



6. *Der OpenMP Standard*

- Direktiven-basiertes API zur Programmierung von Parallelrechnern mit gemeinsamem Speicher
 - für FORTRAN, C und C++
- 

OpenMP Programmiermodell

- OpenMP Direktiven basieren in C and C++ auf `#pragma` Compilerdirektiven.

- Eine Direktive besteht aus einem Namen und einer Klauselliste:

```
#pragma omp directive [clause list]
```

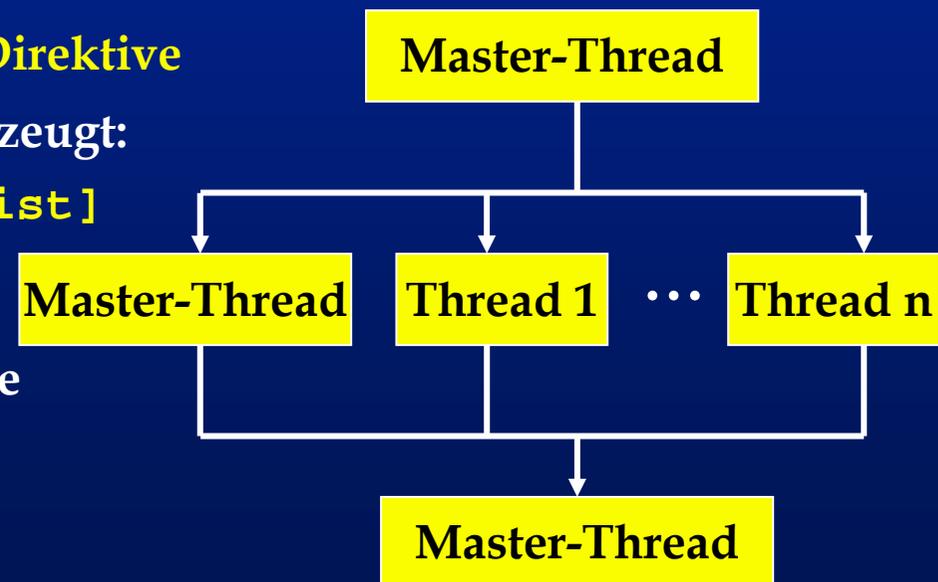
- OpenMP Programme werden sequentiell ausgeführt, bis sie auf eine `parallel` Direktive stoßen, die eine Gruppe von Threads erzeugt:

```
#pragma omp parallel [clause list]
```

```
/* structured block */
```

- Der Thread, der die `parallel` Direktive ausführt, wird zum *Master* der Threadgruppe. Er erhält die ThreadId 0.

Fork-Join-Modell



Beispiel: Hello World

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {
    printf("Anzahl Prozessoren: %d\n", omp_get_num_procs());
    #pragma omp parallel
    { printf ("Thread %d von %d sagt \"Hallo Welt!\"\n",
              omp_get_thread_num(), omp_get_num_threads());
    }
    printf("Fertig.\n"); return 0;
}
```

Parallelisierung von Schleifen

- Haupteinsatzgebiet von OpenMP
- Jeder Thread führt den Schleifencode für eine andere Menge von Iterationen aus.

Bsp: SAXPY: $y = a * x + y$ (scalar a x plus y)

```
for saxpy (const float a, const vector<float>& x,  
          const vector<float>& y)  
{  
    assert (x.size() == y.size());  
    #pragma omp parallel for  
    for (int i = 0; i <x.size(); i++)  
    {  
        y[i] += a * x[i]  
    }  
}
```

Pragmas *for* und *parallel for*

- Die **for Direktive** dient der Aufteilung von Schleifendurchläufen auf mehrere Threads:

```
#pragma omp for [clause list]
```

```
/* for loop */
```

- **parallel for** ist eine Zusammenfassung der **parallel** und der **for** Direktive, wenn genau eine Schleife parallelisiert werden soll.
- Form parallelisierbarer Schleifen:

```
for index = start; index { <
                        <=
                        >=
                        > } end; { index++ / ++ index
                                   index -- / -- index
                                   index += inc / index -= inc
                                   index = index + inc / - inc
                                   index = index - inc }
```

Zugriff auf Variablen

- Default: Alle Threads können auf alle Variablen im parallelen Codeabschnitt zugreifen.
- **Datenzugriffsklauseln** `klausel (variable, variable,...)`
 - **shared und private:**
von allen Threads gemeinsam oder von einem Thread privat genutzt
 - **firstprivate und lastprivate:**
Initialisierung und Finalisierung der Werte privater Variablen
 - **default:**
Abänderung des Defaults
 - **reduction:**
gemeinsam genutzte Variablen, in denen mehrere Threads Werte akkumulieren können

Beispiel: Berechnung von π

```
const double delta_x = 1.0 / num_iter;
double sum = 0.0;
double x, f_x;
int i;
#pragma omp parallel for private(x, f_x) \
                        shared(sum)
for (i = 1; i <= num_iter; i++) {
    x = delta_x * (i-0.5);
    f_x = 4.0 / (1.0 + x*x);
#pragma omp critical
    sum += f_x;
}
return delta_x * sum;
```

Pragma critical

- `#pragma omp critical`
stellt wechselseitigen Ausschluss bei der Ausführung des nachfolgenden Code-Blocks sicher

- => Synchronisation aller Threads
- => Ausbremsen paralleler Threads

- => effizientere Methode: Reduktionsvariablen

Reduktionsklausel

- Die Reduktionsklausel ermöglicht die Reduktion der Werte mehrerer privater Kopien einer Variablen mittels eines Operators zu einem Wert, den die Variable beim Verlassen des parallelen Blocks im Master erhält.
- Die Klausel hat die Gestalt:
`reduction (operator: variable list).`
- Die Variablen in der Liste werden implizit als privat festgelegt.
- Mögliche Operatoren sind: `+`, `*`, `&`, `|`, `^`, `&&`, `||`.

Beispiel:

```
#pragma omp parallel reduction(+: sum) num_threads(8) {  
/* compute local sums here */  
}  
/*sum here contains sum of all local instances of sums */
```

Beispiel: Berechnung von π mit Reduktion

```
const double delta_x = 1.0 / num_iter;
double sum = 0.0;
double x, f_x;
int i;
#pragma omp parallel for private(x, f_x) \
    reduction(+: sum)
for (i = 1; i <= num_iter; i++) {
    x = delta_x * (i-0.5);
    f_x = 4.0 / (1.0 + x*x);
    sum += f_x;
}
return delta_x * sum;
```

Reduktionsoperatoren

Operator	Bedeutung	Typen	Neutrales Element
+	Summe	float, int	0
*	Produkt	float, int	1
&	bitweises Und	int	alle Bits 1
	bitweises Oder	int	0
^	bitweise XOR	int	0
&&	Logisches Und	int	1
	Logisches Oder	int	0

Initialisierung von privaten Variablen

- Private Variablen sind beim Eintritt und beim Verlassen eines parallelen Abschnitts undefiniert.
- Die Klausel **firstprivate** dient dazu, private Variablen mit dem Wert der Variablen im Masterthread zu initialisieren.
- Die Initialisierung erfolgt nur einmal pro Thread und nicht z.B. einmal pro Schleifendurchlauf in einer parallelen Schleife.

Beispiel:

```
x[0] = complex_function();  
#pragma omp parallel for private(j) firstprivate(x)  
for (i=0; i<n; i++) {  
    for (j=1; j<4; j++)  
        x[j] = g(i, x[j-1]);  
    answer[i] = x[1] - x[3];  
}
```

Finalisierung von privaten Variablen

- Die Klausel **lastprivate** ermöglicht es, den Wert der Variablen, den sie bei sequentieller Ausführung in der letzten Iteration angenommen hätte, der Variablen im Masterthread zuzuweisen.

Beispiel:

```
#pragma omp parallel for private(j) lastprivate(x)
for (i=0; i<n; i++) {
    x[0] = 1.0;
    for (j=1; j<4; j++) x[j] = x[j-1] * (i+1);
    sum_of_powers[i] = x[0] + x[1] + x[2] + x[3];
}
n_cubed = x[3];
```

- Eine Variable darf sowohl als **firstprivate** als auch als **lastprivate** deklariert werden. Es wird sichergestellt, dass alle Initialisierungen vor der Finalisierung abgeschlossen sind.

Änderung des Defaults

- Die Klausel `default` erlaubt es, das Standardverhalten (`shared`) für nicht explizite Variablen zu ändern.
- Interessant ist lediglich die Festlegung `default none`.
- Die Option `none` führt dazu, dass der Compiler für jede Variable, die nicht in einer Zugriffsklausel (`private`, `shared`, `firstprivate`, `lastprivate`, `reduction`) auftritt, eine Fehlermeldung ausgibt und die Compilierung unterbricht.
- Bei der nachträglichen Parallelisierung seriellen Codes können auf diese Weise alle Variablen im aktuellen Gültigkeitsbereich gefunden und analysiert werden.

Ablaufpläne mit *schedule*

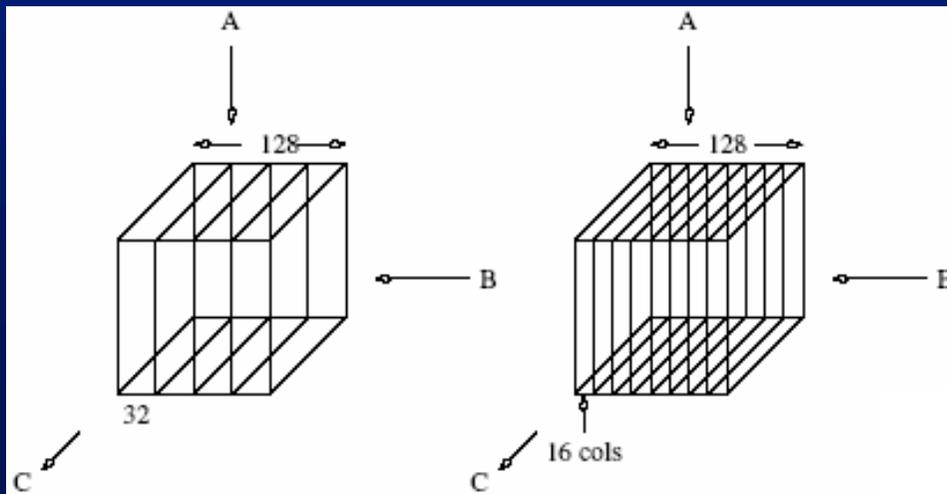
- allgemeine Form: `schedule(scheduling_class[, parameter])`.
- OpenMP unterstützt vier scheduling Klassen: `static`, `dynamic`, `guided`, and `runtime`.

Typ	chunk?	Iterationen pro Stück	Stücke	Bezeichnung
<code>static</code>	nein	n/p	p	einfach, statisch
<code>static</code>	ja	c	n/c	überlappend
<code>dynamic</code>	optional	c	n/c	einfach, dynamisch
<code>guided</code>	optional	anfangs n/c, dann abnehmend	< n/c	geführt
<code>runtime</code>	nein	unterschiedlich	unterschiedl.	unterschiedl.

Kostenzunahme

Beispiel: Matrixmultiplikation

```
/* static scheduling of matrix multiplication loops */  
#pragma omp parallel default(private) shared (a, b, c, dim) \  
    num_threads(4)  
#pragma omp for schedule(static)  
for (i = 0; i < dim; i++) {  
    for (j = 0; j < dim; j++) {  
        c(i,j) = 0;  
        for (k = 0; k < dim; k++) {  
            c(i,j) += a(i, k) * b(k, j); }  
    }  
}
```



schedule(static)

schedule(static,16)

Weitere OpenMP Klauseln

- Bedingte Parallelisierung: `if (skalärer Ausdruck)`
legt fest, ob ein paralleles Konstrukt die Erzeugung von Threads bewirkt
- Nebenläufigkeitsgrad: `num_threads(integer expression)`
spezifiziert die Anzahl der Threads, die erzeugt werden sollen

Beispiel: OpenMP Klauseln

