Informatik und Angewandte Kognitionswissenschaft
Lehrstuhl für Hochleistungsrechnen

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

Rainer Schlönvoigt
Thomas Fogal
Prof. Dr. Jens Krüger

High-Performance Computing

http://hpc.uni-duisburg-essen.de/teaching/wt2013/pp-nbody.html

Exercise 3 - OpenMP N-Body (30 Points)

All assignments are to be uploaded to Moodle. Assignments are due at midnight on the due date. No late assignments will be accepted.

All assignments must include a `Makefile` for compiling your assignments. The assignment specification should include what the default target of your makefile should be. Assignments which do not compile will receive 0 points. Sometimes, we provide sample inputs and outputs; assignments which do not satisfy these test inputs will receive very few points.

Please do not include additional output other than what was requested by the assignment details. *Hint: if you want more debugging output, use a 'debug' flag in your program's arguments and have it only print when that flag is active.*

Your assignment will be graded on the `duecray.uni-due.de` supercomputer. It does not matter if your program runs correctly on another machine; it must run correctly on `duecray` to receive credit.

1 Einführung

Nachdem in der letzten Übung die Parallelisierung mit Hilfe von MPI durchgeführt wurde, soll in dieser Übung stattdessen Parallelisierung mit Hilfe von OpenMP realisiert werden.

2 Message Passing vs Shared Memory

Bei der Verwendung von MPI werden mehrere Instanzen eines Programmes in separaten Prozessen ausgeführt. Daher besitzt jede Instanz ihren eigenen Speicherbereich und ist zunächst völlig unabhängig von den anderen Instanzen. Daher muss sich der Entwickler selbst darum kümmern, an welchen Stellen die Prozesse mit einander kommunizieren, um Daten auszutauschen oder beispielsweise auf einander zu warten.

Bei OpenMP hingegen ist die grundlegende Idee, dass es zunächst einen Hauptthread gibt, der auf Befehl mehrere Nebenthreads zur Bearbeitung einer Aufgabe erstellt. Nach Bearbeitung einer Aufgabe, beziehungsweise dem Ende eines definierten Blockes werden die anderen Threads wieder beendet und das Programm wird wieder alleine auf dem Hauptthread ausgeführt. Threads teilen sich im Gegensatz zu Prozessen den selben Speicher und benötigen daher keinen extra Daten-Abgleich (durch den Entwickler). (Siehe Figure 1)

