1. Einführung in OpenMP

- Übersicht
- Einführung
- Homogene und inhomogene Arbeitsverteilung
- Rekursive Parallelität
- Beispiele

1 - Überblick

- OpenMP: Vereinfachte Parallelisierung mit Threads
  - Idee: Programm durch spezielle Hinweise (Direktiven) parallelisieren
  - Schleifenparallelisierung: Fork-Join-Modell
  - Vorteile
    - » Einfacher als Threads: Höhere Abstraktion
    - » Sequentielle Version weiterhin nutzbar
  - Nachteile
    - » Nur gemeinsamer Hauptspeicher
    - » Eingeschränkte Anwendung (Schleifentypen, C++)
- Sprachen
  - C, C++, Fortran
- Homepage
  - http://www.openmp.org

1 - Überblick

- Entwicklung
  - 1997/1998: Version 1.0
  - 2000/2002: Version 2.0
  - 2005: Version 2.5
  - 2008: Version 3.0
    - » Tasks
- Unterstützung
  - Kommerzielle Compiler: IBM, Portland, Intel
  - OpenSource: GNU gcc
    - » OpenMPI 2.5 ab gcc 4.2
    - » OpenMPI 3.0 ab gcc 4.4

1 - Direktiven

- Spezielle Compileranweisungen
  - In C und C++
    - » #pragma omp <omp anweisung> <optionale argumente>
  - In Fortran
    - » !$omp <omp anweisung> <optionale argumente>

2 – Einführung

- Sequentielles Programm

```
double res[10000];
for (int i=0 ; i<10000 ; i++)
      langwierige_berechnung(&res[i]);
```

- Parallelisiert mit OpenMP

```
double res[10000];
#pragma omp parallel for
for (int i=0 ; i<10000 ; i++)
   langwierige_berechnung(&res[i]);
```

2 – Einführung

- Was passiert im Hintergrund?
- OpenMP nutzt SPMD-Modell (Single program, multiple data)
- Programm startet sequentiell mit einem einzigen Thread
- Direktive parallel startet ein *Team*
  - Aktueller Thread ist *Master*
  - Alle Threads bearbeiten den folgenden Abschnitt
- Direktive for, teilt die Iterationen auf die Team-Mitglieder auf
  - Jeder Thread bearbeitet einen Teil der Iterationen
- Ende das parallelen Blocks
  - Implizite Synchronisation (Barrier)
  - Nur der Master läuft weiter

2 – Einführung: Direktiven

- Parallele Regionen
  - omp parallel
- Definition gemeinsamer und privater Daten
  - omp shared, private,. . .
- Aufteilung von Schleifen
  - omp for
- Synchronisation
  - omp atomic, critical, barrier,. . .
- Bibliotheksfunktionen und Umgebungsvariablen
  - omp_set_num_threads(), omp_set_lock(),. . .
  - OMP_SCHEDULE, OMP_NUM_THREADS, . . .

2 – Einführung

- Festlegung der Anzahl der Threads: Option num_threads

```
double res[10000];
#pragma omp parallel for num_threads(4)
for (int i=0; i<10000 ; i++)
   langwierige_berechnung(&res[i]);
```

- Bedingte Parallelisierung: Option if

```
double res[elemente];
#pragma omp parallel for if(elemente>500)
for (int i=0; i<elemente ; i++)
   langwierige_berechnung(&res[i]);
```

2 – Einführung

- Einige Parameter können auch
  - über OMP-Bibliotheksfunktionen gesetzt werden

```
omp_set_num_threads(4);
double res[10000];
#pragma omp parallel for
for (int i=0; i<10000 ; i++)
   langwierige_berechnung(&res[i]);
```

  - über Umgebungsvariablen beeinflusst werden
    - » `export OMP_NUM_THREADS=4`

3 – Arbeitsverteilung

- Parallele Ausführung mit parallel
  - Folgender Block wird von allen Threads parallel ausgeführt
  - Programmierer verteilt die Arbeit manuell

```
double A[10000];
omp_set_num_threads(4);
#pragma omp parallel
{
int th_id = omp_get_thread_num();
langwierige_berechnung (th_id, A);
}
printf("Beendet.\n");
```

