

# Ekstrem Multiprogrammering 2008

Opgavestiller: Brian Vinter

Opgaveløser: Lars Ole Belhage



# Smith-Waterman sequence alignment

## Basal algoritme

Det grundlæggende resultat af Smith-Waterman er at finde det bedste vægtede match af subset af 2 strenge. Vægtningen tages fra en vægtmatrice, som vægter matchede tegn, tegn for tegn. Desuden kan man indskyde “overspringelser” med en vægtning. Algoritmen beskrives som en metode til at udfylde en matriks med kolonner som den ene streng og rækker som den anden streng. Matricen udfyldes fra øverste venstre hjørne mod nedre højre hjørne. Hver værdi baseres på allerede udregnede værdier efter formlen (i ascii-grafik):

$$H_{ij} = \max\{H_{i-1,j-1} + \text{vægt}(a_i,b_j), \max_{k \geq 1}\{H_{i-k,j} - W_k\}, \max_{l \geq 1}\{H_{i,j-l} - W_l\}, 0\}$$

Hvor “vægt” er vægtmatricen og  $W$  er vægtene for at indsætte en overspringelse.

Det vil sige den nye værdi afhænger af værdien op-venstre (skrot op med Venstre ?), søjlen over og rækken til venstre. Bemærk at en direkte implementering af formlen giver  $O(n^3)$  opførsel.

En praktisk forenkling, som Smith og Waterman også gør, er at lade  $W$  være lineær afhængig af overspringelsens længde – dvs en konstant vægt per “hul”. (Der findes en “Gotoh” variation hvor prisen for åbne et hul er større end en udvidelse af hullet – men det giver samme forenkling i implementeringen). Denne forenkling gør at man kan nøjes med at kigge på værdierne i de 3 naboceller til venstre og op.

## Design

Det stillede problem, går ud på at sammenligne en relativt lang test-streng (6671 protein-tegn) mod en “database” på 100 relativt kortere strenge (64 – 1694, median: 255).

En oplagt, men kedelig, parallelisering er at samtidigt starte udregningen af disse. Problemet med dette er hver udregning er memory-tung ( $n*m$ ), man kan max. starte 100 processer og de er endda meget ulig i størrelse, så det kan være svært at holde mange processer (processorer) fuldt udnyttet.

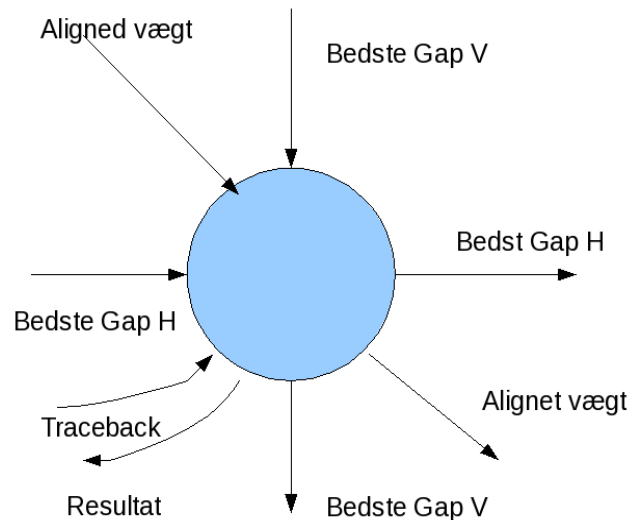
Næste iagttagelse er forskellen i længderne af test-strengen og databasestrengene. Man kan estimere en “worstcase” på matchet ud fra vægtene og priser for et hul – ved at indsætte ligeså mange huller som muligt uden at vægten kommer under 1 (og vælge den største vægt (som altid er ved ens proteiner). Dette har jeg gjort i appendix I. Heraf ses at worstcase længderne er 193 – 5181 median: 781 (75% fraktil: 1258). Dette betyder at man kan opdele test-strengen i mindre, overlappende dele (hvor hver del skal være mindst  $2*worstcase$  og overlappet skal være på mindst  $1*worstcase$ ) og løse disse parallelt (4 samtidige ved medianen og 2 ved 75%). Disse delproblemer har mindre memoryforbrug og kan være mere ensartede i størrelse. Problemet med dette selvfølgelig det meget begrænsede antal opdelinger og at memoryforbruget totalt set er større pga overlappet.

