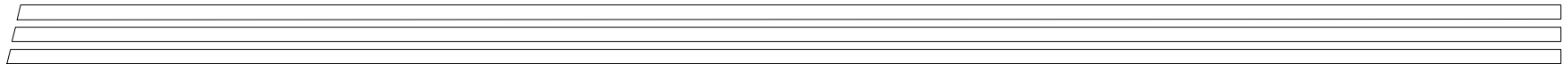


Beispiel: Schleifenparallelisierung

```
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.

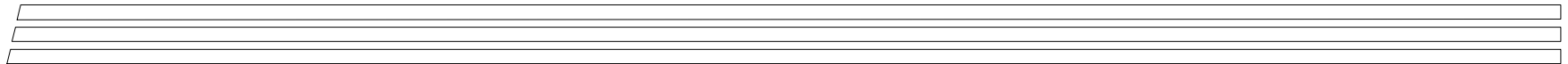
Die innere Schleife wäre parallelisierbar, aber fork/join pro Iteration von äußerer Schleife soll vermieden werden.



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++)
```

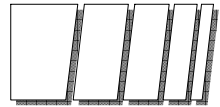
```
                    c[j] = (c[j] - a[i]) / b[i];
```

```
            }
```

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**

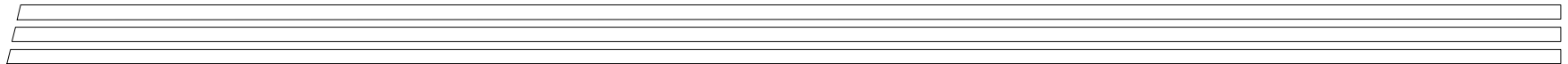
Die **for** Direktive dient der Aufteilung von Schleifendurchläufen auf mehrere Threads:

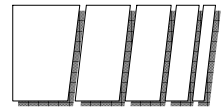
```
#pragma omp for [clause list]
    /* for loop */
```

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.

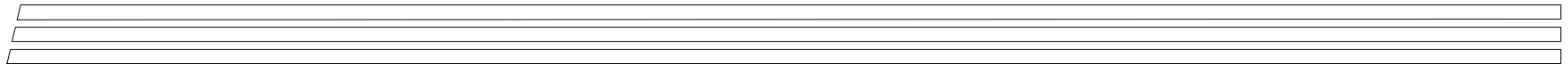




Die sections Direktive

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

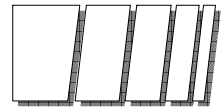
```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```



Beispiel

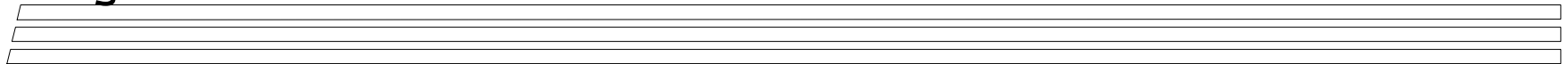
```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
}
```

#pragma omp parallel sections



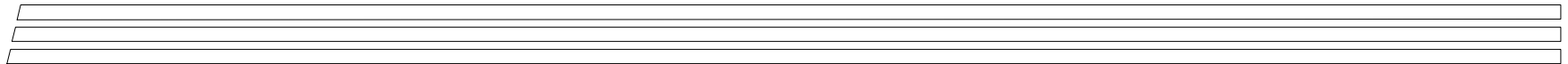
Synchronisationskonstrukte in OpenMP

<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();
```

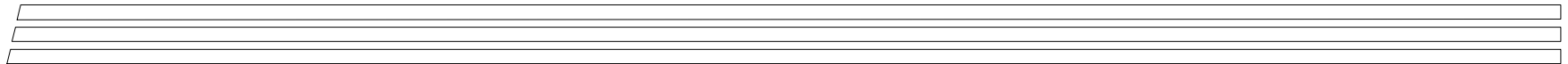


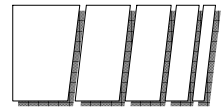
OpenMP Bibliotheksfunktionen

```
/* 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);
```

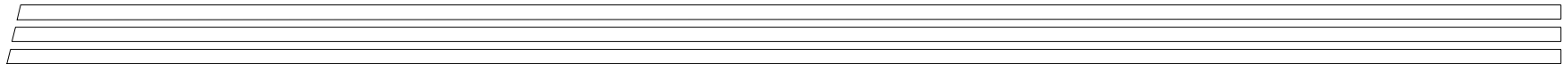
Alle lock Routinen haben ein Gegenstück (`_nest_lock`) für rekursive Mutexe.

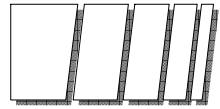




Umgebungsvariablen in OpenMP

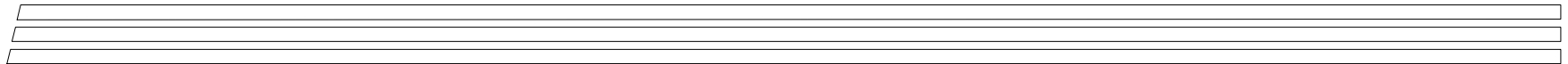
- **OMP_NUM_THREADS**
Festlegung der Standardanzahl zu erzeugender Threads
- **OMP_SET_DYNAMIC**
Festlegung, ob die Threadanzahl dynamisch geändert werden kann
- **OMP_NESTED**
Ermöglichung geschachtelter Parallelität
- **OMP_SCHEDULE**
Scheduling von for-Schleifen falls die Klausel `runtime` festlegt

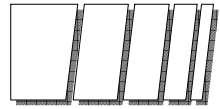




Explizite Threads (PThreads) vs Direktiven (OpenMP)

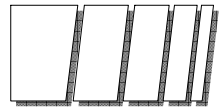
- **Direktiven vereinfachen viele Aufgaben, wie zum Beispiel:**
 - **Initialisierung von Attributobjekten für Threads**
 - **Argumentzuweisung an Threads**
 - **Parallelisierung von Schleifen etc.**
- **Es gibt aber auch Nachteile:**
 - **versteckter Mehraufwand durch impliziten Datenaustausch**
 - **Explizite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





Beispielprogramm: Berechnung von π

```
/* *****  
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 ++;  }
}
```

