationer udføres samtidigt, det vil sige på konstant tid. Kompleksiteten af de n^3 skalarmultiplikationer er derfor $O(1)$.

Vi mangler nu at udregne elementerne

$$c_{ik} = \max_{j=1,2,\ldots,n} a_{ij}b_{jk} = \max_{j=1,2,\ldots,n} C(i,j,k)$$

i den resulterende matrix C.

At finde det største af n tal kan udføres på $\lceil \log_2(n) \rceil$ tidsenheder, hvis n processorer anvendes. Følgende algoritme finder det største af de n tal a_i, der er placeret i de parallelle variable $A(i)$, hvor $i \in \{1, 2, \ldots, n\}$:

Algoritme 2.2: Maksimumoperator

1. /⋆ Initialisering. ⋆/
   $\forall i \in \{1, 2, \ldots, n\}: A(i) \leftarrow a_i$.
   $j \leftarrow 0$.

2. $\forall i \in \{i \mid 2^j + 1 \leq i \leq n\}: \underline{A(i) \leftarrow \max(A(i), A(i - 2^j))}$.

3. $j \leftarrow j + 1$.
   Hvis $j < \lceil \log_2(n) \rceil$, så fortsæt med punkt 2.

4. $\max_{j=1,2,\ldots,n} a_i$ er givet ved variablen $A(n)$.

I vort tilfælde skal maksimumoperatoren anvendes i alt n^2 gange, da der er n^2 elementer i den resulterende matrix. Imidlertid kan disse n^2 operationer udføres samtidigt, da vi netop har n^3 processorer.

Da koordinatskiftet højest tager $\lceil 3 \times \log_2(n) \rceil$ tidsenheder, de n^3 skalarmultiplikationer tager konstant tid, og anvendelsen af maksimumoperatoren tager $\lceil \log_2(n) \rceil$ tidsenheder, er kompleksiteten således $O(1) + O(\log(n))$ for en kube matrixmultiplikation. Vi får følgende algoritme:

Algoritme 2.3: Kube matrixmultiplikation

1. /⋆ Initialisering. ⋆/
   $\forall i,j,k \in \{1,2,\ldots,n\}: A(i,j,k) \leftarrow a_{ij}$.
2. /⋆ Matricerne A og B er i vort tilfælde identiske. ⋆/
   $\forall i,j,k \in \{1,2,\ldots,n\}: \underline{B(i,j,k) \leftarrow A(j,k,i)}$.
3. /⋆ De n^3 multiplikationer udføres. ⋆/
   $\forall i,j,k \in \{1,2,\ldots,n\}: \underline{C(i,j,k) \leftarrow A(i,j,k) \times B(i,j,k)}$.
4. /⋆ Maksimum findes, idet algoritme 2.2 anvendes. ⋆/
   $\forall i,j,k \in \{1,2,\ldots,n\}: \underline{C(i,k,j) \leftarrow \max_{t=1,2,\ldots,n} C(i,t,k)}$.
5. Elementerne i den resulterende matrix C er givet således:
   $\forall k \in \{1,2,\ldots,n\}: c_{ij} = C(i,j,k)$.

Vi har nu beskrevet en algoritme for parallel matrixmultiplikation. Vi mangler nu at parallelisere punkterne 3 og 4 i den sekventielle algoritme 2.1 på side 30.

Paralleliseringen af stopbetingelsen i punkt 3 i algoritme 2.1 har kompleksitet $O(1) + O(\log(n))$. At udføre subtraktionen og tage den numeriske værdi af de n^3 tal i et tre-dimensionalt array kan gøres på konstant tid, da der er n^3 processorer. Disse n^3 tal kan på paralleldatamaten repræsenteres i et en-dimensionalt array — vi behøver *ikke* at kopiere tallene over i et en-dimensionalt array. Ved hjælp af algoritme 2.2 kan vi finde det største af de n^3 tal i et en-dimensionalt array. At finde det størst blandt de n^3 tal tager $\lceil \log_2(n^3) \rceil = \lceil 3 \times \log_2(n) \rceil$ tidsenheder. Stopbetingelsen er nu reduceret til at afgøre, hvorvidt det største af de n^3 tal er større end ε.

Tildelingen i punkt 4 i algoritme 2.1 svarer i kube matrixmetoden til at kopiere et tre-dimensionalt array. Da vi har n^3 processorer, så kan dette foregå i konstant tid. Kompleksiteten af denne tildeling er derfor $O(1)$.

Den samlede kompleksitet af paralleliseringen af punkt 3 og 4 bliver $O(1) + O(\log(n))$. Den samlede kompleksitet af kube matrixmultiplikation og paralleliseringerne af punkt 3 og 4 i algoritme 2.1 er derfor $O(1) + O(\log(n))$.

Kompleksiteten af kube matrixmetoden bliver

$$(O(1) + O(\log(n))) \times O(\log(n)) = O(\log(n)^2),$$

da der højest skal udføre $\lceil \log_2(n-1) \rceil$ matrixmultiplikationer. Processor- og lagerforbruget er $O(n^3)$.

2.1.2.2 Plan matrixmetoden

Vi vil igen parallelisere algoritme 2.1, idet vi konstruerer en parallel algoritme for multiplikation af $n \times n$ matricer. Algoritmen har vi kaldt for plan matrixmultiplikation, da inspirationen er kube matrixmultiplikation (algoritme 2.3). Plan matrixmultiplikationen anvender n^2 processorer.

Betragt igen formlen 2.2 for matrixmultiplikation. Vi kan umiddelbart indføre sekvensen af matricer $C^{(k)}$, $k = 1, \ldots, n$, idet elementerne i $C^{(k)}$ er givet ved

$$c_{ij}^{(k)} = a_{ik} b_{kj}. \tag{2.3}$$

Analogt til den traditionelle matrixaddition, vil vi indføre maksimumoperatoren på matricer: Lad eksempelvis D og E være $n \times n$-matricer. Vi vil anvende maksimumoperatoren på matricerne D og E og resultatet vil vi kalde F. F er en $n \times n$-matrix. Elementerne i $F = \max(D, E)$ er givet ved $f_{ij} = \max(d_{ij}, e_{ij})$.

Ved at anvende maksimumoperatoren på matricer, så kan formel 2.2 umiddelbart omskrives til

$$C = \max_{k=1,2,\ldots,n} C^{(k)}, \tag{2.4}$$

idet vi anvender formel 2.3.

Plan matrixmultiplikation anvender n^2 processorer. De parallelle variable $A(i,j)$, $B(i,j)$ og $C(i,j)$ skal anvendes. Først konstrueres de n matricer fra formel 2.3 — den k'te matrix $C^{(k)}$ konstrueres således: Tildel de parallelle variable $A(i,j)$ værdierne a_{ik} for alle $i,j \in \{1,2,\ldots,n\}$, og tildel $B(i,j)$ værdierne b_{kj} for alle $i,j \in \{1,2,\ldots,n\}$ — det vil sige, at

$$\forall i,j \in \{1,2,\ldots,n\}: A(i,j) \leftarrow a_{ik} \tag{2.5}$$
$$\forall i,j \in \{1,2,\ldots,n\}: B(i,j) \leftarrow b_{kj}. \tag{2.6}$$

De parallelle variable $A(i,j)$ indeholder således n kopier af den k'te søjle fra matricen A, og de parallelle variable $B(i,j)$ indeholder n kopier af den k'te række fra matricen B.

Elementerne $c_{ij}^{(k)}$ i matricen $C^{(k)}$ er givet ved produktet $A(i,j) \times B(i,j)$. Det er således klart, at de n matricer $C^{(k)}$ kan konstrueres i n iterationer. Hvis vi i hver iteration anvender tildelingen

$$C(i,j) \leftarrow \max(C(i,j), A(i,j) \times B(i,j)), \tag{2.7}$$

så vil elementerne c_{ij} i den resulterende matrix C være givet ved $C(i,j)$ i den n'te iteration — forudsat, at alle $C(i,j)$ er initialiseret til 0.

Følgende algoritme for plan matrixmultiplikation kan opstilles:

Algoritme 2.4: Plan matrixmultiplikation

1. /⋆ Initialisering. ⋆/
   /⋆ $A'(i,j)$ indeholder matricen A. ⋆/
   $\forall i,j \in \{1,2,\ldots,n\}: \underline{C(i,j) \leftarrow 0}, A'(i,j) \leftarrow a_{ij}$.
   $k \leftarrow 1$.

2. /⋆ Matricerne A og B er identiske. $B'(i,j)$ indeholder ⋆/
   /⋆ efter tildelingen matricen A. ⋆/
   $\forall i,j \in \{1,2,\ldots,n\}: \underline{B'(i,j) \leftarrow A'(i,j).}$

3. /⋆ Tildelingen 2.5 — kopier den k'te søjle fra matricen A. ⋆/
   $\forall i \in \{1,2,\ldots,n\}: \underline{A(i,1) \leftarrow A'(i,k).}$
   $t \leftarrow 0$.

4. $\forall i \in \{1,2,\ldots,n\}, \forall s \in \{s \mid 1 \leq s \leq 2^t \wedge s + 2^t \leq n\}:$
   $\underline{A(i,s+2^t) \leftarrow A(i,s)}$.

5. $t \leftarrow t + 1$.
   Hvis $t < \lceil \log_2(n) \rceil$, så fortsæt med punkt 4.

6. /* Tildelingen 2.6 — kopier den k'te række fra matricen B. */
   $\forall j \in \{1,2,\ldots,n\}$: $\underline{B(1,j) \leftarrow B'(k,j).}$
   $t \leftarrow 0$.

7. $\forall j \in \{1,2,\ldots,n\}, \forall s \in \{s \mid 1 \leq s \leq 2^t \wedge s + 2^t \leq n\}$:
   $\underline{B(s+2^t,j) \leftarrow B(s,j).}$

8. $t \leftarrow t + 1$.
   Hvis $t < \lceil \log_2(n) \rceil$, så fortsæt med punkt 7.

9. /* Tildelingen 2.7. */
   $\forall i,j \in \{1,2,\ldots,n\}$: $\underline{C(i,j) \leftarrow \max(C(i,j), A(i,j) \times B(i,j)).}$

10. $k \leftarrow k + 1$.
    Hvis $k < n$, så fortsæt med punkt 3.

11. Elementerne c_{ij} i den resulterende matrix er givet ved $C(i,j)$.

Tildelingen i punkt 2 tager konstant tid — vi udnytter at matricerne A og B er identiske. Tildelingen 2.5 (punkt 3 - 5 i algoritme 2.4) og tildelingen 2.6 (punkt 6 - 8) tager hver $\lceil \log_2(n) \rceil$ tidsenheder. Da vi skal konstruere de n matricer i 2.3, resulterer tildelingerne 2.5 og 2.6 i en kompleksitet på $O(n \log(n))$ — og realiseringen af tildeling 2.7 betyder en kompleksitet på $O(n)$. Vi har derfor, at den samlede kompleksitet af matrixproduktet 2.4 er $O(1) + O(n \log(n)) + O(n)$.

Det er nu igen meget nemt at konstruere en parallel algoritme, som kan finde den transitive afslutning. Plan matrixmetoden fremkommer ved at anvende plan matrixmultiplikation, samt at parallelisere punkt 3 og 4 i algoritme 2.1. Vi vil parallelisere disse to punkter på tilsvarende måde som på side 34. Vi kan udføre subtraktionen og tage den numeriske værdi af de n^2 tal i punkt 3 på konstant tid. Vi kan finde det største af disse n^2 tal (ved hjælp af algoritme 2.2) i $\lceil \log_2(n^2) \rceil = \lceil 2 \times \log_2(n) \rceil$ tidsenheder. Paralleliseringen af tildelingen i punkt 4 har kompleksitet $O(1)$, da vi har n^2 processorer. Den samlede kompleksitet af paralleliseringen af punkt 3 og 4 bliver $O(1) + O(\log(n))$.

Vi får den samlede kompleksitet af plan matrixmultiplikation og paralleliseringerne af punkt 3 og 4 i algoritme 2.1 til

$$O(n \log(n)) + O(n) + O(1) + O(\log(n)).$$

Kompleksiteten af plan matrixmetoden bliver

$$(O(n\log(n)) + O(n) + O(1) + O(\log(n))) \times O(\log(n))$$
$$= O(n\log(n)^2),$$

da højest $\lceil \log_2(n-1) \rceil$ matrixmultiplikationer skal udføres. Processor- og lagerforbruget er $O(n^2)$.

2.1.2.3 Cannons matrixmetode

Vi vil igen parallelisere algoritme 2.1. Udgangspunktet er en parallel algoritme for multiplikation af $n \times n$ matricer — det er Cannons matrixmultiplikation. Denne algoritme anvender n^2 processorer.

Formlen for matrixmultiplikation 2.2 kan meget enkelt omskrives til følgende formel

$$c_{ij} = \max_{k=0,1,\ldots,n-1} a_{ik} b_{kj}, \tag{2.8}$$

idet vi blot anvender en anden indeksering af elementerne i matricerne A, B og C. I Cannons matrixmultiplikation anvendes, at k i formel 2.8 må gennemløbe værdierne i sekvensen $0, 1, \ldots, n-1$ på en vilkårlig måde — en vilkårlig permutation af sekvensen $0, 1, \ldots, n-1$ kan anvendes. Derfor kan formel 2.8 omskrives til

$$c_{ij} = \max_{k=0,1,\ldots,n-1} a_{i((((i+j) \bmod n)+k) \bmod n)} b_{((((i+j) \bmod n)+k) \bmod n)j}. \tag{2.9}$$

Formel 2.9 kan vi omskrive til

$$c_{ij} = \max_{k=0,1,\ldots,n-1} \tilde{a}_{i((j+k) \bmod n)} \tilde{b}_{((i+k) \bmod n)j}, \tag{2.10}$$

idet

$$\tilde{a}_{ij} = a_{i((i+j) \bmod n)} \tag{2.11}$$

og

$$\tilde{b}_{ij} = b_{((i+j) \bmod n)j}. \tag{2.12}$$

Vi har nu omskrevet formel 2.8 til formel 2.10, 2.11 og 2.12. Ud fra disse formler kan vi konstruere en effektiv parallel algoritme for matrixmultiplikation.