3 – Arbeitsverteilung

- Aufteilung von Schleifeniteration mit for
  - OpenMP teilt Iterationen den einzelnen Threads zu
  - Nur for-Schleifen mit bestimmter Syntax

```
int i;
#define N 1000
#pragma omp parallel
#pragma omp for
for (i=0 ; i<N ; i++) {
  langwierige_berechnung (i);
}
printf("Beendet.\n");
```

3 – Arbeitsverteilung

- **Beispiel in C++ (OpenMP 3.0)**
  - **Nur random_acess_iterator möglich**

```
vector<int> v(10000);
typedef vector<int>::iterator iter;
#pragma omp parallel
#pragma omp for
for (iter i=v.begin(); i < v.end() ; i++) {
   langwierige_berechnung (*i);
}
printf("Beendet.\n");
```

3 – Arbeitsverteilung

- **Einschränkungen**
  - **Nur ganzahlige Typen (mit oder ohne Vorzeichen)**
  - **Test nur mit <, <=, >, >=**
  - **Schleifenvariable verändern: nur einfache Ausdrücke**
    - » **Operatoren ++,--,+,-+=,-=**
  - **Obere und untere Grenze unabhängig von Schleifendurchlauf**
    - » **Ausdrücke sind möglich**

- **Automatische Synchronisation nach for (barrier)**
  - **can mit nowait unterdrückt werden**

3 – Arbeitsverteilung: Aufteilung von Schleifen

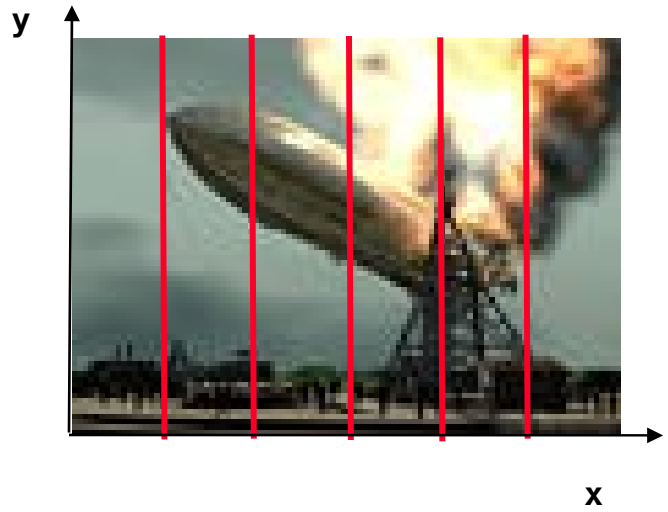
- Konzeptuell: Indexraum 0..N-1 wird aufgeteilt und zugewiesen
- Angabe von schedule
  - schedule(static[,k])
    - » Indexraum wird in Blöcke der Größe k aufgeteilt und reihum an die Threads zugeweisen
  - schedule(dynamic[,k])
    - » Indexraum wird in Blöcke der Größe k aufgeteilt und nach Bedarf an die Threads zugewiesen
  - schedule(guided[,k])
    - » Index wird in Blöcke proportional zur Restarbeit por Thread aufgeteilt und nach Bedarf zugewiesen. k = minimale Iterationszahl
  - schedule(auto)
    - » Implementierung-spezifisch (Standard, wenn keine Angabe)
  - schedule(runtime)
    - » Entscheidung zur Laufzeit (omp_set_schedule, omp_get_schedule, OMP_SCHEDULE)

3 – Arbeitsverteilung: Aufteilung von Schleifen

- Beispiel, 100 Iterationen, 3 Threads, homogene Last
  - schedule(static,1)
    - 012012012.....0
  - schedule(static,5)
    - 00000111112222200000111112222 2...00000
  - schedule(guided,1)
    - Thread 0: 33,1 iterations
    - Thread 1: 22,6,3,1,1 iterations
    - Thread 2: 15,10,4,2,1,1 iterations
  - schedule(guided,5)
    - Thread 0: 33 iterations
    - Thread 1: 22,6,5 iterations
    - Thread 2: 15,10,5,4 iterations

