4. OpenMP

- Datenabhängigkeiten in OpenMP
- Weitere Möglichkeiten in OpenMP
- Fallstudien
  - Einfache Parallelisierung
  - Paralleles dynamisches Programmieren
  - Irreguläre Parallelität

4 - Synchronisation

- Thread-Modell in OpenMP
  - Jeder Thread besitzt einen privaten Kontext
    - » Instruktionszeiger
    - » Stack und lokale Variablen
  - Alle Threads teilen den globalen Kontext
    - » Statische und globale Daten im Hauptspeicher
- Zugriff auf gemeinsame Daten muss synchronisiert werden
- Ansatz in OpenMP
  - Variablen können explizit als privat oder geteilt deklariert werden
  - Speziell Reduktionen werden explizit unterstützt

4 - Synchronisation

- Unsere Annahme bisher: Alle Threads arbeiten unabhängig
  - „Peinlich parallele Programme“ (embarassingly parallel)
- In der Praxis: Abhängigkeiten
  - Zwischen Schleifeniterationen
  - Konkurrierender Zugriff auf globale Variable (Wettlaufbedingung)
- Beispiel konkurrierender Zugriff: Was kommt heraus?

```
int summe = 0, tmp;
int a[10000];
#pragma omp parallel for
for (int i=0; i<10000 ; i++) {
  tmp = a[i]; a[i]=-tmp;
}
```

4 - Synchronisation

- **Beispiel Schleifenabhängigkeit: Was kommt heraus?**

```
int summe = 0;
int a[10000];
f[0]=f[1]=1;
#pragma omp parallel for
for (int i=2 ; i<10000 ; i++)
  f[i] = f[i-1]+f[i-2];
printf("f[9999] : %d", f[9999]);
```

- **Beispiel konkurrierender Zugriff: Was kommt heraus?**

```
int summe = 0;
int a[10000];
#pragma omp parallel for
for (int i=0; i<10000 ; i++)
  summe += a[i];
printf("Ergebnis : %d", summe);
```

4 - Synchronisation

- Lösung erstes Problem: Variable `tmp` ist thread-lokal

```
int summe = 0, tmp;
int a[10000];
#pragma omp parallel for private(tmp)
for (int i=0; i<10000 ; i++) {
  tmp = a[i]; a[i]=-tmp;
}
```

- `private` erzeugt eine Thread-lokale *nicht-initialisierte* Variable
- Zweites Problem: ohne explizite Parallelisierung nicht lösbar
  - Abhängigkeiten erzwingen sequentielle Ausführung

4 - Synchronisation

- Drittes Problem
  - Summe muss anfänglich in allen Threads 0 sein
  - Summe muss am Schluss wieder zusammengefasst werden

```
int summe = 0;
int a[10000];
#pragma omp parallel
#pragma omp for firstprivate(summe) lastprivate(summe)
for (int i=0; i<10000 ; i++)
  summe += a[i];
printf("Ergebnis : %d", summe);
```

  - firstprivate initialisiert lokale Kopie mit globalem Wert
  - lastprivate setzt globale Variable auf Wert der letzten Iteration

4 - Synchronisation

- Drittes Problem
  - Im Grunde: Reduktionsoperation

```
bool exists(bool b[], unsigned n) {
  bool result = false;
#pragma omp parallel for reduction(||:result)
  for (int i=0; i<n ; i++)
    result ||= b[i];
  return result;
}
```

4 - Synchronisation

- Implizite Festlegung innerhalb der lexikalischen Umgebung
  - Lokale Variable (automatic storage) sind privat
  - Schleifenvariable sind privat
  - Statische Variable werden geteilt
  - Dynamisch allozierte Variable sind geteilt
- Implizite Festlegung innerhalb der dynamischen Umgebung
  - Dieselben Grundregeln
  - Referenzparameter erben das Attribut der eigentlichen Variable
- Vorgabewerte festlegen
  - default(typ)
  - mit typ = shared,none,(private,firstprivate)

