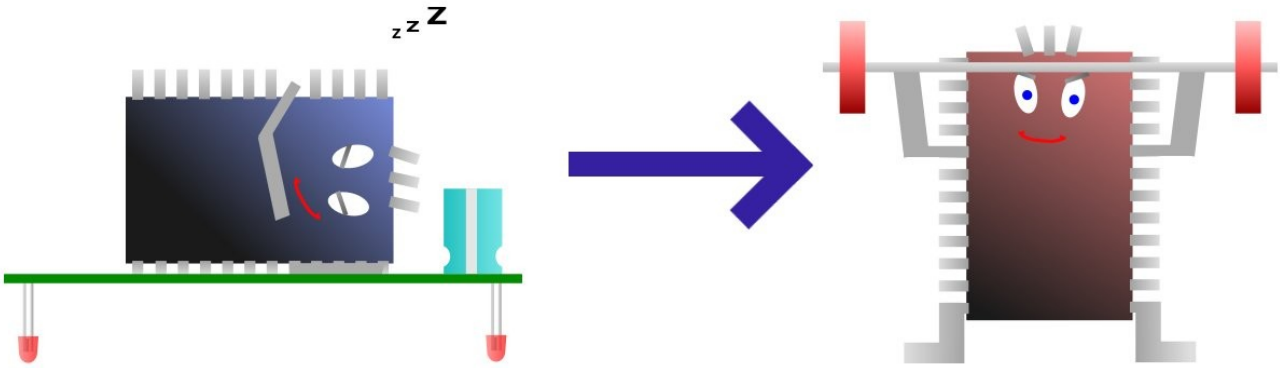
CHALMERS



Parallellisering av algoritmer för flerkärniga processorer

Kandidatarbete inom Data- och informationsteknik

GUSTAF HALLBERG
PER HALLGREN
WUILBERT LOPEZ

Institutionen för Data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2010
Kandidatarbete/rapport nr 2010:001

Abstract

Rewriting programs for computers with multiple cores has been a problem for as long as this type of computer has been available. The parallellisation has mostly been done by programmers already familiar with the code base and the algorithms used. This can be troublesome if the original programmers have left the company or are otherwise unavailable.

Because of this we have worked on parallellisation within OpenFOAM, a free, opensource software package used for computations and simulations in chemistry and physics.

Working on OpenFOAM has shown us the importance of detaild study and analysis of the code and inner workings of the software. This is necessary not only to show where parallellisation is possible put also where it is profitable.
We have also seen how important code structure and documentation are, seeing the relations between varaiables and data structures is vital for the programmer working with your code.

The report also suggests a few tools that would be very useful for profiling and analysing code and some coding standards that helps in rewriting and restructuring programs.

Sammanfattning

Att skriva om programkod för datorer med flera processorkärnor har varit ett problem ända sedan denna typ av datorer uppstod. Parallellisering har oftast utförts av programmerare som redan har god kännedom och förståelse för koden de arbetar i, detta beror dock starkt av att denna typ av programmerare finns tillgänglig.

För att undersöka möjligheterna att parallellisera kod man inte tidigare arbetat med har vi parallelliserat delar av OpenFOAM för att få en uppfattning om vilka problem och möjligheter som uppstår.

OpenFOAM är ett ramverk för att konstruera och definiera simuleringsmiljöer, och används bland annat för att simulera kemiska och fysiska processer. Att parallellisera OpenFOAM har visat på vikten av att kunna profilera och analysera koden i detalj för. Detta krävs för att dels se var det går att parallellisera och var det kan ge goda resultat.

Vi såg även hur viktigt det är med struktur och dokumentation i koden, att parallellisera kod där variablernas funktion inte dokumenterats väl ger mycket dåliga resultat.

Rapporten redogör för de verktyg som saknats under arbetets gång och de kodstandarder man bör följa för att förenkla analys och förståelse av kod i senare skeden.

Innehållsförteckning

1. Introduktion

Projektet började med en hel del instudering och efterforskning för att bättre kunna behandla ämnesområdet. Efterforskningar gjordes bland annat kring parallelliseringsramverk och profileringsverktyg. Instuderingen syftade till att ge en bättre förståelse för parallellprogrammering och en närmare bekantskap med programmeringsspråket.

Slutfasen av projektet har i samråd med Volvo Technology varit en ansträngnig att parallellisera OpenFOAM. Det är en programvara som kan hantera godtyckliga simuleringsproblem gällande flödesdynamik. Programmet utvecklas av företaget OpenCFD vars affärsidé är att sälja service och utbildning i OpenFOAM, samtidigt som man låter källkoden vara öppen.

1.1 Bakgrund

Sedan datorn uppfanns har dess utveckling som bekant följt Moores Lag: ”Antalet transistorer som kan placeras på en yta fördubblas vartannat år.” Detta har inneburit en motsvarande prestandaökning för processorer och andra datorkomponenter.

I många år har denna ökning hållit i sig och datorernas hastighet har ökat exponentiellt. I början på 2000-talet har dock denna trend stött på ett hinder, istället för att utrymmet på kiselchipet hindrar prestandaökning är det värmeutvecklingen från komponenterna som stoppar utvecklingen. Detta är ett problem som varit svårt att avhjälpa med konventionella tekniker och man har därför varit tvunget att ta fram nya lösningar.

Istället för att öka antalet transistorer i varje processor har många tillverkare valt att sätta flera processorer i samma dator. Dessa delar sedan på minne och andra resurser i datorn men kan arbeta med varsin uppgift parallellt.

I servrar och stordatorer för forskning har detta realiserats genom flera helt självständiga processorer med egna cacheminnen och kommunikationsvägar till resten av datorn. I persondatorer är det vanligaste istället att man istället har flera processorkärnor, den del av processorn som utför själva beräkningarna, som delar på cacheminne och kommunikationsmöjligheter.

1.1.1 Parallellismens problematik

En vanlig enkärnig dator kan bara utföra en beräkning i taget, men genom att den arbetar så fort kan den rotera olika uppgifter så att den verkar arbeta med dem parallellt. Detta gör att du kan leva i illusionen av att Spotify spelar musik samtidigt som Firefox laddar en hemsida, trots att all kod körs sekventiellt.

En flerkärnig dator kan däremot utföra flera beräkningar samtidigt. Parallellt exekverad kod introducerar en osäkerhetsfaktor i och med att instruktionernas ordningsföljd inte är deterministisk på samma sätt som när koden körs på den enkärniga datorn. Varje förändring från seriellt exekverad kod till parallellt exekverad kod måste därför kontrolleras noga.