En anden angrebsvinkel er at kigge på den forenkledte algoritme og betragte den enkelte celle som et beregningselement. Dvs et element som modtager data fra sine 3 “forgængere”, beregner og leverer data til de efterfølgende 3 celler (plus en mulighed for at udtrække resultatet).

I CSP kunne denne process se ud som:

Elementerne i første række og første søjle, vil mangle nogle af deres inputkanaler. Ligeledes vil dem på de andre kanter mangle deres output.

For få resultatet er der tilføjet en Resultat kanal som til en central process rapporterer værdien i elementet. Den centrale process kan så vælge den overordnede bedste værdi og starte en traceback derfra. (ikke vist på tegningen er, at den kan "videre-send" traceback til sine forgængere via traceback outputkanaler).



En netværk af disse processer vil så kunne løse opgaven. Der vil kunne opnås et vist mål af samtidighed, idet processer "i diagonalen" vil kunne køre uafhængigt.

Beregningsmønsteret er således:

1	2	3	4	5	...	m-1	m
2	3	4	5	...		m	m+1
3	4	5	...			m+1	m+2
4	5	...				m+2	m+3
...							...
n-1	n	n+1	...			m+n-3	m+n-1
n	n+1	n+2	...			m+n-1	m+n

Det vil sige at der kan opnås op til  $\min(m,n)$  parallelle processer i  $\text{abs}(m-n)+1$  af diagonalerne og lineært stigende/faldende antal i de resterende diagonaler.

Disse processer kan enkelt samles i større processer – hvis man vælger en rektangulær samling (med 1 – n rækker og 1 – m søjler), vil procestegningen være identisk, bare med vektorer af værdier i kanalerne. Dermed kan vælges et vilkårligt mål for samtidigheden.

F.x. ved 4 macroprocesser:

1	2
2	3

Altså en potential forbedring på 25%

## Implementation

Jeg har valgt at bruge C++CSP2 til afprøvning af modellen. Det er et rent unix-commandline program. Macro processerne er implementeret i "plain code" - men kan varieres i størrelse fra  $M*N$  (kun een beregnings process) til  $1*1$  (en hel del processer! – lidt forskelligt antal tråde).

Vægt matricen er implementeret i en "class W" som kan angive vægten mellem 2 proteiner (indlæses fra BLOSUM62 filen) og vægten for gabs, som er hardwired til '5'. Denne matrice

kopieres ud til hver process (måske lidt overkill ift “No shared data-structures” - men jo nødvendigt hvis man f.x. vil afvikle det på en Cell/BE maskine eller andre med opdelt ram).

Jeg har opbygget processen således at den kun modtager data fra 2 andre processer: ovenfor og fra venstre. Disse 2 processer leverer begge en vektor hvor den ene op-venstre værdi er med (lille data redundans, men 30% mindre kommunikation). Af hensyn til randfænomenerne er der knyttet et “isDummy” flag til de 4 kanaler (kunne nok have været implementeret pænere med en “Null-channel”...). Da det ikke vides hvilken af de 2 forgænger processer der kan levere først læser jeg fra dem i parallel.

Hver process skal desuden kunne aflevere “bedste værdi”, dette gør den med en Any2One kanal til “manager” processen.

Til slut skal man kunne lave traceback for at finde de faktiske substreng/huller som udgør matchet. Dette initieres når managerprocessen har modtaget alle beregningsprocessernes “bedste værdi”.

Processen skal (måske) kunne sende traceback til forgængere.

Derefter skal det færdige traceback leveres som resultat – hvilket sendes til managerprocessen (Any2One igen).

Som til slut “sender” poison til traceback kanalen.

