OpenMP

```
#pragma omp parallel [clauses]
```

Man kann parallele Regionen ineinander verschachteln, allerdings wird nur ein Thread der inneren Region zugewiesen. Mit dem Befehl kann mehr als ein Thread eine verschachtelte parallele Region ausführen.

```
void omp_set_nested(int nested)
```

Wichtige Befehle:

Diese Befehle sind *Thread safe*, daher können sie ohne Probleme aufgerufen werden:

```
int omp_get_thread_num(void); // Thread id
int omp_get_num_threads(void); // Anzahl Threads
int omp_get_max_threads(void); //  Max. Anz. an Threads
int omp_set_num_threads(int num_threads); // Anz. der Threads setzen
```

Um die Zeit für eine parallele Ausführung zu messen, gibt es folgende Befehle:

```
double omp_get_wtime(void); // Gibt die Uhrzeit seit einem bestimmten
Zeitpunkt in der Vergangenheit zurück.
double omp_get_wtick(void);
```

Standardmäßig werden alle Variablen die vor dem parallelen Bereich definiert wurden, im parallelen Bereich geshared. Wenn die Variablen **private(..)** gemacht werden, dann erzeugt der Compiler lokale Kopien der Variablen, aber diese Variablen werden nicht initialisiert. Mit **firstprivate(..)** werden die Variablen mit dem Wert initalisiert, den sie vor der parallelen Region hatten. Allerdings kann dies sehr lange dauern.

```
private(<list of variables>)
firstprivate(<list of variables>)
shared(<list of variables>)
default(shared|none)
```

Work Sharing: Master and Single

Die Arbeit in diesem Konstrukt wird vom Master Thread erledigt. Andere Threads überspringen diesen Block. Es gibt keine Barrier Synchronisierung, nach dem Code. Der Code wird nicht unter Mututal Exclusion ausgeführt. Daher können die geteilten Variablen verändert werden.

```
#pragma omp master
<structured statement>
```

Ein Thread führt diesen Block aus. Es können mehrere single Blöcke exisiteren und jeder könnte von einem anderen Thread ausgeführt werden. Die Region wird auch nicht unter Mutual Exclusion augeführt. Aber es gibt eine implizite Barriere nach dem Statement. Das bedeutet, dass alles Threads diesen Punkt erreichen müssen. Mit der **nowait** clause kann die implizite Barriere ausgeschalten werden.

```
#pragma omp single [clauses]
<structured statement>
```

Der Code für **single** oder **master** Threads sollte kurz gehalten werden.

The explicit Barrier

Mit diesem Befehl kann man eine Barriere setzen, wo alle Threads zusammen warten müssen.

```
#pragma omp barrier
```

Work sharing: Sections

Die Schritte eines Algorithmus können manchmal als einzelne unabhängige Teile ausgedrückt werden. Diese Teile können parallel von Threads ausgedrückt werden.

```
#pragma omp sections [clauses]
<section block>
```

Jeder unabhängige Teil des Codes ist als solcher markiert.

```
#pragma omp section [clauses]
<section block>
```

Ein Block von **sections** endet mit einer impliziten Barriere. Kein Thread kann fortfahren, bis nicht alle Sections abgearbeitet wurden. Mit **nowait** kann dieses Verhalten umgangen werden. Variablen können auch mit **private(..)** oder **firstprivate(..)** markiert werden.

Beispiel:

```
#pragma omp parallel
    {

#pragma omp sections
        {
#pragma omp section
            {
                printf("Thread %d ready. pi=%f WAIT\n",
omp_get_thread_num(), calcPi(10));
            }
#pragma omp section
            {
                printf("Thread %d ready. pi=%f WAIT\n",
omp_get_thread_num(), calcPi(10000));
            }
#pragma omp section
            {
                printf("Thread %d ready. pi=%f WAIT\n",
omp_get_thread_num(), calcPi(1000000000));
            }
        }
#pragma omp single
        printf("================ WAIT Section ready. ================\n");

#pragma omp sections nowait
        {
#pragma omp section
            {
                printf("Thread %d ready. pi=%f NOWAIT\n",
omp_get_thread_num(), calcPi(10));
            }
#pragma omp section
            {
                printf("Thread %d ready. pi=%f NOWAIT\n",
omp_get_thread_num(), calcPi(10000));
            }
#pragma omp section
            {
                printf("Thread %d ready. pi=%f NOWAIT\n",
omp_get_thread_num(), calcPi(1000000000));
            }
        }
#pragma omp single
        printf("================ NOWAIT Section ready.
================\n");
    }
}
```

Output:

```
Calculating pi for Thread 0 with 1000000000 iterations...
Calculating pi for Thread 1 with 10 iterations...
Thread 1 ready. pi=3.142426 WAIT
Calculating pi for Thread 3 with 10000 iterations...
Thread 3 ready. pi=3.141593 WAIT
Thread 0 ready. pi=3.141593 WAIT
================ WAIT Section ready. ================
Calculating pi for Thread 0 with 10 iterations...
Calculating pi for Thread 3 with 10000 iterations...
Calculating pi for Thread 2 with 1000000000 iterations...
Thread 3 ready. pi=3.141593 NOWAIT
================ NOWAIT Section ready. ================
Thread 0 ready. pi=3.142426 NOWAIT
Thread 2 ready. pi=3.141593 NOWAIT

Process finished with exit code 0
```

Anhand dieses Beispiels ist ersichtlich, dass jede **section** von einem Thread ausgeführt wird. Wird **nowait** spezifiziert, wird die implizite Barriere am Ende von **sections** deaktiviert, weshalb die Threads 0 und 2 erst nach der *"NOWAIT Section ready."* Ausgabe beenden.