1.2 Syfte

Projektet har syftat till att undersöka möjligheterna för mjukvaruutvecklare, som inte tidigare specialiserat sig i ämnet, att parallellisera kod skriven av någon annan. Detta för att kunna sammanställa tankar, problem och eventuella tips till nästa generations utvecklare.

1.3 Problemformulering

För att ge relevanta och konkreta exempel på arbetsgången i ett parallelliseringsarbete har OpenFOAM parallelliserats. Under arbetet ämnades besvara frågor så som:

- Vilka är de huvudsakliga problem eller principer involverade i parallellisering?
- Finns det möjlighet att formulera en generell guide för parallellisering?
- Vilka verktyg skulle behöva utvecklas för att göra det lättare att parallellisera?
- Vilken del av arbetet är mest tidskrävande?
- Hur långt klarar man sig utan förståelse för själva algoritmen som koden implementerar?

1.4 Fokus och avgränsningar

Rapporten fokuserar på mjukvara och kommer inte diskutera hårdvara i andra syften än för att upprätthålla en korrekt och relevant diskussion kring prestanda - som ju faktiskt är själva syftet med att parallellisera. För att göra detta måste vissa termer så som cache- och RAM-minnen, processorkärnor och registerhantering vid trådväxling belysas.

Rapporten avser att ge tankar rörande parallellisering av kod utan att programmeraren behöver ha en djupare förståelse kring problemet. Därmed menas inte att ge direkta riktlinjer, och långt ifrån en komplett lösning. Problem och möjligheter kommer presenteras, men ingen totallösning kommer att ges.

Rapporten behandlar ej parallelliseringsarbete generellt, utan presenterar istället det arbete som under projektets gång utförts i OpenFOAM för att ge exempel på olika problem och lösningar specifika för just dess kodbas. De resultat som uppnås och de motgångar som uppstår kommer sedan användas för diskussion som kan appliceras på mer generella problem.

1.5 Metod

Valet av programmeringsspråk föll på C och C++. Dessa två språk är nära besläktade och kompileras till maskinkod. Jämfört med ett språk som Java som körs på en virtuell maskin ger detta bättre prestanda vilket gör att C och C++ ofta används till beräkningsintensiva uppgifter, just den typ av uppgifter där prestandaökningar genom parallellisering är mest intressant.

Detta val avklarat måste ett ramverk för parallellisering väljas. OpenMP valdes eftersom det ger ett tydligt och lättförstått stöd till programmeraren, det krävs endast några få rader kod för att parallellisera ett större stycke. Eftersom parallelliseringen sker med hjälp av så kallade pragman, kodstycken som enbart hanteras av kompilatorn om OpenMP-biblioteket är aktiverat, är det enkelt att testa koden i seriellt läge. Detta gör det enklare att avgöra om ett problem kommer ur själva logiken eller ur det parallella flödet.

Det främsta alternativet till OpenMP var Posix Threads som är ett sätt att på väldigt låg nivå hantera trådar i UNIX-system. Detta kan ge bättre parallelliseringsmöjligheter men kräver detaljrik styrning av trådar och mycket mer arbete (Kuhn 2000).

För att profilera och analysera körning program bestämdes att använda gprof, ett profileringsverktyg utvecklat av organisationen GNU. Detta inte att förväxla med UNIX-varianten av samma program med samma namn. gprof presenterar tydligt och detaljerat vilka funktionsanrop som upptar mest tid under en körning.

1.6 Resultatsammanfattning

Det viktigaste resultatet från arbetet med OpenFOAM har varit insikten att vissa av de verktyg som används vid parallelliseringsarbete är långt ifrån fullkomliga. Det främsta exemplet är gprof, som i stor utsträckning kan ses som industristandard vid profilering. gprof var dock inte till alls stor nytta i det här fallet eftersom det ger information endast om hur funktioner behandlas. Då en hel funktion inte alls behöver parallelliseras, utvecklades ett eget verktyg för att tillhandahålla profilering kring egendefinierade sektioner.

Andra viktiga slutsatser har varit de svårigheter som finns vid uppskattning av en möjlig tidsvinst av parallellisering, och att i seriell kod hitta stycken som är parallelliseringsvänliga.

Ett av de största problemen som uppstått under projektets gång har varit sporadiska skillnader vid identiska testkörningar. Dessa uppstår inte på grund av något fel utan kan härledas till att vi har operativsystem och många andra processer körandes i bakgrunden. Det är fördelaktigt att kunna köra testprogram under en tidsrymd av ett fåtal sekunder, men görs detta kommer variationen att vara så stor att en uppsnabbning blir omöjlig att mäta.

1.7 Disposition

Rapporten kommer under kapitel 2. Teori att behandla de olika programvaror som använts för att utföra parallelliseringsarbetet. Kapitlet behandlar även OpenFOAM till sådan grad att läsaren skall kunna ta till sig resultat och diskussion på ett givande sätt.

Under kapitel 3. Resultat presenteras först projektets arbetsgång och sedan de resultat projektet slutligen resulterade i.

Kapitel 4. Diskussion bearbetar de resultat som presenterats i det föregående kapitlet. Här ledes läsaren in i en argumentation till varför nya verktyg bör tas fram och hur dessa skall fungera.

2 Teori

För att utföra parallellprogrammering behövs djup analys och stor förståelse och ofta även analys av stora mängder kod. Därför är goda verktyg ett måste för att utföra sådant arbete. Dessutom är parallellisering en komplicerad process som kräver att man behärskar många tekniker inom ämnesområdet. För att göra paralllellprogrammering hanterbart finns således programvara för att underlätta programmeringen. I det här kapitlet behandlas de verktyg som vi använt oss av under arbetet. Efter kapitlet skall läsaren ha en robust grund för att ta till sig resultaten som presenteras i nästa kapitel.

2.1 OpenMP

OpenMP är ett parallelliseringsramverk för språken C, C++ och Fortran, som underlättar många av de mer komplicerade uppgifterna under ett paralllelliseringsarbete.

