OpenMP

History

OpenMP started as a joined initiative of most of the major hardware and software vendors to provide compiler support for parallel programs, with a main focus on parallelising loops.

- 1997: OpenMP 1.0 for Fortran
- 1998: OpenMP 1.0 for C/C++
- 2000: OpenMP 2.0 for Fortran with fixes and clarifications
- 2002: OpenMP 2.0 for C/C++
- 2005: OpenMP 2.5 combining Fortran and C/C++
- 2008: OpenMP 3.0 introduces tasks
- 2011: OpenMP 3.1
- 2013: OpenMP 4.0rc2 with support for vector units and special targets

Beside the compiler directives (pragmas), OpenMP also contains a set of data types and functions, imported via the header file `omp.h`.

All major C/C++ and Fortran compilers support OpenMP, with Clang being the only important exception.

Hello World

The standard starting point for each programming lecture is:

```
#include <iostream>
int main () {
  #pragma omp parallel
  std::cout << "Hello, world!" << std::endl;
}
```

To enable OpenMP during compilation, a special compiler flag has to be provided:

Intel Compiler

```
> icpc -openmp -o hello -c hello.cc
```

GNU Compiler

```
> g++ -fopenmp -o hello -c hello.cc
```

Possible program output on a Quad-Core CPU may then be:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

```
Hello, world!Hello, world!Hello,
      world!
Hello, world!
```

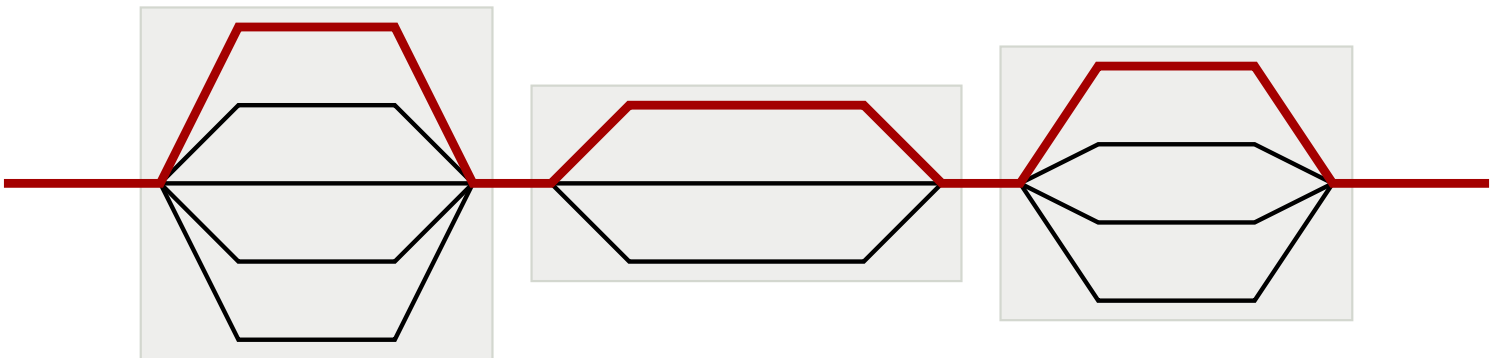
Remark

The garbled output is due to a race condition between all threads competing for the same output channel.

Fork-Join Model

The underlying model of OpenMP is the *fork-join model*:

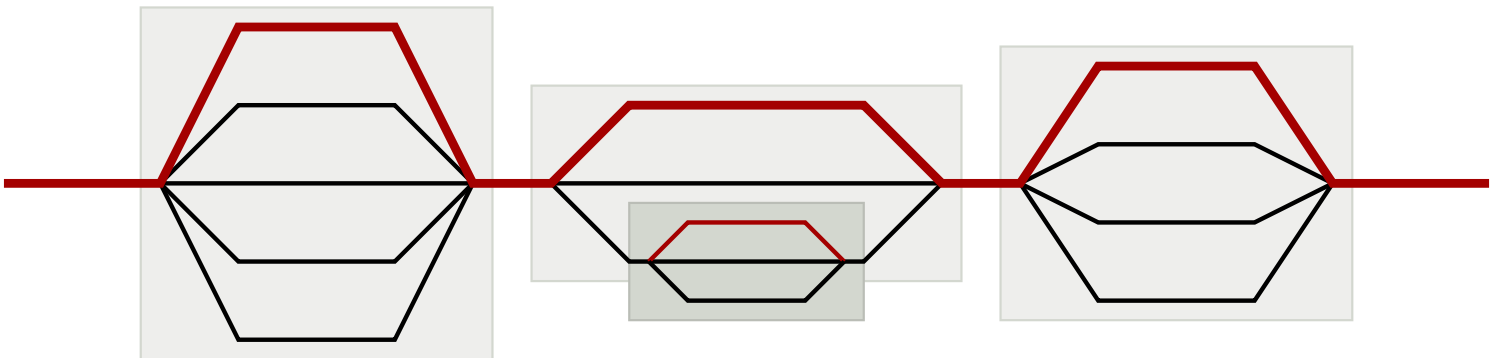
> *A master thread creates a team of worker threads which run in parallel together with the master thread until all worker threads finish.*



Fork-Join Model

The underlying model of OpenMP is the *fork-join model*:

> *A master thread creates a team of worker threads which run in parallel together with the master thread until all worker threads finish.*



The fork-join model is recursive: any thread in a team may create new threads, which form a new team. This is known as *nested parallelism*.

The number of threads in a team is user defined and may vary between different parallel sections of the program.

When all threads of a team are joined, a synchronisation takes place, i.e. the master thread of the team will only proceed when all other team threads have finished.

OpenMP Directives

OpenMP extends the underlying programming language by compiler *directives*.

Each OpenMP directive starts with the pragma

```
#pragma omp <directive> new-line
```

Remark

The new-line is mandatory!

The directive is applied to the immediately following source code block, i.e.

```
#pragma omp <directive>
{
  ...
}
```

or just

```
#pragma omp <directive>
  ...
```

in case of a single-line block.

After the source code block, the worker threads of the thread team will stop and only the master thread will proceed.

OpenMP Directives

Two important concepts appear in the context of OpenMP directives.

Construct

The *construct* of an OpenMP directive includes only the block of source code directly following the directive.

Region

The (parallel) *region* of an OpenMP directive is the set of all code executed by a thread of the created team. This also includes all code executed in called functions.

```
void f () {
  ...                    // part of the region but NOT the construct
}

#pragma omp <directive>
{
  ...                    // construct of the directive
  f();
  ...
}
```

OpenMP Directives

Parallel vs Sequential Mode

In case OpenMP is not supported or explicitly turned off during compilation, all OpenMP directives will be ignored and the output is a standard sequential program.

```
> g++ -Wall -o hello hello.cc
hello.cc:5:0: warning: ignoring #pragma omp parallel [-Wunknown-pragmas]
```

Most OpenMP implementations will support parallel *and* sequential mode of programs containing OpenMP directives. Albeit, it is legal to develop a program, which will only work correctly in parallel mode!

Furthermore, the output of the program may vary between sequential and parallel mode.

Thread Creation

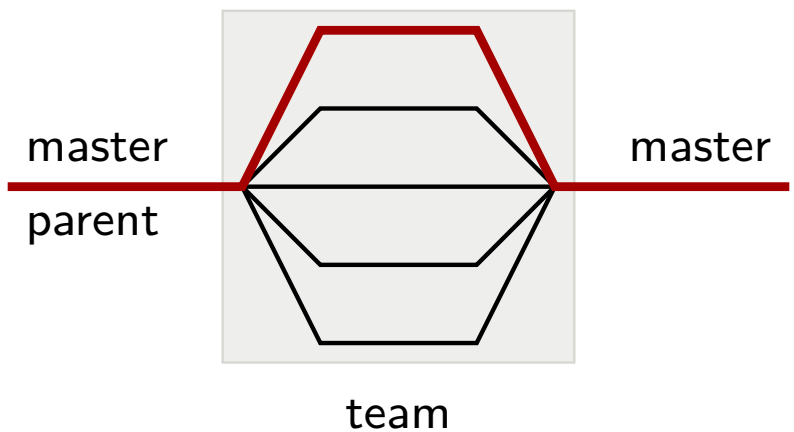
Thread Creation

A new team of threads is created by the directive

```
#pragma omp parallel [clause1 [[,] clause2, ...]]
```

The thread, which initially encounters the `parallel` directive is called the *master* thread and furthermore, becomes the *parent* thread to all threads in the newly created team.

```
void f () {
  ...                          // executed by master thread
  #pragma omp parallel
  {
    ...                        // executed by team
  }
  ...                          // executed by master thread
}
```



Thread Creation

The default number of threads in the team (including the master thread) is chosen by the OpenMP library, and usually equal to the number of processors (cores) in the system.

Instead of this, the following options may be used to change the number of team threads:

- With the clause `num_threads()`:

```
#pragma omp parallel num_threads(4)
```

- With the function `omp_set_num_threads()`:

```
#include <omp.h>
int main () {
  omp_set_num_threads( 4 );
  #pragma omp parallel
  { ... }
}
```

- Using the environment variable `OMP_NUM_THREADS`:

```
> export OMP_NUM_THREADS=4   # bash
```

or

```
> setenv OMP_NUM_THREADS 4   # tcsh
```

Thread Creation

Remark

A typical programming mistake is to forget the `parallel` directive:

```
int main () {
  #pragma omp             // no "parallel"
  {
    ...
  }
}
```

In this case, no new threads are created and the construct will be executed by the master thread alone, i.e. sequential execution.