Cannons matrixmultiplikation anvender n^2 processorer, og hver processor (i,j) kontrollerer de parallelle variable $A(i,j)$, $B(i,j)$ og $C(i,j)$. Først initialiseres de parallelle variable $A(i,j)$ og $B(i,j)$ med henholdsvis værdierne $\tilde{a}_{ij} = a_{i((i+j) \bmod n)}$ og $\tilde{b}_{ij} = b_{((i+j) \bmod n)j}$. Her har vi anvendt formel 2.11 og formel 2.12. Det ses, at elementerne i den i'te række i matricen A bliver roteret netop i gange mod venstre, og elementerne i den j'te søjle i matricen B bliver roteret opad netop j gange, idet $i,j \in \{0,1,\ldots,n-1\}$. i og j kan skrives ud fra deres binære repræsentation på følgende måde, idet den første bit er yderst til højre i den binære repræsentation:

$$i = \sum_{k=0}^{\lceil \log_2(n) \rceil - 1} \begin{cases} 2^k & \text{hvis } k+1\text{'te bit i } i \text{ er 1} \\ 0 & \text{hvis } k+1\text{'te bit i } i \text{ er 0} \end{cases} \tag{2.13}$$

$$j = \sum_{k=0}^{\lceil \log_2(n) \rceil - 1} \begin{cases} 2^k & \text{hvis } k+1\text{'te bit i } j \text{ er 1} \\ 0 & \text{hvis } k+1\text{'te bit i } j \text{ er 0.} \end{cases} \tag{2.14}$$

Idet k i formel 2.13 og 2.14 gennemløber sekvensen $0,1,\ldots,\lceil \log_2(n) \rceil - 1$, når vi frem til følgende algoritme, som kan "vride"[3] matricerne A og B:

Algoritme 2.5: "Vridning"

1. /* De parallelle variable $A(i,j)$ og $B(i,j)$ er initialiseret. */
   $k \leftarrow 0$.

[3]Vi har oversat ordet "preskewing" til "vridning", se Bjørstad (1992).

2. /⋆ Roter 2^k pladser mod venstre i i'te række i matricen A, ⋆/
   /⋆ hvis $k+1$'te bit i den binære repræsentation af i er 1. ⋆/
   $\forall i,j \in \{0,1,\ldots,n-1\}$: Hvis $k+1$'te bit i i er 1, så:
   $A(i,j) \leftarrow A(i,(j+2^k) \bmod n)$.

3. /⋆ Roter 2^k pladser opad i j'te søjle i matricen B, ⋆/
   /⋆ hvis $k+1$'te bit i den binære repræsentation af j er 1. ⋆/
   $\forall i,j \in \{0,1,\ldots,n-1\}$: Hvis $k+1$'te bit i j er 1, så:
   $B(i,j) \leftarrow B((i+2^k) \bmod n, j)$.

4. $k \leftarrow k+1$.
   Hvis $2^k < n$, så fortsæt med punkt 2.

5. Matricerne A og B er blevet "vredet".

Algoritmen "vrider" matricerne A og B i netop $\lceil \log_2(n) \rceil$ iterationer, og algoritmen har derfor kompleksitet $O(\log(n))$.

Vi kan nu færdiggøre resten af Cannons algoritme for matrixmultiplikation: Vi mangler nu blot at parallelisere formel 2.10. Af formlen 2.10 ses det, at k angiver hvor mange gange søjlerne i A skal roteres mod venstre, og hvor mange gange rækkerne i B skal roteres opad. Hvis k gennemløber sekvensen $0,1,\ldots,n-1$ i n iterationer, og der i hver iteration på nær den første roteres én gang i søjlerne i A og én gang i rækkerne i B, så vil søjlerne i A og rækkerne i B være roteret k gange i $k+1$'te iteration. I hver iteration skal hver enkelt processor (i,j) multiplicere de parallelle variable $A(i,j)$ og $B(i,j)$, og variablen $C(i,j)$ skal tildeles dette produkt, dersom værdien af $C(i,j)$ er mindre end dette produkt. De n iterationer har kompleksitet $O(n)$.

Vi har således, at realiseringen af formel 2.11 og 2.12 resulterer en kompleksitet på $O(\log(n))$ og realiseringen af formel 2.10 resulterer en kompleksitet på $O(n)$. Idet vi udnytter, at matricerne A og B i vort tilfælde er identiske, så tager kopieringen konstant tid. Den samlede kompleksitet af den parallelle algoritme er derfor $O(\log(n)) + O(n) + O(1)$. Vi har følgende algoritme:

Algoritme 2.6: Cannons matrixmultiplikation

1. /⋆ Initialisering. ⋆/
   $\forall i,j \in \{0,1,\ldots,n-1\}: A(i,j) \leftarrow a_{ij}$.

2. /⋆ Matricerne A og B er identiske. ⋆/
   $\forall i,j \in \{0,1,\ldots,n-1\}: B(i,j) \leftarrow A(i,j)$.

3. "Vrid" matricerne A og B med algoritme 2.5.

4. /⋆ Ingen rotation i første iteration. ⋆/
   $\forall i,j \in \{0,1,\ldots,n-1\}: C(i,j) \leftarrow A(i,j) \times B(i,j)$.
   $k \leftarrow 0$.

5. /⋆ Rækkerne i A roteres mod venstre og søjlerne i B ⋆/
   /⋆ roteres opad k gange i $k+1$'te iteration. ⋆/
   $k \leftarrow k+1$.
   $\forall i,j \in \{0,1,\ldots,n-1\}: A(i,j) \leftarrow A(i,(j+1) \bmod n)$.
   $\forall i,j \in \{0,1,\ldots,n-1\}: B(i,j) \leftarrow B((i+1) \bmod n,j)$.
   $\forall i,j \in \{0,1,\ldots,n-1\}: C(i,j) \leftarrow \max(C(i,j), A(i,j) \times B(i,j))$.

6. Hvis $k < n-1$, så fortsæt med punkt 5.

7. Elementerne c_{ij} i den resulterende matrix er givet ved $C(i,j)$.

Tilsvarende de to foregående afsnit er det nu nemt at opstille en parallel algoritme for matrixmetoden. Vi skal blot anvende Cannons matrixmultiplikation og paralleliseringerne af punkt 3 og 4 i den sekventielle algoritme 2.1 — disse paralleliseringer er beskrevet på side 37.

Den samlede kompleksitet af Cannons matrixmultiplikation og paralleliseringerne af punkt 3 og 4 i den sekventielle algoritme 2.1 er $O(\log(n)) + O(n) + O(1)$.

Cannons matrixmetode har kompleksitet

$$(O(\log(n)) + O(n) + O(1)) \times O(\log(n)) = O(n \log(n)),$$

da der højest skal udføre $\lceil \log_2(n-1) \rceil$ matrixmultiplikationer. Processor- og lagerforbruget er $O(n^2)$.

2.1.2.4 Permutation matrixmetoden

Vi vil endnu en gang præsentere en parallel algoritme for matrixmetoden. Vi anvender en algoritme for matrixmultiplikation, som vi selv har opfundet — vi har kaldt den permutation matrixmultiplikation.

Ideen i permutation matrixmultiplikation er den samme som i Cannons matrixmultiplikation. Formel 2.8 for matrixmultiplikation fra side 38 omskrives, idet k i denne formel må gennemløbe sekvensen $0, 1, \ldots, n-1$ på en vilkårlig måde — en vilkårlig permutation af sekvensen $0, 1, \ldots, n-1$ kan anvendes. Permutation matrixmultiplikation anvender en "fornuftig" permutation.

Formel 2.8 kan omskrives til

$$c_{ij} = d_{i((j+n-i) \bmod n)}, \tag{2.15}$$

hvor matricen D er givet ved

$$d_{ij} = \max_{k=0,1,\ldots,n-1} a_{ik} b_{k((i+j) \bmod n)}. \tag{2.16}$$

Analogt til Cannons matrixmultiplikation, skal permutation multiplikation anvende n^2 processorer. De parallelle variable $A(i,j)$, $B(i,j)$ og $D(i,j)$ skal anvendes.

Først initialiseres de parallelle variable $A(i,j)$, hvor $i,j \in \{0,1,\ldots,n-1\}$, med værdierne a_{ij}. De parallelle variable $B(i,j)$ skal have værdierne b_{ji}. Bemærk, at matricen B skal transponeres. Da matricerne A og B er identiske i vort tilfælde, så kan vi anvende følgende tildeling efter at have initialiseret de parallelle variable $A(i,j)$: $\forall i,j \in \{0,1,\ldots,n-1\} : B(i,j) \leftarrow A(j,i)$. At transponere matricen B på denne måde tager $\lceil \log_2(n^2) \rceil = \lceil 2 \times \log_2(n) \rceil$ tidsenheder på en hyperkube paralleldatamat.

Formel 2.16 kan realiseres i n iterationer, idet den j'te søjle i matricen D konstrueres i den $j+1$'te iteration. j gennemløber sekvensen $0, 1, \ldots, n-1$. Da vi er i besiddelse af n^2 processorer, så kan de n elementer i den j'te søjle i matricen D findes på $\lceil \log_2(n) \rceil$ tidsenheder, hvis vi benytter algoritme 2.2 til at finde det største af n tal.

Den j'te søjle i D kan konstrueres ved følgende to tildelinger:

$$\forall i \in \{0, 1, \ldots, n-1\} : D(i,j) \leftarrow \max_{k=0,1,\ldots,n-1} A(i,k) \times B(i,k) \quad (2.17)$$

$$B(i,k) \leftarrow B((i+1) \bmod n, k). \quad (2.18)$$

Tildelingen 2.18 betyder, at rækkerne i den transponerede matrix af B bliver roteret opad netop j gange i den $j+1$'te iteration, som formel 2.16 foreskriver. Tildelingen 2.18 har kompleksitet $O(1)$ — og da den skal anvendes n gange, resulterer dette i en kompleksitet på $O(n)$. Tilsvarende vil tildeling 2.17 resultere i en kompleksitet på $n \times O(\log(n)) = O(n \log(n))$. Den samlede kompleksitet er da $O(n \log(n)) + O(n)$.

Efter at have realiseret formel 2.16 mangler vi blot at realisere formel 2.15. Det ses af formel 2.15, at elementerne c_{ij} er givet ved $d_{i((j+n-i) \bmod n)}$. Dette betyder, at elementerne i den i'te række i matricen D skal roteres $n-i$ gange til *venstre* — eller i gange til *højre*. Vi benytter den sidste opfattelse.

Ideen at rotere elementerne i den i'te række i gange til højre i matricen D, ligner meget ideen i algoritme 2.5 (sammenlign punkt 2 i algoritme 2.5 med punkt 6 i algoritme 2.7). Kompleksiteten af de to operationer er den samme, nemlig $O(\log(n))$.

Transponeringen og paralleliseringerne af tildelingerne 2.15, 2.17 og 2.18 bliver således $O(n \log(n)) + O(n) + O(\log(n))$.

Vi kan opstille følgende algoritme til permutation matrixmultiplikation:

Algoritme 2.7: Permutation matrixmultiplikation

1. /⋆ Initialisering. ⋆/
   $\forall i,k \in \{0,1,\ldots,n-1\}: A(i,k) \leftarrow a_{ik}$.
   $j \leftarrow 0$.

2. /⋆ Matricen B transponeres — A og B er jo identiske. ⋆/
   $\forall i,k \in \{0,1,\ldots,n-1\}: \underline{B(i,k) \leftarrow A(k,i).}$

3. /⋆ j'te søjle i D findes, idet maksimum ⋆/
   /⋆ findes ved hjælp af algoritme 2.2. ⋆/
   $\forall i \in \{0,1,\ldots,n-1\}: \underline{D(i,j) \leftarrow \max_{k=0,1,\ldots,n-1} A(i,k) \times B(i,k).}$
   /⋆ Rækkerne i den transponerede matrix af B ⋆/
   /⋆ roteres j gange opad. ⋆/
   $\forall i,k \in \{0,1,\ldots,n-1\}: \underline{B(i,k) \leftarrow B((i+1) \bmod n, k).}$

4. $j \leftarrow j+1$.
   Hvis $j < n-1$, så fortsæt med punkt 3.

5. $t \leftarrow 0$.

6. /⋆ Roter 2^t pladser til højre i i'te række i D, hvis ⋆/
   /⋆ $t+1$'te bit i den binære repræsentation af i er 1. ⋆/
   $\forall i,k \in \{0,1,\ldots,n-1\}$: $\underline{\text{Hvis } t+1\text{'te bit i } i \text{ er 1, så:}}$
   $\underline{D(i,(k+2^t) \bmod n) \leftarrow D(i,k).}$

7. $t \leftarrow t+1$.
   Hvis $2^t < n$, så fortsæt med punkt 6.

8. Elementerne c_{ik} i den resulterende matrix er givet ved $D(i,k)$.

Vi kan nu igen parallelisere matrixmetoden, idet permutation matrixmultiplikation og paralleliseringerne fra side 37 af punk 3 og 4 i algoritme 2.1 anvendes.

Den samlede kompleksitet af permutation matrixmultiplikation og paralleliseringerne af punkt 3 og 4 i algoritme 2.1 er $O(n\log(n)) + O(n) + O(1) + O(\log(n))$.

Permutation matrixmetoden har kompleksitet

$$(O(n\log(n)) + O(n) + O(\log(n)) + O(1)) \times O(\log(n))$$
$$= O(n\log(n)^2).$$

Processor- og lagerforbruget er $O(n^2)$.

2.2 Floyds algoritme

I afsnit 2.2.1 beskrives en sekventiel algoritme for udregning af den transitive afslutning. Der er tale om Floyds algoritme.

I afsnit 2.2.2 paralleliseres den sekventielle udgave af Floyds algoritme.

