Indholdsfortegnelse

1 Resumé

Dette er et projekt på andendelen, udarbejdet i perioden 25/4 til 21/6 – 2008 ved Datalogisk Insititut, Københavns Universitet. Formålet med dette projekt har været at få indsigt i hvordan "Communicating sequential processes"-modellen (CSP) kan bruges til at implementere algoritmer, der kan drage fordel af flere processorer samtidigt. Jeg forventer, at læseren er bekendt med trådet parallelprogrammering.

Jeg har med udgangspunkt i CSP analyseret og implementeret en algoritme til simulation af myrer og deres evne til autonomt at finde den korteste rute mellem myretuen og det mad, der måtte findes i området omkring den. Desuden har jeg implementeret en algoritme til proteinsammenligning "Smith–Waterman algoritmen". Begge algoritmer er meget CPU-intensive og vil derfor kunne drage nytte af flere processorer.

Jeg har testet de to algoritemer på to forskellige computere med hhv. en og to processorer. Jeg har testet, hvordan køretiden ændres, samt hvordan arbejdet fordeler sig på de forskellige processer, når antallet af processer varieres. Desuden har jeg testet, hvor god min implementation af myrer er til at danne den korteste vej til maden i området.

Testene viser, at for Smith–Waterman algoritmen kan man ved brug af flere processer samtidigt opnå en forbedring på 22% på en computer med blot en processor, og en 24% forbedring for en computer med to processorer. For myresimuleringen opnås der ikke en ydelsesforbedring på computere med en processor, mens der for en computer med 2 processorer opnås en 13% forbedring. Testene viser, at man simpelt kan opnå ydelseforbedringer ved at bruge CSP til parallelprogrammering, men at man bør have fokus på overhead og kommunikationsomkostningerne for ikke at miste fordelene.

2 Introduktion

Markedet for processorer er inden i en eksplosiv omvæltning. Tidligere har der udelukkende været fokuseret på hastighed, hvilket har medført stigende temperaturer for processorerne og deraf øget behov for at lede varmen væk. Konsekvensen af dette har været stærkt stigende omkostninger til elektricitet for en minimal forøgelse af hastigheden. Fokus er derfor skiftet til mere energieffektive metoder hvoraf multiprocessorer tegner til at blive fremtiden.

Traditionel programmering er sekventiel i en tråd og kan derfor ikke drager nytte af flere processorer samtidigt. Den eksisterende løsning har været at inkludere flere tråde i programmeringen. Dette har ikke været problemfrit og flertrådet programmering resulterer let i "deadlocks", "livelocks" og "race-condtions". Disse problemer er ikke trivielle at finde og løse og er medvirkende til at flere helt fraråder brug af flere tråde hvor det er muligt (Muller og Walrath 1998).

Inden for programmering med flere tråde findes der to termer. Parallelitet og samtidighed. Parallelitet defineres som to eller flere operationer, der udføres på samme tid . Dette er den mest stringente definition af de to. Algoritmer der er designet parallelt, udnytter hardwaren fuldt ud, og er som sådan garanteret maksimal effektivitet. For at designe parallelle algoritmer skal hardwaren derfor kendes på forhånd og algoritmen designes derefter til netop det system.

Samtidighed defineres som to eller flere operationer, der alle er udført på et målbart sluttidspunkt. Dette betyder ikke, at alle operationer skal udføres på samme tid, men at operationer enten kører samtidigt eller at man ikke kan bestemme i hvilken rækkefølge operationerne bliver udført. Samtidige algoritmer kan udnytte et parallelt system, og har muligheden for at kører parallelt, men er ikke begrænset til konstant at skulle køre parallelt.

De fleste opgaver, der findes i virkerligheden, har en naturlig samtidighed, som eksempelvis det at styre en robot. Denne skal tage billeder og analysere dem, bevæge sig og kommunikere med andre robotter. Alle disse opgaver skal løses samtidigt og kontinueligt. Traditionelt er denne samtidighed ikke udnyttet og er i stedet løst sekventiel, ved at alle opgaverne er samlet i en løkke. Først analyseres et billede, derefter bevæger den sig en distance, for så at kommunikere. Hver af disse opgaver kunne istedet med succes afvikles samtidigt således at robotten samtidigt tager billeder, bevæger sig og kommunikerer. Men med forøgelsen af samtidige opgaver opstår som sagt en risiko for problemer med håndtering af data, der overordnet set kan problemerne inddeles i følgende kategorier: Deadlocks, livelocks, og race-conditions.

I et multitrådet system kan der opstå race-conditions hvis flere processer læser det samme data og minimum en proces skriver til data. Tilstanden af data afhænger således af timing i systemet og hvornår processerne læser og skriver til data. Læser proces 1 data, hvorefter en proces 2 skriver til data, har proces 1 et forkert billede af data. For at undgå sådanne race-conditions bruges traditionelt en mutex til at sikre, at data kun holdes af en proces på samme tid. En sideeffekt ved brugen af mutex er, at der introduceres mulighed for deadlocks, og at processerne aldrig får eneadgang til data.

Definitionen af en deadlock, er når to eller flere processer venter på, at hinanden skal skal slutte, før de selv slutter. Dette sker, når hver proces i deadlock holder en ressource og ønsker en eller flere ressourcer, som holdes af andre processer. E.G Coffman beskrev i 1971 (Coffman, Elphick og Shoshani 1971) fire betingelser, der alle skal være overholdt, for at der kan opstå en deadlock.

1. Mutual exclusion: Kun en proces kan bruge en ressource.
2. Hold and wait: En proces kan holde en ressource og bede om endnu en ressource.
3. No preemption: Det er kun processen, der holder ressourcen, der kan frigive ressoursen.
4. Circular wait: To eller flere processer kan danne en ring, hvor processerne har en ressource og venter på den ressource, som den efterfølgende proces har.

Til forskel fra deadlocks, hvor systemet stopper fuldstændigt, vil et system under livelock umiddelbart virke aktivt og det vil ved inspektion ses, at processerne over tid venter på forskellige ressourcer. Under en livelock bliver der dog ikke foretaget reelt arbejde og systemet vil aldrig terminerer. Livelocks opstår som regel på baggrund af deadlocks, hvor systemet genkender en deadlock og frigiver alle ressourcer, for efterfølgende at optage nye ressourcer, der giver en ny deadlock.