De data som sendes rundt og som gemmes “i maven” på processerne er:

```
class Idx { ...
    int w, i, j;          // Vægt og index i n*m-matrice som gav anledning til værdien (traceback)
};
class Cell { ...
    Idx b;                // Vægt/Traceback for denne celle
    Idx bkV;              // Vægt/Traceback for vertical overspring (overkill, kunne udledes..)
    Idx bkH;              // Vægt/Traceback for horizontal overspring (overkill, kunne udledes..)
};
typedef vector<Cell> vb;  // Vector of Borderelements
```

Dette resulterer i følgende monster-constructor til en beregningsprocess:

```
Ss(const W& w_, const vs& h_, const vs& v_, const Idx& o_, // Vægtmatrice og delstreng
    bool hbInDum_, const AltChanin<vb>& hbIn_,           // Data oppefra
    bool vbInDum_, const AltChanin<vb>& vbIn_,           // Data fra venstre
    bool hbOutDum_, const Chanout<vb>& hbOut_,           // Data nedad
    bool vbOutDum_, const Chanout<vb>& vbOut_,           // Data til højre
    const Chanout<Idx>& resOut_,                          // Bedste resultat
    const Chanin<Trace>& traceIn_,                       // Trace – kan “poison”
    const Chanout<Trace>& htraceOut_,                    // Trace videre opad
    const Chanout<Trace>& vtraceOut_,                    // Trace videre til venstre
    const Chanout<Trace>& traceRes_                      // Færdig trace til Manager
)
: CSProcess(4096), ... // Mindst mulig stack
```

Og følgende skelet for “run()”:

```
læs data fra forgænger <-- hbIn || vbIn (*se nedenfor)
bregn
send data til højre --> hbOut
send data nedad --> vbOut
send data til manager --> resOut (hvis nedre-højre proc – poison resOut)
læs fra trace-request <-- traceIn (eller catch poison og poison v/h-traceOut)
```

```

udfyld trace
hvis nødvendigt send    --> v/h-traceOut
ellers send slutresultat --> traceRes
læs fra trace-request   <-- vent på poison...

```

Manager processen har følgende skelet:

```

while (1)
  læs fra resOut    <-- (stop ved poison)
  send tracereq     --> traceOut
  læs slutresultat  <-- traceRes
  udskriv resultat
  forgift trace kanal --> poison traceOut

```

Parallel læsning af hbIn og vbIn:

Jeg har ikke kunnet finde en simpel løsning på parallel læsning vha C++CSP2. Derfor har jeg implementeret det med en Alternative:

```

list<Guard*> guards;
guards.push_back(hbIn.inputGuard());
guards.push_back(vbIn.inputGuard());
Alternative alt(guards);
switch (alt.fairSelect())           // Ligegyldigt om fair/prio/...
{
  case 0: hbIn >> bh; vbIn >> bv; break; // hbIn, vent på vbIn
  case 1: vbIn >> bv; hbIn >> bh; break; // vbIn, vent på hbIn
}

```

## Iagttagelser

Implementeringen er skrevet og afprøvet på Linux på en lille håndfuld maskiner (intel single core-cpu laptop, amd quadcore discount pc, dual amd dual-core server). Der var ingen praktiske problemer med at hente, kompilere/installere eller anvende C++CPP2 på nogle af platformene.

Jeg har i C++ brugt et supersimpelt 2D-array som gemmer i column-order, hvilket jo passer fint med den umiddelbare for-x/for-y angrebsvinkel (Fortran/C++ Rules!). Betydningen af at læse ram i en uheldig rækkefølge er stor (se appendix II).

C++CSP2 laver default en posix-tråd ud af hver process – jeg har eksperimenteret med at lave hver række “InParallelOneThread”. Processerne i samme række skal alligevel behandles venstre til højre, og vil aflevere til næste række så hurtigt som muligt, så at køre dem i een tråd burde ikke være begrænsende – og burde forbedre performance en del da CSP så kan gøre det non-preemptive, plus at man sparer “thread-ram”).

På 4core discount pc får jeg følgende resultater med “time ./SW -XX:YY BLOSUM62 \*.pro”, hvor XX, YY er division i X og Y (altså XX\*YY beregningsprocesser). Jeg har prøvet både med forkert og “rigtig” array-order.