2.2.1 Den sekventielle udgave

Vi betragter en graf G bestående af n knuder fra mængden af knuder $V = \{v_1, v_2, \ldots, v_n\}$. Vi skal finde den transitive afslutning S^* for grafen G.

Ideen i Floyds algoritme er at konstruere en sekvens af matricer

$$S^{(0)}, S^{(1)}, \ldots, S^{(i)}, \ldots, S^{(n)}, \tag{2.19}$$

hvor elementerne i den i'te matrix $S^{(i)}$ er defineret ud fra iterationsformlen

$$s_{jk}^{(i)} = \max(s_{jk}^{(i-1)}, s_{ji}^{(i-1)} s_{ik}^{(i-1)}), \tag{2.20}$$

idet $j, k \in \{1, 2, \ldots, n\}$. Pointen med at definere elementerne i matricerne i sekvensen 2.19 ud fra iterationsformlen 2.20 er, at hvis vi sætter $S^{(0)} = S$, så vil elementerne i den i'te itererede matrix $S^{(i)}$ repræsentere styrken af den stærkeste sti fra den j'te knude til den k'te knude blandt alle stier, hvor knuder mellem den j'te og den k'te knude (hvis der overhovedet er nogen) kun kan være knuder fra mængden: $\{v_1, v_2, \ldots, v_i\}$.

Vi skal nu vise, at hvis elementerne i $S^{(i)}$ er givet ved iterationsformlen 2.20, så vil vi efter n iterationer have, at $S^{(n)} = S^*$, hvor S^* er den transitive afslutning. Vi vil vise dette ved induktion.

Det første trin i induktionsbeviset:

Betragt den anden matrix $S^{(1)}$ i sekvensen 2.19, hvor elementerne er defineret ud fra iterationsformlen 2.20

$$s_{jk}^{(1)} = \max(s_{jk}^{(0)}, s_{j1}^{(0)} s_{1k}^{(0)}).$$

Elementerne i matricen $S^{(1)}$ indeholder information om styrken af stierne v_j, v_k eller v_j, v_1, v_k, hvor $j, k \in \{1, 2, \ldots, n\}$. Hvis eksempelvis begge stier v_j, v_k og v_j, v_1, v_k eksisterer, så indeholder $S^{(1)}$ information om $\max(L(v_j, v_k), L(v_j, v_1, v_k))$. I $S^{(1)}$ har vi således udvidet informationen i forhold til $S^{(0)}$.

Andet trin i induktionsbeviset:

Antag nu, at elementerne i den $(i-1)$'te matrix $S^{(i-1)}$ i sekvensen 2.19 repræsenterer styrken af den stærkeste sti $v_j, \ldots, v_k$, hvor alle knuder mellem v_j og v_k tilhører mængden $\{v_1, v_2, \ldots, v_{i-1}\}$.

Tredje trin i induktionsbeviset:

Vi ønsker nu at finde styrken af den stærkeste sti $v_j, \ldots, v_k$, men nu vil vi desuden tillade alle knuder mellem v_j og v_k at kunne tilhøre mængden $\{v_1, v_2, \ldots, v_i\}$.

Betragt nu den stærkeste sti $v_j, \ldots, v_i$ og den stærkeste sti $v_i, \ldots, v_k$, hvor det tillades alle knuder mellem den første og den sidste knude i de to stier at kunne tilhøre mængden $\{v_1, v_2, \ldots, v_{i-1}\}$.

I følge antagelsen er styrkerne af disse to stier givet ved $s_{ji}^{(i-1)}$ og $s_{ik}^{(i-1)}$. Da styrken af den stærkeste sti $v_j, \ldots, v_k$, hvor det tillades alle knuder mellem v_j og v_k at kunne tilhøre mængden $\{v_1, v_2, \ldots, v_{i-1}\}$, er givet ved $s_{jk}^{(i-1)}$, må styrken af den stærkeste sti $v_j, \ldots, v_k$, hvor det nu tillades alle knuder mellem v_j og v_k at kunne tilhøre mængden $\{v_1, v_2, \ldots, v_i\}$, være givet ved $\max(s_{jk}^{(i-1)}, s_{ji}^{(i-1)} s_{ik}^{(i-1)})$.

Det er således klart, at elementerne i $S^{(n)}$ repræsenterer styrken af den stærkeste sti $v_j, \ldots, v_k$, hvor alle knuder mellem v_j og v_k tilhører mængden $\{v_1, v_2, \ldots, v_n\}$. Men da alle knuder mellem v_j og v_k tilhører mængden $\{v_1, v_2, \ldots, v_n\}$ er styrken af den stærkeste sti fra v_j til v_k i grafen G givet ved $s_{jk}^{(n)}$. Vi har derfor at $S^{(n)} = S^*$.

Vi har følgende beskrivelse af Floyds algoritme i pseudokode:

Algoritme 2.8: Floyds algoritme — sekventiel

1. /⋆ Initialisering. ⋆/
   $\forall h, l \in \{1, 2, \ldots, n\}: A(h, l) \leftarrow s_{hl}$.

2. For $i \leftarrow 1, 2, ..., n$ udfør:
   For $j \leftarrow 1, 2, ..., n$ udfør:
   Hvis $(A(j, i) > 0)$ og $(i \neq j)$, så udfør:
   For $k \leftarrow 1, 2, ..., n$ udfør:
   $A(j, k) \leftarrow \max(A(j, k), A(j, i) \times A(i, k))$.

3. Den transitive afslutning er fundet: elementerne s_{jk}^* er givet ved $A(j, k)$.

Vi har i algoritme 2.8 udnyttet, at $s_{jk}^{(i)} = s_{ji}^{(i-1)} s_{ik}^{(i-1)} = 0$, hvis $s_{ji}^{(i-1)} = 0$ samt at $s_{jk}^{(i)} = s_{ji}^{(i-1)} s_{ik}^{(i-1)} = s_{jk}^{(i-1)}$, hvis $i = j$. Dette vil bevirke, at der kan spares n multiplikationer, samt at tage maksimum n gange, hvis dette er tilfældet.

Kompleksiteten af Floyds algoritme kan direkte aflæses ved inspektion af koden. Heraf fås at denne maksimalt kan blive $O(n^3)$.

2.2.2 Den parallelle udgave

Vi vil i dette afsnit beskrive, hvordan Floyds algoritme fra afsnit 2.2.1 kan paralleliseres. Betragt endnu engang iterationsformlen 2.20 — denne formel vil vi parallelisere.

Vi indfører en matrix $A^{(i-1)}$, hvis elementer er givet ved:

$$a_{jk}^{(i-1)} = s_{ji}^{(i-1)}, \tag{2.21}$$

hvor $j, k \in \{1, 2, \ldots, n\}$. Vi ser altså at *alle søjlerne* i $A^{(i-1)}$ er identiske med den i'te søjle i $S^{(i-1)}$. Tilsvarende indfører vi en matrix $B^{(i-1)}$, hvis elementer er givet ved:

$$b_{jk}^{(i-1)} = s_{ik}^{(i-1)}, \tag{2.22}$$

hvor $j, k \in \{1, 2, \ldots, n\}$. Vi ser altså at *alle rækkerne* i $B^{(i-1)}$ er identiske med den i'te række i $S^{(i-1)}$. Elementerne i matricen $S^{(i)}$ kan nu udregnes som:

$$s_{jk}^{(i)} = \max(s_{jk}^{(i-1)}, a_{jk}^{(i-1)} b_{jk}^{(i-1)}). \tag{2.23}$$

Formel 2.21, 2.22 og 2.23 kan nu stykkes sammen til en parallel udgave af Floyds algoritme. I algoritmen anvendes n^2 processorer, og hver processor (i, j) kontrollerer de parallelle variable $A(i, j)$, $B(i, j)$ og $C(i, j)$. Vi har følgende algoritme:

Algoritme 2.9: Floyds algoritme — parallel

1. /⋆ Initialisering. ⋆/
   $\forall j, k \in \{1, 2, \ldots, n\} : C(j, k) \leftarrow s_{jk}$.
   $i \leftarrow 1$.

2. /⋆ Matricen $A^{(i-1)}$ findes. Søjlerne i $A^{(i-1)}$ ⋆/
   /⋆ er identisk med den i'te søjle i $S^{(i-1)}$. ⋆/
   $\forall j \in \{1, 2, \ldots, n\} : \underline{A(j, 1) \leftarrow C(j, i).}$
   $t \leftarrow 0$.

3. $\forall j \in \{1, 2, \ldots, n\}, \forall s \in \{s \mid 1 \leq s \leq 2^t \wedge s + 2^t \leq n\} :$
   $\underline{A(j, s + 2^t) \leftarrow A(j, s).}$

4. $t \leftarrow t + 1$.
   Hvis $t < \lceil \log_2(n) \rceil$, så fortsæt med punkt 3.

5. /⋆ Matricen $B^{(i-1)}$ findes. Rækkerne i $B^{(i-1)}$ ⋆/
   /⋆ er identisk med den i'te række i $S^{(i-1)}$. ⋆/
   $\forall j \in \{1, 2, \ldots, n\} : \underline{B(1, j) \leftarrow C(i, j).}$
   $t \leftarrow 0$.

6. $\forall j \in \{1,2,\ldots,n\}, \forall s \in \{s \mid 1 \leq s \leq 2^t \wedge s + 2^t \leq n\}$:
   $\underline{B(s+2^t,j) \leftarrow B(s,j)}$.

7. $t \leftarrow t+1$.
   Hvis $t < \lceil \log_2(n) \rceil$, så fortsæt med punkt 6.

8. /* Matricen $S^{(i)}$ findes. */
   $\forall j,k \in \{1,2,\ldots,n\}$:
   $\underline{C(j,k) \leftarrow \max(C(j,k), A(j,k) \times B(j,k))}$.

9. $i \leftarrow i+1$.
   Hvis $i \leq n$, så fortsæt med punkt 2.

10. Elementerne i s^*_{jk} er givet ved $C(j,k)$.

Kompleksiteten af den parallelle udgave af Floyds algoritme findes som følger: At kopiere en søjle/række fra en matrix til alle søjlerne/rækkerne i en anden matrix resulterer i en kompleksitet på $O(\log(n))$. Denne operation skal udføres 2 gange for hver af de n iterationer. Der skal udføres en skalarmultiplikation og tages maksimum en gang for hver iteration, som i begge tilfælde resulterer i en kompleksitet på $O(1)$. Den samlede kompleksitet bliver derfor

$$O(n\log(n)) + O(n) = O(n\log(n)).$$

Processor- og lagerforbruget er $O(n^2)$.

Bemærkning: Den parallelle udgave af Floyds algoritme egner sig til tykke grafer, da det alene er antallet af knuder som afgør hvor mange operationer, der skal udføres.

Observation: Den parallelle udgave af Floyds algoritme og plan matrixmetoden minder strukturelt set en del om hinanden, idet punkterne 2–8 i algoritme 2.9 og punkterne 3–9 i algoritme 2.4 er identiske.

2.3 De parallelle algoritmers egenskaber

Vi vil benytte Cannons matrixmetode (se afsnit 2.1.2.3) som det gennemgående eksempel i dette afsnit. Betragt den parallelle algoritme for Cannons matrixmetode. Kompleksiteten af denne algoritme er

$$(O(\log(n)) + O(n) + O(1)) \times O(\log(n)) = O(n\log(n)),$$

og der kræves n^2 processorer.

Vor paralleldatamat CM-200 har 8192 processorer — hvis vi har en graf med $n = 128$ knuder, så skal vi anvende i alt $128^2 = 16384$ processorer. Imidlertid har vi jo kun 8192 processorer, så hver processor må arbejde for 2! Det bliver endnu værre når grafen har $n = 256$ knuder — så skal hver processor arbejde for 8, da $\lceil 256^2/8192 \rceil = 8$. Vi vil derfor opstille et andet mål for kompleksiteten, som tager højde for at paralleldatamaten kun har et fast antal processorer p. Vi indfører *den reelle kompleksitet*[4]:

Definition 2.1

> Lad kompleksiteten af en parallel algoritme være givet ved $O(K(n))$. Den reelle kompleksitet er da givet ved
>
> $$\lceil P(n)/p \rceil \times O(K(n)) = O(P(n) \times K(n)),$$
>
> hvor $P(n)$ er det antal processorer, som algoritmen kræver, og hvor p er antallet af fysiske processorer på paralleldatamaten.

Den reelle kompleksitet af Cannons matrixmetode er:

$$\begin{aligned} &\lceil n^2/p \rceil \times (O(\log(n)) + O(n) + O(1)) \times O(\log(n)) \\ &= O(n^3 \log(n)). \end{aligned} \tag{2.24}$$

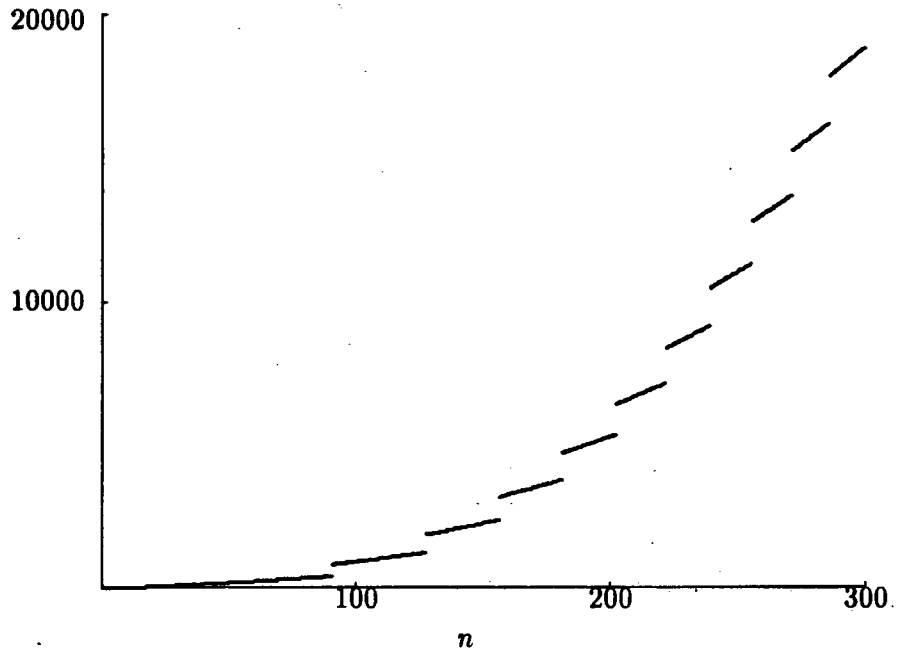
I figur 2.1 er "trappefunktionen"

$$f(n) = \lceil n^2/8192 \rceil \times n\log(n) \tag{2.25}$$

indtegnet. Funktionen 2.25 er afledt af funktion 2.24.

