

Projekt: Shared-Memory Parallelisierung mit OpenMP

Aufgabenstellung und Lehrziele

Aufgabenstellung

In diesem Projekt sollen Teile des Übungscodes mit Hilfe von OpenMP parallelisiert werden. Da eine Parallelisierung nur an den Stellen sinnvoll ist, die viel Rechenzeit benötigen, ist auch Profiling ein Thema. Profiling bedeutet, das Laufzeitverhalten des Codes zu analysieren.

Folgende Aufgaben sind zu lösen:

- Profiling von *cdfv* am Beispiel von *toro4.ini*
- Konzept für die shared-memory Parallelisierung mit OpenMP entwickeln
- OpenMP-Parallelisierung durchführen

Lehrziele

- Erstellen und Lesen von Profiles
- Grundlagen bei Parallelisieren kennenlernen und anwenden
- Laufzeitabhängigkeiten erkennen
- OpenMP zur Parallelisierung von einfachen *do*-Schleifen einsetzen können

Benötigte Software

- Linux
- gfortran
- gprof

1 Aufgabenteil 1: Untersuchen des Laufzeitverhaltens

1.1 Hintergrund

Zunächst müssen wir herausfinden, welche Stellen im Programm viel Rechenzeit verbrauchen. Dazu verwenden wir den einen Profiler, d.h. ein Werkzeug, mit dem das Laufzeitverhalten unseres Programms untersucht werden kann.

Für das Profiling gibt es im Wesentlichen zwei Methoden: Sampling und Instrumentierung.

- **Statistische Auswertung / Sampling:** in bestimmten Intervallen wird das Programm vom Profiler durch Betriebssystem-Interrupts unterbrochen und der Programmzähler ausgelesen. Der Programmzähler ist ein Register, das anzeigt, welche Instruktion gerade ausgeführt wird. Mit Hilfe dieser Messpunkte wird eine statistische Auswertung erstellt, in welchen Routinen wieviel Zeit verbraucht wurde.
- **Instrumentierung:** dabei werden (vom Programmierer / Compiler / Tool) zusätzliche Instruktionen hinzugefügt, um die gewünschten Informationen zu sammeln.
- **Tracing:** eine spezielle Art der Instrumentierung, bei der zusätzlich die zeitliche Abfolge der Ereignisse gespeichert wird. Dies ist vor allem bei der Untersuchung von parallelen Programmen nützlich um Wartezeiten einzelner Prozesse zu identifizieren.

Grundsätzlich gilt: die Verwendung eines Profilers verändert das Laufzeitverhalten. Beim Sampling sind die Einflüsse relativ gering. Eine Instrumentierung hingegen kann das Laufzeitverhalten stark ändern. Ein Ursache ist z.B. das Instrumentieren von sehr kleinen Routinen, die sehr oft aufgerufen werden. Der Overhead führt dazu, dass die Ausführungszeit drastisch zunimmt.

Der GNU Profiler *gprof* verwendet die statistische Auswertung. Er wird angeschaltet, indem man das Programm mit dem Compiler-Flag *-pg* kompiliert und linkt. Beim Ausführen des Programms wird eine Datei *gmon.out* erzeugt, die die gesammelten statistischen Informationen enthält. Den Inhalt dieser Datei kann man sich dann mit dem Profiler *gprof* ansehen.

1.2 Genaue Aufgabenstellung

- Öffnen Sie das Makefile und vergewissern Sie sich, dass bei den `GCC_FLAGS` der Eintrag `F90_CFLAGS` das Flag *-pg* enthält. Falls nicht, ergänzen Sie es.
- Kompilieren Sie *cfdfv*.
- Verwenden Sie das Input-File *toro4.ini* im Verzeichnis *RiemannProblems* auf einem 50x50 Gitter und führen Sie das Programm aus.

```
../.. /cfdfv_toro4.ini
```

Passen Sie bitte die Gittergröße an, wenn das Programm viel zu lange oder nur wenige Sekunden läuft.