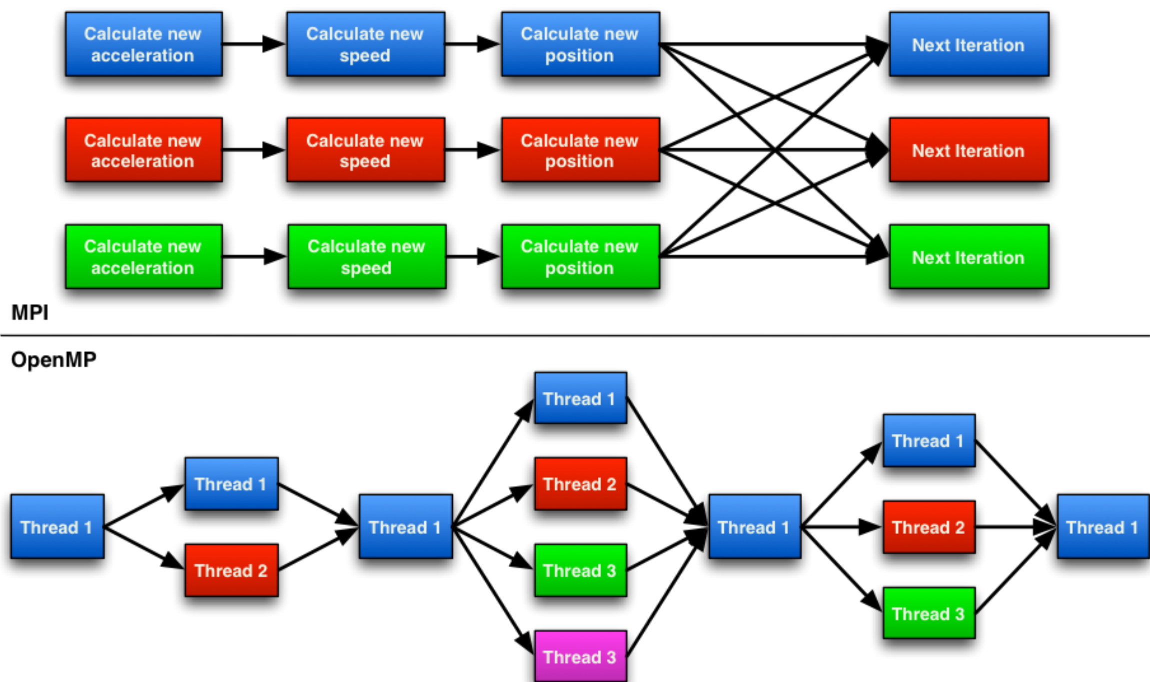

Figure 1: MPI *vs* OpenMP

3 Compiler-Directives

Ein Vorteil und Anwendungszweck von OpenMP ist es, dass der Code im Vergleich zur seriellen Implementation möglichst unverändert bleibt. Stattdessen soll der Compiler die Aufgabe übernehmen. Dies ermöglicht ein flexibles Wechseln zwischen seriellem und parallelem Code und erzielt bereits mit wenig Arbeit ein gutes Resultat.

Um dies zu bewerkstelligen werden dem Code sogenannte Compiler-Directives hinzugefügt. Diese sind Hinweise an den Compiler dafür, wie folgender Code übersetzt werden soll. Unterstützt ein Compiler die verwendeten Hinweise nicht, werden diese ignoriert. Daher kann erstellter Code weiterhin auf Plattformen kompiliert werden, deren Compiler (noch) kein OpenMP unterstützt.

Directives beginnen grundsätzlich mit `#pragma`, der Zusatz `omp` bezeichnet zudem, dass er zur Kategorie des OpenMP gehört. Innerhalb dieser Kategorie gibt es eine Vielzahl an Hinweisen, von denen wir hier nur wenige besprechen/wiederholen. (Unter "http://bisqwit.iki.fi/story/howto/openmp/" findet sich eine etwas ältere aber durchaus Hilfreiche Erklärung vieler Directives.)

3.1 #pragma omp parallel

Der Directive `#pragma omp parallel` ist der wichtigste Befehl, da er definiert, dass für den Bereich des nächsten C/C++ Befehls, beziehungsweise des nächsten Blocks Threads gestartet, verwendet und anschließend wieder beendet werden sollen. Die Anzahl der gestarteten Threads entspricht dabei (wenn nicht anders definiert) der Anzahl der Prozessoren der Maschine, auf der der Code ausgeführt wird.

```
1 void testFunction()
2 {
3   #pragma omp parallel
4   {
5     printf("Hello World\n");
6   }
7 }
```

Listing 1: Parallel Block

Das Code-Listing 1 sorgt dafür, dass *"Hello World"* mehrfach ausgegeben wird. Beachtet dabei, dass der Code weiterhin funktionieren würde, wenn der Compiler omp nicht unterstützen würde oder es absichtlich abgestellt wurde.

Ein weiterer Aufruf von `#pragma omp parallel` innerhalb des Blockes würde keinen Effekt erzielen.

3.2 ID-Based

In manchen Fällen ist es nicht nur wichtig, dass ein Befehl von irgendeinem Thread ausgeführt wird, sondern auch von welchem. ähnlich wie bei MPI lässt sich die eigene Thread-Nummer, sowie die Gesamtzahl an Threads abrufen.

```
1  #include <omp.h>
2
3  void testFunction()
4  {
5    #pragma omp parallel
6    {
7      int threadID = omp_get_thread_num();
8      printf("Hello World from thread %d\n", threadID);
9
10     #pragma omp barrier
11     if (threadID == 0 ) {
12       int nthreads = omp_get_num_threads();
13       printf("There are %d threads\n",nthreads);
14     }
15   }
16 }
```