Skitsen i figur 2.1 skulle i store træk afspejle tidsforbruget (på nær en konstant faktor) på CM-200 for Cannons matrixmetode.

[4] Den reelle kompleksitet bliver (på engelsk) kaldet for "cost", se Aki (1989) side 26.



Figur 2.1: Funktionen $f(n) = \lceil n^2/8192 \rceil \times n\log(n)$ afbildet mod n — denne funktion er i store træk proportional med tidsforbruget for Cannons matrixmetode på paralleldatamaten CM-200.

Vi kan se følgende pointer af figur 2.1:

1. Tidsforbruget er proportional med $n\log(n)$ i alle intervaller mellem "springene" i "trappefunktionen" 2.25. Men — proportionalfaktoren $\lceil n^2/8192 \rceil$ er ikke konstant. Højden på "trappetrinnene" vokser med en faktor n^2. Trenden i funktionen 2.25 er derfor proportional med $n^3\log(n)$.

2. Den sekventielle algoritme 2.1 har kompleksitet $O(n^3\log(n))$. Denne kompleksitet er derfor proportional med trenden af funktionen 2.25.

I følge punkt 1 er trenden af funktionen 2.25 proportional med $n^3\log(n)$ — men inden for intervallerne mellem "springene" er tidsforbruget proportional med $n\log(n)$. Dette taler for, at n skal være lille. Punkt 2 taler også for at n skal være lille, da kompleksiteten af den sekventielle algoritme 2.1 er proportional med trenden i funktionen 2.25.

Vi vil opstille følgende definition af en optimal[5] parallel algoritme:

Definition 2.2

> Lad A_s og A_p være henholdsvis mængden af sekventielle algoritmer og parallelle algoritmer, som kan løse et givet problem. Blandt de sekventielle algoritmer i A_s udvælges algoritmen med den mindste kompleksitet $O(K_s(n))$. Ligeledes udvælges den parallelle algoritme fra A_p med mindste kompleksitet $O(K_p(n))$. Den parallelle algoritme er optimal hvis og kun hvis $P(n) \times O(K_p(n)) = O(K_s(n))$, hvor $P(n)$ er antallet af processorer som problemet kræver.

Den optimale parallelle algoritme kan konkurrere med den bedste sekventielle algoritme selv når problemet er stort (det vil sige, at problemets størrelse går mod uendelig). Når problemet er lille, det vil sige, når antallet af processorer som algoritmen kræver er mindre end eller lig med antallet af fysiske processorer, så er det fordelagtigt at anvende den parallelle algoritme.

Vi så før, at den reelle kompleksitet af Cannons matrixmetode var det samme som kompleksiteten af den sekventielle algoritme 2.1. Imidlertid har den sekventielle algoritme 2.1 ikke den mindste kompleksitet blandt de sekventielle algoritmer — for eksempel har den sekventielle udgave af Floyds algoritme (algoritme 2.9) mindre kompleksitet. Cannons matrixmetode er derfor ikke optimal. Den sekventielle udgave af Floyds algoritme er derfor fordelagtig at anvende, når n er stor fremfor den parallelle Cannons matrixmetode.

I tabel 2.1 er angivet de parallelle algoritmers kompleksitet, reelle kompleksitet, samt processor- og lagerforbrug. Vi kan se, at ingen af de parallelle algoritmer er optimale, da den sekventielle udgave af Floyds algoritme har lavere kompleksitet (nemlig $O(n^3)$) end den reelle kompleksitet af de parallelle algoritmer.

Cannons matrixmetode og den parallelle udgave af Floyds algoritme har tilsyneladende de samme egenskaber. Plan matrixmetoden og permutation matrixmetoden har også de samme egenskaber, men Cannons matrixmetode og den parallelle udgave af Floyds algoritme er bedre.

[5] Aki bruger betegnelsen "cost optimal", se Aki (1989) side 26. Den optimale parallelle algoritme ikke nødvendigvis entydig — der kan eksistere flere.

	Kompleksitet	Reel kompleksitet	Processor- og lagerforbrug
kube matrixmetoden	$O(\log(n)^2)$	$O(n^3 \log(n)^2)$	$O(n^3)$
plan matrixmetoden	$O(n \log(n)^2)$	$O(n^3 \log(n)^2)$	$O(n^2)$
Cannons matrixmetode	$O(n \log(n))$	$O(n^3 \log(n))$	$O(n^2)$
permutation matrixmetoden	$O(n \log(n)^2)$	$O(n^3 \log(n)^2)$	$O(n^2)$
(parallel) Floyds algoritme	$O(n \log(n))$	$O(n^3 \log(n))$	$O(n^2)$

Tabel 2.1: Kompleksitet, den reelle kompleksitet, samt processor- og lagerforbrug for de parallelle algoritmer.

Kube matrixmetoden adskiller sig fra de andre, idet den har en virkelig god kompleksitet, men til gengæld har den et stort behov for processorer. Den reelle kompleksitet af kube matrixmetoden er af samme orden som den reelle kompleksitet af plan matrixmetoden og permutation matrixmetoden. Endvidere vil kube matrixmetode krævet meget lager — n gange så meget som de andre.

Kube matrixmetode er derfor den bedste algoritme, når n^3 er mindre end eller lig med antallet af fysiske processorer på paralleldatamaten; i dette tilfælde kan den transitive afslutning findes på $O(\log(n)^2)$ tidsenheder. På vor paralleldatamat CM-200 skal n være mindre end eller lig med 20, hvis 8192 processorer skal være nok[6].

Hvis n^2 er mindre end eller lig med antallet af fysiske processorer, og n^3 er større end antallet af fysiske processorer, så kan Cannons matrixmetode eller den parallelle udgave af Floyds algoritme anvendes; tidsforbruget er $O(n \log(n))$ tidsenheder. På CM-200 skal n være mellem 21 og 90.

[6]Vi stiller måske nogle lidt urimelige krav — i praksis må n^3 gerne være nogle gange større end 8192, da båndbredden på paralleldatamaten er meget større end på en sekventiel datamat (se afsnit 1.2). Vi vil dog se stort på dette i denne fremstilling, da vi foretrækker at præsentere resultaterne teoretisk.

Hvis n^2 er større end antallet af fysiske processorer, bliver den sekventielle udgave af Floyds algoritme[7] en konkurrent til Cannons matrixmetode og den parallelle udgave af Floyds algoritme — i dette tilfælde skal den sekventielle algoritme ikke udføres på paralleldatamaten, men naturligvis på en sekventiel datamat.

	Algoritme	Tidsforbrug	Datamat
$n^3 \leq p$	kube matrixmetoden	$O(\log(n)^2)$	Parallel
$p < n^3 \wedge p \geq n^2$	Cannons matrixmetode	$O(n \log(n))$	Parallel
$p < n^3 \wedge p \geq n^2$	(parallel) Floyds algoritme	$O(n \log(n))$	Parallel
$p < n^2$	(sekventiel) Floyds algoritme	$O(n^3)$	Sekventiel

Tabel 2.2: *Skitse over hvordan algoritmerne til at finde en grafs transitive afslutning vælges, når antallet af knuder n er kendt. p er antallet af processorer på en CM-200–lignende paralleldatamat. Bemærk, at vi også vælger type af datamat — sekventiel eller parallel!*

I tabel 2.2 er en skitse af hvordan en passende algoritme vælges ud fra antallet af knuder n i grafen og antallet af processorer p på paralleldatamaten. Bemærk, at i skemaet indgår også valget af type datamat.

Referencer

Aki (1989) beskriver på side 259 matrixmetoden (i princippet både den sekventielle og parallelle) — han anvender kube matrixmultiplikation, som han har beskrevet på side 183. Den sekventielle matrixmetode er også beskrevet i Larsen (1990). Endvidere opstiller Aki (1989) på side 47 en algoritme, som kan addere n tal parallelt. Udgangspunktet for algoritme 2.2 (maksimumoperatoren) har været denne algoritme.

Vi har forsøgt med andre algoritmer for matrixmultiplikation. Cannons matrixmultiplikation har vi fra Bjørstad (1992), mens vi selv har konstrueret plan matrixmultiplikation og permutation matrixmultiplikation.

[7] Vi har stiltiende antaget, at de betragtede grafer er tykke. Imidlertid ville vi ikke anvende Floyds algoritme på en sekventiel datamat, hvis graferne er tynde. Vi ville i stedet vælge Dijkstras algoritme (Sedgewick (1988) side 457). Kompleksiteten af denne algoritme er $O(n(n+e)\log(n))$, hvor e er antallet af kanter.

Floyds algoritme (den sekventielle) har vi fra Sedgewick (1988) side 477. Vi har selv konstrueret den parallelle udgave af Floyds algoritme

I Aki (1989) afsnit 1.3 defineres kompleksitet, reel kompleksitet og optimale parallelle algoritmer.

Kapitel 3

Tidsforbrug på CM-200

I afsnit 2.1 og 2.2 analyserede vi de opstillede parallelle algoritmer og fandt frem til et mål for de parallelle algoritmers kompleksitet. Da vi fandt frem til dette mål, antog vi, at paralleldatamaten havde processorer nok. I afsnit 2.3 indførte vi den reelle kompleksitet. Dette mål tager højde for, at paralleldatamaten kun har et fast antal processorer.

I dette kapitel vil vi verificere det reelle kompleksitetsmål, idet vi netop med udgangspunkt i dette vil opstille en model over tidsforbruget for hver af de parallelle algoritmer, som blev behandlet i kapitel 2.

Ved at anvende statistiske metoder vil vi dels vurdere, om der er overensstemmelse mellem det observerede og estimerede tidsforbrug, samt undersøge om modellerne kan anvendes til at forudsige tidsforbruget på paralleldatamaten. Vi skal være opmærksomme på følgende:

Hvis vi ikke kan afvise den opstillede model over tidsforbruget, har vi en vis garanti for, at kompleksitetsmålet (der hvor vi har processorer nok) fra afsnit 2.1 og 2.2 er rigtigt. Vi kan dog godt være i den situation at kompleksitetsmålet (og dermed den opstillede model) ved et tilfælde et korrekt, men at analysen af algoritmen er fejlagtig eller mangelfuld.

Hvis vi afviser den opstillede model over tidsforbruget er kompleksitetsmålet forkert. Dette kan enten skyldes at præmisserne for opstillingen af kompleksitetsmålet er forkerte eller, at implementeringen forårsager, at kompleksiteten bliver anderledes end forventet. Hvis det sidste er tilfældet kan det være svært at forklare årsagen hertil, da den ikke nødvendigvis er synlig for programudvikleren.

Bemærkninger: Vi måler det samlede tidsforbrug for udførsel af hver af de implementerede algoritmer fra afsnit 2.1 og 2.2. Vi måler tidsforbruget på den parallelle enhed af CM-200 (se side 20) under tidsdeling (se afsnit B.3). En begrundelse for at vi ikke måler tidsforbruget på front-end'en kan findes i afsnit B.4.

For at kunne sammenligne de forskellige algoritmer, bliver tidsmålingerne for hver implementeret algoritme så vidt muligt udført på samme sæt af $n \times n$-matricer S (se afsnit B.2). I implementeringen af kube matrixmetoden er dette dog ikke muligt, da n på grund af lagerforbruget skal være mindre end eller lig med 256 (se afsnit B.2).

3.1 Kube matrixmetoden

Fra afsnit 2.1.2.1 ved vi, at kompleksiteten af kube matrixmetoden er $(O(1) + O(\log(n))) \times O(\log(n))$. Vi ved, at antallet af processorer, som algoritmen kræver, er n^3. Den reelle kompleksitet er derfor

$$(O(\log(n)) + O(1)) \times O(\log(n)) \times \left\lceil \frac{n^3}{p} \right\rceil. \tag{3.1}$$

Idet vi betinger med, at n^3 skal være et multiplum af det fysiske antal processorer p, er ligning 3.1 identisk med

$$(O(\log(n)) + O(1)) \times O(\log(n)) \times \frac{n^3}{p}. \tag{3.2}$$

I ligning 3.2 har vi antaget, at antallet af matrixmultiplikationer er proportional med $\log(n)$. Som vi så i afsnit 2.1.1, skal der højest udføres $\log(n)$ matrixmultiplikationer. Imidlertid afhænger antallet af matrixmultiplikationer af grafens "udseende".

Når vi skal opstille en model over tidsforbruget, er vi interesseret i at kende det eksakte antal matrixmultiplikationer. I praksis kan antallet af matrixmultiplikationer selvfølgelig tælles. I modellen over tidsforbruget vil vi betegne antallet af matrixmultiplikationer ved størrelsen $s(n)$. Derved kan ligning 3.2 omskrives til

$$(O(\log(n)) + O(1)) \times O(s(n)) \times \frac{n^3}{p}$$
$$= O(n^3 s(n) \log(n)) + O(n^3 s(n)),$$

hvilket giver os følgende model over tidsforbruget

$$T(n) = k_1 n^3 s(n) \log(n) + k_2 n^3 s(n). \tag{3.3}$$

Da vi ikke har så mange observationer til rådighed for estimation af parametrene, har vi valgt at simplificere modellen over tidsforbruget. Dette har vi gjort ved at udelade det mindst betydende led i modellen 3.3, hvilket giver os følgende model over tidsforbruget for kube matrixmetoden

$$T(n) = k_1 n^3 s(n) \log(n). \tag{3.4}$$

Vi har estimeret parameteren k_1 til 1.867×10^{-8} ved mindste kvadraters metode, idet vi har anvendt alle 4 observationer som grundlag for estimationen. I tabel 3.1 har vi vist en række resultater — størrelsen n, det observerede tidsforbrug, det forventede tidsforbrug, samt differensen mellem det observerede og det forventede tidsforbrug.