XX:YY	All Database		P25870					
	Thread/Process		Thread/Row		Thread/Process		Thread/Row	
	Forkert Array	Rigtig Array	Forkert Array	Rigtig Array	Forkert Array	Rigtig Array	Forkert Array	Rigtig Array
1:1	111/106/8.1	17/12/4.8	111/106/8.1	18/13/4.7	5.7/5.5/0.2	0.9/0.7/0.2	5.7/5.5/0.2	0.9/0.7/0.2
2:2	81/100/4.3	15/14/3.7	81/101/3	15/14/4.0	4.3/5.4/0.2	0.8/0.7/0.2	4.5/5.6/0.2	0.9/0.7/0.2
4:4	47/81/8	15/14/4.6	45/80/6	14/15/3.8	3.0/5.8/0.3	0.7/0.7/0.3	2.8/5.9/0.3	0.8/0.8/0.3
8:8	29/37/20	24/21/22	26/37/14	20/19/12	2.1/4.3/1.2	0.7/0.7/0.3	1.9/4.2/0.6	0.7/0.8/0.3
16:16	85/58/124	101/65/175	50/44/74	45/38/65	0.9/1.2/1.4	0.9/0.9/0.9	1.0/1.3/0.6	1.0/1.0/1.8
32:32 (*)			305/206/577	336/227/663	4.0/2.8/5.5	7.0/4.2/12.6	3.0/2.0/4.2	2.8/2.0/3.6
16:4	85/59/136	23/18/16	20/25/8	16/17/7.3	1.0/1.1/0.2	0.7/0.7/0.3	0.9/1.1/0.4	0.7/0.8/0.3
4:16	49/85/8.5	22/21/18	25/34/11	17/19/0.6	2.4/4.8/1.5	0.8/0.8/0.3	2.3/4.7/0.8	0.7/0.8/0.3

Tabel over “time” fra forskellige kørsler på 4-core maskine (de 3 time tider er real(“stopur”), cpu tid brugt på brugerprocesse(n/erne) og cpu tid brugt i os-kernen. De 2 sidste tal skal deles med 4 for at kunne sammenlignes med real-time).

Umiddelbart ser det jo noget deprimerende ud...

Den største effekt ses når der er “dårlig” ram-access (men her er forbedringen ikke pga parallelisme, men at ramområdet bliver mindre per process og dermed straffen for forkert adgang mindskes).

Der er dog en lille forbedring af den total køretid ved opsplnitning i 16 processer.

Specielt hvis man tager den “interne” tid som det tager at regne på P25870 fås:

1:1 500ms  
 2:2 411ms  
 4:4 401ms  
 8:8 350ms  
 4:16 302ms

For at afprøve om der var desiderede fejl i CSP layout, har jeg lagt “spildtidsløkke” ind i align-funktionen – og iagtaget cpu-forbrugsmønstret (med “top”). Dette viser tydeligt at alle 4 cpu kan fuldt udnyttes (og at vi springer fra 1 cpu i brug, 2 cpu'er, 3-4 cpu'er og ned igen til 2 og 1).

SW vil være velegnet til afvikling på flere platforme, specielt bør Cell/BE være god (der skal så vælges en passende blokstørrelse som fylder de 256K-ram ud). Yderligere bør det være muligt at udnytte SIMD instruktioner (ihvertfald til gap-delen).

# Ant-hill simulation

## Basal algoritme

Grundtanken i myre-simulationer (og andre “simple”-agent netværk) er at en myre har et begrænset intellektuelt liv, men at den udfra ganske enkle regler (10 bud?) kan danne nyttige adfærds-mønstre i kraft af et samarbejde med et relativt stort antal ligesindede. (se appendix III).

I denne verden er myrer udstyret med et lugteorgan, en “føle”-sans (kan mærke når jeg rammer mad eller når jeg rammer hjemmet (tuen)), en begrænset retningssans (kan vendes 180deg), en simpel hukommelse (jeg har fundet mad – måske endda: jeg har afleveret mad 1-2bit) og en tisser.

Der er en del parametre man kan sætte (og/eller opfinde):

Hvor langt rækker lugteorganet

Hvor retningsbestemt er lugteorganet

Hvor langt rækker “føle” organet (støde ind i, eller lede efter)

Hvor god er retningssansen – (og er det en orthogonal eller polær hukommelse (Turtles?))

Hvor god er hukommelsen (måske slet ingen, jeg vender bare når jeg rammer Mad/Tue)

Hvor stor er tisseren – et vigtigt spørgsmål for en myre

og andre faktorer (uden for myrens “nu”-oplevelse):

Bliver maden spist op

Hvor længe/langt kan en myre gå, før den skal have mad eller vende hjem

Hvor mange skal der til, før vi er ligesindede nok

Hvor hurtigt formerer myrer sig

Hvor hurtigt løber en myre – og er det med konstant hastighed

Hvor bange er myrer for hinanden – skal de gå omveje når de møder hinanden

Hvor meget blæser det – dvs, hvor hurtigt fordamper “tisset”

## Design

Udover ovenstående parametre skal der vælges en overordnet model.