> *“OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.”* (The OpenMP Architecture Review Board 2010)

En viktig insikt är att OpenMP inte är en produkt, utan endast en specifikation. Det är upp till varje kompilator för sig att implementera OpenMP. OpenMP-specifikationen utvecklas och godkänns av OpenMP Architecture Review Board, som består av representanter från många IT-jättar så som Microsoft, IBM, Cray, HP, AMD, Intel, Texas Instruments och Oracle. (The OpenMP Architecture Review Board 2010)

OpenMP opererar på enskilda kodblock, och inte på ett helt program. Ett parallellt block har ett tillhörande så kallat *lag* av trådar, som utför den inneslutna koden. Om inga direktiv ges vid skapandet av blocket, kommer samtliga trådar i laget att utföra koden. Det är lätt att styra hur trådar skall hantera kod inom ett parallellt

block genom att skapa kontrollblock inom det parallelliserade blocket eller ange direktiv i det parallella blockets huvud. Kontrollblock kan till exempel vara till för att synkronisera eller dela upp arbetet (Chapman B 2008).

För C och C++ nyttjar sig OpenMP av förkompileringsdirektivet pragma för att skapa och styra sina parallelliserade block. Pragma är en förkortning av pragmatisk och är till för att begära en kompilator-specifik åtgärd. Användantet av pragma-direktivet är en stor fördel då kompilatorn ignorerar dessa pragmatiska direktiv om den inte vet vad de innebär (Internation Standardization Organisation 1998). Således kan man kompilera kod som innehåller OpenMP-instruktioner på en kompilator som inte stödjer OpenMP och fortfarande få ett fungerande program som resultat.

2.1.2 Uppgiftsuppdelande block — *"*Worksharing*"*

En av de viktigaste egenskaperna med OpenMP är att dess möjlighet att tillhandahålla en väl skalande parallellism (Kuhn 2000). Detta genom att de tidigare nämnda uppgiftsuppdelande kontrollblocken, och då specifikt möjligheten att låta varje iteration av en for-loop skötas parallellt. Se de två nedanstående kodstyckena som exemplifierar detta.

```
#pragma omp sections
{
      #pragma omp section
      {
            // Work
      }
      #pragma omp section
      {
            // Work
      }
}
```

Listning 1. Uppgiftsfördelning genom sektionsblock

```
#pragma omp for
for(int i=0; i<N; ++i)
{
      // Work
}
```

Listning 2. Uppgiftsuppdelning genom att dela upp en for-loop

Antag att tidsåtgången för trådskapande och synkronisering är försumbar samt att arbetet som utförs i varje block är exakt lika tidskrävande. Då skulle koden från listning 1 köras dubbelt så fort på en multicore-processor som en enkärnig processor. Koden från listning 2 skulle dock resultera i ett program som körs C gånger snabbare, där C är antalet processorkärnor på den arkitektur man exekverar programmet.

Då antalet trådar ej är kopplat till programmet utan till arkitekturen den körs på kan man använda koden från listning 2 på vilken dator som helst och förvänta sig

en uppsnabbning med faktor C. Dock förutsatt att for-loopen utför ett antal iterationer är jämnt delbart med antalet processorkärnor i just den arkitekturen.

2.2 Profilering

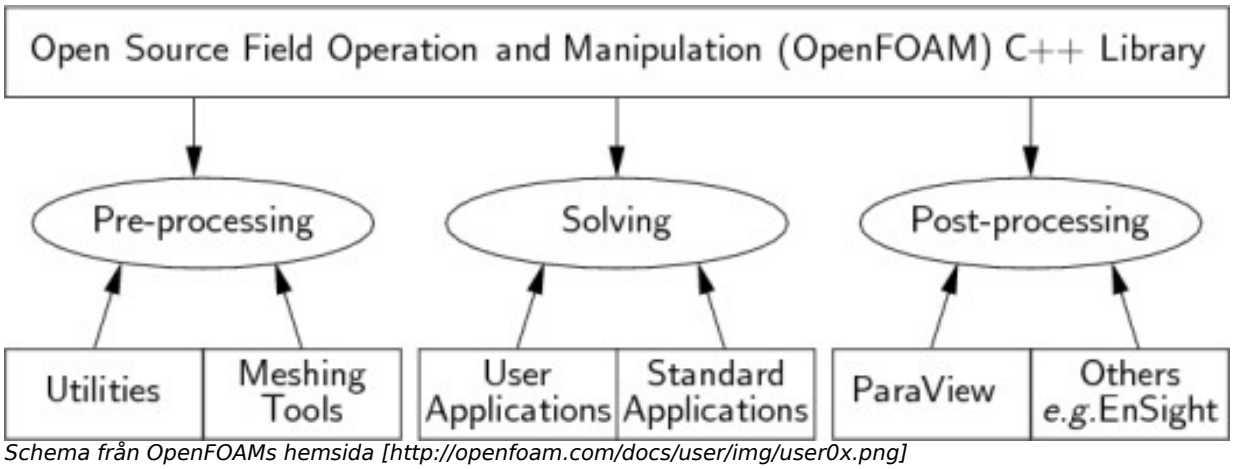
Profilering motsvarar dynamisk kodanalys. En insikt i vad detta innebär fås lättast genom att betrakta motsatsen - statisk kodanalys. Satisk kodanalys innebär att man helt enkelt analyserar koden, och betraktar hur programfödet ser ut. Dynamisk kodanalys innebär alltså att man *inte* tittar på koden, utan bara betraktar hur programmet beter sig när det faktiskt är under exekvering.

För att underlätta profileringsarbetet har gprof använts. Verktyger är enkelt, snabbt och har väldigt många inställningar för att finjustera nogrannhet. Programmet mäter tidsåtgång och antal anrop för varje funktion, samtidigt som den spara information om anropsgrafen.

2.3 OpenFOAM

OpenFOAM är ett ramverk för att konstruera och definiera simuleringsmiljöer, och tillhandahåller verktyg för att skapa representationer av den fysiska miljön programmet skall köras i och för att visualisera utdata. Dessutom ges goda möjligheter att representera en fysikalisk eller kemisk process genom att skapa s.k. lösare.

En fysikalisk eller kemisk process kan givetvis ske under många olika förhållanden och i många olika miljöer. Därför kan man med OpenFOAM lätt konfigurera flera olika saker inför en simulation. Förutom att definiera den fysiska miljön och vad som påverkar den, kan man även finjustera med hur stor precision beräkningar under simulationen skall utföras.



Schema från OpenFOAMs hemsida [http://openfoam.com/docs/user/img/user0x.png]

Arbetsgången för OpenFOAM följer figuren ovan. Rapporten kommer att behandla simuleringsfall och lösare, och utelämnar allt förarbete och efterarbete.