Vi har udført kørsler med $n \times n$-matricer, hvor $n = 32, 64, 128$ og 256. En begrundelse for valget af netop disse værdier kan findes i afsnit B.2. Det forventede tidsforbrug er givet ved ligning 3.4.

n	$s(n)$	Observeret tidsforbrug	Estimeret tidsforbrug	Afvigelse fra forventet tidsforbrug
32	4	0.007 s	0.008 s	−0.002 s
64	6	0.053 s	0.122 s	−0.069 s
128	5	0.481 s	0.950 s	−0.469 s
256	4	7.012 s	6.947 s	0.065 s

Tabel 3.1: Tidsforbruget i sekunder for kube matrixmetoden.

Vi har udregnet et skøn over variansen[1], som vi vil anvende til at vurdere modellens brugbarhed. Hvis variansen er "stor", vil vi forkaste modellen, hvis den er "lille", vil vi ikke forkaste den.

Vi vil understrege at en vurdering af om variansen er stor eller lille er et subjektivt spørgsmål. Vi har udregnet variansen til 0.076. Dette vil vi betegne som en "lille" størrelse, så vi vil altså ikke forkaste modellen 3.4. Derfor har vi også prøvet at undersøge, om modellen kan bruges til at forudsige tidsforbruget for en kørsel. Dette har vi gjort ved at estimere k_1, idet vi nu kun har anvendt de 3 første observationer som grundlag for estimationen.

Med et estimat på 9.432×10^{-9} har vi brugt modellen til at forudsige tidsforbruget for $n = 256$. Vi har dernæst udregnet den relative afvigelse fra det forudsagte tidsforbrug i procent, det vil sige

$$\left| \frac{\text{observeret tidsforbrug} - \text{forudsagt tidsforbrug}}{\text{forudsagt tidsforbrug}} \right| \times 100. \qquad (3.5)$$

[1]Et centralt skøn over variansen opnås ved at udregne $\frac{1}{m-b}\sum_{i=1}^{m}(y-\hat{y}_i)^2$, hvor y_i er det observerede tidsforbrug og $\hat{y}_i$ er det estimerede tidsforbrug. m er antallet af observationer og b antallet af parametre i modellen.

Dette gav en relativ afvigelse på næsten 100%. At modellen 3.4 ikke giver en særlig god forudsigelse af tidsforbruget kan skyldes, at der er for få observationer (der er kun 3 observationer) til estimation af k_1, eller at der er for stor unøjagtighed på tidsmålingerne (de observerede tider for $n = 32$ og 64 er jo meget små).

3.2 Plan matrixmetoden

Fra afsnit 2.1.2.2 ved vi, at kompleksiteten af plan matrixmetoden er $(O(n\log(n)) + O(n) + O(\log(n)) + O(1)) \times O(\log(n))$. Endvidere ved vi, at antallet af processorer, som algoritmen kræver, er n^2. Den reelle kompleksitet er derfor

$$(O(n\log(n)) + O(n) + O(\log(n)) + O(1)) \times O(\log(n)) \times \frac{n^2}{p}. \tag{3.6}$$

Idet antallet af matrixmultiplikationer betegnes ved $s(n)$, kan ligning 3.6 omskrives til

$$\begin{aligned}&(O(n\log(n)) + O(n) + O(\log(n)) + O(1)) \times O(s(n)) \times \frac{n^2}{p}\\ &= O(n^3 s(n)\log(n)) + O(n^3 s(n)) + O(n^2 s(n)\log(n)) + O(n^2 s(n)),\end{aligned}$$

hvilket giver os følgende model over tidsforbruget

$$\begin{aligned}T(n) = {} & k_1 n^3 s(n)\log(n) + k_2 n^3 s(n) + \\ & k_3 n^2 s(n)\log(n) + k_4 n^2 s(n).\end{aligned} \tag{3.7}$$

Med samme begrundelser som i afsnit 3.1 simplificerer vi modellen 3.7 ved at udelade det mindst betydende led. Således har vi følgende model over tidsforbruget for plan matrixmetoden

$$T(n) = k_1 n^3 s(n) \log(n) + k_2 n^3 s(n) + k_3 n^2 s(n) \log(n). \qquad (3.8)$$

Vi har estimeret parametrene k_1, k_2 og k_3 til henholdsvis 9.269×10^{-10}, 5.663×10^{-9} og 8.251×10^{-7}, idet vi har anvendt 6 observationer som grundlag for estimationen. I tabel 3.2 har vi vist resultaterne.

n	$s(n)$	Observeret tidsforbrug	Estimeret tidsforbrug	Afvigelse fra forventet tidsforbrug
128	5	1.208 s	0.434 s	0.773 s
256	4	2.848 s	1.924 s	0.923 s
512	5	15.65 s	14.43 s	1.228 s
1024	5	94.16 s	94.88 s	−0.716 s
2048	5	678.8 s	678.7 s	0.115 s
4096	4	4136 s	4136 s	−0.007 s

Tabel 3.2: Tidsforbruget i sekunder for plan matrixmetoden.

Vi har udregnet variansen til 1.162, hvilket vi vil betegne som en "lille" størrelse. Vi har derfor estimeret k_1, k_2 og k_3 ud fra de 5 første observationer til henholdsvis 5.625×10^{-9}, -3.396×10^{-8} og 1.847×10^{-6}, samt forudsagt tidsforbruget til 4557 s for $n = 4096$. Den relative afvigelse har vi udregnet til 9.2%.

Der er tilsyneladende en rimelig overensstemmelse mellem det observerede og det estimerede tidsforbrug. Der er også en rimelig overensstemmelse mellem det observerede og det forudsagte tidsforbrug.

3.3 Cannons matrixmetode

Fra afsnit 2.1.2.3 ved vi, at kompleksiteten af Cannons matrixmetode er $(O(n)+O(\log(n))+O(1)) \times O(\log(n))$. Endvidere ved vi, at antallet af processorer, som algoritmen kræver, er n^2. Den reelle kompleksitet er derfor

$$(O(n) + O(\log(n)) + O(1)) \times O(\log(n)) \times \frac{n^2}{p}. \qquad (3.9)$$

Idet antallet af matrixmultiplikationer betegnes ved $s(n)$, kan ligning 3.9 omskrives til

$$(O(n) + O(\log(n)) + O(1)) \times O(s(n)) \times \frac{n^2}{p}$$
$$= O(n^3 s(n)) + O(n^2 s(n) \log(n)) + O(n^2 s(n)),$$

hvilket giver os følgende model over tidsforbruget

$$T(n) = k_1 n^3 s(n) + k_2 n^2 s(n) \log(n) + k_3 n^2 s(n). \qquad (3.10)$$

Denne model simplificeres til følgende model over tidsforbruget for Cannons matrixmetode

$$T(n) = k_1 n^3 s(n) + k_2 n^2 s(n) \log(n). \qquad (3.11)$$

Parametrene k_1 og k_2 er estimeret til henholdsvis 8.926×10^{-9} og 6.200×10^{-7} med 6 observationer som grundlag for estimationen. I tabel 3.3 har vi vist resultaterne.

n	$s(n)$	Observeret tidsforbrug	Estimeret tidsforbrug	Afvigelse fra forventet tidsforbrug
128	5	0.428 s	0.340 s	0.088 s
256	4	1.631 s	1.500 s	0.130 s
512	5	11.27 s	11.06 s	0.208 s
1024	5	69.48 s	70.45 s	−0.973 s
2048	5	482.8 s	482.5 s	0.318 s
4096	4	2800 s	2800 s	−0.031 s

Tabel 3.3: Tidsforbruget i sekunder for Cannons matrixmetode.

Vi har udregnet variansen til 0.2794, som betegnes som "lille". Da vi således ikke vil afvise modellen 3.11, har vi også prøvet at forudsige tidsforbruget for $n = 4096$, idet vi har anvendt de 5 første observationer som grundlag for estimationen af parametrene. Dette gav estimater for k_1 og k_2 på henholdsvis 9.121×10^{-9} og 5.696×10^{-7}, samt et forudsagt tidsforbrug på 2825 s. Den relative afvigelse er ved formel 3.5 udregnet til 0.9%.

Der er altså en rimelig overensstemmelse mellem det observerede, det estimerede, samt det forudsagte tidsforbrug.

3.4 Permutation matrixmetoden

Fra afsnit 2.1.2.4 ved vi, at kompleksiteten af permutation matrixmetoden er $(O(n\log(n))+O(n)+O(\log(n))+O(1))\times O(\log(n))$. Endvidere ved vi, at antallet af processorer, som algoritmen kræver, er n^2. Den reelle kompleksitet er derfor

$$(O(n\log(n)) + O(n) + O(\log(n)) + O(1))\times O(\log(n)) \times \frac{n^2}{p}. \tag{3.12}$$

Idet antallet af matrixmultiplikationer betegnes ved $s(n)$, kan ligning 3.12 omskrives til

$$\begin{aligned}&(O(n\log(n)) + O(n) + O(\log(n)) + O(1)) \times O(s(n)) \times \frac{n^2}{p}\\ &= O(n^3 s(n)\log(n)) + O(n^3 s(n)) + O(n^2 s(n)\log(n)) + O(n^2 s(n)),\end{aligned}$$

hvilket giver os følgende model over tidsforbruget

$$\begin{aligned}T(n) \;=\; & k_1 n^3 s(n)\log(n) + k_2 n^3 s(n) + \\ & k_3 n^2 s(n)\log(n) + k_4 n^2 s(n).\end{aligned}$$

Denne model simplificeres til følgende model over tidsforbruget for permutation matrixmetoden

n	$s(n)$	Observeret tidsforbrug	Estimeret tidsforbrug	Afvigelse fra forventet tidsforbrug
128	5	0.984 s	0.630 s	0.354 s
256	4	3.081 s	2.643 s	0.437 s
512	5	18.99 s	18.56 s	0.433 s
1024	5	114.1 s	114.3 s	−0.269 s
2048	5	776.4 s	776.4 s	0.044 s
4096	4	4577 s	4577 s	−0.003 s

Tabel 3.4: Tidsforbruget i sekunder for permutation matrixmetoden.

$$T(n) = k_1 n^3 s(n) \log(n) + k_2 n^3 s(n) + k_3 n^2 s(n) \log(n). \qquad (3.13)$$

Vi har estimeret parametrene k_1, k_2 og k_3 til henholdsvis 1.174×10^{-9}, 4.199×10^{-9} og 1.324×10^{-6} med 6 observationer som grundlag for estimationen. I tabel 3.4 har vi vist resultaterne.

Vi har udregnet variansen til 0.1925. Med en så "lille" størrelse har vi estimeret k_1, k_2 og k_3 ud fra de 5 første observationer til henholdsvis 2.978×10^{-9}, -1.102×10^{-8} og 1.717×10^{-6}, samt forudsagt tidsforbruget til 4738 s for $n = 4096$. Den relative afvigelse har vi udregnet til 3.4%.

Der er altså en rimelig overensstemmelse mellem det observerede, det estimerede, samt det forudsagte tidsforbrug.

3.5 Den parallelle udgave af Floyds algoritme

Fra afsnit 2.2.2 ved vi, at kompleksiteten af den parallelle udgave af Floyds algoritme er $O(n \log(n)) + O(n) = O(n \log(n))$. Endvidere ved vi, at antallet af processorer, som algoritmen kræver, er n^2. Den reelle kompleksitet er derfor

$$\begin{aligned} &(O(n \log(n)) + O(n)) \times \frac{n^2}{p} \\ &= O(n^3 \log(n)) + O(n^3), \end{aligned}$$

hvilket giver os følgende model over tidsforbruget

$$T(n) = k_1 n^3 \log(n) + k_2 n^3. \tag{3.14}$$

Med 6 observationer som grundlag for estimationen, har vi estimeret parametrene k_1 og k_2 til henholdsvis -1.399×10^{-9} og 2.632×10^{-8}. I tabel 3.5 har vi vist resultaterne.

n	Observeret tidsforbrug	Estimeret tidsforbrug	Afvigelse fra forventet tidsforbrug
128	0.237 s	0.041 s	0.196 s
256	0.711 s	0.311 s	0.400 s
512	3.193 s	2.361 s	0.832 s
1024	18.90 s	17.85 s	1.051 s
2048	134.1 s	134.4 s	−0.305 s
4096	1009 s	1009 s	0.020 s

Tabel 3.5: *Tidsforbruget i sekunder for den parallelle udgave af Floyds algoritme.*

Vi har udregnet variansen til 0.5224. Denne vil også betegne som "lille" og vi har derfor ud fra de 5 første observationer estimeret k_1 og k_2 til henholdsvis -3.065×10^{-9} og 3.898×10^{-8}, samt forudsagt tidsforbruget for $n = 4096$ til 927 s. Dette giver en relative afvigelse på 8.8%.

Der er altså en rimelig overensstemmelse mellem det observerede og det estimerede tidsforbrug. Der er også en rimelig overensstemmelse mellem det observerede og det forudsagte tidsforbrug, omend resultatet ikke er nær så overbevisende som i Cannons matrixmetode og permutation matrixmetoden.

3.6 Sammenfatning af måling af tidsforbruget

I afsnit 2.3 opstillede vi definitionen på den reelle kompleksitet. I kapitel 3 opstillede vi med udgangspunkt i den reelle kompleksitet en model over tidsforbruget på CM-200 for hver af de parallelle matrixmetoder, samt den parallelle udgave af Floyds algoritme. Her kontrollerede vi ved at udføre en række kørsler, hvor vi målte det samlede tidsforbrug, om der var overensstemmelse mellem det observerede og det estimerede tidsforbrug.