- Geben Sie die statistischen Daten aus:

```
grof_././cfdv
```

bzw. für eine seitenweise Ansicht (nächste Seite durch Drücken der Leertaste):

```
grof_././cfdv_|_more
```

Die Ausgabe ist in zwei Teile unterteilt. Aus dem *flat profile* können Sie entnehmen, in welcher Funktion wieviel Zeit verbracht wurde. Im zweiten Teil ist die Ausführungszeit den nach den Funktionsaufrufen im Programm unterteilt.

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
28.46	1.69	1.69	16518600	0.00	0.00	__exact_riemann_mod__exact_riemann
16.13	2.64	0.96	41427400	0.00	0.00	__exact_riemann_mod__prefun
11.40	3.32	0.68	3236	0.00	0.00	__fluxcalculation_mod__fluxcalculation
10.05	3.91	0.60	16503600	0.00	0.00	__flux_godunov_mod__flux_godunov
...						

Es ist nicht verwunderlich, dass der Riemannlöser selbst am meisten Zeit beansprucht.

2 Aufgabenteil 2: Konzept für eine Parallelisierung

2.1 Hintergrund

Nur weil eine Funktion ganz oben im Profil auftaucht, bedeutet das nicht, dass genau diese Routine parallelisiert werden soll. Vielmehr muss ein Konzept entwickelt werden, dass dafür sorgt, dass diese Routine parallel ausgeführt wird. Je grobgranularer die Parallelisierung ist, desto besser, da weniger Overhead entsteht.

Dazu rufen wir uns nochmal die Struktur des Programms ins Gedächtnis:

- Zeitschritt bestimmen
- Randbedingungen
- Rekonstruktion

- Cauchy-Kovalevskaya-Prozedur
- Flussberechnung
- Zeitupdate auf das nächste Zeitlevel
- Analyse und Output

Grundsätzliche Fragen, die zu klären sind:

- welche der obigen Punkte lohnt es sich als erstes zu parallelisieren, welche danach, welche zum Schluss oder gar nicht?
- sind diese Routinen mit OpenMP parallelisierbar, d.h. gibt es dort *do*-Schleifen?
- sind alle Iteration der Schleifen unabhängig oder wird z.B. in jeder Iteration eine skalare Variable aktualisiert?

2.2 Genaue Aufgabenstellung

Füllen Sie unter Zuhilfenahme des erzeugten *call graph*-Profiles und des Quell-Codes folgende Tabelle aus.

	Arbeitsaufwand	parallelisierbar	Schleifen	Abhängigkeiten	Reihenfolge
Zeitschritt bestimmen					
Randbedigungen					
Rekonstruktion					
CK-Prozedur					
Flussberechnung					
Zeitupdate					
Analyse und Output		nein	-	-	-

3 Aufgabenteil 3: die OpenMP-Parallelisierung

3.1 Hintergrund

Mit OpenMP kann man Arbeit auf mehrere Threads aufteilen. Dazu muss die Arbeit in bestimmter Form vorliegen, z.B. in Form von *do*-Schleifen oder in Form von unabhängigen Code-segmenten. Um OpenMP zu verwenden, müssen Kommentare in einer bestimmten Form, sogenannte Direktiven, im Quellcode ergänzt werden und mit den entsprechenden Compiler-Flags kompiliert und gelinkt werden. Die Parallelisierung mit OpenMP kann stückweise erfolgen, d.h. man kann jede einzelne Schleife separat parallelisieren. Dies ist jedoch nicht vorteilhaft.

3.2 Genaue Aufgabenstellung

3.2.1 Integration von OpenMP-Direktiven im Quellcode

Zunächst werden wir uns der Parallelisierung einer *do*-Schleife zuwenden. Zuerst müssen die Threads erzeugt und am Ende wieder vernichtet werden. Fassen Sie ihre Schleife also ein mit:

```
!$omp parallel default(none)
do i = 1 ,n
    ...
end do
!$omp end parallel
```

Da wir die Arbeit noch nicht aufgeteilt haben wird, die Schleife von allen Threads durchlaufen. Der Zusatz *default(none)* gibt an, dass alle Variablen die in der *do*-Schleife vorkommen, entweder als *shared* oder als *private* deklariert werden müssen. Dies gibt unsere nächste Zeile:

```
!$omp parallel default(none)                &
!$omp shared(var1,var2) private(var3,var4)
do i = 1 ,n
    ....
```

Beachten Sie das *&* am Ende der ersten Zeile!