Listing 2: Thread-ID Dependent OpenMP

In Code-Listing 2 wird zunächst mit Hilfe von `omp_get_thread_num` die eigene Thread-Nummer abgerufen und anschließend innerhalb der “Hello World”-Ausgabe mit ausgegeben. Die Reihenfolge in der die einzelnen Threads dies ausgeben ist jedoch unbestimmt. Der Befehl `#pragma omp barrier` sorgt dafür, dass alle Threads warten, bis alle anderen Threads ebenfalls an dieser Stelle angekommen sind, bevor die Ausführung fortgesetzt wird. Grundsätzlich sollte dies in der Parallelprogrammierung vermieden werden, wenn die Reihenfolge nicht relevant ist. Anschließend wird lediglich von einem einzelnen Thread ausgeben, wie viele Threads insgesamt existieren.

In diesem Beispiel wird tatsächlicher C/C++ Code von omp benötigt und kann daher nur kompiliert werden, wenn OpenMP Support vorhanden ist.

3.3 `#pragma omp for`

In der MPI-übung musstet ihr selbst aufteilen, welcher Prozess für welche Objekte verantwortlich ist. Zumindest für for-Schleifen kann OpenMP dies mit Hilfe des Directive `#pragma omp for` selbst übernehmen. Dies wird einfach innerhalb eines `parallel` blocks vor eine for-Schleife geschrieben.

```
#include <omp.h>
void simpleForLoop()
{
  #pragma omp parallel
  {
    #pragma omp for
    for(int i = 0; i < totalObjectCount; i++)
    {
      calculateNewPositionForObject(i);
    }
  }

  #pragma omp parallel for
  for(int i = 0; i < totalObjectCount; i++)
  {
    calculateNewPositionForObject(i);
  }
}
```

Listing 3: Automatische Arbeitsaufteilung

Falls lediglich die Schleife parallelisiert werden soll, können beide Directives zu `#pragma omp parallel for` zusammengefasst werden. (Siehe Listing 3)

3.4 Nested Loops

Wenn ihr mehrere For-Schleifen ineinander verwendet, so lässt sich bei älteren OpenMP-Versionen nur eine der beiden Schleifen parallelisieren. Die zweite Parallelisierung erzielt dabei keinen Effekt, da die Durchläufe der zweiten Schleife nur auf einen einzelnen Thread verteilt werden. Ihr müsst euch daher entscheiden, ob ihr die innere oder die äußere Schleife parallelisieren wollt.

3.5 How to compile

Wenn ein Compiler OpenMP unterstützt, muss ihm mitgeteilt werden, dass er omp-Directives auch tatsächlich übersetzen soll.

Dies wird bewerkstelligt, indem GCC das Flag `-fopenmp` sowohl als Compiler-Flag, als auch als Linker-Flag mitgegeben wird.

Der Cluster verwendet standardmäßig einen anderen Compiler. Wenn ihr den Code dort kompilieren möchtet, denkt daran nach dem Login zuerst den Befehl "`module swap PrgEnv-pgi PrgEnv-gnu`" (ohne Anführungszeichen) auszuführen, um auf GCC zu wechseln. (Aus Platzgründen wird in diesem Kapitel lediglich beschrieben, wie dies im Falle des Gnu Compilers funktioniert, andere Compiler verwenden meist andere Flags => Google)

4 Your own code

Wenn ihr die Compiler- und Linkerflags in euer Makefile eingetragen habt, könnt ihr damit beginnen, euren eigenen Code auf OpenMP umzustellen. Fangt damit an, eine Funktion wie in Listing 2 zu implementieren, damit ihr überprüfen könnt, ob die omp-Directives tatäschlich vom Compiler beachtet werden, und um zu sehen, wie viele Threads verwendet werden. Erstellt anschließend eine Kopie eures seriellen Codes und fügt nacht eigenem Ermessen OpenMP Code hinzu, wo immer ihr denkt, dass der Code von Parallelisierung profitieren würde. Denkt dabei an mögliche *Race-Conditions*.