Lösare i OpenFOAM är program som löser specifika problem inom flödesdynamiken. Dessa använder sig av verktyg som tar hand om

datamanipulering och algebraiska beräkningar. För att programmera lösare i OpenFOAM är att det är viktigare att förstå sig på flödesdynamik än att vara en duktig programmerare. Grundkunskaper i programmering är det enda som krävs (OpenCFD 2010a).

```
1. Initiera gränsförhållanden
2. Lös ut hastigheten ur den diskretiserade rörelsemängdsekvationen
3. Beräkna massflödena i modellens celler
4. Lös tryckekvationen
5. Korrigera massflödena i modellens celler
6. Korrigera modellens hastigheter baserat på den nya trycket
7. Uppdatera gränsförhållandena
8. Upprepa tills konvergens uppnåtts
```

Listing 3. Översikt av SIMPLE-algoritmen (Passalacqua 2007)

```
while (runTime.loop())
{
      ...
      #include "Ueqn.H"               // 2&3
      #include "pEqn.H"               // 4&5
      ...
      #include "convergenceCheck.H"   // 8
      ...
}
```

Listning 4. Utdrag ur simpleFoam.C (OpenCFD 2009a)

Det är inte direkt uppenbart hur koden i listning 4 korresponderar till algoritmen i listning 3. Analys av koden behövs för förståelse men läsaren behöver nu bara bekanta sig med den allmänna formen på en lösare. While-loopen på översta raden återfins i någon form i samtliga lösare, och benämns fortsättningsvis tidsloopen. Ekvationsfilerna UEqn.H och pEqn.H sköter ekvationslösning åt oss. I dessa finns en funktion benämnd solve(). Alla ekvationsfiler behöver anropa solve() för att lösa den ekvation de representerar.

Lösare använder sig av bibliotek. Beroende på den modell man vill simulera väljer man bibliotek med de nödvändiga funktionerna (OpenCFD 2010a). Det är t.ex. skillnad mellan en termofysisk simulering och en rent mekanisk simulering.

Ett exempel på en funktion i ett bibliotek är solve(). Den får som argument ekvationer (i form av matriser) som beskriver någon egenskap i modellen (t.ex. hastighet och tryck) och beräknar lösningen (OpenCFD 2010a). Nedan visas två ekvationer och deras motsvarighet, skriven med OpenFOAMs notation.

```
∇2 φ = f
      <=>
solve(fvm::laplacian(phi) == f)
```

Listning 5. Poissons ekvation (OpenCFD 2009b)

```
∂φ = κ∇2 φ
      <=>
solve(fvm::ddt(phi) == kappa*fvm::laplacian(phi))
```

Listning 6. En diffusionsekvation (OpenCFD 2009b)

Läsaren behöver inte förstå ekvationernas innebörd utan bara bekanta sig med representationer av ekvationer i OpenFOAM.

Lösare körs alltid i en mapp som representerar ett simuleringsfall. Ett simuleringsfall är helt enkelt en mapp som innehåller konfigurationsfiler. I dessa filer bestäms ett flertal saker. Av intresse för denna rapport är möjligheten att styra den simulerade tiden, förändra ekvationslösarnas tolerans, ange vilka ekvationslösare som ska användas samt förfina modellens upplösning (OpenCFD 2010b).

3. Resultat

Det praktiska arbetet har nästan uteslutande berört OpenFOAM och det är endast detta som tas upp i rapporten.

3.1 Parallellisering av OpenFOAM

OpenFOAM är mycket modulärt och parallellisering i de olika delarna medför väldigt skilda resultat. En bedömning av hur förändringarna i koden påverkar projektet i helhet behövs för flertalet aspekter för att avgöra vilken del som är lämpligast att parallellisera.

OpenFOAMs lösare är anpassade för att författare av dessa inte ska behöva några vidare programmeringskunskaper för att skriva dem. Således medför parallelliseringsarbete i dessa möjligen besvär för eventuella fysiker, matematiker eller kemister som uppdaterar applikationslösare. Dock kan det diskuteras huruvida ett fysikalisk beskrivning av ett flöde någonsin skulle behöva.

Att parallellisera en lösare och dess ekvationsfiler ger endast effekt för just den lösaren, och möjligen några ytterligare som brukar dennes ekvationer. Effekten av parallellisering i en lösare blir hur som helst begränsad till mycket få applikationer. Om en viktig kärnmekanism däremot skulle parallelliseras skulle en uppsnabbning av samtliga lösare uppnås.

Kontinuitetsproblem kan uppstå vid framtagandet av en parallelliserad kärnmekanism och därmed en ny version av OpenFOAMs kärna. När OpenCFD sendan släpper ny version skulle det innebära att den parallella versionen ersätts och skrivs över. OpenCFD uppdaterar dock inte fysikaliska omständigheter och således behövs inga förändringar i lösare vid uppdatering av OpenFOAM. Därför skulle inte dessa problem uppstå vid parallellisering av en lösare och man slipper helt kontinuetitsproblem vid framtagandet av en parallelliserad version av en lösare.

För projektets ändamål valdes att parallellisera lösaren dieselFoam. Detta var grundas på att parallellisera en lösare är mindre komplext än att parallellisera en algoritm i en kärnmekanism, som till exempel gauss-elimination, matrismultiplikation, etc.. Andra orsaker till att valet parallellisera lösare var att det tar betydligt kortare tid - ca 30 sekunder istället för ett par timmar - att kompilera en lösare än att kompilera om kärnan och att de ovan nämnda kontinuetitsproblemen undviks.

3.1.1 Analys

För att bedömma vilken form av parallellisering som skulle ge mest effekt är profilering och analys av koden en självklarhet. Parallellisering av en lösare är problematiskt då den är ämnad att lösa ett fysikaliskt problem, och därmed förenklat kan beskrivas som en serie uträkningar. Dessa uträkningar måste ofta utföras i korrekt ordning. Det är nu av värde att titta på koden för dieselFoam:

```
while(runTime.run()) //tidsloopen
{
      #include "rhoEqn.H"
      #include "UEqn.H"

      for (label ocorr=1; ocorr <= nOuterCorr; ocorr++)
      {
            #include "YEqn.H"
            #include "hEqn.H"

            // --- PISO loop
            for (int corr=1; corr<=nCorr; corr++)
            {
                  #include "pEqn.H"
            }
      }
}
```

Listning 7. Förenklad version av dieselFoam.C (OpenCFD 2009a)

Listning 4 är ej ett fullständingt utdrag ur dieselFoam.C utan har kortats ner för att öka läsbarheten. Den verkliga koden står att finna i appendix tillsammans med innehållet från varje ekvationsfil. Vid en första anblick lägger man märke till den nästlade for-loopen. Iterationer av detta slag har stor potential vid parallellisering. Men det är inte tillräckligt att identifiera potential då den potentialen är beroende av vad tidsanalysen kommer fram till. Således tar vi analysverktyg till hjälp.