Thread Creation

Remark

A typical programming mistake is to forget the `parallel` directive:

```
int main () {
  #pragma omp             // no "parallel"
  {
    ...
  }
}
```

In this case, no new threads are created and the construct will be executed by the master thread alone, i.e. sequential execution.

Thread ID

Each thread of a team has an id $< p$ which is unique for the corresponding parallel region. The master thread will always have id 0.

This id may be obtained using the function `omp_get_thread_num()`:

```
#include <omp.h>
int main () {
  #pragma omp parallel
  {
    const int  thread_id = omp_get_thread_num();
    ...
  }
}
```

Nested Parallelism

The `parallel` directive may also be *nested*:

```
void f () {
  #pragma omp parallel
  {
    ...
    #pragma omp parallel
    {
      ...
    }
    ...
  }
}
```

Usually, this nested mode has to be enabled *explicitly* using either the environment variable `OMP_NESTED`:

```
> export OMP_NESTED=TRUE   # bash
> setenv OMP_NESTED TRUE   # tcsh
```

or with the OpenMP function `omp_set_nested()`:

```
omp_set_nested( true );
```

If nested parallelism is not enabled, the nested directives will be handled by only one thread, i.e. sequentially.

Nested Parallelism

Since, by default, the number of threads for a parallel region equals the number of processors, the second parallel region will *not* spawn new threads.

Hence, nested parallel regions should be used together with the `num_threads` clause:

```
void f () {
  #pragma omp parallel num_threads(4)
  {
    ...
    #pragma omp parallel num_threads(2)
    {
      ...
    }
    ...
  }
}
```

Here, each of the 4 worker threads will create a new team with 2 threads each, i.e. the inner construct is executed by 8 threads.

Nested Parallelism

Nested parallelism may be used within the whole region of a `parallel` directive:

```
void g () {
  #pragma omp parallel      // inside parallel region of f()
  {
    ...
  }
}

void f () {
  #pragma omp parallel
  {
    ...
    g();
    ...
  }
}
```

This also allows recursive computations to be handled by nested parallel regions:

```
void f () {
  #pragma omp parallel num_threads(2)
  {
    f();
  }
}
```

Here, each call of `f()` will spawn 2 new threads until all processors are used.

Shared vs. Private Data

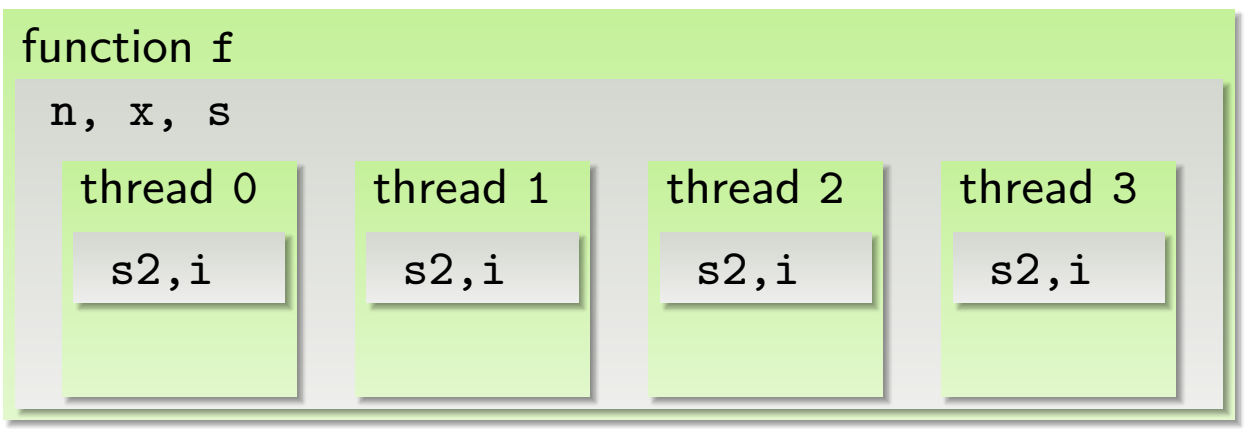
All data defined in the surrounding block of a parallel construct is *shared* by all threads.

Data defined within a parallel construct is *private* to a specific thread.

At line 4, the variables `n`, `x` and `s` are defined. In the parallel construct 5-12, these variables are shared by all threads.

The variables `s2` and `i` are defined within the parallel construct and are therefore, private to each thread.

```
1  void f ( int n, double * x ) {
2    double  s = 0.0;
3
4    #pragma omp parallel
5    {
6      double s2 = 0;
7
8      for ( int i = 0; i < n; ++i )
9        s2 += x[i];
10
11     s = s+s2;
12   }
13 }
```



Shared vs. Private Data

Any change to a shared variable will affect all threads of the thread team, and hence, defines a critical section.

```
1  void f ( int n, double * x ) {
2    double  s = 0.0;
3
4    #pragma omp parallel
5    {
6      double s2 = 0;
7
8      for ( int i = 0; i < n; ++i )
9        s2 += x[i];
10
11     s = s+s2;
12   }
13 }
```

In the example, at line 11 the shared variable `s` is updated, yielding a race condition.

In the `for` loop at lines 8 and 9, the shared variables `n` and `x` are accessed read-only. As long as the content of both variables is not changed by a thread, such accesses are uncritical.

Shared and Default Clause

With the clause

```
#pragma omp parallel shared(x1,x2,...)
```

a list of variables is defined, which exist in the scope of the code block *before* entering the parallel construct and which should be shared within the parallel construct.

Since by default, all such variables are shared, the statement would normally have no effect.

This standard behaviour can be changed by the clause

```
#pragma omp parallel default(shared|none)
```

with

`default(shared)`: representing the default behaviour, i.e. variables are shared,

`default(none)`: disable automatic variable sharing.

In the latter case, the status of any non-private variable referenced within the parallel construct has to be defined explicitly, e.g. by the `shared` clause.

Shared and Default Clause

The combination of both clauses allows to limit the risk of unwanted side effects, e.g. if shared variables are erroneously accessed:

```
void f ( int n, double * x ) {
  double  x1[10], x2[100];
  double  s = 0.0;

  #pragma omp parallel default(none) shared(n,x,s)
  {
    double s2 = 0;
    for ( int i = 0; i < n; ++i )
      s2 += x1[i];   // error reported by compiler
    s = s+s2;
  }
}
```

Without `default` and `shared`, the compiler would not report any error.

Private Clause

Outside variables may also be declared *private* for parallel constructs:

```
#pragma omp parallel private(var1,var2,...)
```

For each variable listed in the `private` clause, a new thread-private variable is created but *not initialised*. In particular, the thread-private copy will not have the same value as the variable in the surrounding scope.

This clause is therefore useful for automatic creation of thread-private variables, which are mainly used for local work and not for data sharing.

Some limitations apply to the `private` clause:

- The private variable must not appear inside a compound

  ```
  struct { double  real, imag; } c;

  #pragma omp parallel private(c.real) // illegal: struct member
  ```

  or array data structure

  ```
  double y[10];

  #pragma omp parallel private(y[5])   // illegal: array member
  ```

Private Clause

- The variable must not be declared `const`.

```
const double  x = 1.0;
#pragma omp parallel private(x)   // illegal: const variable
```

The only exception are classes with `mutable` members.

```
const struct {
  double         x;
  mutable const  n;
} s;

#pragma omp parallel private(s)   // legal: s.n is mutable
```

- If the variable is a class, an accessible default constructor must exist.

```
struct A {
  double  x, y;

  A ( double x, double y );

private:
  A ();
};

A  a( 1, 2 );

#pragma omp parallel private(a)   // illegal: A() is not accessible
```

- Reference types are not allowed.

```
void f ( double & y ) {
  #pragma omp parallel private(y)   // illegal: reference type
  ...
}
```

Private Clause

Construct vs. Region

The `private` clause applies to an OpenMP construct. It is not defined how the access to a variable is handled in the remaining parallel region.

```
int  n;

void f ( int i ) {
  n = i;                          // undefined, may refer to private or global variable
}
int main () {
  #pragma omp parallel private(n)
  {
    n = 1;                        // refers to private variable
    f( 2 );
  }
}
```

Private Clause

Construct vs. Region

The `private` clause applies to an OpenMP construct. It is not defined how the access to a variable is handled in the remaining parallel region.

```
int  n;

void f ( int i ) {
  n = i;                          // undefined, may refer to private or global variable
}
int main () {
  #pragma omp parallel private(n)
  {
    n = 1;                        // refers to private variable
    f( 2 );
  }
}
```

Access to the Original Variable

The original variable of a private copy may still be accessed in the parallel region, e.g. via pointers:

```
void f ( int i ) {
  int *  i_ptr = & i;

  #pragma omp parallel private(i)
  {
    i      = 1;                   // refers to the private variable
    *i_ptr = 2;                   // refers to the original variable
  }
}
```

Firstprivate Clause

To initialise private variables with the value the variable has outside the parallel construct the clause `firstprivate` is provided:

```
#pragma omp parallel firstprivate(var1,var2,...)
```

The initialisation of each variable is performed before the parallel construct is executed.