Myrerne lever i en verden centreret om deres tue. Med mindre de bor på en meget lille planet (så de kan gå hele vejen rundt om planeten og komme hjem igen), så må vi antage at deres verden er “uendelig” stor. Dette er ubehageligt at modellere i en computer – så vi må enten lade myren vide noget om hvor langt den er hjemmefra (det ved den nok ift den energi den har tilbage – men den kunnen var ikke tillagt den i opgaven), lade den støde ind i “verdens kant” (hvor den så ellers er) og vende om eller - trist nok - bare falde ud over kanten...

En oplagt process model for simulationen er at simulere hver myre som en process. Dette giver mening idet den enkelte myre kun er meget lokal i sin adfærd. Dog skal den have adgang til at sniffe efter myrespor og opdage mad/tue.

Det kan gøres ved at have en “geografi” process som kender til myretue, maddepoter og myrespor. Problemet er at “geografi”-processen kan blive en knap resource (selvom den kan være underdelt i andre processer, så bliver den simple interaktion (er der mad her ?, er der spor her ?, ...) tung i kommunikation). Det er også problematisk i en skaleringsammenhæng at een process skal kende “hele” verden.

Så kan man opdele geografien i mindre dele – f.x. i kagestykker og ringe (som en Dart-skive). Men

så får myren et problem – hvilke dele skal den spørge om info (Nu er det myren som skal kende hele verden).

En naturlig sammenkædning af geografi og myre er ønskelig – så fås maksimal “lokal afhængighed”.

Denne sammenkædning kan være at en myre “ejer” et stykke jord – men myren flytter sig hele tiden, og andre myrer kunne også gøre deres “ejer-ret” gældendene. Så denne model vil kræve et snedigt system til at håndtere opdeling af geografiske områder og udveksling af “trespasser” rettigheder (Det lyder jo næsten bekendt ;).

Omvendt kunne være at lade lagkagestykkerne administrere myren(erne) der pt. er i deres område...

Det er denne sidste model jeg har valgt. Dvs jeg lader landområdet håndtere myrerne, som så “bare” er data.

Næste design spørgsmål er data-repræsentationen – min første indskydelse var at lave alt som vektorer evt i et polært koordinatsystem. Men jeg endte med at lave et rektangulært grid, med maddepoter, tue og myrespor markeret som værdier i griddet.

Jeg har ladet en myre være bevidst om sin nuværende retning, kigge 8 steps frem i en 45deg graders vinkel til højre og venstre for den nuværende retning – og ændre retning proportionalt med forskellen i det samlede spor på højre/venstre side. Retningsændringen er størst når myren har fundet mad - så den er mere tilbøjelig til at følge sporet.

Myren tager så et tilfældigt step i enten x eller y afhængigt af den aktuelle retning – dir:

```
if (rand() <= fabs(RAND_MAX * cotan(dir) / 2)) x += sign(cos(dir)) else y += sign(sin(dir));
```

Myren markerer det nye gridpunkt med en mængde feromoner.

Når myren rammer en gridcelle markeret med MAD, vender den retning 180deg og sætter sit mad flag, når den rammer Tuen vender den 180deg (men beholder MAD flaget – så den har større sandsynlighed for at følge sporet tilbage til maden).

Landjorden opdeles i matrikler og hver kommunikerer kun med sine naboer. Kommunikationen består af at overdrage en myre fra et område til det næste når en myre går ud over kanten (Hvis det en kantmatrikel overdrages myren til en kirkegårdsprocess, som så genopliver myren i myretue matriklen). For at undgå kommunikation med naboområdet når en myre sniffer efter spor i nærheden af kanten overføres der også fra nabocellen det ekstra overlappende område som svarer til myrens udsyn (8). Når alle myrer i en matrikel er behandlet, nedskrives alle myrespor med een (fordampning) og matriklen venter på kommunikation med alle naboområder.

Dette betyder at hvert landområde kan udregnes uafhængigt af de andre – parallelt, og at “tiden” i hele verdenen synkroniseres af kommunikationen (klokken behøver ikke at være det samme overalt, men den er højst et tidsskridt væk).