3 CSP

"Communicating sequential processes" CSP er en programmeringsmodel udviklet af C.A.R. Hoarce i 1978 (Hoare 1978) til beskrivelse af systemer, der kører samtidigt og som gør det nemmere at omgå de tre overnævnte problemer. Ideen bag CSP er modulært, at indkapsle programmet i mindre processer, der fungerer uafhængig af hinanden. I en proces er alt data og kode privat og kan dermed hverken ses eller ændres af andre processer. Processerne kan udelukkende kommunikere gennem eksplicit definerede kanaler, hvor der sendes beskeder. CSP blev oprindeligt implementeret i occam (May 1983), og siden er ideerne blevet udgivet som biblioteker i andre sprog som JCSP, PyCSP og C++CSP. Hvert bibliotek er implementeret forskelligt afhængigt af det underliggende sprog. Hvert bibliotek har sine ulemper og fordele som JSCP, der implementeret processer som objekter, hvilket kan give problemer, når man referer til objektet og ikke værdien. PyCSP er under kraftig udvikling og er dermed ikke så modent som de andre biblioteker, f.eks kan det kun emulere samtidighed ved brug af en tråd. C++CSP2 (Brown 2007) bygger som navnet antyder på C++ og bruger POSIX standarden til at håndtere flere tråde samtidigt. Det er næsten, lige så hurtig som Occam, og meget hurtigere end JCSP (Brown og Welch 2003, s. 151-152). Fordi hastigheden ligger så tæt på Occam og samtidigt er et højniveau sprog har jeg har valgt at bruge C++CSP i min implementering, hvorfor jeg vil fokusere på hvordan C++CSP har implementeret CSPmodellen.

Hver proces kan starte et vilkårligt antal underprocesser, og det kan frit vælges hvordan disse skal køres. Hvilke der skal køres samtidigt, hvilke der skal køres sekventiel, samt i hvilken rækkefølge processerne skal køre og først når alle underprocesserne er afsluttet, slutter processen selv. Hver proces bliver som standard oprettet som en sepereat kernetråd, men C++CSP giver også mulighed for at specificere at processem skal oprettes som en brugertråd.

Den basale kanal i CSP er en direkte kanal mellem to processer uden buffer. Enten: Først oprettes en kanal mellem to processer og så længe begge processer er i live, kan kanalen bruges. For at de to processer kan kommunikere, skal de begge være klar. På et hvert tidspunkt kan senderen og modtageren indikere, at de ønsker at kommunikere og starte overførslen. Processen er så blokeret, indtil overførslen er sluttet. Kommunikationen foregår dermed kun, når begge parter har indikeret, at de er klar til at foretage en overførsel af en besked.

De to processer, der kommunikere, har ikke kendskab til hinanden og til selve kanalen. De interagere med endepunkterne af kanalen, der kan være en af to typer. Enten kan det sende eller modtage og hver proces kender typen af endepunkt den har, og ved derfor om den kan forvente at modtage data eller kan sende data igennem sit kanalendepunkt.

Endepunkterne kan udvides, så flere processer kan skrive og modtage beskeder på den samme kanal. I en Any2One kan der være flere sende-endepunkter, mens der kun findes et modtage-endepunkt. Modsat gælder One2Anykanalen, hvor der der findes et sende-endepunkt, og flere modtage-endepunkter. Det er her vigtigt at bemærke, at der kun er en processer, der modtager den enkelte besked, men at det ikke nødvendigvis er den samme proces, der modtager den næste besked i kanalen. Processerne ser ikke samme data, og dermed er der ikke mulighed for race-condition (se også 3). Endeligt findes også Any2Anykanalen - der som navnet antyder - kan have flere sende-endepunkter og flere modtage-endepunkter.

For at forbedrer ydelsen af kanaler, kan de udvides med en buffer. I C++CSP findes der 3 typer buffere; en fast buffer, der kan indeholde et endeligt antal elementer; en buffer med uendeligt plads; samt en overskrivende buffer, der overskriver den ældste data hvis bufferen er fyldt. Ønsker en proces at sende data og der findes en fast buffer og denne ikke er fyldt, gøres dette umiddelbart og processen kan fortsætte. Er kanalen derimod fyldt, venter processen, indtil der igen er plads i bufferen; svarende til at modtagerkanalen melder klar. En proces, der ønsker at modtage data, kan med det samme modtage data fra bufferen og afventer ellers indtil der kommer data i bufferen. Fra processernes synspunkt sker der ikke en ændring i funktionalitet set ift.til den originale kanal uden buffer.

For at have muligheden for på tværs af processer at stoppe processer, introducerede Occam: "Poisoning" (Welch 1989), som også er implementeret i C++CSP. Problemet opstår, når en proces ønsker at kommunikere. Her er den låst, indtil kommunikationen er færdig. Er der ingen kanal til at kommunikere med, da den/de er stoppet, opstår en deadlock. For at garantere at der altid er processer i hver ende af en kanal introduceres poisoning. Når en proces ønsker at stoppe, forgifter den de kanaler den har endepunkter til og således kan de ikke bruges igen. Når en anden proces ønsker at sende eller modtage data på en af de kanaler, der er, forgiftet, modtager den i stedet forgiftningen og kan sørge for at stoppe korrekt i stedet havne i deadlock. Det er værd at bemærke, at både sender og modtager kan forgifte en kanal. Det er altså muligt for modtageren at forgifte kanalen mod "strømmen".

Ved at hver proces kun har adgang til sine interne variabler og denne proces er den eneste der har adgang til disse variabler har processen altid alene adgang til alle sine ressourcer, og race-conditions kan ikke eksistere.

Derimod er det ikke garanteret, at man kan undgå deadlocks og livelocks med CSP. Den modulære metode giver i stedet mulighed for simpelt at resonerer sig frem til, om en proces indeholder mulighed for deadlocks og livelocks. Ved at opbygge netværk, bestående af flere processer, man ved ikke indeholder deadlocks, kan man ligeledes let resonere sig frem til om netværket indeholder deadlocks og livelocks. Man kan nu bygge, komplekse netværker af de delnetværk, der alle er uden deadlocks og på denne måde opbygge et komplekst program uden mulighed for deadlocks og livelocks.

Når en algoritme skal implementeres parallelt er den traditionelt skrevet til en bestemt grad af samtidighed. Hele algoritmen er meget stramt paralleliseret, hvor antallet af tråde, der kører samtidigt, ligger helt fast og ikke siden kan ændres. Algoritmen er dermed lavet til en bestemt maskine, der har et givent antal processorer til rådighed. Hvis maskinen senere bliver opgraderet eller algoritmen bliver kørt på en anden og større maskine, vil den ikke kunne bruge det større antal processorer. En åben parallelisering er derimod

ikke implementeret til en bestemt maskine og et præcist antal processer. Den skal derimod kunne køres på både en lille maskine og en stor server, og begge steder udnytte de ressourcer der er til rådighed.

4 Implementering

Jeg ønsker at implementere to algoritmer, der ved hjælp af CSP er uden deadlocks, livelocks og race-conditions, og skal garantere en åben parallelisering. Den første er Smith-Watermans algoritme fra 1981 (Smith og Waterman 1981), der er en bioalogoritme, der sammenligner to proteinstrenge og som resultat giver en delstreng der har det bedste overlap. Den anden algoritme er en simpel myresimulering (Beckers, Deneubourg og Goss 1992; Dorigo 1992), der skal simulere myrer i en myretue, og hvordan de finder føde.