Variables of elementary data types are initialised with standard copy assignments, whereas arrays are initialised element wise. For classes the copy constructor is used.

The same restrictions as for the `private` clause apply to the `firstprivate` clause.

Thread Overhead

Spawning and finishing threads creates overhead, e.g. the operating system has to copy internal data for managing threads etc..

As an example, the following program which uses C++11 threads will take 33 seconds to run on a 2-CPU Intel Xeon E5-2640 (Hexa-Core):

```
for ( size_t i = 0; i < 100000; ++i ) {
    for ( int p = 0; p < 12; ++p ) threads[p] = std::thread( f );
    for ( int p = 0; p < 12; ++p ) threads[p].join();
}
```

In contrast to this, the equivalent OpenMP version:

```
for ( size_t  i = 0; i < 100000; ++i ) {
    #pragma omp parallel
    {
        f();
    }
}
```

only takes *0.6* seconds.

The reason for this difference is, that OpenMP will *not* spawn new threads for each parallel directive, but instead keep a *pool* of threads running all the time and only assign the corresponding tasks to these threads. Nevertheless, the actual task should have some minimal runtime for efficient OpenMP parallelisation.

Parallelising Loops

Parallelising Loops

Initially, OpenMP was focused on parallelising loops.

The corresponding directive is

```
#pragma omp for [clause1 [[,] clause2, ...]]
```

and applies to the immediately following `for` loop:

```
#pragma omp for
for ( size_t  i = 0; i < n; ++i ) {
  ...
}
```

Furthermore, the `for` directive must be inside the region of a `parallel` construct:

```
#pragma omp parallel
{
  #pragma omp for
  for ( size_t  i = 0; i < n; ++i ) {
    ...
  }
}
```

The loop will then be split automatically into individual chunks, which are mapped to the threads of the current team, e.g. each worker thread will handle n/p indices of the `for` loop.

Parallelising Loops

Remark

Common mistakes for the `for` directive are

- *Forgotton `parallel` clause, which results in sequential execution:*

```
// no "parallel" directive
{
  #pragma omp for
  for ( size_t  i = 0; i < n; ++i ) {
    ...
  }
}
```

- *Forgotton `for` keyword, in which case all threads will execute the whole loop:*

```
#pragma omp parallel
{
  #pragma omp                          // no "for" directive
  for ( size_t  i = 0; i < n; ++i ) {
    ...
  }
}
```

Unfortunately, the compiler will not warn about such errors.

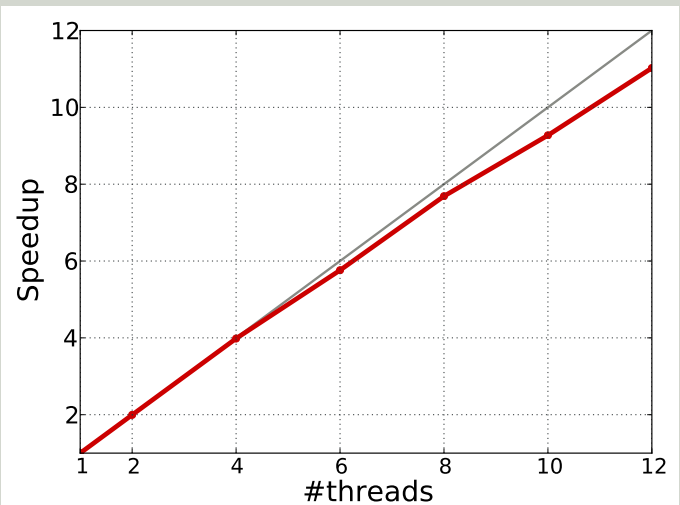
Parallelising Loops

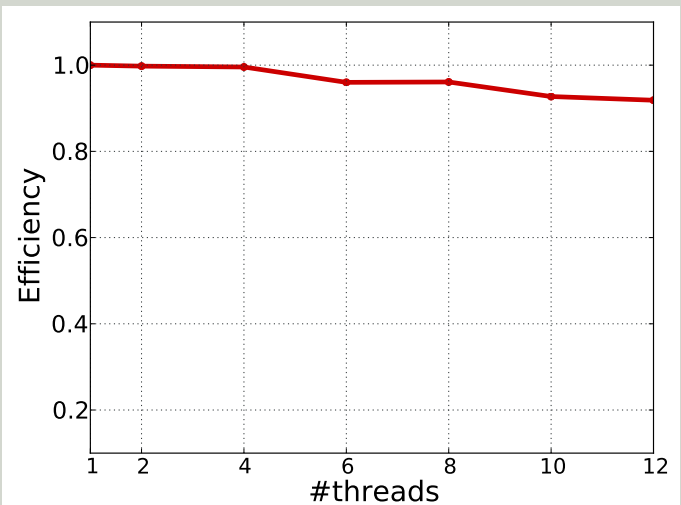
Example: Matrix Multiplication

Let $A, B, C \in \mathbb{R}^{n \times n}$ and compute the product $C = A \cdot B$:

```
void mat_mul ( const size_t n, const Matrix & A, const Matrix & B, Matrix & C ) {
  #pragma omp parallel
  #pragma omp for
  for ( size_t  i = 0; i < n; ++i ) {
    for ( size_t  j = 0; j < n; ++j ) {
      double  c_ij = 0;

      for ( size_t  k = 0; k < n; ++k )
        c_ij += A(i,k) * B(k,j);

      C(i,j) = c_ij;
} } }
```

For $n = 1000$ on a 2-CPU Intel Xeon E5-2640 this yields:





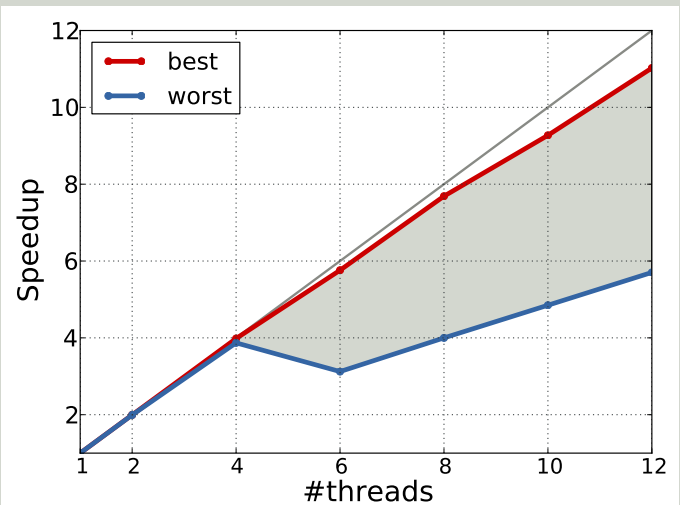
Parallelising Loops

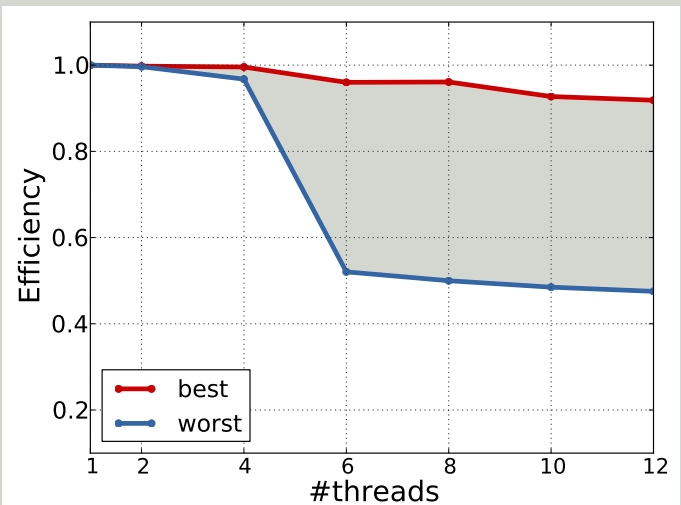
Example: Matrix Multiplication

Let $A, B, C \in \mathbb{R}^{n \times n}$ and compute the product $C = A \cdot B$:

```
void mat_mul ( const size_t n, const Matrix & A, const Matrix & B, Matrix & C ) {
  #pragma omp parallel
  #pragma omp for
  for ( size_t  i = 0; i < n; ++i ) {
    for ( size_t  j = 0; j < n; ++j ) {
      double  c_ij = 0;

      for ( size_t  k = 0; k < n; ++k )
        c_ij += A(i,k) * B(k,j);

      C(i,j) = c_ij;
} } }
```

For $n = 1000$ on a 2-CPU Intel Xeon E5-2640 this yields:





Loop Restrictions

The data type of the index variable must be

- a signed or unsigned integer type, e.g. `int` or `size_t`

```
#pragma omp for
for ( size_t i = 0; i < n; ++i ) {
  ...
}
```

- a random access iterator, e.g. for `std::vector` (not for `std::list`!)

```
#pragma omp for
for ( std::vector< double >::iterator iter = x.begin(); iter != x.end(); ++iter ) {
  ...
}
```

- a pointer type

```
#pragma omp for
for ( double * p = & x[0]; p < & x[n]; ++p ) {
  ...
}
```

Loop Restrictions

Only constant index changes are supported, e.g. `i++`, `-i` or `i += 5`.

```
#pragma omp for
for ( size_t i = 0; i < n; i = 2*i ) {          // error
  ...
}
```

Changing the index variable inside the loop body is not allowed.

```
#pragma omp for
for ( size_t i = 0; i < n; ++i ) {
  ...
  i += 2;                                       // error
  ...
}
```

The loop test must be simple, e.g. `i < n` or `i >= n`.

```
#pragma omp for
for ( size_t i = 0; i < n && i > n/2; ++i ) {    // error
  ...
}
```

Also, C++11 range-based loops are *not* supported:

```
#pragma omp for
for ( auto & v : vec ) {                        // error
  ...
}
```

Loop Restrictions

Due to the implicit barrier at the end of the construct, *all* team threads must encounter the `for` directive during their execution or none at all:

```
#pragma omp parallel
{
  if ( do_loop() ) {  // conditional loop execution
    #pragma omp for
    for ( ... ) {
      ...
    }
  }
}
```

Here, `do_loop` has to return the same values for *all* threads. Otherwise, the program may block because of the barrier.