```
#pragma omp parallel if (is_parallel==1) num_threads(8) \  
  private (a) shared (b) firstprivate(c) {  
  /* structured block */  
}
```

- Falls die Variable `is_parallel` den Wert 1 hat, werden 8 Threads erzeugt.
- Jeder dieser Threads erhält private Kopien der Variablen `a` und `c`. Variable `b` wird gemeinsam genutzt.
- Der Wert von jeder Kopie von `c` wird mit dem Wert von `c` vor der parallelen Direktive initialisiert.
- Ohne Angabe werden Variablen gemeinsam genutzt. Mit der `default` Klausel kann dies geändert werden.

OpenMP -> PThreads

```
int a, b;
main() {
  // serial segment
  #pragma omp parallel num_threads (8) private (a) shared (b)
  {
    // parallel segment
  }
  // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
  // serial segment
  Code inserted by the OpenMP compiler
  for (i = 0; i < 8; i++)
    pthread_create (....., internal_thread_fn_name, ...);
  for (i = 0; i < 8; i++)
    pthread_join (.....);
  // rest of serial segment
}
void *internal_thread_fn_name (void *packaged_argument) {
  int a;
  // parallel segment
}
```

Corresponding Pthreads translation

Datenabhängigkeiten

- notwendige Voraussetzung für Parallelisierung:
wechselseitige Unabhängigkeit der Ergebnisse der einzelnen Iterationen einer Schleife
- Seien zwei Anweisungen A1 und A2 gegeben, wobei A1 in der sequentiellen Ausführung vor A2 liege.
 - **echte Datenabhängigkeit (Flussabhängigkeit, RAW- read after write)**
A1 schreibt in eine Speicherzelle, die von A2 gelesen wird.
 - **Gegenabhängigkeit (WAR – write after read)**
A1 liest eine Speicherzelle, die anschließend von A2 geschrieben wird.
 - **Ausgabeabhängigkeit (WAW – write after write)**
A1 und A2 schreiben in dieselbe Speicherzelle.

Beispiel: Schleifenparallelisierung

```
for (i = 0; i < m; i++)
{
    low = a[i]; high = b[i];
    if (low > high) {
        printf ("Exiting during iteration %d\n", i);
        break;
    }
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i]) / b[i];
}
```

Wegen `break` ist die äußere Schleife nicht parallelisierbar.

Die innere Schleife wäre parallelisierbar, aber `fork/join` pro Iteration von äußerer Schleife soll vermieden werden.

Beispiel: Lösungsansatz

```
#pragma omp parallel private (i,j)
  for (i = 0; i<m; i++)
    { low = a[i]; high = b[i];
      if (low > high) {
        printf ("Exiting during iteration %d\n",i);
        break;
      }
    }
#pragma omp for
  for (j=low;j<high;j++)
    c[j] = (c[j] - a[i]) / b[i];
}
```

Aber die Fehlermeldung wird von jedem Thread ausgegeben.

Pragma single

Das Pragma `#pragma omp single` weist den Compiler an, dass der nachfolgende Codeblock nur von einem Thread ausgeführt werden soll.

```
#pragma omp parallel private (i,j)
```

```
    for (i = 0; i < m; i++)  
        { low = a[i]; high = b[i];  
          if (low > high) {
```

```
#pragma omp single
```

```
    printf ("Exiting during iteration %d\n", i);  
    break;  
    }
```

```
#pragma omp for
```

```
    for (j=low; j < high; j++)  
        c[j] = (c[j] - a[i]) / b[i];
```

```
    }
```

Klausel *nowait*

Die Klausel `nowait` lässt den Compiler die implizite Barrierensynchronisation am Ende einer Schleife aufheben.

```
#pragma omp parallel private (i,j, low, high)
  for (i = 0; i<m; i++)
    { low = a[i]; high = b[i];
      if (low > high) {
#pragma omp single
          printf ("Exiting during iteration %d\n",i);
          break;
      }
#pragma omp for nowait
          for (j=low;j<high;j++)
              c[j] = (c[j] - a[i]) / b[i];
    }
}
```

Arbeit aufteilende Direktiven

- **Parallelisierung von Schleifen**

Die for Direktive dient der Aufteilung von Schleifendurchläufen auf mehrere Threads:

```
#pragma omp for [clause list]
    /* for loop */
```

Mögliche Klauseln sind dabei: `private`, `firstprivate`, `lastprivate`, `reduction`, `schedule`, `nowait`.

- **Parallele statische Abschnitte**

Eine Menge voneinander unabhängiger Codeblöcke (**sections**) wird auf die Threads eines Teams aufgeteilt und von diesen nichtiterativ parallel ausgeführt.

Die sections Direktive

- OpenMP unterstützt nicht-iterative Parallelisierungen mittels der `sections` Direktive.
- allgemeine Form:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```



Beispiel

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

 #pragma omp parallel sections

Synchronisationskonstrukte in OpenMP

<code>#pragma omp barrier</code>	explizite Barriere
<code>#pragma omp single [clause list] structured block</code>	Ausführung durch einen Thread
<code>#pragma omp master structured block</code>	Ausführung durch Master Thread (keine implizite Barriere)
<code>#pragma omp critical [(name)] structured block</code>	kritischer Abschnitt mit globalem Namen -> wechselseitiger Ausschluss in allen kritischen Abschnitten gleichen Namens
<code>#pragma omp atomic assignment</code>	atomare Zuweisung

OpenMP Bibliotheksfunktionen

```
/* Thread- und Prozessorzaehler */  
void omp_set_num_threads (int num_threads);  
int omp_get_num_threads ();  
int omp_get_max_threads ();  
int omp_get_thread_num ();  
int omp_get_num_procs ();  
int omp_in_parallel();
```

OpenMP Bibliotheksfunktionen

```
/* Dynamische Threadanzahl / Geschachtelte Par. */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();

/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

Alle lock Routinen haben ein Gegenstück (`_nest_lock`) für rekursive Mutexe.

Umgebungsvariablen in OpenMP

- **OMP_NUM_THREADS**
Festlegung der Standardanzahl zu erzeugender Threads
- **OMP_SET_DYNAMIC**
Festlegung, ob die Threadanzahl dynamisch geändert werden kann
- **OMP_NESTED**
Ermöglichung geschachtelter Parallelität
- **OMP_SCHEDULE**
Scheduling von for-Schleifen falls die Klausel `runtime` festlegt

Explizite Threads (PThreads) vs Direktiven (OpenMP)

- Direktiven vereinfachen viele Aufgaben, wie zum Beispiel:
 - Initialisierung von Attributobjekten für Threads
 - Argumentzuweisung an Threads
 - Parallelisierung von Schleifen etc.
- Es gibt aber auch Nachteile:
 - versteckter Mehraufwand durch impliziten Datenaustausch
 - Explizite Threads bieten ein umfangreicheres API, zum Beispiel
 - condition waits
 - verschiedene Sperrmechanismen (locks)
 - Explizite Threads bieten mehr Flexibilität zur Definition eigener Synchronisationsoperationen.

Beispielprogramm: Berechnung von pi

```
/* *****  
An OpenMP version of a threaded program to compute PI.  
***** */  
sum=0; sample_points_per_thread = sample_points / num_threads;  
#pragma omp parallel \  
private(rand_no_x, rand_no_y, seed, i) \  
shared(sample_points, sample_points_per_thread) \  
reduction(+: sum) num_threads(num_threads)  
{ seed = omp_get_thread_num();  
  for (i = 0; i < sample_points_per_thread; i++) {  
    rand_no_x =(double)(rand_r(&seed))/(double)((2<<30)-1);  
    rand_no_y =(double)(rand_r(&seed))/(double)((2<<30)-1);  
    if (( (rand_no_x - 0.5) * (rand_no_x - 0.5) +  
        (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)  
        sum ++; }  
}
```

Beispiel: Berechnung von π mit Schleifenparallelisierung

```
sum = 0;
#pragma omp parallel private(rand_no_x, rand_no_y, seed) \
    shared(sample_points) reduction(+:sum) \
    num_threads(num_threads)
{ num_threads = omp_get_num_threads();
  seed = omp_get_thread_num();
#pragma omp for
  for (i = 0; i < sample_points; i++) {
    rand_no_x = (double)(rand_r(&seed)) / (double)((2<<30)-1);
    rand_no_y = (double)(rand_r(&seed)) / (double)((2<<30)-1);
    if (( (rand_no_x - 0.5) * (rand_no_x - 0.5) +
          (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
        sum ++; }
}
```