Myresimuleringen er kendetegnet ved at være modulært opbygget og derfor forventer jeg, at den egner sig godt til kunne beregnes samtidigt. Modsat findes der i Smith-Waterman algoritmen mange afhængigheder som gør det sværere at have mange samtidige beregninger og der stilles derfor større krav til designet af denne algoritme, for at opnå en acceptabel samtidighed. Opgaverne er inspirerede af eksamensopgaven i kurset ekstrem multiprogrammering, der blev afholdt i 2008 på Datalogisk Institut på Københavns Universitet.

4.1 Myresimulering

Myresimulering er interessant fra et akademisk synspunkt, da det har vist sig at kunne bruges som en heuristik til at finde den korteste rute mellem to punkter, selv når omgivelserne ændres over tid. Desuden er myre simulering et yderst uafhængigt problem hvorfor det i teorien nemt kan udvikles programmel der effektivt udnytter flere processorer på samme tid.

Myresimulering arbejder med tre simple regler for myrerne, som hver myre skal overholde for at skaffe mad til myretuen.

- En myre bevæger sig tilfældigt rundt, men når feromonniveauet stiger, skal sandsynligheden for, at myren bevæger sig den vej stige, og når den bevæger sig lægger den et feromonspor,
- Hvis en myre finder mad, skal den samle det op og vende sig 180 grader.
- Hvis en myre finder myretuenm skal den lægge maden, den måtte have, og vende sig 180 grader.

Over tid bør disse lokale regler for myrerne give en rute mellem myretuen og føden. Afstanden mellem myretuen og føden kan være vilkårlig lang, og det er ikke påkrævet at der findes en lige rute mellem punkterne. Tesen i kunstig inteligens for fremspirende opførsel (emergent behavior) er at mange simple individer tilsammen kan skabe en ny kompleks opførsel (Reynolds 1987). Derfor vælger jeg at myrerne skal være så simple som muligt og at de på ingen måde skal vide, hvor myretuen eller maden er i forvejen, så de kun ved tilfældigt at gå rundt opdager en rute og danner et feromonspor mellem tuen og maden.

Jeg antager at flere myrer kan befinde sig det samme sted på samme tidspunkt. Dette ville svare til, at myrerne kravler over hinanden for at komme videre. Desuden øger det

myrenes ufafhængighed, da alle myrerne kan bevæge sig rundt uafhængigt af, hvor de andre myrer fysisk befinder sig.

Myrernes placering og bevægelse kan ses om en vektor, hvor myrerne har fuldstændig frihed til at vælge retning og fart. Alternativt kan deres bevægelser og placering diskretiseres, så bevægelserne for myren begrænses til et af nabofelterne. Jeg har valgt den diskrete løsning med firkantede felter, dels fordi beregningerne hurtigere kan foretages, når bevægelserne begrænses til en endelig mængde og dels fordi en diskret baseret verden nemt vil kunne opsplittes. Myrernes verden består af det felt, den står på og de otte omgivende felter. Dette anser jeg som det minimale en myre kan opfatte.

For at skabe fremdrift har jeg har valgt at give myrerne etfeltshukommelse om hvilken felt, de kom fra i skridtet før. Myren ved derfor i hvilket retning den bevæger sig. Myren kan vælge mellem de tre felter der befinder sig foran den, samt de to felter til hver af dens sider. Den kan dermed ikke bevæge sig tilbage i samme retning som den kom fra, med undtagelse af når den finder mad eller myretuen og derfor bliver vendt om jf. reglerne for myrerne. Dette er gjort for at hindrer myren i konstant at følge sit eget feromonspor. Det er dog ikke nok til at hindrer, at myrerne skaber en løkke og derfor skaber et selvforstærkende spor. Har den to gange i træk valgt at dreje til den samme side, har den vendt sig om, og vil kunne detektere sit eget feromonspor. Dette vil jeg ikke begrænse, da det harmonere med filosofien om at have simple myrer, der kan foretage dumme/simple valg. For at sikrer myrerne ikke går rundt i en evig løkke har myrerne en alder som måles i antal skridt de har foretaget, og når denne nås dør myrene.

Alle myrerne starter på samme tid, og derfor til en start have samme alder. Har de den samme maksimale alder, vil det resultere i at de forsvinder på samme tid når de har gået deres antal skridt. For at skabe et kontinuerligt antal myrer i hele simulationsområdet har jeg valgt et bredt interval hvor myrens maksimale alder ligger mellem. For desuden at belønne de myrer der arbejder, nulstilles myrens alder, hver gang den kommer til maden eller myretuen.

For at holde myrerne inden for simulationsområdet skal myrernes bevægelser begrænses når de prøver at bevæge sig uden for det specificerede område. Enten kan man antage, at der findes en høj væg omkring området og myren derfor ikke kan komme længere end til kanten, ellers kan man opstille et "elektrisk" hegn omkring området og alle myrer der rører området dør. Jeg har valgt at lade myrerne dø, når de rammer kanten, da jeg mener at den er mest realtisk, at de forsvinder, når de bevæger sig for langt væk. Dette svarer til at myren har bevæget sig så langt væk fra myretuen at den ikke kan finde tilbage igen, og derfor ikke efterfølgende påvirker de andre myrer. For at have et konstant antal myrer i simulation, vil døde myrer umiddelbart genopstå i myretuen. Ved at flytte myrerne tilbage til myretuen forstærkes sporet omkring myretuen, så andre myrer nemmere kan finde tilbage til tuen.

Simulationen består af to dele. Dels et simulationsområde, der agere de fysiske rammer med myretue og mad, og dels myrerne. For at kunne øget antallet af processer, skal problemet opsplittes, og dette kan man tilgå fra to sider. Enten kan man vælge at dele antallet af myrer, så hver proces står for det samme antal myre. Alternativt kan hver proces have ansvaret for et område af simuleringen og de myrer, der befinder sig i det område.

Ved at opdele myrerne i grupper og lade hver proces stå for en gruppe myrer, opnås en ligelig fordeling af arbejdsbyrden, fordi hver proces vil have lige mange myrer og dermed arbejde lige lang tid i hvert trin. Dette giver teoretisk den bedste arbejdsfordeling. Ulempen er, at fordi myrerne skal kende deres omgivelse, skal dette være den samme for alle myrer. Antag at to forskellige processer styrer to myrer, der befinder sig i samme felt. De to processer skal have kendskab til den samme del af simulationen, svarende til at de skal dele

information. Desuden påvirker begge processer simulationen, idet begge deres myrer afsætter et feromonspor. En dekopling mellem myrernes bevægelser og afgivelsen af feromonspor er derfor ønskelig. En sådan kan laves i en producent/arbejder-model. Producenten kender hele simuleringsområdet og har et globalt billede af feromonsporerne. Arbejdsprocesserne kan bede producenten om de delområder, hvor den har myrer, og myrernes bevægelser sendes tilbage til produceren. Når alle arbejderprocesserne har sendt deres myrers handling, opdateres kortet af producenten og processen gentages. Desværre resulterer dette efter min mening i, at en stor del af arbejdet foregår centralt i producenten og dermed kun kan varetages af en proces.