Um die Schleife auf alle Threads zu verteilen bzw. das wieder zu beenden, fügen wir zwei weitere Zeile ein, so dass schließlich Folgendes darsteht:

```
!$omp parallel default(none)                &
!$omp shared(var1,var2) private(var3,var4) &
!$omp do schedule(runtime)
```

```
do i = 1 ,n
    ...
end do
!$omp end do
!$omp end parallel
```

Der Zusatz *schedule(runtime)* gibt an, dass wir die Umgebungsvariable *OMP_SCHEDULE* setzen werden. Mehr dazu später.

An zwei Stellen im Code wird eine globale Summe bzw. ein globales Minimum gebildet. Bei diesen Schleifen fügen Sie davor bitte noch

```
!$omp reduction(+:myvar1)
```

bzw.

```
!$omp reduction(min:myvar2)
```

ein.

Es empfiehlt sich, die OpenMP-Direktiven in *#ifdefs* einzufassen.

```
#ifdef OPENMP
!$omp parallel default(none) &
!$omp shared(var1,var2) private(var3,var4) &
!$omp do schedule(runtime)
#endif
do i = 1 ,n
    ...
end do
#endif
!$omp end do
!$omp end parallel
#endif
```

3.2.2 Ausführen des Programms

Bevor Sie das Programm ausführen können, benötigen Sie ein neues Makefile, das um OpenMP-Optionen erweitert wurde. Bitte laden Sie sich es von der Homepage herunter. Kompilieren Sie *cfdfv* mit:

```
make □gnu-openmp
```

Danach sind noch die beiden Umgebungsvariable zu setzen:

```
export OMP_NUM_THREADS=2
export OMP_SCHEDULE=static
```

Damit werden 2 Threads verwendet und *OMP_SCHEDULE=static* teilt die Iterationen einer Schleife gleichmäßig auf alle Threads auf.

Sie können jetzt das Programm wie gewohnt ausführen:

```
../../cfdv_toro4.ini
```

Stimmen die Ergebnisse noch? Wie ändert sich die Laufzeit?

3.2.3 Weitere Schritte

Nun können Sie nach und nach weitere *do*-Schleifen parallelisieren. Achten Sie auf die Korrektheit und die Laufzeit ihres Programms. Erhalten Sie einen vernünftigen Speedup? Was passiert, wenn man statt 1. Ordnung in Raum und Zeit mit 2. Ordnung rechnet?

Im Allgemeinen ist das Ziel, die Erzeugung und das Vernichten der Threads möglichst weit nach außen zu ziehen, da dies sehr teuer ist. Nehmen wir also an, wir haben zwei *do*-Loops mit etwas Output, Initialisierungen von Variablen oder Ähnlichem dazwischen:

```
!$omp parallel default(none) &
!$omp shared(var1,var2) private(var3,var4) &
!$omp do schedule(runtime)
do i = 1 ,n
    ...
end do
!$omp end do
!$omp end parallel

print *, "intermediate Results"

!$omp parallel default(none) &
!$omp shared(var1) private(var3,var5) &
!$omp do schedule(runtime)
do i = 1 ,n
    ...
end do
!$omp end do
!$omp end parallel
```

Dies kann man wie folgt zusammenfassen:

```
!$omp parallel default(none)                &
!$omp shared(var1,var2) private(var3,var4,var5)

!$omp do schedule(runtime)
do i = 1 ,n
    ...
end do
!$omp end do

!$omp single
print *, "intermediate Results"
!$omp end single

!$omp do schedule(runtime)
do i = 1 ,n
    ...
end do
!$omp end do

!$omp end parallel
```

Alle Vereinbarung von *shared* und *private* Variablen der einzelnen Loops werden zusammengefasst und stehen gleich nach der Erzeugung der Threads. Das Zwischenstück wird in *single*-Direktiven eingefasst, da es nur von einem Thread ausgeführt werden soll.