## Implementation

Jeg har igen valgt C++CSP2, denne gang sammen med wxWidgets for at kunne vise fremdriften grafisk.

Jeg har valgt at lade wxWidgets køre sin eventloop i en CSProcess (WXCore), som så

kommunikerer med en WXExtern process – der igen kan modtage grafiske ændringer fra myre-matriklerne (WorldTile). Denne mellemlid er nødvendigt data wxWidgets kører sin egne kommunikation udenom CSP (frem og tilbage med X11). WXCore “poller” som om der nye data fra Wxextern for hver opdatering af grafik-vinduet (som er sat til at refresh 4 gange i sekundet):

```
bool pollData(WxCmdNData * wcd)
{
    list<Guard*> gds;
    gds.push_back(res.inputGuard());
    gds.push_back(new SkipGuard());
    Alternative alt(gds);
    if (alt.priSelect() == 0) { res >> *wcd; return 1; }
    return 0;
}
```

Hver matrikel har One2One kanaler sende og modtage til de 8 naboer (hjørnenaboerne er med så vi kan have “sniffe”-data). En Any2One kanal til WXExtern (til at tegne på skærmen) og en Any2One kanal til kirkegårds processen (Cementary) – som i sin tur har en One2One kanal til myretuematriklen (valg som center-matriklen). En myre overføres som en pointer til et myreobject (Ant) (kun een matrikel eller kirkegården har denne pointer).

Dvs den samlede mængde kanaler per matrikel er: 8+2 send-ender og 8(9 for tuen) modtage ender. Efter hver beregning kommunikere matriklen i en bestemt rækkefølge med naboerne:

- Send evt døde myrer på kirkegården
- modtag/send fra naboer til venstre og oppefra
- send/modtag til naboer til højre og nedefter

Dvs at kommunikationen vil “bølge” fra øvre højre hjørne frem til nedre venstre.

Jeg har valgt at lade myrerne afsætte et spor som er næsten 6 gange afstanden fra tuen i midten og ud til kanten. Dette gør at sporet ikke er fordampet helt før myren har en chance for at finde tilbage (med mindre den røgget ud over kanten på verdenen).

Når en myre fødes har den en tilfældig udklækningstid, så der ikke opstår “raids” af myrer som forlader tuen.

Det totale antal myrer har jeg valgt til 64 – men mønstret optræder ligeså fint med nedtil 16 myrer og forstærkes ved en endnu større antal (prøvet op til 1024). Jeg har også afprøvet en lidt anden model for myreliv – hvor myren dør helt når falder over kanten, men at der til gengæld fødes nye myrer hele tiden. Dette “sparer” Cementary processen – men gav ingen særlig forskel, hverken i hastighed eller mønstre.

Et andet eksperiment var at give myren lidt hukommelse (det sidste N koordinater), så den evt kunne undgå at gå i ring. Men da dette ser ud til at være tilfældet, havde det ikke den store effekt.