For at imødekomme problemet ved et globalt feromonkort som varetages af en proces, har jeg valgt en anden metode, der opdeler simuleringsområdet. Man er ikke garanteret, at hver proces har en ligelig fordeling af myrer, og dermed heller ikke en ligelig arbejdsfordeling mellem processerne. Hver proces har ansvaret for et delområde og står for både at opdatere feromonkortet og at bevæge de myrer, der befinder sig i dens ansvarsområde. I grænseområdet mellem to processer er begge processer afhængig af hinanden. Det skyldes, at en myre, der befinder sig på kanten af et delområde, skal bruge feromonniveauet fra de felter ved siden af den selv, som bliver administreret af naboprocessen. Når en myre bevæger sig uden for delområdet, skal processerne desuden flytte myren mellem sig, således at den anden proces får ansvaret for myren.

processerne udveksler derfor deres kantområde med deres naboer, så hver proces er sikret den kender hele det område som dens myrer kender. Nu kan procsserne uafhængigt flytte alle myrerne og opdatere feromonkortet. De myrer, der vælger at bevæge sig ind i et andet område, kan samlet sendes til naboprocessen når de har bevæget sig. Kommunikationen mellem processerne foregår samtidigt, hvorfor kommunikationstiden ikke øges, når antallet af processer øges. For at kunne kommunikere samtidigt og for at sørge for, at myrerne baserer deres valg af retning på det nyeste feromonniveau i grænseområderne, er det nødvendigt at have en synkronisering mellem processerne, således at alle myrerne kun bevæger sig et skridt mellem kommunikationsfaserne.

Der er flest myrer tæt på myretuen og derfor mest arbejde i området tættest på myretuen. Omvendt vil der sjældent være mange myrer i områderne langt fra myretuen. Der vil derfor være stor forskel på hvor meget arbejde, der er i de forskellige processer. Er der lige mange processer og processorer, vil de processorer, med mindst arbejde vente på de processorer, der har meget arbjede. Dette er ikke ønskværdigt, da en optimal udnyttelse af alle ressourcer kræver, at alle processorer arbejder lige meget, hvilket svarer til, at hver processor bevæger det samme antal myrer. Ved at øget antallet af processer gøres områderne mindre og hver processor skal dermed håndtere flere processer Arbejdet for de enkelte processer er stadigt meget forskelligt, men gruppen af processer hver proccesor skal håndtere, kan sammensættes så de, indeholder ca. lige meget arbejde.

Jeg har valgt at implementere koden som en producent/arbejder-model, hvor den enlige producer står for at lave den initielle matrice over simuleringsområdet samt at oprette alle myrerne, mens arbejdsprocesserne står for at bevæge myrerne og opdatere feromonkortet. En producent/arbejder-model egner sig godt til at sikre en åben parallelisering, da antallet af arbejdere kan justeres efter hvor meget arbejde, der skal foregå samtidigt. En producents kerneopgave er udelukkende at klagøre området ved at opsplitte det i delområder med de dertilhørende myrer og uddele disse til arbejderne. Når alle delområder er uddelt, har producenten ikke længere nogen funktion. Arbejderene melder til producenten, når de er klar til at modtage et delområdet.

Det er som sådan ikke nødvendigt at samle alle resultaterne for delområderne igen, hvis

man blot ønsker at kende mængden af mad der er blevet transporteret til myretuen, men for at kunne vise et print af verdenen, er det nødvendigt at samle al information et sted og derfor sender arbejdsprocesserne periodisk data videre. Til implementering af myresimuleringen har jeg valgt at lade producenten være modtageren af denne data. Således sender arbejderne løbende delområdesresultat tilbage til producenten, som sætter det sammen og viser resultatet på skærmen.

Som beskrevet i afsnit 3 på side 4 er kanaler retningsorienterede, og en proces kan enten modtage eller sende data igennem sit kanal-endepunkt. Derfor kræver det to kanaler, for at to processer kan kommunikere med hinanden Jeg har valgt at implementere en løsning hvor området kun opdeles i en dimension. Derved er der maksimalt to naboprocesser mod otte naboprocesser ved en opdeling i to dimensioner. Derved minimere jeg, antallet af kanaler der skal oprettes, samt processernes indbyrdes afhængighed. Hastighedsforøgelse, ved at opdele området i to dimensioner, vil også øger implementeringskompleksiteten kraftigt, uden at der designmæssigt er forskel på de to løsninger. Hver arbejder har derfor i alt syv kanal-endepunkter.

- To kanal-endepunkter til producenten.
- Fire kanal-endepunkter til naboprocesserne. To til hver.
- Et kanal-endepunkt til processen der har myretuen.

Hver arbejder har tre dele. En område-kommunikationsdel, en bevægelsesdel samt en myrerudvekslingsdel. Arbejderen starter med at udveksle information, om hvordan området ser ud langs kanten af sit eget område. Dermed kan den flytte myrerne rundt langs kanten af sit eget område uden yderligere informationer. Hver arbejder flytter alle myrerne et felt, der baseres på feromonniveauet i de fem omkring den. De fem felters feromonniveau akkumuleres og der vælges et tilfældigt tal (*target*) mellem 0 og det akkumulerede tal. Felternes feromonniveau akkumuleres fra venstre mod højre, indtil det er større end *target*. Dette felt udvælges som det, myren bevæger sig hen på. Når myrerne har bevæget sig, opsamles de myrer, der ikke længere skal kontrolleres, af processen. Myrene, der skal videresendes, kan inddeles i tre grupper. To grupper bruges til de myrer der har bevæget sig over i et naboområdet og en gruppe består af de myrer der er døde, enten fordi de har ramt kanten af simulationsområdet, eller fordi de er blevet for gamle. Gruppen af døde myrer sendes til procssen med myretuen, hvor de straks kan genopstå. De tre grupper sendes samlet af sted, og ligeledes modtager man myrer fra naboprocesserne samlet. Tilslut tilføjes feromonstoffet, til de felter som myrerne befinder sig i og arbejderen gentager processen.

4.2 Smith-Waterman sequence alignment

Smith-Waterman sequence alignment (SW) er en algoritmen opfundet af Temple Smith og Michael Waterman i 1981 (Smith og Waterman 1981) til sammenligning af proteiner og neuklitiders opbygning. Der foretages en lokal sammenligning af en sekvens, men modsat tidligere algoritmer, der sammenligner to komplette strenge i sin helhed, foretager SW en sammenligning af samtlige kombinationer af delstrenge i de to proteinstrenge. Resultatet af SW er de to delstrenge, der har den største lighed med hinanden, hvor delstrengene kan have et eller flere mellemrum.

Kort fortalt beregnes der først en kostmatrice, hvor de to proteiner er hhv. længden og bredden af matricen. Hver felt repræsentere dermed et index til begge proteiner, og

værdien i hver felt repræsenterer det højeste resultat fremmkommet, ved at sammenligne samtlige delstrenge af proteinerne op til det felts position i matricen. Når matricen er beregnet, kan man finde feltet med det højeste resultat, og fra den finde de to delstrenge af proteinerne. Jeg ønsker at sammenligne proteinet NEBU_HUMAN med andre proteinstrenge. Det vil være trivielt at implementere en samtidig løsning, hvor flere proteinstrenge sammenlignes samtidigt. Denne metode vil dog give et urealstisk højt hukommelseforbrug, hvorimod hukommelsesforbruget ideelt set ikke ændrer sig nævneværdigt ved indførslen af samtidighed, hvor der kun sammenlignes to proteinstrenge ad gangen.