4 - Synchronisation

- **Reduktionsoperationen in OpenMP**
  - **Syntax: reduce(op:var)**
  - **Variable wird mit Standardwert initialisiert**
  - **Nur wenige Operatoren unterstützt (kein C++-Overloading!)**

Operator	Default	Operator	Default
+	0	*	1
-	0	&	~0
|	0	^	0
&&	1	||	0

  - **Globale Variable erhält nach der parallelen Region den Gesamtwert**
  - **Vorsicht: Ausführungsreihenfolge nicht garantiert, numerische Ergebnisse können abweichen**

4 – Zusammenfassung Direktiven und Klauseln

	parallel	for	sections	single	parallel do	parallel sectior
if	x				x	x
private	x	x	x	x	x	x
shared	x	x			x	x
default	x				x	x
firstprivate	x	x	x	x	x	x
lastprivate		x	x		x	x
reduction	x	x	x		x	x
copyin	x				x	x
schedule		x			x	
ordered		x			x	

4 – Thread-private Variable

- Weitere Möglichkeit: Daten die über die gesamte Laufzeit mit einem Thread assoziiert sind
- Deklaration

```
#pragma omp threadprivate(var-list)
```

- Vergleiche
  - private: Thread-lokale Variable, kein Zugriff auf globale Variable gleichen Namens
  - threadprivate: Thread-private, globale Variable

5 – Weitere Synchronisationsmöglichkeiten

- Nur Master-Thread führt aus
  #pragma omp master
- Kritischer Abschnitt: Wird einzeln ausgeführt
  #pragma omp critical
- Synchronisation aller Threads mit Schranke
  #pragma omp barrier
- Auf Ende von Unter-Tasks warten
  #pragma omp taskwait
- Atomares Schreiben (“kleiner kritischer Abschnitt”)
  #pragma omp atomic
- Synchronisation Thread-Sicht mit Speicher (schwach konsistentes Speichermodell)
  #pragma omp flush
- Ausführen von Teilen einer Schleife in Iterationsreihenfolge
  #pragma omp ordered

5 - Beispiel

```
 int n = 0;
 int num_threads;
#pragma omp parallel
  {
#pragma omp single nowait
    num_threads = omp_get_num_threads();
 #pragma omp critical
     n++;
  }
// hier: n == num_threads
```

5 - Beispiel

```
int x = 0;
#pragma omp parallel shared(x)
{
#pragma omp single
  {
    x++;
  }
#pragma omp single
  {
    x++;
  }
printf("x is %d", x); // x==2 nicht garantiert
}
```

5 - Beispiel

- OpenMP definiert ein schwach konsistentes Speichermodell
  - Insbesondere können gemeinsame Variabel in Registern gehalten werden
- Um globale Konsistenz herzustellen, muss flush verwendet werden
- Im vorigen Beispiel zwischen den single-Abschnitten

```
#pragma omp critical
```

einzufügen

5 - Beispiel

```
#pragma omp parallel for nowait
for(int i=0, i<100; i++)
    a[i] = b[i]+c[i]
#pragma omp parallel for nowait
for(int i=0, i<100; i++)
    d[i] = b[i]+a[i]
```

- Vorsicht mit nowait
- Erstes Beispiel schlägt fehl, zweites nicht. Warum?
  - Barrier im ersten Fall notwendig!

```
#pragma omp parallel for nowait
for(int i=0, i<100; i++)
    a[i] = b[i]+c[i]
#pragma omp parallel for nowait
for(int i=0, i<100; i++)
    d[i] = b[i]+c[i]
```

5 – Beispiel: Parallele Listenverarbeitung

```
ListElement t = head;
#pragma omp parallel
{
#pragma omp single
  {
    while (t != 0) {
#pragma omp task firstprivate(t)
      {
        compute(t);
      }
    } // single
  } // parallel
```