Work sharing: Loops of Independant Iterations

```
#pragma omp for [clauses] for (<canonical form loop range>)
<loop body>
```

Alle Threads müssen die gleichen Werte für die Start- und Enditerationen berechnen können und müssen den selben increment verwenden. Upper Bound Condition muss die Form i<n, i<=n, i>n, i>=n, i!=n haben. Increment muss folgende Form haben i++, i+=inc, i=i+inc. Schleifen, die diese Bedingung erfüllen, sind kanonisch.

Kurzform für eine parallele Region mit Schleifen. Eine Schleife erlaubt kein **nowait**.

```
#pragma omp parallel for [clauses]
for (<canonical form loop range>)
<loop body>
```

Schleifeniterationen dürfen keine Dataraces verursachen. Schleifeniterationen müssen unabhängig sein. Einfache Regeln für unabhängige Schleifen sind, dass nur Array Elemente upgedatet werden und das in jeder Iteration maximal ein Element geändert wird.

//TODO

Loop Scheduling

Loop scheduling bezeichnet die Zuweisung von Schleifeniterationen zu Threads.

```
// Spezifiziert eine statische Verteilung der Iterationen. Die Anzahl der
Iteration wird durch die chunksize dividiert und so aufgeteilt. Wenn keine
chunksize angegeben wird, dann werden die Iterationen gleichmäßig
aufgeteilt.
schedule(static, chunksize)

// Dynamische Zuweisung von Iterationen an Threads. Wenn keine chunksize
angegeben wurde, dann wird chunksize = 1 verwendet.
schedule(dynamic, chunksize)

// Spezifiziert eine dynamische Zuweisung von Iterationen an Threads mit
sinkender Größe.
schedule(guided, chunksize)
```

```
// Die Scheduling Strategie wird dem Compiler/Runtime-System überlassen.
schedule(auto)

// Scheduling wird während der Laufzeit bestimmt. Zur Laufzeit wird die
variable *OMP_SCHEDULE* ausgelesen.
schedule(runtime)
```

Collapsing nested loops

Viele Berechnungen werden mithilfe von verschachtelten Schleifen ausgeführt.

```
#pragma omp parallel for collapse(2)
for (i=0; i<n; i++) {
    // OpenMP will make one loop out of two
    for (j=0; j<m; j++) {
        x[i][j] = f(i,j);
    }
}
```

Reductions

Erlaubte Operatoren sind: +, - , *, &, |, ^, &&, ||, min und max.

```
reduction(<reduction operator>:<reduction variables>)
```

Beispiel:

```
double calcPi(int iterations) {
    printf("Calculating pi for Thread %d with %d iterations...\n",
omp_get_thread_num(), iterations);
    long double pi = 0.0l;
    const double deltaX = 1.0l / iterations;
    double x, fx;
    int i;

    #pragma omp parallel for private (x, fx) reduction(+:pi)
    for (i = 1; i <= iterations; i++) {
        x = deltaX * (i - 0.5);
        fx = 4.0 / (1.0 + x * x);
        pi += fx;
    }
    return deltaX * pi;
}
```

Work sharing: Tasks and Task Graphs

```
#pragma omp task [clauses]
<structured statement>
```

//TODO

Mutual Exclusion Constructs

Um Data Races zu vermeiden, bietet OpenMP Unterstützung für Mutual Exclusion.

```
#pragma omp critical [(name)]
<structured statement>
```

Threads führen eine Critical Section unter Mutual Exclusion aus. Maximal ein Thread kann den Code einer critical section ausführen. In einer critical section können shared Variablen gelesen, geupdated werden. Solche Sektionen sollten spärlich eingesetzt werden, denn sie können zu Serialisierung führen.

Wenn die Operation in der kritischen Sektion eine bestimmte Form hat, dann ist es auch möglich eine atomare Operation auszuführen.

```
#pragma omp atomic [read|write|update|capture]
<structured statement>
```

Locks

OpenMP bietet Locks.

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

Special Loops

```
#pragma omp parallel for simd [clauses]
for (<canonical form loop range>)
<loop body>
```

Parallelzing Loops with Hopeless Dependencies

Mit diesem Befehl kann ein Teil einer Schleife iterativ ausgeführt werden.

```
#pragma omp ordered
<structured statement>
```