Even if the barrier is removed using `nowait`, the resulting program is non-conforming and may not work correctly.

Loop Scheduling

When encountering a `for` loop:

```
#pragma omp for
for ( size_t i = 0; i < n; ++i ) {
  ...
}
```

the range of the loop index is split into *chunks* of (almost) equal size, each forming a task. These tasks are then assigned to the threads of the team.

The typical chunk size is n/p, e.g. the loop range is split as:

Loop Scheduling

When encountering a `for` loop:

```
#pragma omp for
for ( size_t i = 0; i < n; ++i ) {
  ...
}
```

the range of the loop index is split into *chunks* of (almost) equal size, each forming a task. These tasks are then assigned to the threads of the team.

The typical chunk size is n/p, e.g. the loop range is split as:

This *static* scheduling can be changed by the `schedule` clause:

```
#pragma omp for schedule(static|dynamic|guided|auto|runtime)
```

For `static`, `dynamic` and `guided` an optional chunk size may be specified:

```
#pragma omp for schedule(static|dynamic|guided [, chunksize])
```

Loop Scheduling

The definition of the different scheduling algorithms is:

static: Divide the loop range into chunks of size `chunksize` and assign the resulting task to the team threads in a round-robin fashion:

If no `chunksize` is specified, the default size is $\approx n/p$.

dynamic: Divide the loop range into chunks of size `chunksize`. Tasks are assigned as each thread requests them one after the other:

If no `chunksize` is specified, the default size is 1.

guided: Start by building chunks of a size $\geq$ `chunksize` proportional to the unassigned indices. Tasks are assigned as each thread requests them.

If no `chunksize` is specified, the default size is 1.

Loop Scheduling

Further scheduling types are:

auto: Let the compiler or runtime system decide about best scheduling.

runtime: The runtime system decides about best scheduling.

In case of runtime scheduling, the user may change the default behaviour using the `OMP_SCHEDULE` environment variable, which should contain one of the above scheduling types:

```
> export OMP_SCHEDULE="static,16"   # bash
```

or

```
> setenv OMP_SCHEDULE "dynamic"     # tcsh
```

Remark

The environment variable `OMP_SCHEDULE` applies to all `for` directives in the program using `auto` or `runtime` scheduling, e.g. no control of single loops is possible.

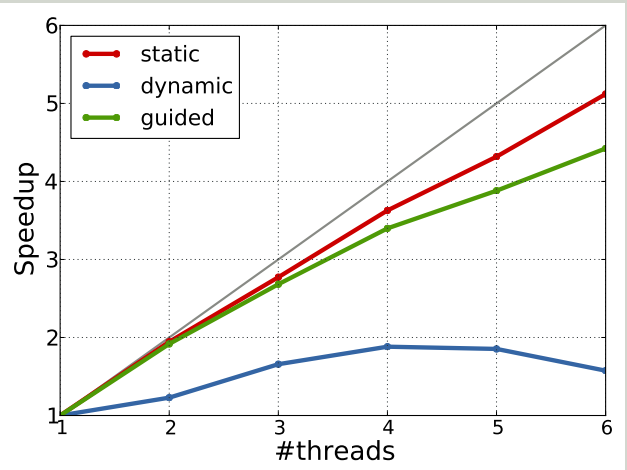
Loop Scheduling

If the work per loop index is equal, static scheduling yields an optimal load balance between all team threads.

Example: Matrix-Vector Multiplication

Let $A \in \mathbb{R}^{n \times n}$ and $x, y \in \mathbb{R}^n$. Compute $y = A \cdot x$:

```
void mat_vec ( const size_t n, const Matrix & A,
               const Vector & x, Vector & y ) {
  #pragma omp parallel
  #pragma omp for schedule(● ● ●)
  for ( size_t  i = 0; i < n; ++i ) {
    double  y_i = 0;
    for ( size_t  k = 0; k < n; ++k )
      y_i += A(i,k) * x(k);
    y(i) = y_i;
  }
}
```



The performance of dynamic and guided scheduling is due to the additional overhead induced by assigning tasks to threads during runtime.

If the work per task is increased, this overhead can often be neglected, e.g. increasing the above dimension n by a factor of 4 yields the same speedup for all scheduling schemes.

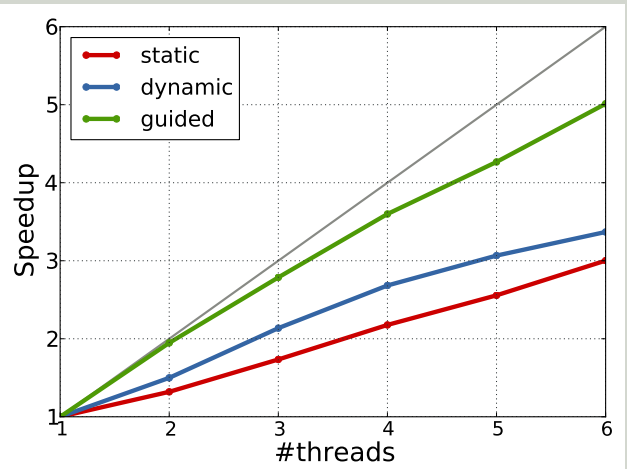
Loop Scheduling

The situation changes, if the work is unevenly distributed with respect to the loop index.

Example: Triangular Matrix-Vector Multiplication

Let $A \in \mathbb{R}^{n \times n}$ be a lower triangular matrix and $x, y \in \mathbb{R}^n$. Compute $y = A \cdot x$:

```
void mat_vec ( const size_t n, const Matrix & A,
               const Vector & x, Vector & y ) {
  #pragma omp parallel
  #pragma omp for schedule(● ● ●)
  for ( size_t  i = 0; i < n; ++i ) {
    double  y_i = 0;
    for ( size_t  k = 0; k <= i; ++k )
      y_i += A(i,k) * x(k);
    y(i) = y_i;
  }
}
```



A static mapping leads to an unbalanced load between all threads since the load increases with the loop index.

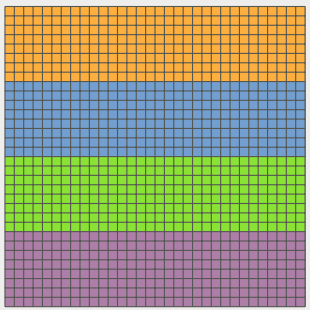
In contrast to this, with dynamic or guided scheduling more tasks are constructed such that idling threads may request more work, yielding a more even load balance.

Loop Scheduling

Remark

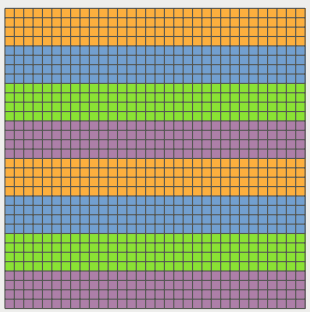
For the matrix-vector multiplication, the default static mapping corresponds to a row-wise block decomposition:

```
#pragma omp for
for ( size_t  i = 0; i < n; ++i ) {
  for ( size_t  k = 0; k < n; ++k )
    y(i) += A(i,k) * x(k);
}
```



By specifying a chunk size, this can be changed to a cyclic row-wise block decomposition, which would also result in a better load balance in the triangular matrix case:

```
#pragma omp for schedule(static,4)
for ( size_t  i = 0; i < n; ++i ) {
  for ( size_t  k = 0; k < n; ++k )
    y(i) += A(i,k) * x(k);
}
```



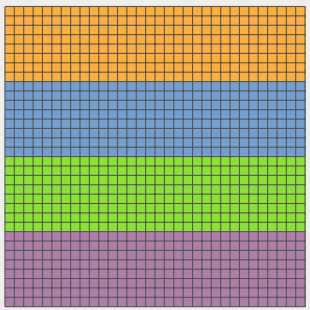
Dynamic or guided scheduling corresponds to a randomised block decomposition and is an example of Online Scheduling.