5 Hybrid MPI/OpenMP

Ihr solltet gemerkt haben, dass die Verwendung von OpenMP sehr einfach zu realisieren ist. OpenMP basiert auf Shared Memory, alle Threads teilen sich daher den selben Speicher. Dies führt zu Problemen, sobald Code nicht nur auf einem einzelnen Rechner, sondern auf einem gesamten Cluster ausgeführt werden soll. Selbst in einem System in dem der Arbeitsspeicher mehrerer Rechner in Form eines großen Speichers virtuell zusammengeführt wird, leidet die Performance stark darunter. Unter Umständen müssten Berechnungen, die auf dem einen Computer stattfinden, ständig auf den Arbeitsspeicher des anderen Rechners zugreifen, falls sich dort die relevanten Daten befinden.

Dieses Problem ist bei MPI deutlich geringer, da es sich um mehrere Prozesse handelt, die alle einen lokalen Speicher besitzen, auf dem sie Berechnungen durchführen können. Die verschiedenen Prozesse müssen aber untereinander Nachrichten verschicken, um sich zu synchronisieren und zu kommunizieren. Im Falle, dass alle Prozesse auf der selben Maschine laufen, kann es sein, dass diese Art der Kommunikation mehr Arbeit benötigt und langsamer ist, als der simple Speicherzugriff bei OpenMP.

Hybrid-Systeme, die MPI und OpenMP kombinieren, können daher unter Umständen von Vorteil sein. Ein stark vereinfachter Fall ist es beispielsweise, einen (oder mehrere) MPI-Prozess(e) pro Maschine zu starten und die anfallende Arbeit wie gewohnt auf die Prozesse zu verteilen, aber innerhalb der Berechnungen eines Prozesses (bzw. einer Maschine) auf OpenMP zurückzugreifen. (Siehe Figur 2).

Ziel der nächsten Übung wird es sein, die bestmögliche Performanz aus dem Cluster heraus zu holen. Daher solltet ihr auch versuchen, ein Hybrid-

System zu realisieren. (Im sehr simplen Falle entspricht dies lediglich einer Zeile pro For-Schleife in eurem MPI-Code.)

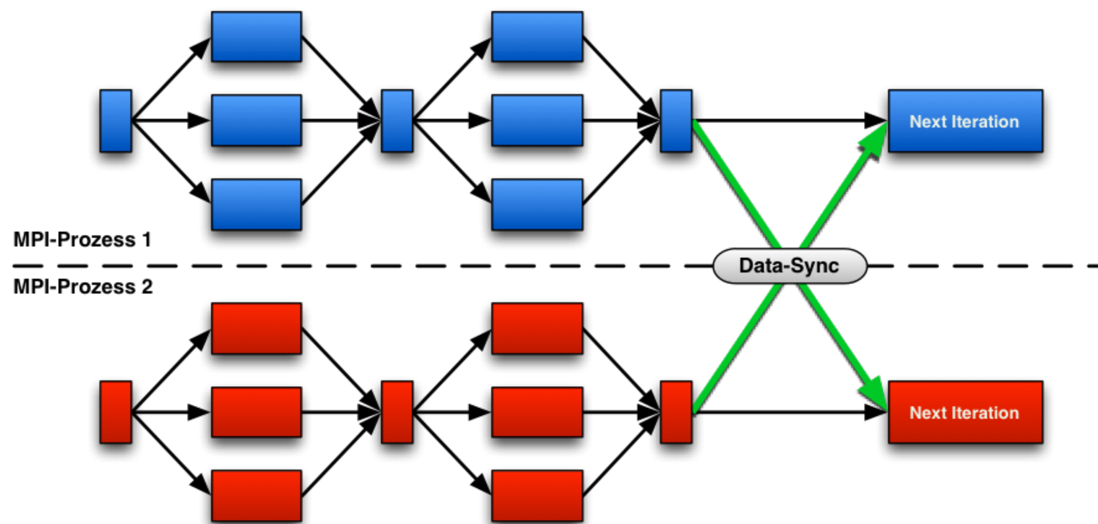

Figure 2: MPI/OpenMP-Hybrid

6 *sine qua non*

Pack your file into a single compressed tarball for submission. The tarball should extract all files into a single subdirectory. Upload your tarball to Moodle.