Analys med hjälp av gprof gav oss en tabell med funktioner sorterade efter tidsåtgång i procent. Samtliga av dessa funktioner är kärnmekanismer och inte av intresse då strategin var att fokusera på en lösare istället för på kärnmekanismer. Dessutom tog dessa funktioner upp stor del av tiden därför att de anropas ofta, inte för att varje enskild körning av funktionen tar mycket tid.

Då gprof inte kunde producera de data som behövdes användes istället tidtagning kring så gott som varje instruktion i dieselFoam.C och samtliga ekvationsfiler. De

viktigare delarna av resultatet från tidtagning under en iteration av tidsloopen kan beskådas nedan.

```
What                                        Time            Count
UEqn.H..................................... 3.9358          1
UEqn.H > UEqn()............................ 2.2435          1
UEqn.H > solve()........................... 1.6923          1
YEqn.H..................................... 2.0518          1
YEqn.H > for............................... 2.0381          1
YEqn.H > for > if > solve()................ 2.0067          4
hEqn.H..................................... 1.3548          1
hEqn.H > solve()........................... 0.69628         1
pEqn.H > else > for........................ 1.5857          2
pEqn.H > else > for > pEqn.solve()......... 1.1958          2
pEqn.H > if/else........................... 2.3624          2
rhoEqn.H > for............................. 0.062782        3
rhoEqn.H > solve()......................... 0.28001         3
```

Listning 8. Tidtagning av dieselFoams olika delar

Av den data som presenteras i lisning 8 kan man utläsas hur många anrop som görs för varje iteration i tidsloopen, och således få en uppfattning av hur god skalaning parallellisering av olika delar skulle ge. Ett högt antal iterationer ger god skalning. Därför inser man att god skalning förmodligen inte går att uppnå genom parallellisering av denna lösare.

3.1.2 Genomförande

Efter att ha fastställt hur stor tidsåtgång varje del av koden kräver, ses till vilket beroende som finns mellan olika delar av koden. Givetvis är det första stället att leta på den nästlade for-loopen. Se till koden längre upp. Dock skulle parallellisering av denna loop skulle ge stora bekymmer, då varje ekvationsfil uppdaterar en eller flera globala variabler.

Men som man kan se i listning 7 kan det finnas goda parallelliseringsmöjligheter i ekvationsfilerna. Loppar hittas i ekvationsfilerna rhoEqn.H, YEqn.H samt pEqn.H och dessa måste alla analyseras för att bedömma lönsamheten av att parallellisera dem. rhoEqn.H kan snabbt försbises, då denna ekvationsfil uppdat så pass lite tid, samt att loopen i den upptar en bråkdel av dess tidsåtgång, se listning 8. Då kvarstår YEqn.H och pEqn.H.

YEqn.H borde kunna parallelliseras med framgång, dock är antalet iterationer i YEqn.H är lika med antalet kemiska substanser som ingår i simuleringsmiljön. I vårat fall var detta begränsat till endast fyra, vilket inte ger god skalning. Parallellisering i YEqn.H kan möjligen vara exceptionellt i en simuleringsmiljö där man har betydligt fler kemiska substanser.

pEqn.H är nästlad i båda de två looparna som finns i dieselFoam.C, och innehåller även själv en loop. Detta kan finnas mycket intressant, men då ekvationen uppdaterar den globala variabeln p i sin innersta loop, skulle parallellisering av denna fil bli knapphändig.

Endast ett fåtal stycken utan beroenden finnes, och inga uppenmbara parallelliseringsmöjligheter hittas. Dock inser man att just solve()-funktionen upptar en stor del av varje körning, och då anropen sker på ett något mystiskt sätt där man brukar överlagrade operatorer, valdes att undersöka detta närmare.

3.1.2.1 Uppdelning av solve():s parametrar

Solve()-funktionen anropas med argument i form av ekvationer. Dessa ekvationer byggs upp av mindre delar som beräknas var för sig. I originalimplementationen beräknades ekvationsdelarna inom solvefunktionens anrop, detta visade sig vara en möjlighet till parallellisering.

Efter att ha analyserat beräkningarna som gjordes för att skapa ekvationsdelarna fastslogs det att de inte modifierade några gemensamma variabler utan bara den variabel de var till för att räkna ut. Genom att flytta ut beräkningen av ekvationsdelarna ur solveanropet gick det att beräkna ett antal av delarna parallellt.

```
What                                        Time              Count
UEqn.H..................................... 2.4761            1
UEqn.H > UEqn.............................. 1.1755            1
UEqn.H > UEqn > left1 + right.............. 0.44766           1
UEqn.H > UEqn > left2...................... 0.64868           1
UEqn.H > UEqn()............................ 0.074448          1
UEqn.H > solve()........................... 1.3006            1
YEqn.H..................................... 2.3613            1
YEqn.H > for > if > left1 & right......... 0.24937           4
YEqn.H > for > if > left2.................. 0.37565           4
YEqn.H > for > if > left3.................. 0.41695           4
YEqn.H > for > if > solve.................. 2.3031            4
YEqn.H > for > if > solve()................ 1.2441            4
hEqn.H..................................... 1.5597            1
hEqn.H > solve............................. 0.7065            1
hEqn.H > solve > left1..................... 0.18121           1
hEqn.H > solve > left2 + left3 + right.... 0.16508           1
hEqn.H > solve()........................... 0.36016           1
```

Listning 9. Tidtagning av solve():s parametrar

Nedan följer ett fullständigt utdrag av UEqn.H, med kommentarer för att beskriva vad tidtagningen mäter. Förutom det som beskrivs i kommentarerna bör nämnas att `UEqn` syftar på den totala tiden för `UEqn > left1 + right`, `UEqn.H > UEqn > left2` och `UEqn.H > Ueqn()`.

```
fvVectorMatrix UEqn                             // Ueqn()
(
    fvm::ddt(rho, U)                            // left1
  + fvm::div(phi, U)                            // left2
  + turbulence->divDevRhoReff(U)                // left3
 ==
    rho*g                                       // right
  + dieselSpray.momentumSource()                // right
);

if (momentumPredictor)
{
      solve(UEqn == -fvc::grad(p));             // solve()
}
```

Listning 10. Utdrag av UEqn.H (OpenCFD 2009a)