Loop Scheduling

Remark

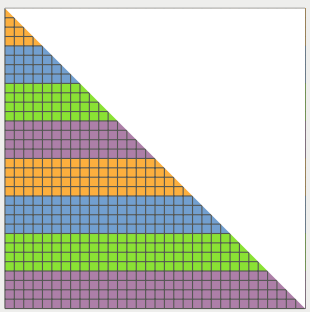
For the matrix-vector multiplication, the default static mapping corresponds to a row-wise block decomposition:

```
#pragma omp for
for ( size_t  i = 0; i < n; ++i ) {
  for ( size_t  k = 0; k < n; ++k )
    y(i) += A(i,k) * x(k);
}
```



By specifying a chunk size, this can be changed to a cyclic row-wise block decomposition, which would also result in a better load balance in the triangular matrix case:

```
#pragma omp for schedule(static,4)
for ( size_t  i = 0; i < n; ++i ) {
  for ( size_t  k = 0; k < n; ++k )
    y(i) += A(i,k) * x(k);
}
```



Dynamic or guided scheduling corresponds to a randomised block decomposition and is an example of Online Scheduling.

False Sharing

If the chunk size in the `schedule` clause is too small, *false sharing* may happen:

```
std::vector< double >  x( n );

#pragma omp parallel
#pragma omp for schedule(static,1)
for ( size_t  i = 0; i < n; ++i )
  x[i] = f(i);
```

Here, each index position is cyclically scheduled to a different thread:

cache line

Hence the update of the corresponding array entries will affect the cache lines of several other threads. Depending on the runtime for each call of `f`, this may severely limit the parallel speedup:

	chunk size			
	default	1	16	128
`static`	10.09	7.89	10.01	10.33
`dynamic`	0.51	0.51	6.82	10.17
`guided`	10.23	10.23	10.28	10.36

Speedup on 2-CPU Xeon E5-2640

Nested Loops

Since `parallel` directives may be nested, this also applies to nested loops.

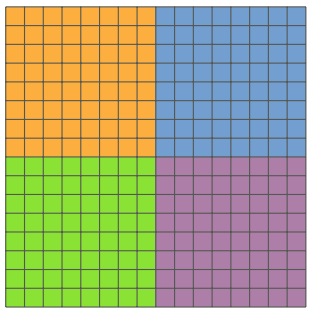
In the matrix multiplication algorithm, the two outer loops apply to independent parts of the destination matrix C. Hence, these should be natural candidates for nested parallel loops:

```
void mat_mul ( const size_t n, const Matrix & A, const Matrix & B, Matrix & C ) {
  #pragma omp parallel num_threads(2)
  #pragma omp for
  for ( size_t  i = 0; i < n; ++i ) {
    #pragma omp parallel num_threads(2)  // nested, parallel loop
    #pragma omp for
    for ( size_t  j = 0; j < n; ++j ) {
      double  c_ij = 0;

      for ( size_t  k = 0; k < n; ++k )
        c_ij += A(i,k) * B(k,j);

      C(i,j) = c_ij;
} } }
```

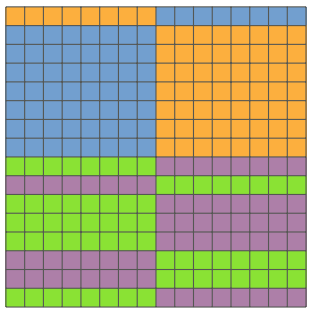
In this example, the rows and the columns will be split in half and 4 threads will handle the corresponding inner dot product computations in parallel.

Nested Loops

Unfortunately, the scheduling of the second loop, e.g. the columns, is applied to each row *individually*:

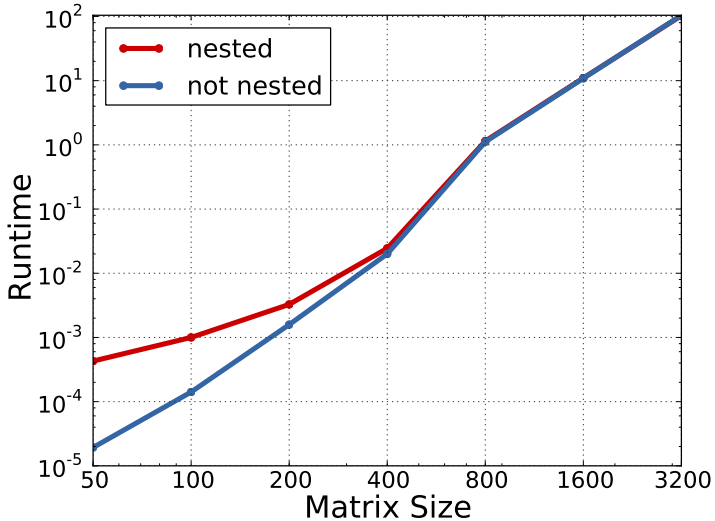


nested and `schedule(static)`



nested and `schedule(dynamic,8)`

This may induce a significant scheduling overhead. For the matrix multiplication example, especially for small dimensions, this overhead severly limits the parallel efficiency.



Nested Loops

Since the previous loops result in independent computations, both loops form a larger iteration space over rows and columns together.

OpenMP supports such loop *collapsing* via the `collapse` clause:

```
#pragma omp for collapse(n)
```

where n specifies the number of nested loops to collapse.

The loops must be perfectly nested and not depend on each other, e.g.

```
#pragma omp for collapse(2)
for ( size_t i = 0; i < n; ++i ) {
  for ( size_t j = 0; j < n; ++j ) {
    ...
  }
  for ( size_t j = 0; j < n; ++j ) {          // not perfectly nested
    ...
  }
}
```

or

```
#pragma omp for collapse(2)
for ( size_t i = 0; i < n; ++i ) {
  for ( size_t j = 0; j < i; ++j ) {          // loop dependence
    ...
} }
```

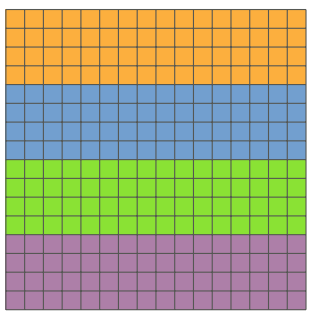
are not allowed.

Nested Loops

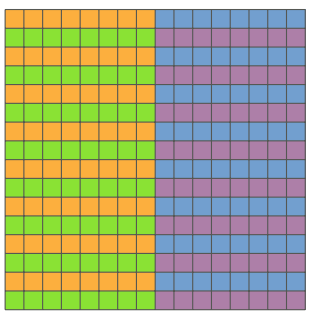
For the matrix multiplication, the outer loops are perfectly nested and independent, hence, they may be collapsed as an alternative to nested parallelism:

```
void mat_mul ( const size_t n, const Matrix & A, const Matrix & B, Matrix & C ) {
  #pragma omp parallel num_threads(4)
  #pragma omp for collapse(2)
  for ( size_t  i = 0; i < n; ++i ) {
    for ( size_t  j = 0; j < n; ++j ) {
      double  c_ij = 0;

      for ( size_t  k = 0; k < n; ++k )
        c_ij += A(i,k) * B(k,j);

      C(i,j) = c_ij;
} } }
```

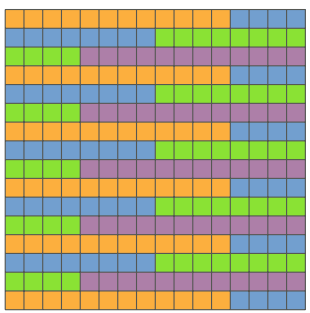
The resulting mapping depends on the specified scheduling and especially the chunk size:



`collapse(2)`



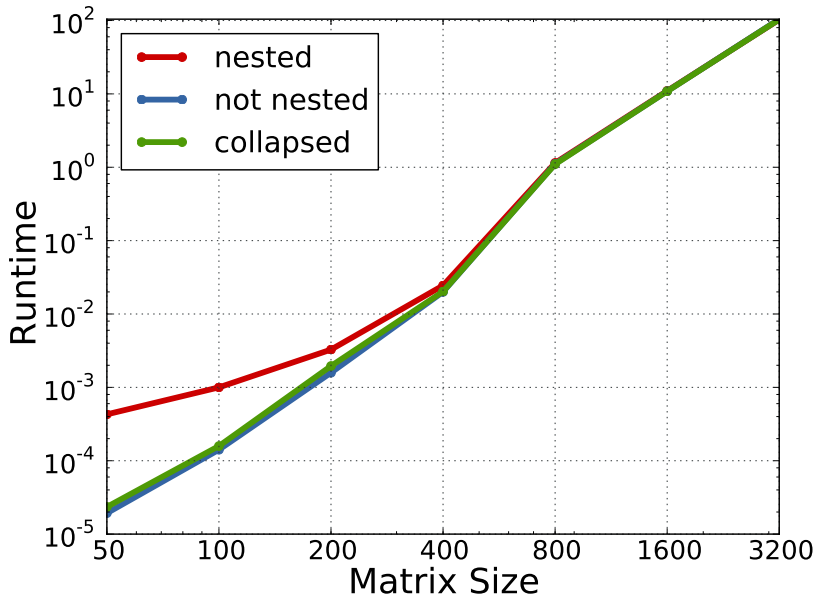
`collapse(2) schedule(static,8)`



`collapse(2) schedule(static,12)`

Nested Loops

As the scheduling is now applied to all rows and columns simultaneously, the additional overhead of the nested parallelism is eliminated:

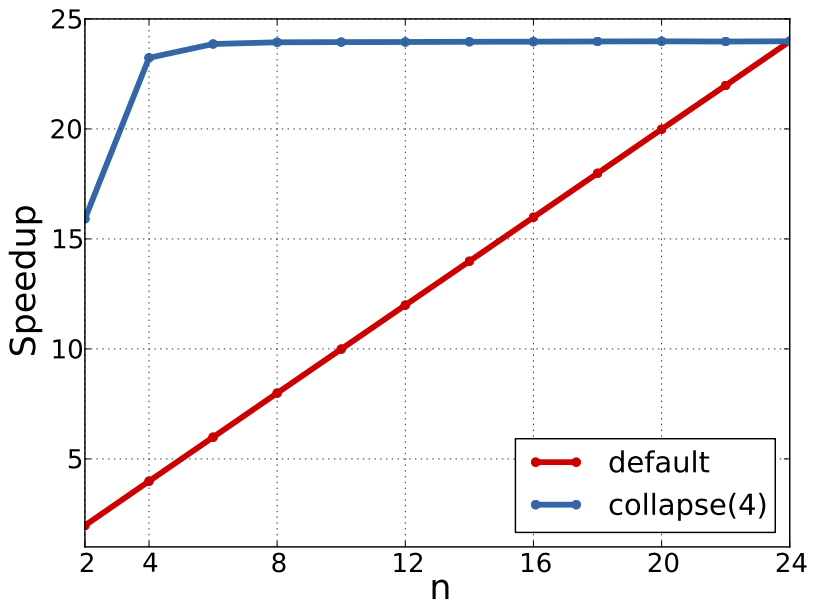


Nested Loops

Especially for nested loops with *small* loop ranges the `collapse` clause useful to enlarge the iteration space and thereby, decrease the granularity and increase the number of tasks:

```
for ( size_t  i = 0; i < n; ++i ) {
  for ( size_t  j = 0; j < n; ++j ) {
    for ( size_t  k = 0; k < n; ++k ) {
      for ( size_t  l = 0; l < n; ++l ) {
        compute( i, j, k, l );
} } } }
```

A single `for` directive for the outer most loop would permit at most n threads. With the `collapse` clause, n^4 threads can be used efficiently:



Speedup on 2-CPU Intel Xeon E5-2640

Reductions

Reduction operations are supported by OpenMP by the `reduction` clause:

```
#pragma omp for reduction(op: var1, ...)
```

For each variable of a reduction clause, a thread-private copy is created and initialised. The values of all private variables are combined at the end of the loop using the specified operator.

Example: Dot Product

Compute $\sum_i x_i \cdot y_i$ for $x, y \in \mathbb{R}^n$:

```
void dot ( const size_t n, double const * x, double const * y ) {
  double  sum = 0.0;

  #pragma omp parallel
  #pragma omp for reduction(+: sum)
  for ( size_t i = 0; i < n; ++i )
    sum += x[i] * y[i];                // "sum" is private

  return sum;
}
```

Since each copy of the reduction variable is thread-private, the variable update does not form a critical section.

Reductions

Supported operators are:

$$+/-/*, \quad \min/\max, \quad \&/|/\hat{}, \quad \&\&/||$$

The order in which the private copies are combined is *not* defined, e.g. different results may be computed in different runs of the program:

```
#pragma omp parallel for reduction(+: sum)
for ( size_t i = 1; i < n; ++i ) {
    sum += std::pow(-1.0,i+1) / double(i);
}
```

may yield

```
> reduction
6.9319718305994504e-01
> reduction
6.9319718305994582e-01
> reduction
6.9319718305994626e-01
```

Reductions

Multiple Reductions

More than one reduction clause is supported for a single loop, e.g.

```
#pragma omp parallel for reduction(+: sum) reduction(*: prod)
for ( size_t i = 0; i < n; ++i ) {
  sum  = sum  + ... ;
  prod = prod * ... ;
}
```

Reductions

Multiple Reductions

More than one reduction clause is supported for a single loop, e.g.

```
#pragma omp parallel for reduction(+: sum) reduction(*: prod)
for ( size_t i = 0; i < n; ++i ) {
  sum  = sum  + ... ;
  prod = prod * ... ;
}
```

Restrictions

Some restrictions apply to the `reduction` clause:

- A reduction variable must be shared in the surrounding `parallel` region.
- A reduction variable must not appear in multiple `reduction` clauses.
- A reduction variable must not be declared `const`.
- Compound types are not supported.

Reductions

Remark

The `reduction` clause is also available for the `parallel` directive without a loop, hence the following implementation is equivalent to the previous parallel dot product:

```
void dot ( const size_t n, double const * x, double const * y ) {
  double  sum = 0.0;

  #pragma omp parallel reduction(+: sum)
  #pragma omp for
  for ( size_t i = 0; i < n; ++i )
    sum += x[i] * y[i];

  return sum;
}
```

The difference is, that instead of combining the values at the end of the loop, the reduction is performed at the end of the parallel region.

Loop Synchronisation

By default, threads will synchronise with the end of the loop, e.g. all threads will wait for all others to finish their computations.

Using the `nowait` clause, this implicit barrier can be eliminated:

```
#pragma omp for nowait
```

A typical application of this clause is several loops with an uneven load per task:

```
#pragma omp parallel
{
  #pragma omp for nowait
  for ( size_t i = 0; i < n; ++i ) {
    ...
  }

  #pragma omp for nowait
  for ( size_t j = 0; j < m; ++j ) {
    ...
  }

  #pragma omp for nowait
  for ( size_t l = 0; l < k; ++l ) {
    ...
  }
}
```

Here, the team threads may proceed to the next loop without waiting for other threads.

Loop Synchronisation

Reduction without Barrier

If `nowait` is used for a loop with a `reduction` clause, a race condition will occur if the reduction variable is accessed outside the loop:

```
#pragma omp parallel
{
  double  sum = 0;

  #pragma omp for nowait reduction(+: sum)
  for ( size_t i = 0; i < n; ++i ) {
    ...
  }

  f( sum );       // race condition
}
```

Since not all updates to `sum` may have been applied, the corresponding value is undefined.

Loop Serialisation

Using the `ordered` clause together with a corresponding block, some part of the loop body may be executed sequentially in the order defined by the loop index:

```
#pragma omp for ordered
for ( ... )
{
  ...                              // parallel region
  #pragma omp ordered
  {
    ...                            // ordered, sequential region
  }
  ...                              // parallel region
}
```

Since the code within the `ordered` region is executed in loop order, threads will *wait* at the entry to the `ordered` region until the `ordered` region of all previous iterations have been completed.

Code before the `ordered` region is not affected by this and may be executed as soon as the corresponding task is mapped to a thread.

Due to this *serialisation*, `ordered` blocks should contain only fast computations and be used only at the end of a loop.

Loop Serialisation

Using the `ordered` clause, *deterministic* behaviour can be enforced:

```
#pragma omp for ordered reduction(+: global_val)
for ( size_t i = 0; i < n; ++i )
{
  const double  local_val = compute_value();

  #pragma omp ordered
  global_val += local_val;   // sum up in sequential order
  }
}
```

It is also usefull for printing intermediate values, e.g. for debugging:

```
#pragma omp for ordered
for ( size_t i = 0; i < n; ++i )
{
  value = compute_value();

  #pragma omp ordered
  std::cout << value << std::endl;
}
```

Otherwise, access to the I/O channel induces a race condition which will lead to scrambled output.

Loop Serialisation

Restriction

During the iteration of a loop, each thread must executed at most *one* `ordered` block, i.e. the following code is not allowed:

```
#pragma omp for ordered
for ( ... )
{
  #pragma omp ordered
  { ... }
  #pragma omp ordered    // error
  { ... }
}
```

However,

```
  #pragma omp for ordered
  for ( ... )
  {
    if ( condition ) {
      #pragma omp ordered
      { ... }
    } else {
      #pragma omp ordered
      { ... }
    }
  }
```

is legal code, since only one `ordered` block is executed.

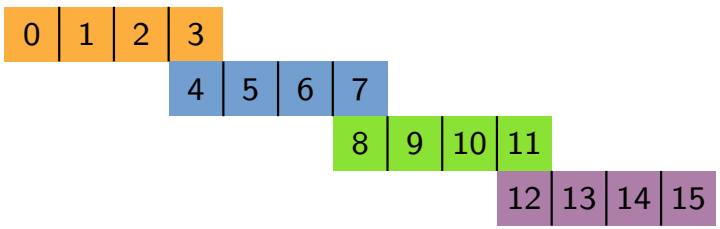
Loop Serialisation

Loop Scheduling and `ordered`

Using the default `static` scheduling together with `ordered` blocks will often *prevent* parallel execution, since neighboured indices are executed by the same thread *in order*.

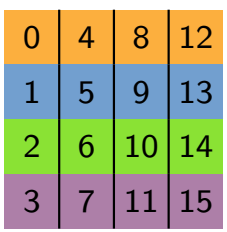
For example, the following loop is executed almost sequentially while using 4 threads:

```
#pragma omp for schedule(static) ordered
for ( size_t i = 0; i < 16; ++i )
{
  double  x = f();

  #pragma omp ordered
  std::cout << x << std::endl;
}
```



With `dynamic` scheduling (or "`static,1`") all threads run in parallel:

```
#pragma omp for schedule(dynamic) ordered
for ( size_t i = 0; i < 16; ++i )
{
  double  x = f();

  #pragma omp ordered
  std::cout << x << std::endl;
}
```



Loop Serialisation

Remark

A common programming mistake is to forget an `ordered` block although the `ordered` clause was specified:

```
int main () {
  #pragma omp parallel
  #pragma omp for ordered
  for ( ... )
  {
    ...             // no "ordered" block
  }
}
```

Combined Parallel Loops

Since parallel construct often contain a single loop, both directives may be joined into a single OpenMP directive:

```
#pragma omp parallel for
```

which replaces the separate statements:

```
#pragma omp parallel
#pragma omp for
```

All clauses of both the `parallel` and the `for` directive may be used as classes for the combined directive.