Det estimerede tidsforbrug blev fundet ud fra de opstillede modeller over tidsforbruget, idet vi havde estimeret parametrene i modellerne ved mindste kvadraters metode. Vi fandt i alle tilfælde, at der tilsyneladende var god overensstemmelse mellem det observerede og det estimerede tidsforbrug.

Vi havde desuden prøvet at undersøge om modellerne kunne bruges til at forudsige tidsforbruget for en kørsel. Vi kunne herved konstatere, at der var en rimelig overensstemmelse mellem det observerede og det forudsagte tidsforbrug undtaget for kube matrixmetoden.

I kube matrixmetoden kunne vi ikke afgøre, om det dårlige resultat (en relativ afvigelse på cirka 100%) skyldtes: For få observationer (der var kun 3 observationer) til estimation af parameteren, for stor unøjagtighed på tidsmålingerne eller en kombination af de to muligheder.

Referencer

Larsen (1993) giver i kapitel 8 en grundlæggende indførsel i lineær regressionsanalyse. Draper (1966) kan anbefales for personer, der er interesseret i en mere generel og langt mere omfangsrig indførsel i samme emne.

Konklusion

Vi havde følgende overordnede mål med dette projekt: At konstruere (eller finde i litteraturen) parallelle algoritmer, finde deres kompleksitet, samt at implementere disse algoritmer på en paralleldatamat.

De parallelle algoritmer skulle kunne implementeres effektivt på paralleldatamaten CM-200. I kapitel 1 gav vi en beskrivelse af denne paralleldatamat. Vi opnåede derved en forståelse af, at problemet skulle formuleres som operationer på matrix-lignende strukturer.

Med udgangspunkt i traditionelle sekventielle algoritmer, som kunne finde en grafs transitive afslutning, konstruerede vi i kapitel 2 parallelle udgaver af disse algoritmer til en CM-200–lignende paralleldatamat. Det var tale om følgende sekventielle algoritmer: Matrixmetoden og Floyds algoritme.

For matrixmetoden i afsnit 2.1.2 opstillede vi fire forskellige algoritmer, idet vi anvendte fire forskellige algoritmer for parallel matrixmultiplikation. Kube matrixmultiplikation og Cannons matrixmultiplikation var kendte algoritmer, mens vi selv opfandt plan matrixmultiplikation og permutation matrixmultiplikation. I afsnit 2.2.2 paralleliserede vi Floyds algoritme — vi opfandt derved en ny algoritme.

Vi forsøgte at give en fuldstændig beskrivelse af algoritmernes kompleksitet. Ligeledes beskrev vi algoritmerne således, at de skulle være umiddelbare at implementere (for eksempel i C^*) til en CM-200–lignende paralleldatamat.

Som det fremgår af afsnit 2.3, havde plan matrixmetoden og permutation matrixmetoden højere kompleksitet end kube matrixmetoden og Cannons matrixmetode. Imidlertid var vor parallelle udgave af Floyds algoritme forbløffende god. Kompleksiteten var af samme orden som Cannons matrixmetode.

Den nedenstående tabel er en gengivelse fra afsnit 2.3. Tabellen viser kompleksitet, den reelle kompleksitet, samt processor- og lagerforbrug for de parallelle algoritmer.

	Kompleksitet	Reel kompleksitet	Processor- og lagerforbrug
kube matrixmetoden	$O(\log(n)^2)$	$O(n^3 \log(n)^2)$	$O(n^3)$
plan matrixmetoden	$O(n \log(n)^2)$	$O(n^3 \log(n)^2)$	$O(n^2)$
Cannons matrixmetode	$O(n \log(n))$	$O(n^3 \log(n))$	$O(n^2)$
permutation matrixmetoden	$O(n \log(n)^2)$	$O(n^3 \log(n)^2)$	$O(n^2)$
(parallel) Floyds algoritme	$O(n \log(n))$	$O(n^3 \log(n))$	$O(n^2)$

I afsnit 2.3 opstillede vi også retningslinier for, hvilke af de parallelle algoritmer der skulle anvendes, idet parametrene var antallet af knuder n i den betragtede graf, samt antallet af fysiske processorer p på den pågældende paralleldatamat. Den nedenstående tabel er en gengivelse fra afsnit 2.3.

	Algoritme	Tidsforbrug	Datamat
$n^3 \leq p$	kube matrixmetoden	$O(\log(n)^2)$	Parallel
$p < n^3 \wedge p \geq n^2$	Cannons matrixmetode	$O(n \log(n))$	Parallel
$p < n^3 \wedge p \geq n^2$	(parallel) Floyds algoritme	$O(n \log(n))$	Parallel
$p < n^2$	(sekventiel) Floyds algoritme	$O(n^3)$	Sekventiel

Af denne tabel kan vi se, at vi ikke ukritisk skal vælge paralleldatamaten CM-200 fremfor sekventielle datamater. Når n er stor, så er det fordelagtig at anvende en sekventiel algoritme på en sekventiel datamat. Det skal dog nævnes i denne forbindelse, at paralleldatamater har forholdsvis større båndbredde end sekventielle datamater. Proportionalfaktoren, som vi ser bort fra i det teoretiske kompleksitetsmål, er derfor meget lille på paralleldatamater. Dette betyder i praksis, at n^2 skal være en del gange større end p, før vi anvender sekventielle datamater.

Vi mener at have verificerede algoritmernes kompleksitet. I kapitel 3 opstillede vi ud fra algoritmers reelle kompleksitet matematiske modeller over tidsforbruget på paralleldatamaten. Modellerne var således ikke ad hoc modeller. Ved at sammenligne resultaterne fra disse modeller med det observerede tidsforbrug, kunne vi ikke forkaste disse modeller.

Appendiks A

Notation

Vi vil i afsnit A.1 indføre den grafteoretiske notation, som er anvendt i denne fremstilling. I afsnit A.2 indføres kort "sproget" eller "pseudokoden", som algoritmerne i kapitel 2 er angivet i.

A.1 Grafteori og notation

I dette afsnit indføres den grafteoretiske notation, som er anvendt i denne fremstilling.

Vi vil betragte orienterede, vægtede grafer $G = (V, E, W)$. $V = \{v_1, v_2, \ldots, v_n\}$ er mængden af n knuder, og E er mængden af kanter: $E \subseteq V \times V$. Mængden E er således en mængde af talpar (v_i, v_j), hvor $v_i, v_j \in V$. En sti er en sekvens af knuder $v_a, v_b, v_c, \ldots, v_d, v_e$, idet $\{(v_a, v_b), (v_b, v_c), \ldots, (v_d, v_e)\} \subseteq E$, og en knude højest optræder én gang i sekvensen — dog med den undtagelse, at den første og sidste knude gerne må være den samme.

Lad mængden R være følgende relation:

$$R = \{(v_i, v_j) \in V \times V \mid \text{ der eksisterer en sti fra } v_i \text{ til } v_j\}.$$

Denne relation er transitiv:

$$\forall v_i, v_j, v_k \in V\colon ((v_i, v_k) \in R) \wedge ((v_k, v_j) \in R) \Rightarrow (v_i, v_j) \in R.$$

Det vil sige, at hvis der er en sti fra v_i til v_k, og en sti fra v_k til v_j, så er der også en sti fra v_i til v_j.

Relationen R er også refleksiv: $\forall v_i \in V$: $(v_i, v_i) \in R$, idet vi kun betragter grafer, hvor de n kanter (v_i, v_i) tilhører E.

$W(v_i, v_j)$ er en funktion, som til hver kant $(v_i, v_j) \in E$ knytter et reelt tal: W: $E \mapsto]0, 1]$. Specielt gælder for de n kanter $(v_i, v_i) \in E$, at $W(v_i, v_i) = 1$. Det bemærkes, at W er strengt større end 0.

Ud fra funktionen W kan elementerne i $n \times n$-matricen S indføres på følgende måde:

$$s_{ij} = \begin{cases} W(v_i, v_j) & \text{hvis } (v_i, v_j) \in E \\ 0 & \text{hvis } (v_i, v_j) \notin E, \end{cases}$$

hvor $i, j \in \{1, 2, \ldots, n\}$. Konstanterne s_{ij} kaldes grafens vægte. Det bemærkes, at $s_{ij} = 0$, hvis kanten (v_i, v_j) ikke eksisterer.

Styrken af stien $v_a, v_b, v_c, \ldots, v_d, v_e$ er givet ved $L(v_a, v_b, v_c, \ldots, v_d, v_e) = s_{ab} s_{bc} \cdots s_{de}$. Hvis $(v_i, v_j) \in R$, så kaldes den sti, som har den største styrke, for den stærkeste sti fra v_i til v_j. Trods superlativen er den stærkeste sti ikke nødvendigvis entydig. Styrken af den stærkeste sti er givet ved $\max_{p_{ij} \in P_{ij}} L(p_{ij})$, hvor P_{ij} er mængden af stier fra v_i til v_j. Ved den transitive afslutning for grafen $G = (V, E, W)$ forstås $n \times n$-matricen S^*, hvor elementerne s^*_{ij} er givet således:

$$s^*_{ij} = \begin{cases} \max\limits_{p_{ij} \in P_{ij}} L(p_{ij}) & \text{hvis } (v_i, v_j) \in R \\ 0 & \text{hvis } (v_i, v_j) \notin R. \end{cases}$$

Vi bemærker, at hvis der ikke er en sti fra v_i til v_j, så er $s^*_{ij} = 0$.

Det kan i øvrigt nævnes, at problemet at finde "styrken af den stærkeste sti" er analogt med problemet at finde "længden af den korteste sti". I det første tilfælde bruger vi maksimumoperatoren og multiplikation, mens vi i det sidste tilfælde bruger minimumoperatoren og addition.

A.2 Algoritmer og notation

Vi vil i dette afsnit indføre pseudokoden, som bruges i forbindelse med algoritmerne i kapitel 2. Bemærk, at notationen benyttes både for sekventielle og parallelle algoritmer.

Paralleldatamaten skal være af samme type som CM-200 (se afsnit 1.4).

Vi vil ikke give en stringent og fuldstændig beskrivelse af vor pseudokode. Vi mener imidlertid, at pseudokoden er så ukompliceret, at læseren kan læse kapitel 2 med fuldt udbytte ved at betragte det efterfølgende eksempel:

Lad os som eksempel konstruere en algoritme, der kan addere to 2×2-matricer A og B, og tildele resultatet til matricen C. Vi har indført et lignende eksempel i afsnit 1.4.5.

Lad matricerne A og B være givet ved:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}.$$

Matricen C er naturligvis:

$$C = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix},$$

da elementerne i matricen C er givet ved:

$$c_{ij} = a_{ij} + b_{ij},$$

hvor $i, j \in \{1, 2\}$.

Når vi skal konstruere algoritmer til CM-200, så skal vi anvende den datastruktur, som i datalogiske termer kaldes et "array". Tankegangen er den, at hver position i arrayet varetages af netop én processor.

I vort tilfælde skal vi addere to 2×2-matricer, så vor datastruktur kan betragtes som et array af størrelsen 2×2.

Det er klart, at vi kan finde de 4 elementer i C uafhængigt af hinanden — hvis vi har 4 processorer, så er disse i stand til at udføre matrixadditionen i ét hug.

Elementerne i matricerne A og B skal fordeles på de 4 processorers lager. Lad os nummerere processorerne på tilsvarende måde som elementerne i matricerne A og B. Den første processor kan vi nummerere med taltuplen $(1,1)$, den anden med taltuplen $(1,2)$ — og så videre.

Pointen er nu, at processor nummer (i,j) skal kontrollere tre parallelle variable, som vi kan kalde $A(i,j)$, $B(i,j)$ og $C(i,j)$ — den parallelle variabel $A(i,j)$ indeholder værdien af elementet a_{ij}, og $B(i,j)$ indeholder værdien af b_{ij}. Når additionen er udført, så indeholder den parallelle variabel $C(i,j)$ værdien af elementet c_{ij}. Vi bemærker, at de elementer, som skal adderes, kontrolleres af netop én processor. I vort tilfælde kan dette skitseres som i figur A.1.

Processor nummer (1,1) $A(1,1) = 1$ $B(1,1) = 5$ $C(1,1)$ = "ikke defineret"	Processor nummer (1,2) $A(1,2) = 2$ $B(1,2) = 6$ $C(1,2)$ = "ikke defineret"
Processor nummer (2,1) $A(2,1) = 3$ $B(2,1) = 7$ $C(2,1)$ = "ikke defineret"	Processor nummer (2,2) $A(2,2) = 4$ $B(2,2) = 8$ $C(2,2)$ = "ikke defineret"

Figur A.1: I lageret til processor nummer (i,j) er værdierne a_{ij} og b_{ij}.

Det ses, at vi har tildelt $A(i,j)$ og $B(i,j)$ de respektive værdier, samt at $C(i,j)$ ingen værdi har endnu. I vor pseudokode anvender vi følgende notation:

$$\forall i,j \in \{1,2\}: A(i,j) \leftarrow a_{ij}, B(i,j) \leftarrow b_{ij}.$$

Dette betyder, at vi tildeler $A(i,j)$ og $B(i,j)$ de respektive værdier. Denne initialisering er sekventiel — det kunne for eksempel være, at værdierne skulle indlæses fra en fil. For at kunne kende forskel på parallelle og sekventielle operationer, så vedtager vi, at parallelle operationer bliver understregede i teksten. Den næste operation er parallel:

$$\forall i,j \in \{1,2\}: \underline{C(i,j) \leftarrow A(i,j) + B(i,j).}$$

Dette er selve additionen af matricerne A og B — de 4 processorer udfører samtidigt de 4 additioner af elementerne i A og B. Derefter tildeles $C(i,j)$ værdien c_{ij}.