3.1.2.2 Parallellisering längsmed tidsaxeln

Varje simulation i OpenFOAM utför ett visst antal tidsiterationer, vilket ger ett visst potential till parallellism. För att testa detta tillvägagångssätt användes simpleFoam, vilket är en mycket minder komplex applikationslösare än t.ex. dieselFoam. Dock visade sig resultaten långt ifrån tillräckkliga eftersom endast små delar i början och slutet av varje tidsiteration i den algoritm som simpleFoam (SIMPLE) använder kunde parallelliseras. Tidigt i varje iteration behöver man läsa från en variabel som skrivs till i slutet av föregående iteration.

```
while (runTime.loop())
{
    ...
    tmp<fvVectorMatrix> UEqn
    (
        // Använder U
        fvm::div(phi, U) + turbulence->divDevReff(U)
    );
    ...
    // Initierar U
    volScalarField AU = (*UEqn)().A();
    U = UEqn().H()/AU;
    ...
    // Uppdatering inför nästa iteration
    U -= fvc::grad(p)/AU;
    U.correctBoundaryConditions();
    ...
}
```

Listning 11. Förenklad version av simpleFoam.C (OpenCFD 2009a)

Man beräknar UEqn, som användes för att initiera U under varje iteration. Uträkning av UEqn kräver dessutom att U från föregående iteration är fullständigt utförd. Eftersom U inte beräknas förän i slutet av varje iteration och UEqn beräknas i ett tidigt skede av varje iteration. Finns inte mer än ett fåtal iterationer att tjäna på parallellisering längs tidsaxeln i fallet av simpleFoam.

Det finns dock ett möjligt potential; om man finner en lösare där varje instruktion i algoritmen endast påverkar någon eller några nästkommande instruktioner. Då projektet var under stor tidspress hann inte fler lösare undersökas närmare för detta ändamål.

3.2 Resultatredovisning

Tester har gjorts med det färdiga Aachenbomb-exemplet som följer med OpenFOAM-distributionen. Testerna har utförts i Ubuntu 10.04 körandes på en maskin med Intel Core2 Quad och 4 GB RAM-minne.

För att ekvationerna som de olika anropen till solve() löser ska ta en icke försumbar tid att ställa upp, måste problemet man försöker simulera vara tillräckligt stort. Samtidigt för inte körningen ta allt för lång tid för att testen skulle hinna bli behandlade i projektet. För att öka problemstorleken har modellens storlek ökats, noggrannheten för varje ekvation ökats och därtill har antalet tidsiterationer minskats

Ett stort problem med att mäta tidsåtgång för OpenFOAM är att programmet hela tiden skriver ut data till både skärmen och till diverse utfiler. I/O-operationer tar en något sporadiskt tid att utföras och är beroende av många saker i systemet. Detta har varit ett stort problem under projektet, och för att få ett trovärdigt resultat har flera mätningar gjorts, och ett medeltal tagits fram.

```
What                                                Time
UEqn.H.............................................. 92.388
UEqn.H > UEqn....................................... 30.282
UEqn.H > UEqn > left1 + right...................... 9.9924
UEqn.H > UEqn > left2............................... 18.507
UEqn.H > UEqn()..................................... 1.6698
UEqn.H > solve()..................................... 62.106
```

Listning 12. Tidtagning av UEqn.H med en aktiv tråd

```
What                                                Time
UEqn.H.............................................. 83.861
UEqn.H > UEqn....................................... 20.653
UEqn.H > UEqn > left1 + right...................... 15.119
UEqn.H > UEqn > left2............................... 18.701
UEqn.H > UEqn()..................................... 1.7938
UEqn.H > solve()..................................... 63.208
```

Listning 13. Tidtagning av UEqn.H med två aktiva trådar

Listning 12 och Listning 13 visar körningar med endast UEqn.H parallelliserad. Listning 12 visar resultat med endast en tråd, och listning 13 visar tiden för körning med två trådar.

4 Diskussion

Under arbetets gång har många problem uppstått. Utmaningar vid kodanalys samt otilfäckliga profileringsverktygs har varit de främsta. Många av de problemen går att till stor grad motverka genom bättre verktyg. Avsnitten nedan kommer att behandla de största problemen under projektets gång och ge förslag på hur nya verktyg skulle kunna tänkas fungera.

4.1 Problematik under arbetet med OpenFOAM

Det mesta av tiden under parallelliseringsarbete i allmänhet gått åt till att analysera kod och identifiera beroenden i den. I en välstrukturerad kod ska dessa beroenden vara relativt lätta att finna. Speciellt om koden är väldokumenterad. OpenFOAM har en ovanlig kodstruktur, särskilt på lösarnivå eftersom den är tänkt för fysiker och inte programmerare. Detta gör att koden är svår att analysera eftersom den bryter mot många grundläggande principer för hur programkod bör skrivas.

Exempelvis finns det ett antal globala variabler som används utan dokumentation. Denna typ av kod gör det mycket svårt att analysera beroenden och utröna vilka kodstycken som modifierar samma platser i minnet eftersom den variabel som modifieras kan finnas i en importerad fil, ett stort antal steg längre in i programmet.

Utvecklarna har också valt att överlagra ett antal operatorer så som == och *, i vissa fall är det motiverat. Till exempel när de använt det för att implementera matrismultiplikation. I andra fall är det dock otydligt och svårläst och överlagrandet av == verkar ha gjorts enbart för att få koden att likna en matematisk ekvation. Att förenkla koden för matematiker, fysiker och kemiker gör den mer svårhanterlig för programmerare och är en stor anledning till att det eventuellt hade varit bättre att parallellisera i kärnan istället.

Det absolut viktigaste för en programmerare är att programmet ger korrekt resultat. En fördel med OpenMP är att man kan utföra en förändring i taget (se 4.2 Utvärdering av verktyg). D.v.s. man parallelliserar t.ex. en for-loop av fyra möjliga och analyserar resultatet innan man förändrar nästa for-loop.

Resultatet från en simulering i OpenFOAM (se 2.3 OpenFOAM) sparas i mappar som representerar tidssteg i simuleringen. För att verifiera att en seriellt exekverad simulering gav samma resultat som en parallelliserad version hashade vi bägge resultaten och jämförde.

Som uppdagat i avdelning 3.1.2.1 parallelliserades OpenFOAM under projektet genom att låta olika delar av en funktions parametrar beräknas parallellt. Detta tillvägagångssätt är förmodligen inte att föredra i nästan alla andra projekt, då det

är ovanligt att krävande operationer krävs för att beräkna en funktions parametrar. Dock kan denna insikt vara av väldigt god nytta i andra programspråk. Ett exempel är MATLAB, där man ofta tillåter programmeraren att skriva matriser eller vektorer såväls som skalärer och sedan upprepar funktionen för varje element i matrisen eller vektorn.