The only exception is the `nowait` clause, which is only supported for separate `for` directives.

Example: N-Body Problem

The implementation of the N-body program using the SOA approach mainly consists of the timestep, updating the velocity and position of all particles:

```
1  void timestep ( const double  dt ) {
2    for ( size_t  i = 0; i < n_particles; ++i ) {
3      const vector3_t  pos_i( pos_x[i], pos_y[i], pos_z[i] );
4      vector3_t        force;
5
6      #pragma vector always                          // for auto-vectorisation
7      for ( size_t  j = 0; j < n_particles; ++j ) {
8        if ( i != j ) {
9          const vector3_t  pos_j( pos_x[j], pos_y[j], pos_z[j] );
10         const vector3_t  dist( pos_j - pos_i );
11
12         force.add( mass[j] / dist.norm_cubed(), dist );
13       }
14     }
15
16     vel_x[i] += dt * force.x;
17     vel_y[i] += dt * force.y;
18     vel_z[i] += dt * force.z;
19   }
20
21   #pragma ivdep                                    // for auto-vectorisation
22   for ( size_t  i = 0; i < n_particles; ++i ) {
23     pos_x[i] += dt * vel_x[i];
24     pos_y[i] += dt * vel_y[i];
25     pos_z[i] += dt * vel_z[i];
26   }
27 }
```

The main work is done in the two loops at line 2 and 7, respectively.

Example: N-Body Problem

The outer loop and the update loop are candidates for OpenMP loop parallelisation. The inner loop at line 7 performs a reduction w.r.t. the compound variable `force`, for which OpenMP reduction is not directly possible.

```
void timestep ( const double  dt ) {
  #pragma omp parallel
  {
    #pragma omp for
    for ( size_t  i = 0; i < n_particles; ++i ) {
      const vector3_t  pos_i( pos_x[i], pos_y[i], pos_z[i] );
      vector3_t        force;

      #pragma vector always
      for ( size_t  j = 0; j < n_particles; ++j ) {
        if ( i != j ) {
          const vector3_t  pos_j( pos_x[j], pos_y[j], pos_z[j] );
          const vector3_t  dist( pos_j — pos_i );

          force.add( mass[j] / dist.norm_cubed(), dist );
        }
      }

      vel_x[i] += dt * force.x;
      vel_y[i] += dt * force.y;
      vel_z[i] += dt * force.z;
    }

    #pragma omp for
    #pragma ivdep
    for ( size_t  i = 0; i < n_particles; ++i ) {
      pos_x[i] += dt * vel_x[i];
      pos_y[i] += dt * vel_y[i];
      pos_z[i] += dt * vel_z[i];
    }
  }
```

Example: N-Body Problem

The parallel speedup of this approach is

	SSE2 (Xeon X5650) (24 threads)	AVX (Xeon E5-2640) (24 threads)	MIC (XeonPhi 5110P) (240 threads)
Vectorisation:	1.97x	2.05x	6.84x
OpenMP:	11.86x	11.11x	102.55x

Remark

All systems use hyperthreading, i.e. support usage of currently unused processor units (load, store, arithmetic) by other threads.

Instead of static scheduling, dynamic or guided scheduling can be used. The resulting speedup is:

	SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
`dynamic` vs. `static`:	1.01x	1.00x	1.16x
`guided` vs. `static`:	1.01x	1.00x	1.11x

Example: N-Body Problem

Replacing the *force* variable by an elementary type, the inner loop permits OpenMP reduction:

```
for ( size_t  i = 0; i < n_particles; ++i ) {
  const vector3_t  pos_i( pos_x[i], pos_y[i], pos_z[i] );
  double force_x = 0.0;
  double force_y = 0.0;
  double force_z = 0.0;

  #pragma omp parallel for reduction(+: force_x, force_y, force_z)
  #pragma vector always
  for ( size_t  j = 0; j < n_particles; ++j ) {
    if ( i != j ) {
      const vector3_t  pos_j( pos_x[j], pos_y[j], pos_z[j] );
      const vector3_t  dist( pos_j - pos_i );
      double           d = mass[j] / dist.norm_cubed();

      force_x += d * dist.x; force_y += d * dist.y; force_z += d * dist.z;
    }
  }

  vel_x[i] += dt * force_x;
  vel_y[i] += dt * force_y;
  vel_z[i] += dt * force_z;
}
```

But this leads to *no* further speedup.

Example: N-Body Problem

The total speedup gained compared to the initial version or the program is:

SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
23.6x	22.7x	815x

The corresponding absolute runtimes are

	SSE2 (Xeon X5650)	AVX (Xeon E5-2640)	MIC (XeonPhi 5110P)
original program	50.67s	36.69s	315.70s
final program	2.15s	1.61s	0.39s

Remark

All speedup numbers are w.r.t. to the exact timings, not rounded.

Sections

Sections

If the computation consists of independent but *non-iterative* parts, the `sections` directive can be used to distributed the work to several threads:

```
#pragma omp sections [clause1 [[,] clause2, ...]]
{
  #pragma omp section              // first section region
  {
    ...
  }
  #pragma omp section              // second section region
  {
    ...
  }
  ...                              // more sections
}
```

Each `section` block inside the construct of the `sections` directive will be executed *once* by *one* team thread.

By default, all team threads will synchronise at the end of the `sections` block.

Sections

Remark

The `sections` directive must be placed inside a `parallel` region, otherwise it is executed sequentially.

Sections

Remark

The `sections` directive must be placed inside a `parallel` region, otherwise it is executed sequentially.

The construct of a `sections` directive may contain *only* `section` blocks, e.g. the following will lead to an error:

```
#pragma omp sections
{
  #pragma omp section
  { ... }
  #pragma omp section
  { ... }

  f();                    // error
}
```

Sections

Remark

The `sections` directive must be placed inside a `parallel` region, otherwise it is executed sequentially.

The construct of a `sections` directive may contain *only* `section` blocks, e.g. the following will lead to an error:

```
#pragma omp sections
{
  #pragma omp section
  { ... }
  #pragma omp section
  { ... }

  f();                    // error
}
```

The optional clauses of the `sections` directive include:

- `private`: create unintialised thread-private variables,
- `firstprivate`: create copy-initialised thread-private variables,
- `lastprivate`:
- `reduction`: combine thread-private copies at end of `sections` construct,
- `nowait`: remove implicit barrier at end of `sections` construct.

For all clauses, the standard restrictions apply.

Combined Parallel Sections

The `sections` directive can also be combined with the `parallel` directive:

```
#pragma omp parallel sections
```

to replaces the separate statements:

```
#pragma omp parallel
#pragma omp sections
```

All clauses of both the `parallel` and the `sections` directive, except `nowait`, may be used as classes for the combined directive.

Single Execution

Single Execution

With the `single` directive

```
#pragma omp single
```

the corresponding block is executed by only *one* thread of the current team.

All other threads will *wait* at the end of the `single` block until the executing thread finishes.

In the example:

```
#pragma omp parallel
{
  f();                  // executed by all threads

  #pragma omp single
  {
    std::cout << "finished f()" << std::endl;
  }

  g();                  // executed by all threads _after_ single block
}
```

`f()` will be executed in parallel, followed by a single thread printing the message. Only then will `g()` be executed again in parallel.

Single Execution

Directly inside `for` or `sections` blocks, the `single` construct is *not* allowed, e.g.

```
#pragma omp parallel for
for ( size_t i = 0; i < n; ++i )
{
  ...

  #pragma omp single       // error
  { ... }
}
```

Instead, the `single` block may be in the region of a `for` or `sections` directive but associated to a new thread team:

```
#pragma omp parallel sections
{
  #pragma omp section
  {
    #pragma omp parallel   // new thread team
    {
      ...
      #pragma omp single   // no error
      { ... }
    }
  }
  #pragma omp section
  {
    #pragma omp parallel   // new thread team
    {
      ...
      #pragma omp single   // no error
      { ... }
    }
  }
}
```

Single Execution

The optional clauses of the `single` directive include:

`nowait`: remove implicit barrier at end of `single` construct

```
#pragma omp parallel
{
  #pragma omp single nowait
  {
    ...
  }

  g();          // may be executed _before_ single block in other threads
}
```

`private`: create unintialised thread-private variables,

`firstprivate`: create copy-initialised thread-private variables,

For all clauses, the standard restrictions apply.

Copyprivate Clause

A special clause is available for the `single` directive:

```
#pragma omp single copyprivate(var1,var2,...)
```

The value of any variable in the list of the `copyprivate` clause will be copied to all other threads at the end of the `single` clause:

```
#pragma omp parallel
{
  double x = 2;

  #pragma omp single copyprivate(x)
  {
    x = 1;
  }                    // copy value of x to all other threads

                       // x == 1 in all threads
}
```

All variables in the `copyprivate` clause have to be *private* in the surrounding parallel region.

Remark

The copy operation is performed using standard copy assignment for elemental types, entry-wise copy for arrays or using the copy operator for classes.