De to strenge består af en række karaktere, som kan individuelt kan sammenlignes ved et opslag i en erstatnings matrice. I denne implementering bruger jeg BLOSUM62 matricen til at sammenligne enkelte proteinkaraktere med hinanden. For at kunne beregne matricen effektivt, skal det sikres, at det samme felt ikke udregnes flere gange af forskellige processer. Udregningen af matricen starter i øverste venstre hjørne og udregnes dels mod højre og dels nedad. For at kunne beregne et givet felt, skal man kende værdierne i kolonnen over feltet samt værdierne til venstre for feltet i samme række, samt det felt der findes en over og en venstre for feltet.

Felterne i matricen adressere jeg som række-først tupler, hvor (0,0) angiver feltet øverst til venstre i matricen. Første række i matricen (*,0) kan udregnes ved at starte i felet (0,0) og udregne et felt ad gangen. Hvor hvert beregnet resultat åbner mulighed for at beregne det efterfølgende felt. For at have flere samtidige processer, der beregner på den samme matrice, skal det udnyttes, at et beregnet felt kan åbne op for, at to nye felter kan beregnes. Efter at have beregnet første felt (0,0) i matricen kan både felt (0,1) og (1,0) beregnes. (0,0) har dermed åbnet for to samtidige beregninger. Beregnes (0,1) er der fortsat mulighed for to samtidige beregninger (2,0) og (0,1). Det er dermed ikke givet at hver felt åbner op for flere nye beregninger. Hvis det tredje felt, der beregnes, er (1,0), vil der efterfølgende være tre mulige felter der kan beregnes (2,0), (1,1), (0,2). Ved hele tiden, at beregne et felt i hver række, maksimeres antallet af felter, der kan beregnes samtidigt. Her har jeg taget udgangspunkt i at udregne felterne rækkevis, men dette kan naturligvis også foregå per kolonne.

Har man viden om, hvilken type maskine programmet skal fungere på, kan det udnyttes. Eksempelvis hvis man har en maskine med delt hukommelse, kan man designe programmet til at arbejde yderst effektivt, men det strækker ideen bag CSP-definitionen og bryder efter min mening også smed den. Race-conditions kan som beskrevet tidligere ikke finde sted i CSP, fordi alt data kun kan tilgås af en proces af gangen. Når en proces er den eneste, der har en reference til data, vil den kunne skrive til data uden mulighed for race-conditions.

Ved at sende referencer igennem kanaler opnår man at flere processer kan tilgå samme data. Så længe flere processer har den samme reference, må ingen af dem skriver til den, da dette vil kunne resultere i en race-condition. Hver proces skal derfor vide hvilke referencer den har eneret over, og dermed kan skrive til, og hvilke den kun må læse fra.

Disse fakta kan udnyttes til en effektiv implementering af SW. For at have styr på hvilke referencer, der må skrives til og hvilke, der kun må læses fra, introduceres egenskaberne "single access" og "read only". En producent opretter matricen, der skal beregnes. Alle felter i matricen er markeret som "single acces" og kun en proces må have en reference til hvert felt, som tilgengæld både må læse og skrive til det.Omvent markere et felt med "read only", at der ikke må skrives til det, men at flere processer må have referencen til feltet på samme tid.

Producentens opgave er at vedligeholde en liste med de felter, der ikke er beregnede, men hvor alle de krævede felter, er det. Producenten vælger kontinuerligt felter fra listen,

der hvor det er muligt, vil give en forøgelse af felter, der er klar til at blive beregnet, og sender dem til arbejderprocesserne.

Arbejderprocessen modtager et felt markeret som "single access", som skal beregnes, samt en række referencer til de felter, der er påkrævet for at udregne feltet. Arbejdesprocessen udregner værdien for feltet og gemmer det i referencen, hvilket svarer til at gemme direkte i matricen hos producenten. Når arbejdsprocessen er færdig med at skrive data, returneres referencen til producenten, som markerer feltet som "read only", da ingen igen skal skrive til feltet i referencen.

Dataoverførselen mellem processerne er hurtig og hukommelsesforbruget minimalt, da alle bruger det samme hukommelse og kun sender referencer rundt. Fordi hver proces kun arbejder på et felt ad gangen, vil mange proccesser desuden hurtigt komme til at arbejde samtidigt.

Ulempen ved denne metode er, som det tydeligt fremgår, at det kun virker for arkitekturer, som har delt hukommelse, og at den derfor ikke vil virke på eksempelvis en cell processor. Desuden synes jeg ikke, det overholder ideerne i CSP at markerer referencer som værende enten "read only" eller "singel acces". Markering af referencer er ikke understøttet i nogle af CSP-bibliotekerne, hvorfor en programmeringsfejl hurtigt vil kunne lede til race-condition.

Jeg har valgt at designet en model, der ikke forudsætter delt hukommelse. Designet er en producent/arbejdermodel, hvor der kun sendes værdier. Producenten initiere matricen, der skal beregnes, og sender den første række, til den første arbejder, som beregner rækken og sender den tilbage til producenten. Hver proces arbejder på en enkelt række og arbejder systematisk fra venstre mod højre. For at give mulighed for at have flere processer arbejdende samtidigt, er det nødvendigt, at arbejderne sender resultaterne af felterne, videre inden rækken er færdig. Når første felt er beregnet hos den første arbejder, kan værdien sendes til næste arbejder, som kan starte på sin række og sende videre. På denne måde vil k processer kører samtidigt, når den første proces har beregnet k værdier.

Ulempen ved denne metode, er at der sendes en værdi efter hver beregning og derfor vil kommunikationstiden overstige arbejdstiden. En bedre metode er derfor, at lade hver proces beregne et antal felter i en række, men ikke hele rækken, inden resultatet af felterne sendes videre. Jeg har valgt at lade antallet af felter, der skal beregnes, være bestemt af hvor mange arbejdsprocesser skal arbejde samtidigt, samt hvor lang proteinstrengene er. Præcist beregnes proteinstrengenes længde / antal processer. Hermed er man sikret, at hver proces har mulighed for at foretage en flere beregninger i træk uden at bruge tiden på kommunikation, og at når første række er beregnet, arbejder alle processer samtidigt.

Som nævnt ønsker jeg ikke at lave en løsning, så den kun kan bruges på en delt hukommelsesstruktur, men faktum er, at de maskiner, jeg tester programmet på, har en delt hukommelse. For at opnå en pænere kode, har jeg valgt, at hver arbejdsproces har en reference til samme BLOSUM62 matrice, da denne på alle tidspunkter i programmet er statisk. Desuden har alle processer en referencer til de to tekststrenge, der skal evalueres. Disse kunne også været placeret i hver sin arbejderproces, hvor eneste forskel ville være et øget hukommelsesforbrug.