4.2 Utvärdering av verktyg

gprof är något av industristandard när det kommer till profilering. Dock utför det inte alls det arbete som behövs för att utföra parallellisering i en lösare eftersom den mäter tidsåtgången vid funktionsanrop. Det vore av större intresse att mäta tidsåtgång för kodblock. Tidsåtgång per funktionsanrop kan till vara väldigt bra när man skall optimera ett program. Dock är det inte alls lika relevant när man vill parallellisera ett program, se Appendix A.

Ett profileringsverktyg som operar på annan basis än funktion-för-funktion, som gprof gör, skulle passat projektet mycket bättre. Ett sådant verktyg skulle gärna profilera tiden kring varje block, såväl som varje rad. På så vis skulle det göra nästan precis det jobb gprof gör, fast med större detaljrikedom.

Valet av OpenMP har fungerat bra under projektets gång. Det har varit lätt att använda och levt upp till de förväntningar som ställts på det. Det primära alternativet, POSIX Threads hade eventuellt kunnat ge bättre resultat, men med tanke på den begränsade tidstillgången hade det antagligen inte hjälpt nämnvärt. Vid en eventuell parallellisering av kärnan hade den lägre overheaden kunnat ge bättre prestanda, men på den nivå som projektet rört sig hade resultatet knappast förbättrats i någon större utsträckning.

Att verifiera huruvida ett parallelliserat program ger rätt utdata är ofta krångligt och tidskrävande. Detta skulle kunna implementeras i någon sorts parallelliseringsramverk, på så sätt att man markerar vad man vill kontrollera och när, den kan då samla all dessa värden, hasha resultatet och spara undan det i en specifik mapp för körningen. Denna hash kan man sedan spara undan och jämföra med resultat från andra körningar.

- Förslag på verktyg:
    - profilering block-by-block
    - Variabel-relations-verktyg?

4.2.3 Approach efter hårdvara?

Cahcemissar är mindre kritiska i vissa system, osv.

5 Referenser

Chapman B, Jost G och Van der Pas R. (2008) *Using OpenMP*. MIT Press: Cambridge

Internation Standardization Organisation och International Electrotechnical Commission (1998) *Programming languages — C++*. ISO/IEC 14882:1998(E), American National Standards Institute, New York. Tillgänglig: http://www-d0.fnal.gov/~dladams/cxx_standard.pdf (2010-05-16)

Kuhn B., Petersen P. och O'Toole E. (2000) OpenMP versus Threading in C/C++. *Concurrency*, vol 12, nr 12, ss. 1165-1176. Tillgänglig: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.5676&rep=rep1&type=pdf (2010-05-16)

OpenCFD Ltd. (2009a) *OpenFOAM-1.6*. http://downloads.sourceforge.net/foam/OpenFOAM-1.6.General.gtgz?use_mirror=mesh (2010-05-17)

OpenCFD Ltd. (2009b) *Programmer's Guide*. http://foam.sourceforge.net/doc/Guides-a4/ProgrammersGuide.pdf (2010-05-17)

OpenCFD Ltd. (2010a) Features of OpenFOAM. *OpenFOAM® - The Open Source Computational Fluid Dynamics (CFD) Toolbox*. *http://www.openfoam.com/features/* (2010-05-17)

OpenCFD Ltd. (2010b) Lid-driven cavity flow. *OpenFOAM® - The Open Source Computational Fluid Dynamics (CFD) Toolbox*. http://openfoam.com/docs/user/cavity.php (2010-05-18)

Passalacqua A. (2007) The SIMPLE algorithm in OpenFOAM. *OpenFOAMWiki*. http://openfoamwiki.net/index.php/The_SIMPLE_algorithm_in_OpenFOAM (2010-05-10)

The OpenMP Architecture Review Board (2010) *The OpenMP API specification for parallel programming*. http://openmp.org/wp/ (2010-05-17)

Appendix A

dieselFoam.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Application
    dieselFoam

Description
    Solver for diesel spray and combustion.

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "hCombustionThermo.H"
#include "turbulenceModel.H"
#include "spray.H"
#include "psiChemistryModel.H"
#include "chemistrySolver.H"

#include "multivariateScheme.H"
#include "IFstream.H"
#include "OFstream.H"
#include "Switch.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
```

```
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "readGravitationalAcceleration.H"
    #include "readCombustionProperties.H"
    #include "createSpray.H"
    #include "initContinuityErrs.H"
    #include "readTimeControls.H"
    #include "compressibleCourantNo.H"
    #include "setInitialDeltaT.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info << "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
        #include "readPISOControls.H"
        #include "compressibleCourantNo.H"
        #include "setDeltaT.H"

        runTime++;
        Info<< "Time = " << runTime.timeName() << nl << endl;

        Info << "Evolving Spray" << endl;

        dieselSpray.evolve();

        Info << "Solving chemistry" << endl;

        chemistry.solve
        (
            runTime.value() - runTime.deltaT().value(),
            runTime.deltaT().value()
        );

        // turbulent time scale
        {
            volScalarField tk =
                Cmix*sqrt(turbulence->muEff()/rho/turbulence->epsilon());
            volScalarField tc = chemistry.tc();

            // Chalmers PaSR model
            kappa = (runTime.deltaT() + tc)/(runTime.deltaT()+tc+tk);
        }

        #include "rhoEqn.H"
        #include "UEqn.H"

        for (label ocorr=1; ocorr <= nOuterCorr; ocorr++)
        {
            #include "YEqn.H"
            #include "hEqn.H"

            // --- PISO loop
            for (int corr=1; corr<=nCorr; corr++)
```

```
            {
                #include "pEqn.H"
            }
        }

        turbulence->correct();

        #include "spraySummary.H"

        rho = thermo.rho();

        if (runTime.write())
        {
            chemistry.dQ()().write();
        }

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


// ************************************************************************* //
```

UEqn.H

```
    fvVectorMatrix UEqn
    (
        fvm::ddt(rho, U)
      + fvm::div(phi, U)
      + turbulence->divDevRhoReff(U)
     ==
        rho*g
      + dieselSpray.momentumSource()
    );

    if (momentumPredictor)
    {
        solve(UEqn == -fvc::grad(p));
    }
```

YEqn.H