Processor nummer (1,1)	Processor nummer (1,2)
$A(1,1) = 1$	$A(1,2) = 2$
$B(1,1) = 5$	$B(1,2) = 6$
$C(1,1) = 6$	$C(1,2) = 8$
Processor nummer (2,1)	**Processor nummer (2,2)**
$A(2,1) = 3$	$A(2,2) = 4$
$B(2,1) = 7$	$B(2,2) = 8$
$C(2,1) = 10$	$C(2,2) = 12$

Figur A.2: Efter additionen er elementet c_{ij} i lageret til processor nummer (i,j).

Figur A.2 viser situationen efter denne tildeling.

Algoritmen vil i vor pseudokode se således ud:

1. /⋆ Initialisering. ⋆/
   /⋆ Vi indlæser elementerne fra matricerne A og B. ⋆/
   $\forall i,j \in \{1,2\}: A(i,j) \leftarrow a_{ij}$, $B(i,j) \leftarrow b_{ij}$.

2. /⋆ Selve additionerne af elementerne i matricerne ⋆/
   /⋆ A og B udføres. Operationen er parallel ⋆/
   /⋆ (og er derfor understreget i teksten). ⋆/

3. $\forall i,j \in \{1,2\}: \underline{C(i,j) \leftarrow A(i,j) + B(i,j).}$

4. /⋆ Matricen $C = A + B$ er fundet — vi er færdige! ⋆/
   Elementerne i c_{ij} er givet ved $C(i,j)$.

I vor pseudokode er også andre konstruktioner, som ikke fremgår af det viste eksempel. Der er betingende konstruktioner, "hop" konstruktioner, samt diverse matematiske konstruktioner.

Referencer

Sedgewick (1988) kapitel 29 og Aki (1989) kapitel 10 beskriver begge den nødvendige grafteori, men vor fremstillingsform er meget mere stringent end deres.

Vi har konstrueret vor pseudokode efter at have studeret programmeringssproget C^*, som er en "parallel" udgave af C. C^* er konstrueret til CM-200. Tankegangen i programmeringssproget C^* og i vor pseudokode er den samme. Thinking Machines Corporation (1991b og 1991c) har beskrevet C^* — C^* er også beskrevet i upublicerede skrifter af Malcolm Brown (1993).

Appendiks B

Test og tidsforbrug

B.1 Kontrol af programmer

For at overbevise os selv om, at de parallelle programmer virkede som de skulle, sammenlignede vi resultaterne fra traditionelle sekventielle programmer og resultaterne fra de parallelle programmer. Vi skrev resultaterne ud på filer, og brugte (officielle) programmer til at sammenligne disse filer (bit for bit!).

Ovenstående måde kræver, at vi er helt sikre på, at de sekventielle programmer fungerer. Vi har derfor implementeret de sekventielle algoritmer: Floyds algoritme, matrixmetoden og Dijkstras algoritme[1]. Disse tre sekventielle algoritmer er blevet implementeret uafhængigt af hinanden og testet grundigt (med "håndkraft").

Matricen S (se afsnit A.1) bliver genereret ved hjælp af en talgenerator. Ved hjælp af denne talgenerator kan vi bestemme hvor tyk eller hvor tynd matricen skal være — en matrix er tynd hvis mange af elementerne er identisk med 0, en matrix er tyk hvis mange af elementerne er forskellig fra 0.

[1] Se Sedgewick (1988) side 457.

Vi har forsøgt med en række forskellige matricer: tynde og tykke, samt forskellige mellemstader. For de forskellige algoritmer — på nær kube matrixmetoden — har vi anvendt matricer af størrelsen 128×128 og af størrelsen 256×256. For kube matrixmetoden har vi anvendt 32×32-matricer og 64×64-matricer.

B.2 Grafernes størrelse n

Antallet af processorer, som algoritmen kræver, bør være et multiplum af 8192. Hvis dette ikke er tilfældet, så kan vi godt "snyde" (se side 22) paralleldatamaten ved at udføre en række overflødige "dummy-operationer", men dette er vi ikke interesseret i.

På nær kube matrixmetoden, så kræver algoritmerne n^2 processorer og 2 eller 3 gange så meget lager. Da n^2 skal være et multiplum af 8192, samt at vi ikke ønsker at overskride lagerkapaciteten på 1 Gbyte (ram), så har vi følgende valg af n: 128, 256, 512, 1024, 2048 og 4096.

Anderledes står det til med kube matrixmetoden. Først og fremmest kræver algoritmen en lagerkapacitet på $O(n^3)$. Med en lagerkapacitet på 1 Gbyte (ram) kan vi ikke anvende så store n, som i de andre algoritmer. Endvidere er der en anden teknisk begrænsning, som siger, at $\log_2(n)$ skal være et helt tal ($n = 160$ er derfor ingen kandidat, trods 160^3 er et multiplum af 8192). Her af får vi følgende valg af n: 32, 64, 128 og 256.

B.3 Tidsdeling på paralleldatamaten

CM-200 er et "fler-bruger system". De viste tider fra kapitel 3 er den samlede tid, som paralleldatamaten har brugt på det pågældende job under tidsdeling.

En forudsætning for at kunne verificere kompleksiteten af vore algoritmer er, at det målte tidsforbrug er uafhængig af, om der er andre brugere på paralleldatamaten.

Vi har fået lov til at foretage eksklusive kørsler, hvor vi er de eneste brugere på paralleldatamaten. I den begrænsede tid, som vi havde til rådighed, blev der udført 9 kørsler. Tidsforbruget for de eksklusive kørsler uden tidsdeling, samt tidsforbruget for kørsler med tidsdeling fremgår af tabel B.1.

	Cannons matrixmetode		(parallel) Floyds algoritme	
n	Med tids-deling	Uden tids-deling	Med tids-deling	Uden tids-deling
128	0.43 s	0.43 s	0.24 s	0.24 s
256	1.63 s	1.62 s	0.71 s	0.71 s
512	11.27 s	11.09 s	3.19 s	3.10 s
1024	69.48 s	69.14 s	18.90 s	18.70 s
2048	482.82 s	478.21 s		

Tabel B.1: Tidsforbrug i sekunder med tidsdeling og uden tidsdeling på paralleldatamaten.

Da der ikke er de helt store forskelle mellem tidsforbruget med og uden tidsdeling, så vil vi betragte resultaterne fra kapitel 3 som værende upåvirket af, at der er andre brugere på paralleldatamaten.

B.4 Tidsforbrug på front-end'en

Som beskrevet i kapitel 1 består CM-200 systemet af en eller flere front-end'er og af en eller flere CM enheder. Dette kan give problemer, når vi skal undersøge vore programmers tidsforbrug. Vi vil i det følgende beskrive, hvorfor vi vælger kun at se på tidsforbruget på CM enheden.

Det er ønskeligt, at så store dele af programmet som muligt afvikles på CM enheden, da vort mål er at lave effektive parallelle algoritmer. Så længe front-end'ens tidsforbrug ikke er markant stort, kan vi se bort fra dette. I tabel B.2 har vi vist tidsforbruget på CM-200 og på front-end'en for kørslerne uden tidsdeling (se afsnit B.3). Vi har også bragt forholdet mellem tidsforbruget på front-end'en og CM-200:

$$f(n) = \frac{\text{tidsforbrug på front-end}}{\text{tidsforbrug på CM-200}}.$$

Som det ses, så er forholdet mellem tidsforbruget på front-end'en og CM-200 en aftagende funktion af n.

Front-end'en på CM-200 på UNI•C er server for flere af datamater på UNI•C. Endvidere deler front-end'en fastdisklager med en anden datamat. Dette gør, at tiderne, der returneres fra front-end'en, ikke nødvendigvis afspejler den tid front-end'en alene har brugt på at løse vort "problem", da den også bruger tid på andre opgaver. Dette gør det umuligt at estimere tidsforbruget på front-end'en.

	Cannons matrixmetode			(parallel) Floyds algoritme		
n	front-end	CM-200	$f(n)$	front-end	CM-200	$f(n)$
128	2.17	0.43	5.047	1.99	0.24	8.292
256	2.86	1.62	1.765	2.36	0.71	3.324
512	9.60	11.09	0.865	4.69	3.10	1.513
1024	21.02	69.14	0.304	20.31	18.70	1.086
2048	61.47	478.21	0.128			

Tabel B.2: Tidsforbrug i sekunder på front-end'en, tidsforbrug på CM-200, samt forholdet: $f(n) = \frac{\text{tidsforbrug på front-end}}{\text{tidsforbrug på CM-200}}$.

Appendiks C

De implementerede algoritmer

I det følgende vil vi angive de implementerede algoritmer. Det drejer sig om de parallelle matrixmetoder fra afsnit 2.1.2 samt den parallelle udgave af Floyds algoritme fra afsnit 2.2.2. Alle algoritmerne er implementeret i C^*.

I alle programmerne har vi anvendt det for C^* specielle biblioteksmodul `cscomm.h`. I dette findes de specielle biblioteksfunktioner, som vi anvender. Vi vil hverken give en beskrivelse af de for C^* specielle biblioteksfunktioner eller af programmeringssproget C^* i øvrigt, men i stedet henvise læseren til C^*-manualerne fra Thinking Machines Corporation (1991b) og (1991c).

Vi arbejder med parallelle variable navngivet med bogstaver først i alfabetet: `a`, `b`, `c` og `d`. I alle programmerne findes der i hovedprogrammet `main` et kald til funktionen `initial`. Denne funktion genererer sekventielt matricen S (se afsnit A.1). Funktionen `initial` kan implementeres på mange måder og er uden betydning for forståelsen af de implementerede algoritmer.

Kube matrixmetoden

```
#include "cscomm.h"
#include "stdlib.h"
#define N 32
#define EPSILON 0.000001
shape [N][N][N]matrix;
float:matrix a, b;

void initial(void) {
  ...
}

float:matrix square(float:matrix a) {
  float:matrix b;

  b = [pcoord(1)][pcoord(2)][pcoord(0)]a;
  a = [pcoord(0)][pcoord(2)][pcoord(1)]
      spread(a*b, 1, CMC_combiner_max);
  return (a);
}


void main() {

  with (matrix) {
    initial();
    do {
      b = a;
      a = square(a);
    } while (|= (a - b > EPSILON));
  }
}
```

Plan matrixmetoden

```
#include "cscomm.h"
#include "stdlib.h"
#define N 128
#define EPSILON 0.000001
shape [N][N]matrix;
float:matrix a, b;

void initial(void) {
  ...
}

float:matrix square(float:matrix a) {
  int i;
  float:matrix b;

    b = 0;
    for(i = 0; i < N; i++)
      b >?= copy_spread(&a, 1, i)*copy_spread(&a, 0, i);
    return (b);
}

void main() {

  with (matrix) {
    initial();
    do {
      b = a;
      a = square(a);
    } while (|= (a - b > EPSILON));
  }
}
```

Cannons matrixmetode

```
#include "cscomm.h"
#include "stdlib.h"
#define N 128
#define EPSILON 0.000001
shape [N][N]matrix;
float:matrix a, b;

void initial(void) {
  ...
}

float:matrix square(float:matrix b) {
  int i, t;
  float:matrix c;

  t = 1;
  i = 0;
  while (t < N) {
    where ((pcoord(0) >> i) % 2)
      a = [pcoord(0)][(pcoord(1) + (1 << i)) %% N]a;
    where ((pcoord(1) >> i) % 2)
      b = [(pcoord(0) + (1 << i)) %% N][pcoord(1)]b;
    t = 2*t;
    i++;
  }
  c = a*b;
  for (i = 0; i < N-1; i++) {
    a = [pcoord(0)][(pcoord(1) + 1) %% N]a;
    b = [(pcoord(0) + 1) %% N][pcoord(1)]b;
    c >?= (a*b);
  }
  return (c);
}

void main() {

  with (matrix) {
    initial();
    do {
      b = a;
      a = square(a);
    } while (|= (a - b > EPSILON));
  }
}
```

Permutation matrixmetoden

```
#include "cscomm.h"
#include "stdlib.h"
#define N 128
#define EPSILON 0.000001
shape [N][N]matrix;
float:matrix a,b;

void initial(void) {
  ...
}

float:matrix square(float:matrix b) {
  int i, t;
  float:matrix c, d;

  b = [pcoord(1)][pcoord(0)]a;
  for (i = 0; i < N; i++) {
    c = a*b;
    reduce(&d, c, 1, CMC_combiner_max, i);
    b = [(pcoord(0) + 1) %% N][pcoord(1)]b;
  }
  i = 0;
  t = 1;
  while (t < N) {
    where ((pcoord(0) >> i) % 2)
      d = [pcoord(0)][(pcoord(1) - (1 << i)) %% N]d;
    t = 2*t;
    i++;
  }
  return (d);
}

void main() {

  with (matrix) {
    initial();
    do {
      b = a;
      a = square(a);
    } while (|= (a - b > EPSILON));
  }
}
```

Den parallelle udgave af Floyds algoritme

```
#include "cscomm.h"
#define N 128
shape [N][N]matrix;
float:matrix a;

void initial(void) {
  ...
}

void main() {
  int i;

  with (matrix) {
    initial();
    for(i = 0; i < N; i++)
      a >?= copy_spread(&a, 1, i)*copy_spread(&a, 0, i);
  }
}
```

Referenceliste

Aki, Selim G.
The design and analysis of parallel algorithms
Printice Hall, New jersey 1989
ISBN 0-13-200056-3

Bjørstad, P.
Effecient matrix multiplication on SIMD computers
SIAM J. Matrix Anal. Appl. vol. 13 no. 1 pp. 586-401 1992

Brown, Malcolm
Upubliceret materiale til parallelprogrammerings kursus på UNI•C
Februar 1993

Carey, Graham F.
Parallel supercomputing: Methods, Algorithms and applications
John Wiley & sons Inc. Texas 1989
ISBN 0-471-92436-9

Draper N. R.
Applied regression analysis
John Wiley & sons Inc. New York 1966
ISBN 0-471-22170-8

Hansen, Per Christian
Upubliceret materiale til parallelprogrammerings kursus på UNI•C
Februar 1993

Larsen, Henrik Legind
An approach to customized end-user views in multi-user information retrieval systems
Roskilde University Centre 1990
ISBN 0109-9779-23