Producenten har en kanal til den første arbejder for at kunne initiere arbejderne. Desuden har producenten en kanal, hvor den modtager de færdige rækker fra alle arbejderne. Hver arbejder har tre kanaler, de kommunikerer over. En kanal, hvor de modtager beregnede værdier af felterne fra en foregående arbejder. En kanal, hvor de selv sender data videre til næste arbejder samt en kanal til producenten, hvor de sender den færdigberegnede række.

Workflowet i algoritmen er, at data sendes fra producent til den første arbejder. Denne beregner en række og sender løbende data til næste arbejder. Et gennemløb med k arbejderprocesser beregner derfor k rækker af kostmatricen, som sendes tilbage til producenten. Resultaterne sendes derefter til arbejderproces 0 til udregning af den $k+1$ række og så fremdeles. Når rækken er færdigberegnet af hver arbejderproces, sendes rækken til producenten. Jeg har siden indset, at dette er unødvendigt, da den sidste arbejder kræver alle værdier, der er beregnet i matricen. Det er derfor tilstrækkelig, at den sidste arbejder sender data videre enten til første arbejderproces eller til producenten. Denne øgede kommunikation er dog minimal set i forhold til arbejdet med at beregne hele rækken og påvirker derfor ikke i nævneværdig grad beregningshastigheden, og jeg har derfor ikke reimplementeret det.

Den opbyggede matrice giver ikke i sig selv de to delstrenge, som har det bedste overlap. For at finde det bedste overlap er det nødvendigt at kende den position i den færdigtberegnede matrice, der har den højeste værdi. Dette sted markere afslutning på de to delstrenge der har det bedste overlap. De to bogstaver i proteinerne der gav værdien i feltet markere de sidste bogstaver i de to delstrenge. Fra position løbes baglæns og for hvert felt printes bogstaverne ud indtil feltet værdi når startværdien 0.

For at kende placeringen af feltet med den højeste værdi, holder hver proces øje med hvor den har set den maksimale værdi og sender denne position til produceren sammen med rækken. Produceren sammenligner den maksimale fra hver arbejder og når alle rækker er færdigberegnet og producenten har samlet matricen, kender producenten positionen i matricen uden at skulle løbe matricen igenne. Producenten står som det sidste for at printe de to delstrenge ud som giver det bedste overlap.

Som nævnt i begyndelsen har jeg valgt kun at fokusere på to protein strenge ad gangen. Når der startes kan kun en proces arbejde på matricen, men som arbejdet skrider frem kan flere processer samtidigt arbejde på samme matrice. Ligeledes vil der i slutningen af matricen igen være færre processer der kan arbejde samtidigt. En udvidelse er derfor at lade producenten kende alle proteiner der skal sammenlignes med og når der ikke er mere arbejde til alle processer i en streng startes beregninger på det næstes protein. Så er det kun i starten af det første protein der er begrænsning på antallet af processer der kan arbejde samtidigt

I opbygningen af den valgte producent arbejder model, er hver arbejder implementeret som en proces. hvis man ønskede en endnu større grad af samtidighed kan man lade arbejderen være delt op i flere af processer. Specielt kan beregning af hvor mange mellemrum det er optimalt at have, foretages samtidigt i både kolonne og række retning.

5 Test

På den medfølgende cd er kildekoden til implementeringen af de to algoritmer. For myresimuleringen er den grafiske del lavet ved hjælp af grafik bibliotektet QT. Det er påkrævet at man har installeret QT for køre programmet med grafik. For at køre simuleringen, når man ikke har QT, har jeg lavet en kopi af programmet hvor grafikken er fjernet. Som inddata til algoritmerne angives antallet af arbejdprocesser, dimensionerne på området, antallet af myrer, antal tidsskridt for simulationen, samt hvor ofte der skal vises et billede.

For Smith-Waterman findes der ikke grafik og det kan derfor køres på alle maskiner. Inddata til denne algoritme er antal arbejdsproceser, prisen per mellemrum, samt stien til de to proteiner.

5.1 Myrer

For at teste den udviklede algoritme har jeg kørt algoritmen et antal gange, hvor der med jævne mellemrum blev udskrevet et billede at simulationen. Billedet viser hele simulationsområdet med myretuen i midten og maden som store røde firkanter. Hver myre er vist som en gul firkant og feromonsporet er indtegnet som en nuance mellem hvid og blå, hvor blå indikere det maksimale feromonspor og hvid er det minimale feromonspor. Billeder fra en kørsel kan ses i billaget på side 18.

Når simulationen kører ses i en sekvens af statusbilleder, hvordan myrerne bevæger sig. Her kan man kan se, at myrene deler sig i grupper. En stor del bevæger sig rundt og opretholde feromonniverauet, der altid er en rute tilbage til myrertuen. Og en lille del af myrerne afsøger området omkring tuen. Dette skyldes, at myren som regel ønsker at holde sig på feromonsporet og kun en sjælden gang bryder ud af sporet og går tilfældigt rundt indtil den rammer et eksisterende spor, maden eller udkanten af området. Den eneste metode til at forstærke en rute mellem maden og myretuen fremkommer ved, at myren bliver vendt om, når den samler mad op, og når den afleverer det. Desuden foretrækker myrerne en gammelkendt vej, som har et stærkt feromonspor, ift. en ny, hurtigere vej med kun lidt feromon.

Ved inspektion af sekvensen af statusbilleder kan jeg konstatere, at myrerne har en tendens til at fange deres eget homonspor og går i uendelig løkke. Dette beskrev jeg allerede i analysen, men jeg forventede, at de i de fleste tilfælde selv ville slippe ud af løkken, men dette har vist sig ikke at være sandt. Man kan påvirke myrernes bevægelse ved at manipulere med de feromonniveauer, som myren opfatter i de fem felter omkring sig. Jeg har øget feromonniveauet i det felt som myrerne opfatter som lige foran, således er der en større sandsynlighed for, at den bevæger sig lige ud og opdager et større område.

Myrerne har stadig en klar tendens til at bevæge sig, hvor der har gået myrer før. Dette bevirker, at i flere tilfælde transporteres der ikke nødvendigvis mere mad til myretuen, hvis man afsætter et feromonspor end at lade myrerne bevæge sig rundt i fuldstændig tilfælde retning; resultatet forbedres altså ikke nødvendigvis. Dette er mærkværdigt og jeg formoder, det skyldes, at konstanterne for feromonsporet ikke er optimalt sat, og/eller at der skal indlægges ekstra intelligens i myrerne. Dette kunne være en retningsangivelse til myretuen, så den nemmere kan komme direkte tilbage med maden. Det kunne være, at der udskilles et andet eller kraftigere feromonspor, når myren går med mad. Eller det kunne være, at myren kunne detektere omgivelser i en større radius omkring sig selv end de nærmeste fem felter. Alle disse optimeringer ændrer dog ikke på, hvordan paralleliseringen er implementeret.