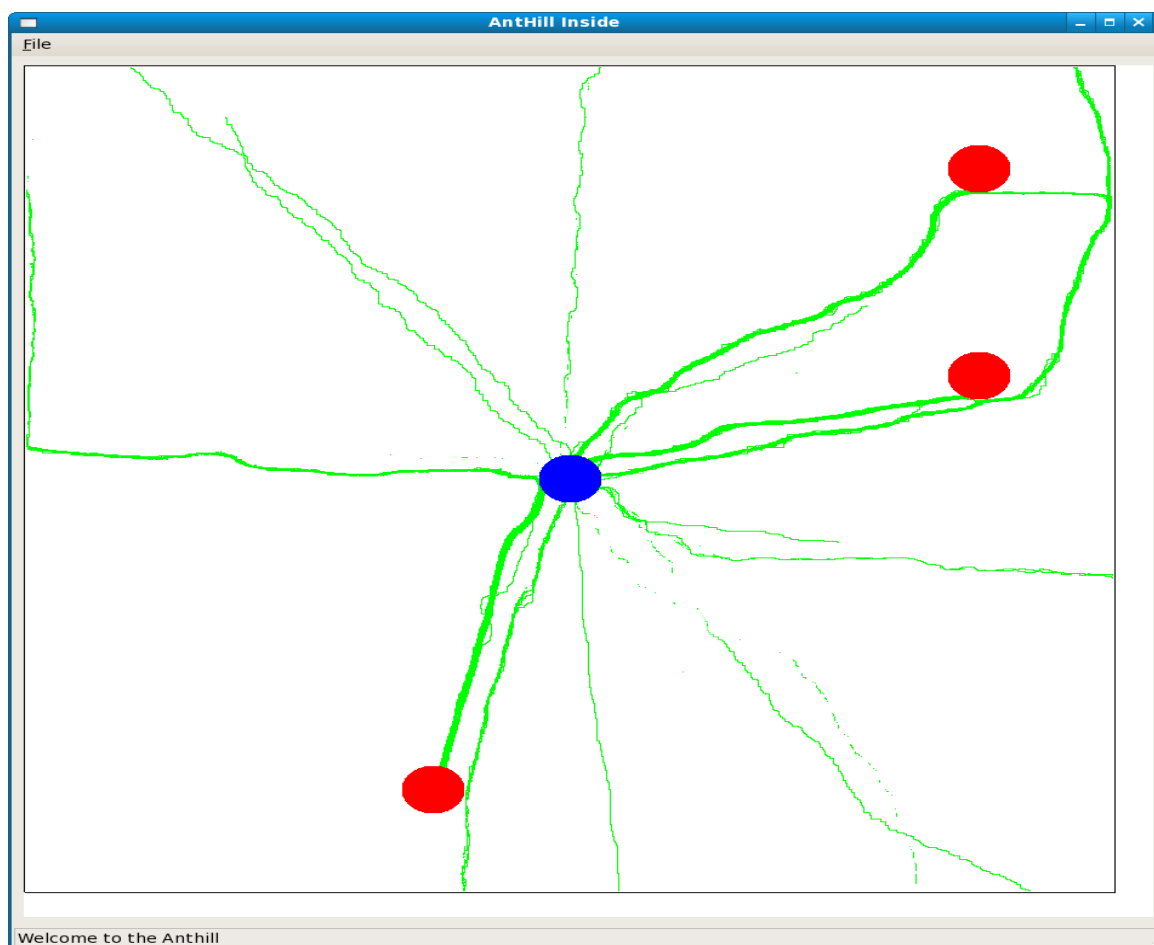
## Iagttagelser

Jeg har afprøvet det på 4core maskinen – og den viser god udnyttelse af af cpu. Med fra 9 tiles optil 225 tiles – hvorefter process overhead tager over (stadigt stigende andel af cpu brugt i “system”). Process overhead i “virtuelt” ramforbrug tager også overhånd ved stigende antal tiles. Generelt er den samlede ydelse faldende ved stigende antal tiles. Det skyldes igen at overheadet er stort i

forhold til den egentlige beregnings opgave i hver tile. (Fordi min samlede verden ikke er større end 900\*900 (for at kunne vises på skærmen ;)).

Modellen vil kunne skaleres meget op – enten ved at øge størrelsen af verdenen, eller ved at mindske tilestørrelse (skal dog være større end 2\*sniffe afstand – ellers overlappes der med andre end kun naboområdet). Den vil kunne udnytte maskiner med stort kommunikationsoverhead – hvis bare tile-størrelsen kan sættes højt. (Dvs ikke så egnet til Cell/BE – da den har relativt lille ramme per process).

Måske kunne man opnå en gevinst ved at undlade gridding og bruge vektorer istedet (vil mindske data som skal overføres mellem naboer, vil klart mindske ramforbrug – mod noget mere cpuforbrug i håndtering af myresporene (kunne evt gemmes i oct-træer))



Flere billeder i appendix IV.

## Referencer

Udleveret eksamens materiale, specielt Smith-Waterman's original paper...