3 – Arbeitsverteilung

- **Anwendungsbeispiel: Raytracing**
  - Idee: Jeder Thread rechnet einen Streifen
  - Problem: Ungleichmäßige Last
  - Mit OpenMP: Jeder Thread berechnet 10 Spalten




```
#pragma omp parallel
#pragma omp for schedule(dynamic,10)
for (int x=0; x <= Breite; x++) {
   berechne_spalte (x);
}
```

3 – Arbeitsverteilung

- Zusammenfassung von Schleifen
  - for-Direktive wirkt nur auf nächste Schleife
  - collapse(k) erlaubt, kombiniert Indexraum der folgenden k Schleifen

```
#pragma omp parallel
#pragma omp for schedule(dynamic,20) collapse(2)
for (int x=0; x <= Breite; x++) {
  for (int y=0; x <= Höhe; y++) {
    berechne_pixel (x,y);
  }
}
```

  - Jeder Thread berechnet 20 Pixel
  - Vorteil: Feinere Lastverteilung

3 - Arbeitsverteilung

- **Anwendungsbeispiel: Game of life**
  - **Dasselbe Muster**
  - **Durch zwei Arrays: Lese- aber keine Schreibkonflikte**

```
#pragma omp parallel
#pragma omp for schedule(dynamic,10)
for (int i=1; i <= height; i++)
  for (int j=1; j <= width; j++)
    update_cell(i,j);
```

3 - Arbeitsverteilung

- **Anwendungsbeispiel: LU-Zerlegung**

```
for(k=0; k<=n-2; k++) {
#pragma omp parallel for
  for (i=k+1; i<=n-1; i++)
    M[i][k] = -M[i][k] / M[k][k];
#pragma omp parallel for collapse(2)
  for(j=k+1; j<=n-1; j++)
    for(i=k+1; i<=n-1; i++)
      M[i][j] = M[i][j] + M[i][k]*M[k][j];
}
```

3 – Arbeitsverteilung

- **Parallelisierung ohne Schleifen mit Abschnitten**
  - Jeder Abschnitt wird einem Thread zugewiesen
  - Nur statische Parallelität möglich: Abschnitt gilt für ganzes Team

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section    // erstes #pragma optional
  Arbeit1( );
#pragma omp section
  Arbeit2( );
#pragma omp section
  Arbeit3( );
}
```

3 – Arbeitsverteilung

- Einfache Synchronisation
- Folgenden Abschnitt nur von einem Thread ausführen

```
#pragma omp single
```

3 – Arbeitsverteilung

- **Parallelisierung ohne Schleifen mit Tasks (OpenMP 3.0)**
  - **Jeder Task wird (reihum) einem Thread zugewiesen**
  - **Unterschied zu sections**
    - » **Task kann von beliebigem Thread im Team definiert werden**
    - » **Ein Thread im Team führt den Task aus**
    - » **Definition eines Tasks an beliebiger Stelle**
- **Syntax:**

  #pragma omp task

3 – Arbeitsverteilung

- **Beispiel: Quicksort mit Task**

```
void quicksort(int a[], int l, int r) {
 if(r>l) {
    int i = partition(a,l,r)
#pragma omp task
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
 }
}

int main() {
#pragma openmp parallel
  {
#pragma openmp single nowait
    quicksort(a,1, 99);
  }
}
```

3 – Arbeitsverteilung

- Zum Vergleich: Quicksort mit verschachtelten Teams

```
void quicksort(int a[], int l, int r) {
 if(r>l) {
    int i = partition(a,l,r);
#pragma omp parallel sections
   {
#pragma omp section
      quicksort(a, l, i-1);
#pragma omp section
      quicksort(a, i+1, r);
   }
 }
}
int main() {
  quicksort(a,1, 99);
}
```

Reference: Some useful library functions

```
#include <omp.h>
omp_get_num_procs()          // get number of processors
omp_get_num_threads()        // # of running threads
omp_set_num_threads()        // set # of running threads
omp_get_max_threads()        // maximum # of threads
omp_get_thread_num()         // current thread's id
```

Reference: Some useful environment variables

```
OMP_NUM_THREADS      # number of threads to use
OMP_SCHEDULE         # scheduling policy for
                  # loop with schedule(runtime)
OMP_STACKSIZE        # size of thread stacks
```