5.1.1 Arbejdsfordeling

Jeg ønsker at teste arbejdsbelastningen for hver process, dvs. antallet af myrer hver process skal flytte. I testen er myretuen i midten og processerne skære området igennem $\frac{1}{4}$del fra toppen i midten og$\frac{3}{4}$ fra toppen. Tabellerne 1 på den følgende side og 2 på næste side viser antallet af myrer hver process har ansvaret for under gennemkørslen af programmet. Hvert forsøg er kørt med fire arbejdsprocesser og der køres 4000 iterationer.

I tabel 1 på den følgende side arbejdes på et 100*100 område og af tabellen ses det, at til at starte med befinder alle myrerne sig i en process. Efter forsøget har P_2 og P_3 hver ca. 400 myrer og de to andre processer har 100 myrer hver. De to processer tættest på myretuen står samlet for 80% af arbejdet.

Iteration	P_1	P_2	P_3	P_4
0	0	0	1000	0
1000	68	412	428	92
2000	79	409	416	96
3000	103	409	416	96
4000	89	398	412	101

Tabel 1: 4 processer med et 100*100 område, 1000 myrer, 4000 iterationer

Iteration	P_1	P_2	P_3	P_4
0	0	0	1000	0
1000	0	491	509	0
2000	0	497	502	0
3000	0	496	504	0
4000	0	486	514	0

Tabel 2: 4 processer med et 500*500 område, 1000 myrer, 4000 iterationer

I tabel 2 er området udviddet til 500*500. I tabellen kan man se at der på intet tidspunkt er nogle myrer i de to yderste processer og de to midterste processer står for 100% af myrene. Denne umiddelbart dårlige fordeling af arbejdt skyldes, at der altid vil være flere myrer tæt på myretuen og blive færre som afstanden øges. Når området øges vil afstanden til naboprocesserne øges, og i dette tilfælde er afstanden så stor, at myrere ikke bevæger sig ud til de yderste processers områder. Som forventet skal man derfor øge antallet af processer, når størrelsen af området øges.

Som beskrevet i implementering har jeg valgt, kun at opdele området i en dimension, dermed skal de fire processer i tabel 1 hver bestyre 25 linier. Dette var i tabel 2 udviddet fem gange til 125 linier per process. I tabel 3 er antallet processer ligeledes udviddet fem gange til 20 processer og her kan det ses, at fordelingen af myrer på de fire midterste processer $P_8 - P_{11}$ ligner fordelingen af myrer i tabel 1. Der er altså en linær sammenhæng med arbejdsfordelingen og størrelsen af området.

Iteration	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}
0	0	0	0	0	0	0	0	0	0	0	0	
1000	2	2	8	81	374	415	104	11	3			
2000	1	2	9	79	386	414	105	2		2		
3000	2	5	17	87	378	381	120	8	1			1
4000	1	4	6	96	349	396	130	12	5	1		

Tabel 3: 20 processer 500*500 område, 1000 myrer, 4000 iterationer

5.1.2 Kørselstider

I både tabel 4 på den følgende side og tabel 5 på næste side ses, at kørselstiden ikke er jævnt stigende. Når antallet af processer er få, kan man se, at kørselstiden er lavere ved lige antal processer end ved ulige. Dette skyldes, at ved et lige antal processer deles simulationsområdet igennem i midten, og myretuen ligger på kanten mellem to processer. Myrene vil derfor med det samme være fordelt ligeligt mellem to processer. Ved et ulige antal processer befinder myretuen sig midt i en proces' område og dermed vil de fleste myrer befinder sig i denne proces.

I tabel 4 på den følgende side vises kørselstiderne på en computer med en processor, når antallet af processer øges fra 1 til 8. Man kan sammenligne kørselstiderne i to grupper, dem med et lige antallet af processer og dem med et ulige. Her ses det at kørselstiderne øges, når antallet af processer stiger. Algoritmen kan dermed ikke drage nytte af at sim-

Processorer	Processer	tid(sek.)
1	1	75
1	2	96
1	3	237
1	4	146
1	5	364
1	6	226
1	7	414
1	8	301

Tabel 4: Kørselstider i et 500 * 500 område, 20.000 myrer og 5.000 iterationer.

Processorer	Processer	tid(sek.)
2	1	103
2	2	89
2	3	111
2	4	165
2	5	123
2	8	230
2	10	280
2	20	390

Tabel 5: Kørselstider i et 500 * 500 område, 20.000 myrer og 10.000 iterationer.

ulationsområdet bliver delt i mindre områder, men når antallet af processer øges, øges mængden af kommunikation og dermed stiger kørselstiden.

I tabel 5 vise køretiderne på en computer med to processorer. Som forventet falder køretiden ved to processer, men kun med 13,5%. Myretuen er placeret i midten af området og med to processer, vil de dele simulationsområdet lige igennem myretuen. Derfor er der en lige fordeling af arbejde mellem de to processer. Optimalt skulle der have været tæt på en halvering af kørselstiden. Da dette ikke er tilfældet må man antage at kommunikation mellem processerne tager en stor del af gevinsten ved at køre på to processorer.

5.2 Smith-Waterman

Den optimale delstreng mellem to proteiner afhænger af erstatningsmatricen og straffen for at indsætte et mellemrum. Med BLOSUM62 som erstatnings matrice og en straf på 5 for hvert mellemrum, er den højeste værdi 58:

```
max is: (88,3602) = 58
Assignment1_data/P56134.pro     RGYYRYYNKYINVKKGS-ISGITMVLA--CYVLFSYSFSYKHLKHE
Assignment1_data/NEBU_HUMAN.pro REYKKEFEKW-KTKFSSPVDMLGVVLAKKCQILVS-DIDYKHPLHE
It took 18 seconds (0.28 min)for 1 threads
```

For en computer med en proccesor forventes det, at kørselstiden er optimalt, når der kun er en arbejderproces, fordi der dermed ikke bruges tid på at skifte mellem procceserne. Som forventet stiger kørselstiden også når antallet af processer øges fra 1 til 10. Derefter kommer der mod forventning et fald i kørselstiden og mellem 75 og 100 proceser opnås de hurtigste kørselstider. Når antallet af processer øges, falder blokstørelsen, hver arbejdsproces skal behandle. Jeg formoder derfor, at forbedringen i køretid opstår, når blokkene kan være direkte i cachen, og at man dermed har færre "cache miss". Når procesantallet øges over 100 stiger kørselstiden som forventer, pga den øgede kommunikation processerne imellem.

I tabel 7 på den følgende side vises kørselstider for Smith–Waterman på en maskine med to processorer. Da processorne er væsentligt hurtigere, vælger jeg at udvide testen ved at sammeligne NEMU_HUMAN med et længere protein for således at tydeligøre forskellene, når antallet af processer øges. Her ses som forventet og modsat tabel 6 på næste side (hvor der kun findes en processor) et fald i kørertiden, når antallet af processer øges fra en til

Processorer	Processer	tid(sek.)
1	1	18
1	2	19
1	5	21
1	10	23
1	25	17
1	50	15
1	75	14
1	100	14
1	150	24
1	200	26
1	250	29
1	1000	39

Tabel 6: Kørselstider i for at finde bedste overlap mellem proteinet NEBU_HUMAN og P56134.pro.