Larsen, Jørgen
Lineær regressionsanalyse
Roskilde Universitetscenter 1993

McBryan, Oliver A.
The Connection Machine
Parallel Computing 9 (1988/89) 1-24 (tidsskrift)

Sedgewick, Robert
Algorithms
Addison-Wesley 1988
ISBN 0-201-06673-4

Tanenbaum, Andrew S.
Structured computer organization, third edition
Printice-Hall international, Inc 1990
ISBN 0-13-852872-1

Thinking Machines Corporation 1991a
CM-200 User's quide, version 6.1
Thinking Machines Corporation, Cambridge, Massachusetts

Thinking Machines Corporation 1991b
C* Programming guide, version 6.0.2
Thinking Machines Corporation, Cambridge, Massachusetts

Thinking Machines Corporation 1991c
Getting started in C*
Thinking Machines Corporation, Cambridge, Massachusetts

Liste over tidligere udkomne tekster tilsendes gerne. Henvendelse herom kan ske til IMFUFA's sekretariat tlf. 46 75 77 11 lokal 2263

217/92 "Two papers on APPLICATIONS AND MODELLING IN THE MATHEMATICS CURRICULUM"
by: Mogens Niss

218/92 "A Three-Square Theorem"
by: Lars Kadison

219/92 "RUPNOK - stationær strømning i elastiske rør"
af: Anja Boisen, Karen Birkelund, Mette Olufsen
Vejleder: Jesper Larsen

220/92 "Automatisk diagnosticering i digitale kredsløb"
af: Bjørn Christensen, Ole Møller Nielsen
Vejleder: Stig Andur Pedersen

221/92 "A BUNDLE VALUED RADON TRANSFORM, WITH APPLICATIONS TO INVARIANT WAVE EQUATIONS"
by: Thomas P. Branson, Gestur Olafsson and Henrik Schlichtkrull

222/92 On the Representations of some Infinite Dimensional Groups and Algebras Related to Quantum Physics
by: Johnny T. Ottesen

223/92 *THE FUNCTIONAL DETERMINANT*
by: Thomas P. Branson

224/92 *UNIVERSAL AC CONDUCTIVITY OF NON-METALLIC SOLIDS AT LOW TEMPERATURES*
by: Jeppe C. Dyre

225/92 *"HATMODELLEN" Impedansspektroskopi i ultrarent en-krystallinsk silicium*
af: Anja Boisen, Anders Gorm Larsen, Jesper Varmer, Johannes K. Nielsen, Kit R. Hansen, Peter Bøggild og Thomas Hougaard
Vejleder: Petr Viscor

226/92 *"METHODS AND MODELS FOR ESTIMATING THE GLOBAL CIRCULATION OF SELECTED EMISSIONS FROM ENERGY CONVERSION"*
by: Bent Sørensen

227/92 "Computersimulering og fysik"
af: Per M.Hansen, Steffen Holm, Peter Maibom, Mads K. Dall Petersen, Pernille Postgaard, Thomas B.Schrøder, Ivar P. Zeck
Vejleder: Peder Voetmann Christiansen

228/92 "Teknologi og historie"
Fire artikler af:
Mogens Niss, Jens Høyrup, Ib Thiersen, Hans Hedal

229/92 "Masser af information uden betydning"
En diskussion af informationsteorien i Tor Nørretranders' "Mærk Verden" og en skitse til et alternativ basseret på andenordens kybernetik og semiotik.
af: Søren Brier

230/92 "Vinklens tredeling - et klassisk problem"
et matematisk projekt af
Karen Birkelund, Bjørn Christensen
Vejleder: Johnny Ottesen

231A/92 "Elektrondiffusion i silicium - en matematisk model"
af: Jesper Voetmann, Karen Birkelund, Mette Olufsen, Ole Møller Nielsen
Vejledere: Johnny Ottesen, H.B.Hansen

231B/92 "Elektrondiffusion i silicium - en matematisk model" Kildetekster
af: Jesper Voetmann, Karen Birkelund, Mette Olufsen, Ole Møller Nielsen
Vejledere: Johnny Ottesen, H.B.Hansen

232/92 "Undersøgelse om den simultane opdagelse af energiens bevarelse og især om de af Mayer, Colding, Joule og Helmholtz udførte arbejder"
af: L.Arleth, G.I.Dybkjær, M.T.Østergård
Vejleder: Dorthe Posselt

233/92 "The effect of age-dependent host mortality on the dynamics of an endemic disease and
Instability in an SIR-model with age-dependent susceptibility
by: Viggo Andreasen

234/92 "THE FUNCTIONAL DETERMINANT OF A FOUR-DIMENSIONAL BOUNDARY VALUE PROBLEM"
by: Thomas P. Branson and Peter B. Gilkey

235/92 OVERFLADESTRUKTUR OG POREUDVIKLING AF KOKS
- Modul 3 fysik projekt -
af: Thomas Jessen

236a/93 INTRODUKTION TIL KVANTE HALL EFFEKTEN

af: Anja Boisen, Peter Bøggild

Vejleder: Peder Voetmann Christiansen
Erland Brun Hansen

236b/93 STRØMSSAMMENBRUD AF KVANTE HALL EFFEKTEN

af: Anja Boisen, Peter Bøggild

Vejleder: Peder Voetmann Christiansen
Erland Brun Hansen

237/93 The Wedderburn principal theorem and Shukla cohomology

af: Lars Kadison

238/93 SEMIOTIK OG SYSTEMEGENSKABER (2)

Vektorbånd og tensorer

af: Peder Voetmann Christiansen

239/93 Valgsystemer - Modelbygning og analyse

Matematik 2. modul

af: Charlotte Gjerrild, Jane Hansen, Maria Hermannsson, Allan Jørgensen, Ragna Clauson-Kaas, Poul Lützen

Vejleder: Mogens Niss

240/93 Patologiske eksempler.
Om sære matematiske fisks betydning for den matematiske udvikling

af: Claus Dræby, Jørn Skov Hansen, Runa Ulsøe Johansen, Peter Meibom, Johannes Kristoffer Nielsen

Vejleder: Mogens Niss

241/93 FOTOVOLTAISK STATUSNOTAT 1

af: Bent Sørensen

242/93 Brovedligeholdelse - bevar mig vel

Analyse af Vejdirektoratets model for optimering af broreparationer

af: Linda Kyndlev, Kare Fundal, Kamma Tulinius, Ivar Zeck

Vejleder: Jesper Larsen

243/93 TANKEEKSPERIMENTER I FYSIKKEN

Et 1.modul fysikprojekt

af: Karen Birkelund, Stine Sofia Korremann

Vejleder: Dorthe Posselt

244/93 RADONTRANSFORMATIONEN og dens anvendelse i CT-scanning

Projektrapport

af: Trine Andreasen, Tine Guldager Christiansen, Nina Skov Hansen og Christine Iversen

Vejledere: Gestur Olafsson og Jesper Larsen

245a+b /93 Time-Of-Flight målinger på krystallinske halvledere
Specialerapport

af: Linda Szkotak Jensen og Lise Odgaard Gade

Vejledere: Petr Viscor og Niels Boye Olsen

246/93 HVERDAGSVIDEN OG MATEMATIK
- LÆREPROCESSER I SKOLEN

af: Lena Lindenskov, Statens Humanistiske Forskningsråd, RUC, IMFUFA

247/93 UNIVERSAL LOW TEMPERATURE AC CONDUCTIVITY OF MACROSCOPICALLY DISORDERED NON-METALS

by: Jeppe C. Dyre

248/93 DIRAC OPERATORS AND MANIFOLDS WITH BOUNDARY

by: B. Booss-Bavnbek, K.P.Wojciechowski

249/93 Perspectives on Teichmüller and the Jahresbericht Addendum to Schappacher, Scholz, et al.

by: B. Booss-Bavnbek

With comments by W.Abikoff, L.Ahlfors, J.Cerf, P.J.Davis, W.Fuchs, F.P.Gardiner, J.Jost, J.-P.Kahane, R.Lohan, L.Lorch, J.Radkau and T.Söderqvist

250/93 EULER OG BOLZANO - MATEMATISK ANALYSE SET I ET VIDENSKABSTEORETISK PERSPEKTIV

Projektrapport af: Anja Juul, Lone Michelsen, Tomas Højgård Jensen

Vejleder: Stig Andur Pedersen

251/93 Genotypic Proportions in Hybrid Zones

by: Freddy Bugge Christiansen, Viggo Andreasen and Ebbe Thue Poulsen

252/93 MODELLERING AF TILFÆLDIGE FÆNOMENER

Projektrapport af: Birthe Friis, Lisbeth Helmgaard Kristina Charlotte Jakobsen, Marina Mosbæk Johannessen, Lotte Ludvigsen, Mette Hass Nielsen

253/93 Kuglepakning

Teori og model

af: Lise Arleth, Kåre Fundal, Nils Kruse

Vejleder: Mogens Niss

254/93 Regressionsanalyse

Materiale til et statistikkursus

af: Jørgen Larsen

255/93 TID & BETINGET UAFHÆNGIGHED

af: Peter Harremoës

256/93 Determination of the Frequency Dependent Bulk Modulus of Liquids Using a Piezoelectric Spherical Shell (Preprint)

by: T. Christensen and N.B.Olsen

257/93 Modellering af dispersion i piezoelektriske keramikker

af: Pernille Postgaard, Jannik Rasmussen, Christina Specht, Mikko Østergård

Vejleder: Tage Christensen

258/93 Supplerende kursusmateriale til "Lineære strukturer fra algebra og analyse"

af: Mogens Brun Heefelt

259/93 STUDIES OF AC HOPPING CONDUCTION AT LOW TEMPERATURES

by: Jeppe C. Dyre

260/93 PARTITIONED MANIFOLDS AND INVARIANTS IN DIMENSIONS 2, 3, AND 4

by: B. Booss-Bavnbek, K.P.Wojciechowski

261/93 OPGAVESAMLING
Bredde-kursus i Fysik
Eksamensopgaver fra 1976-93

262/93 Separability and the Jones Polynomial
by: Lars Kadison

263/93 Supplerende kursusmateriale til "Lineære strukturer fra algebra og analyse" II
af: Mogens Brun Heefelt

264/93 FOTOVOLTAISK STATUSNOTAT 2
af: Bent Sørensen

265/94 SPHERICAL FUNCTIONS ON ORDERED SYMMETRIC SPACES
To Sigurður Helgason on his sixtyfifth birthday
by: Jacques Faraut, Joachim Hilgert and Gestur Olafsson

266/94 Kommensurabilitets-oscillationer i laterale supergitre
Fysikspeciale af: Anja Boisen, Peter Bøggild, Karen Birkelund
Vejledere: Rafael Taboryski, Poul Erik Lindelof, Peder Voetmann Christiansen

267/94 Kom til kort med matematik på Eksperimentarium - Et forslag til en opstilling
af: Charlotte Gjerrild, Jane Hansen
Vejleder: Bernhelm Booss-Bavnbek

268/94 Life is like a sewer ...
Et projekt om modellering af aorta via en model for strømning i kloakrør
af: Anders Marcussen, Anne C. Nilsson, Lone Michelsen, Per M. Hansen
Vejleder: Jesper Larsen

269/94 Dimensionsanalyse en introduktion
metaprojekt, fysik
af: Tine Guldager Christiansen, Ken Andersen, Nikolaj Hermann, Jannik Rasmussen
Vejleder: Jens Højgaard Jensen

270/94 THE IMAGE OF THE ENVELOPING ALGEBRA AND IRREDUCIBILITY OF INDUCED REPRESENTATIONS OF EXPONENTIAL LIE GROUPS
by: Jacob Jacobsen

271/94 Matematikken i Fysikken.
Opdaget eller opfundet
NAT-BAS-projekt
vejleder: Jens Højgaard Jensen

272/94 Tradition og fornyelse
Det praktiske elevarbejde i gymnasiets fysikundervisning, 1907-1988
af: Kristian Hoppe og Jeppe Guldager
Vejledning: Karin Beyer og Nils Hybel

273/94 Model for kort- og mellemdistanceløb
Verifikation af model
af: Lise Fabricius Christensen, Helle Pilemann, Bettina Sørensen
Vejleder: Mette Olufsen

274/94 MODEL 10 - en matematisk model af intravenøse anæstetikas farmakokinetik
3. modul matematik, forår 1994
af: Trine Andreasen, Bjørn Christensen, Christine Green, Anja Skjoldborg Hansen. Lisbeth Helmgaard
Vejledere: Viggo Andreasen & Jesper Larsen

275/94 Perspectives on Teichmüller and the Jahresbericht
2nd Edition
by: Bernhelm Booss-Bavnbek

276/94 Dispersionsmodellering
Projektrapport 1. modul
af: Gitte Andersen, Rehannah Borup, Lisbeth Friis, Per Gregersen, Kristina Vejrø
Vejleder: Bernhelm Booss-Bavnbek

277/94 PROJEKTARBEJDSPÆDAGOGIK - Om tre tolkninger af problemorienteret projektarbejde
af: Claus Flensted Behrens, Frederik Voetmann Christiansen, Jørn Skov Hansen, Thomas Thingstrup
Vejleder: Jens Højgaard Jensen

278/94 The Models Underlying the Anaesthesia Simulator Sophus
by: Mette Olufsen(Math-Tech), Finn Nielsen (RISØ National Laboratory), Per Føge Jensen (Herlev University Hospital), Stig Andur Pedersen (Roskilde University)

279/94 Description of a method of measuring the shear modulus of supercooled liquids and a comparison of their thermal and mechanical response functions.
af: Tage Christensen

280/94 A Course in Projective Geometry
by Lars Kadison and Matthias T. Kromann

281/94 Modellering af Det Cardiovaskulære System med Neural Pulskontrol
Projektrapport udarbejdet af:
Stefan Frello, Runa Ulsøe Johansen, Michael Poul Curt Hansen, Klaus Dahl Jensen
Vejleder: Viggo Andreasen