Den implementerede kode kan hentes på:

<http://midibel.com/SW.cpp>

<http://midibel.com/AHI.cpp>

## Appendix I

Org Len	Worst Case	Org Len	Worst Case	Org Len	Worst Case	Org Len	Worst Case
1694	5181	414	1257	257	768	160	502
1115	3382	408	1228	243	731	150	456
934	2837	403	1221	239	723	143	428
876	2668	372	1172	237	715	135	415
862	2632	382	1165	226	712	137	413
846	2562	376	1159	230	704	131	398
777	2367	371	1147	226	701	130	391
767	2299	359	1125	226	685	123	369
659	2015	356	1094	214	659	118	360
603	1836	361	1086	220	659	117	348
554	1756	355	1082	214	658	117	345
571	1735	335	1032	209	636	113	344
567	1716	334	1023	212	635	112	337
536	1673	334	1014	197	614	110	335
522	1631	326	1009	192	598	109	324
527	1626	323	978	201	598	94	295
500	1524	326	975	190	576	91	264
490	1495	293	919	188	565	81	240
485	1493	303	916	186	554	79	235
493	1476	291	889	177	553	73	233
475	1446	289	869	184	543	75	228
462	1405	283	850	175	522	72	213
447	1370	272	838	173	519	72	212
420	1291	255	790	170	516	70	207

Tabel 1. Strenglængder – og max længde med gaps

## Appendix II

Jeg har lavet et lille test program for at se effekten af uheldig ramtilgang i matricer. Programmet afprøver nogle forskelle “mønstre” og timer disse.

F.x:

```
int * h = (int*)malloc(n * m * sizeof(int));  
memset(h, 0, n * m * sizeof(int)); // Sikrer at ram er (forsøgt) paged in
```

```
t0 = microsec_db(); // “God” adgang  
for (int r = 0, j = 0; j < m; r += n, ++j)  
  for (int k = r, i = 0; i < n; ++k, ++i)  
    h[k] = 0;  
cout << "Sequential time: " << (int)((microsec_db() - t0) / 1000.0) << endl;
```

```
t0 = microsec_db(); // “Dårlig” adgang  
for (int i = 0; i < n; ++i)  
  for (int j = 0; j < m; ++j)  
    h[j * n + i] = 0;  
cout << "Stride time: " << (int)((microsec_db() - t0) / 1000.0) << endl;
```

Og med C++:

```
template <class T> class arrT
{
public:
    arrT(int row, int col) : rnum(row), cnum(col), data((row != 0 && col != 0) ? new T[row * col] :
0) {}
    ~arrT() { if (data) delete [] data; }
    inline T* operator[](int i) { return data + cnum * i; }
    // inline T const*const operator[](int i) const {return data + cnum * i; }
private:
    arrT& operator=(const arrT&);
    const int rnum, cnum;
    T * data;
};
```

```
arrT<int> H(n, m); // "God" adgang – bemærk i column-order
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
        H[i][j] = 0;
cout << "H time: " << (int)((microsec_db() - t0) / 1000.0) << endl;

for (int j = 0; j < m; ++j) // "Dårlig" adgang
    for (int i = 0; i < n; ++i)
        H[i][j] = 0;
cout << "Stride-H time: " << (int)((microsec_db() - t0) / 1000.0) << endl;
```

Målinger (med en 7000 \* 2000 matrice (ca. som SW-matricerne)) – på min 4core maskine (som har "billig" discount ram):

```
[lob@qp64 SW]$ ./tst 3 7000 2000
memset time:      17
Sequential time:  23
Stride time:      5521
H time:           21
Stride-H time:    5506
```

Altså en faktor 240 !!! i hastigheds forskel.

Det er også sjovt at iagtagte at det "pæne" (syntaktisk sukker ;) ) C++ array er mindst lige så hurtigt som almindelig C.

På en serverplatform får jeg:

```
[root@svr17 ~]# ./tst 3 7000 2000
memset time: 17
Sequential time: 30
Stride time: 323
H time: 26
Stride-H time: 333
```

En del bedre, men stadig over en faktor 10.

Programmet tester også en række andre “optimerede” måder (loopunrolling, større datablokke, ...)  
Alle med nogenlunde samme resultater...

<http://midibel.com/tst.cpp>

## Appendix III

Lidt snak om denne slags intelligente netværk af simple-agenter...  
Er det ikke lidt det samme som Neurale Netværk  
Er det ikke lidt det samme som ....  
Er det ikke lidt det samme som monte carlo metoder.  
Som alle er det det samme som adaptive filtre!!!

(og vi vil hurtigt finde på “smartere” træningsmetoder fordi vi ikke har tålmod – eller at det ikke virker...)

## Appendix IV

Flere billeder fra AntHill...