Processorer	Processer	tid(sek.)	Processorer	Processer	tid(sek.)
2	1	69	2	100	66
2	2	52	2	200	65
2	3	53	2	250	62
2	5	53	2	300	61
2	6	57	2	500	61
2	7	61	2	600	62
2	10	62	2	750	62
2	20	63	2	1000	73
2	30	64	2	1500	81
2	50	64			
2	75	64			

Tabel 7: Kørselstider i for at finde bedste overlap mellem proteinet NEBU_HUMAN og Q8G5F3.pro.

to. Derfra stiger kørselstiden som forventligt indtil der nås 100 processer, pga. den øgede kommunikation. Fra 100 processer falder køretiden til et lokalt minimum mellem 300 og 500 processer. Dette lokale minimum tilskriver jeg ligesom før, at blokkenes størelse falder til de kan være i cachen. Den forbedrede blokstørrelse bliver overskygget af den øgede kommunikation, og den hurtigste køretid findes, når der kun er to processer.

6 Konklusion

Jeg har med succes implementeret to algoritmer gennem CSP-modellen, så de kan bruge et vilkårligt antal processorer. De er implementeret med en forventning om, at ydelsesforbedringen ved flere processer automatisk fremkommer, når algoritmerne køres på computere med flere processorer. Testene viser, at dette har været en naiv tilgang og selvom algoritmerne udnytter flere processorer til en hastighedsforbedring er det er vigtigt, igennem designet har fokus på ydelsen og kommunikationen, for at opnå en optimal algoritme.

I CSP gælder, at alle processerne som standard oprettes som seperate kernetråde, men efter at have gennemført testen, er jeg kommet til den konklusion, at man kan opnå forbedringer ved at have processer, der både er kernetråde og brugertråde. Med en optimal fordeling vil der i Smith–Waterman-algoritmen på en computer med to processorer være to processer med kernetråde, og de resterende processer vil starte som brugertråde. Hver brugerproces kommunikerer hurtigt med dens naboer i seperate brugertråde og kun de to processer, der kommunikere på tværs af kernetrådene lider af langsom kommunikation. Den hurtigere kommunikation samt hurtigere skift brugertråde imellem vil give en betragtelig hastighedsforøgelse.

At designe algoritmer uden at tage hensyn til antallet af processorer, har vidst sig at medføre et ydelsetab og man bør istedet designe algoritmerne med henblik på kun at oprette det antal kernetråde, som der findes processorer. For at få dette til at harmonere med, at algoritmerne ikke skal designes til en specifik computer, men skal kunne bruges uafhængigt af computerenm er det vigtigt, at man på kørselstidspunktet angiver antallet af processorer.

Man kunne også forbedre algoritmen med implementering af buffere. Alle kommunikationskanalerne er implementeret uden brug af buffer. Dermed tvinges processen til at vente på at den modsatte process er klar til at modtage/sende, før den kan fortsætte. Ved at ændre implementationen til at bruge buffere, ville processer oftere kunne fortsætte deres kørsel og deraf forbedre ydelsen.

CSP er en interessant metode til at designe algoritmer, der kan køre på computere med flere processorer. Det kan være med til både at sikre forbedringer på computere med flere processorer, men nok så interessant også i nogle tilfælde for computere med kun en processor. Med tilførslen af begrebet processer sikres en høj grad af isolation, som er kendt for at være "best pracsis". Jeg finder, at man med CSP nemmere og hurtigere end med konventionel tråd programmering kan designe algoritmer, der udnytter flere processorer. CSP løser ikke alle problemerne ved parallelprogrammering, og for at drage fuld nytte af flere proccesorer er det nødvændigt at medtænke,s at algoritmen ikke skal afvikles på en abstract processor, men på en fysisk processor med dertilhørende begrænsninger.

A Billag

Figur 1: Kort efter start. Myretuen befinder sig i centrum. Øverst til venstre findes et område med føde. feromonsporet ligger fortrinsvist tæt på myretuen. 8 myrer udforsker området længere væk fra myretuen.

Figur 2: Feromonsporet omkring myretuen er udviddet. Flere myrer er nu begyndt at udforske områddet. I bunden ses en myre der bevæger sig i en løkke.

Figur 3: Begyndelsen på hvad der kunne blive til et spor mellem myretuen og føden

Figur 4: Sporet fra foregående figur er forsvundet og feromonsporene ligger omkring myretuen.

Litteraturliste

Beckers, R., J.L Deneubourg og S. Goss (1992). „Trails and U-turns in the selection of the shortest path by the ant Lasius niger“. I: *Journal of theoretical biology* 159, s. 397–415.

Brown, N.C.C. og P.H. Welch (2003). „An Introduction to the Kent C++CSP Library“. I: *Communicating Process Architectures 2003*. Udg. af J.F. Broenink og G.H. Hilderink. Bd. 61. Concurrent Systems Engineering Series. Amsterdam, The Netherlands: IOS Press, s. 139–156. ISBN: 1-58603-381-6. URL: http://www.cs.kent.ac.uk/pubs/2003/1784.

McEwan, Alistair A., Wilson Ifill og Peter H. Welch, red. (2007). *C++CSP2: A Many-to-Many Threading Model for Multicore Architectures*, s. 183–205. ISBN: 978-1586037673.

Coffman, E. G., M. Elphick og A. Shoshani (1971). „System Deadlocks“. I: *ACM Comput. Surv.* 3.2, s. 67–78. ISSN: 0360-0300. DOI: http://doi.acm.org/10.1145/356586.356588.

Dorigo, M. (1992). „Optimization, Learning and Natural Algorithms“. Ph.d.-afh. Politecnico di Milano.

Hoare, C. A. R. (1978). „Communicating Sequential Processes“. I: *Commun. ACM* 21.8, s. 666–677.

May, David (1983). „OCCAM“. I: *SIGPLAN Notices* 18.4, s. 69–79. URL: www.trier (Lokaliseret d. 06.04.08).

Muller, Hans og Kathy Walrath (1998). „Threads and Swing“. I: *Sun's hjemmeside*. URL: http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html (Lokaliseret d. 17.06.2008).

Reynolds, Craig W. (1987). „Flocks, herds and schools: A distributed behavioral model“. I: s. 25–34. DOI: http://doi.acm.org/10.1145/37401.37406.

Smith, T. F. og M. S. Waterman (1981). „Identification of common molecular subsequences.“ I: *J Mol Biol* 147.1, s. 195–197.

Welch, P.H. (1989). „Graceful Termination – Graceful Resetting“. I: *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*. ISBN 90 5199 007 3. Occam User Group. Enschede, Netherlands: IOS Press, Netherlands, s. 310–317. URL: http://www.cs.kent.ac.uk/pubs/1989/252.