```
tmp<fv::convectionScheme<scalar> > mvConvection
(
```

```
    fv::convectionScheme<scalar>::New
    (
        mesh,
        fields,
        phi,
        mesh.divScheme("div(phi,Yi_h)")
    )
);

{

    label inertIndex = -1;
    volScalarField Yt = 0.0*Y[0];

    for (label i=0; i<Y.size(); i++)
    {
        if (Y[i].name() != inertSpecie)
        {
            volScalarField& Yi = Y[i];

            solve
            (
                fvm::ddt(rho, Yi)
              + mvConvection->fvmDiv(phi, Yi)
              - fvm::laplacian(turbulence->muEff(), Yi)
              ==
                dieselSpray.evaporationSource(i)
              + kappa*chemistry.RR(i),
                mesh.solver("Yi")
            );

            Yi.max(0.0);
            Yt += Yi;
        }
        else
        {
            inertIndex = i;
        }
    }

    Y[inertIndex] = scalar(1) - Yt;
    Y[inertIndex].max(0.0);
}
```

hEqn.H

```
{
    solve
    (
        fvm::ddt(rho, h)
      + mvConvection->fvmDiv(phi, h)
      - fvm::laplacian(turbulence->alphaEff(), h)
     ==
       DpDt
```

```
     + dieselSpray.heatTransferSource()
    );

    thermo.correct();
}pEqn.H
rho = thermo.rho();

volScalarField rUA = 1.0/UEqn.A();
U = rUA*UEqn.H();

if (transonic)
{
    surfaceScalarField phid
    (
        "phid",
        fvc::interpolate(psi)
       *(
            (fvc::interpolate(U) & mesh.Sf())
          + fvc::ddtPhiCorr(rUA, rho, U, phi)
        )
    );

    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix pEqn
        (
            fvm::ddt(psi, p)
          + fvm::div(phid, p)
          - fvm::laplacian(rho*rUA, p)
         ==
            Sevap
        );

        pEqn.solve();

        if (nonOrth == nNonOrthCorr)
        {
            phi == pEqn.flux();
        }
    }
}
else
{
    phi =
        fvc::interpolate(rho)
       *(
            (fvc::interpolate(U) & mesh.Sf())
          + fvc::ddtPhiCorr(rUA, rho, U, phi)
        );

    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix pEqn
        (
            fvm::ddt(psi, p)
          + fvc::div(phi)
```

```
          - fvm::laplacian(rho*rUA, p)
         ==
            Sevap
        );

        pEqn.solve();

        if (nonOrth == nNonOrthCorr)
        {
            phi += pEqn.flux();
        }
    }
}

#include "rhoEqn.H"
#include "compressibleContinuityErrs.H"

U -= rUA*fvc::grad(p);
U.correctBoundaryConditions();

DpDt = fvc::DDt(surfaceScalarField("phiU", phi/fvc::interpolate(rho)), p);
```

rhoEqn.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 1991-2009 OpenCFD Ltd.
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Global
    rhoEqn

Description
    Solve the continuity for density.
```

```
\*---------------------------------------------------------------------------*/

volScalarField Sevap
(
    IOobject
    (
        "Sevap",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("zero", dimensionSet(1, -3, -1, 0, 0), 0.0)
);

for (label i=0; i<Y.size(); i++)
{
    if (dieselSpray.isLiquidFuel()[i])
    {
        Sevap += dieselSpray.evaporationSource(i);
    }
}

{
    solve
    (
        fvm::ddt(rho)
      + fvc::div(phi)
      ==
        Sevap
    );
}

// ************************************************************************* //
```

Appendix B

Orginal

```
#include <stdio.h>
/* Computes the length of Collatz sequences */
unsigned int step (unsigned int x)
{
    if (x % 2 == 0)
    {
        return (x / 2);
    }
    else
    {
        return (3 * x + 1);
    }
}
unsigned int nseq (unsigned int x0)
{
    unsigned int i = 1, x;

    if (x0 == 1 || x0 == 0)
      return i;
    x = step (x0);
    while (x != 1 && x != 0)
    {
        x = step (x);
        i++;
    }
    return i;
}
int main (void)
{
    unsigned int i, m = 0, im = 0;
    for (i = 1; i < 500000; i++)
    {
        unsigned int k = nseq (i);
        if (k > m)
        {
            m = k;
            im = i;
            printf ("sequence length = %u for %u\n", m, im);
        }
    }
    return 0;
}
```

Profilering via gprof

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
```

```
 55.92     68.92    68.92 39999980     1.72     3.07  nseq(unsigned int)
 43.71    122.78    53.87 1248562184     0.04     0.04  step(unsigned int)
  0.37    123.24     0.46                             main
```

Optimering

```
unsigned int step (unsigned int x)
{
    if (x % 2 == 0)
    {
        return (x / 2);
    }
    else
    {
        return (3 * x + 1);
    }
}

unsigned int nseq (unsigned int x0)
{
    unsigned int i = 1, x;

    if (x0 == 1 || x0 == 0)
      return i;
    x = step (x0);

    while (x != 1 && x != 0)
    {
        x = step (x);
        i++;
    }

    return i;
}
```

Bytes mot

```
unsigned int step (unsigned int x)
{
    return (x & 1) == 0
        ? x / 2
        : 3 * x + 1;
}
unsigned int nseq (unsigned int x0)
{
    unsigned int i = 1;
    if (x0 == 1 || x0 == 0)
      return 1;
    for(unsigned int x = step(x0);
        x != 1 && x != 0;
        x = step (x), ++i
        )
    ;
    return i;
```

```
}
```

Resultat

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 54.93     64.94    64.94 39999980     1.62     2.94  nseq(unsigned int)
 44.64    117.70    52.77 1248562184     0.04     0.04  step(unsigned int)
  0.43    118.21     0.51                             main
```

Parallellisering

```
int main (void)
{
    unsigned int i, m = 0, im = 0;
    for (i = 1; i < 500000; i++)
    {
        unsigned int k = nseq (i);
        if (k > m)
        {
            m = k;
            im = i;
            printf ("sequence length = %u for %u\n", m, im);
        }
    }
    return 0;
}
```

Bytes mot

```
int main (void)
{
    unsigned int i, m = 0, im = 0;
    #pragma omp parallel for private(i) default(shared)
    for (i = 1; i < 500000; i++)
    {
        unsigned int k = nseq (i);
        #pragma omp critical
        {
            if (k > m)
            {
                m = k;
                im = i;
                printf ("sequence length = %u for %u\n", m, im);
            }
        }
    }
    return 0;
}
```