Copyprivate Clause

A similar effect has a shared variable. But this may lead to a race condition *before* the `single` block since the barrier is only at it's end:

```
double x = 2;

#pragma omp parallel
{
  ...
  f(x);                // x == 1 or x == 2
  ...
  #pragma omp single
  {
    x = 1;
  }
                       // x == 1 in all threads
}
```

Furthermore, using `nowait` together with `single`, the race condition may also appear *after* the `single` block:

```
double x = 2;

#pragma omp parallel
{
  #pragma omp single nowait
  {
    x = 1;
  }
                       // x == 1 or x == 2
}
```

Copyprivate Clause

The `copyprivate` clause provides a safe alternative since the update of all copies takes place at the barrier:

```
double x = 2;

#pragma omp parallel firstprivate(x)
{
  f(x);                // x == 2

  #pragma omp single copyprivate(x)
  {
    x = 1;
  }                    // update of thread-local copy when thread reaches barrier
                       // x == 1 in all threads
}
```

Copyprivate Clause

The `copyprivate` clause provides a safe alternative since the update of all copies takes place at the barrier:

```
double x = 2;

#pragma omp parallel firstprivate(x)
{
  f(x);                // x == 2

  #pragma omp single copyprivate(x)
  {
    x = 1;
  }                    // update of thread-local copy when thread reaches barrier
                       // x == 1 in all threads
}
```

Furthermore, the barrier at the end of the construct can *not* be removed:

```
#pragma omp parallel
{
  double x = 2;

  #pragma omp single copyprivate(x) nowait  // "nowait" without effect
  {
    x = 1;
  }
                                            // x == 1 in all threads
}
```

Copyprivate Clause

Since the `single` directive has an implicit barrier, *all* team threads must encounter the directive during their execution or none at all:

```
#pragma omp parallel
{
  if ( do_loop() ) {  // conditional execution
    #pragma omp single
    for ( ... ) {
      ...
    }
  }
}
```

Removing the barrier by the `nowait` clause will usually work, but the resulting program is non-conforming.

Master Directive

A `single` block may be executed by `any` thread of the team.

Using the `master` directive:

```
#pragma omp master
```

the corresponding construct will be executed `only` by the master thread of the team, i.e. the thread with thread id 0.

```
#pragma omp parallel
{
  ...                 // executed by all threads

  #pragma omp master
  {
    ...               // executed _only_ by master thread
  }

  ...                 // executed by all threads
}
```

In contrast to the `single` directive, no implicit barrier exists at the end of the `master` block. Also no clauses are available for the `master` directive.

Thread Synchronisation

Thread Synchronisation

Beside thread creation and scheduling, thread synchronisation is equally important due to the shared address space and the potential for race conditions.

OpenMP provides a wide variety of synchronisation methods, namely

- Mutexes, directive and type/function based,
- Barriers,
- Atomic Operations and
- Memory Flushes

Another form of synchronisation for loops is the `ordered` clause, which enables loop serialisation (see above).

Critical

A critical section in a program can be guarded using the `critical` directive:

```
#pragma omp critical [(name)]
```

In contrast to other OpenMP directives, the `critical` directive applies to *all* running threads and not just to the threads in the current team:

```
int main () {
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      #pragma omp parallel for      // team 1
      for ( int i = 0; i < n; ++i )
        f( i );
    }
    #pragma omp section
    {
      #pragma omp parallel for      // team 2
      for ( int j = 0; j < m; ++j )
        f( j );
    }
  }
}
```

```
void f ( int i ) {
  ...
  #pragma omp critical
  {
    ...    // one thread from team 1 _or_ team 2
  }
  ...
}
```

Even though `f` is called from different thread teams, the critical section may only be entered by a single thread at a time.

Critical

Without a name, *all* critical sections are considered to have the *same* name, i.e. all anonymous critical sections form a single critical section:

```
int main () {
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      #pragma omp parallel for      // team 1
      for ( int i = 0; i < n; ++i )
        f( i );
    }
    #pragma omp section
    {
      #pragma omp parallel for      // team 2
      for ( int j = 0; j < m; ++j )
        g( j );
    }
  }
}
```

```
void f ( int i ) {
  ...
  #pragma omp critical
  {
    ...    // one thread from team 1 _or_ team 2
  }
  ...
}
```

```
void g ( int i ) {
  ...
  #pragma omp critical
  {
    ...    // one thread from team 1 _or_ team 2
  }
  ...
}
```

Critical

Without a name, *all* critical sections are considered to have the *same* name, i.e. all anonymous critical sections form a single critical section:

```
int main () {
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      #pragma omp parallel for      // team 1
      for ( int i = 0; i < n; ++i )
        f( i );
    }
    #pragma omp section
    {
      #pragma omp parallel for      // team 2
      for ( int j = 0; j < m; ++j )
        g( j );
    }
  }
}
```

```
void f ( int i ) {
  ...
  #pragma omp critical
  {
    ...    // one thread from team 1 _or_ team 2
  }
  ...
}
```

```
void g ( int i ) {
  ...
  #pragma omp critical
  {
    ...    // one thread from team 1 _or_ team 2
  }
  ...
}
```

With the optional name, this program global mutual exclusion can be limited to critical sections with an equal name:

```
void f ( int i ) {
  ...
  #pragma omp critical (f)
  {
    ...    // one thread from team 1
  }
  ...
}
```

```
void g ( int i ) {
  ...
  #pragma omp critical (g)
  {
    ...    // one thread from team 2
  }
  ...
}
```

Critical

Example: Pipeline

In the pipeline algorithm model, several computations are successively applied to some input data, e.g. for $v \in \mathbb{R}^n$ the value $f_3(f_2(f_1(f_0(v_i)))), i \leq n$, is sought.

This can be done in OpenMP with several queues and parallel blocks, synchronised with critical sections:

```
bool                    end = false;
std::queue< double >    in0, in1, in2, in3;

#pragma omp parallel sections
{
  #pragma omp section      // f0
  { ... }

  #pragma omp section      // f1
  { ... }

  #pragma omp section      // f2
  { ... }

  #pragma omp section      // f3
  { ... }
}
```

```
#pragma omp section                // f1
{
  do {
    double  v;

    #pragma omp critical (in1)
    {
      if ( in1.empty() ) “continue”;
      v = in1.front(); in1.pop();
    }

    v = f1( v );

    #pragma omp critical (in2)
    in2.push( v );
  } while ( ! end );
}
```

Critical

Remark

Recursive function calls within critical sections may lead to deadlocks:

```
void f ( int i ) {
  ...
  #pragma omp critical  // deadlock at second call
  {
    ...
    f();                // recursion
    ...
  }
  ...
}
```

Critical

Remark

Recursive function calls within critical sections may lead to deadlocks:

```
void f ( int i ) {
  ...
  #pragma omp critical  // deadlock at second call
  {
    ...
    f();                // recursion
    ...
  }
  ...
}
```

Example: Dot Product

The previous dot product compution can also be implemented using `critical`:

```
double dot ( const std::vector< double > & x, const std::vector< double > & y ) {
  double sum = 0.0;
  #pragma omp parallel
  {
    double s = 0.0;

    #pragma omp for
    for ( size_t i = 0; i < x.size(); ++i )
      s += x[i] * y[i];

    #pragma omp critical
    sum += s;
  }
  return sum;
}
```

Mutexes

Using the `critical` directive will often lead to unneccessary thread blocking, since it is bound to functions and not to data.

In the following example, the access to the output stream is critical, *not* the function call:

```
void log ( std::ostream & os, const char * text ) {
  #pragma omp critical
  os << text;
}
```

As an alternative, OpenMP provides mutex types and corresponding functions. The data type for an OpenMP mutex is:

```
omp_lock_t
```

Functions for locking and unlocking mutexes are:

```
void omp_set_lock   ( omp_lock_t * mutex );   // lock mutex
void omp_unset_lock ( omp_lock_t * mutex );   // unlock mutex
```

Both are imported via the OpenMP header file `omp.h`.

Mutexes

Before using an OpenMP mutex, it has to be initialised with:

```
void omp_init_lock ( omp_lock_t * mutex );
```

After initialisation, the mutex is unlocked.

Remark

No other OpenMP lock function must be called with an uninitialised mutex!

Finally, if the mutex is no longer used, it should be destroyed using

```
void omp_destroy_lock ( omp_lock_t * mutex );
```

Before calling `omp_destroy_lock`, the mutex has to be *unlocked*.

Furthermore, a test function is provided, which either locks an unlocked mutex or immediately returns:

```
int omp_test_lock ( omp_lock_t * mutex );
```

If the mutex could successfully be locked, the function returns a non-zero value, i.e. `true`. Otherwise, 0 is returned, i.e. `false`.

Mutexes

Using OpenMP mutexes, the above function can be implemented without a function based lock. Instead, a data centered lock is used:

```
#include <omp.h>

void log ( std::ostream & os, const char * text, omp_lock_t * mutex ) {
  omp_set_lock( mutex );
  os << text;
  omp_unset_lock( mutex );
}
```

```
void f ( int i ) {
  omp_lock_t     mutex_f;
  std::ofstream  out_f( "f.txt" );

  omp_init_lock( & mutex_f );

  #pragma omp parallel
  {
    ...

    log( out_f, "message f()", & mutex_f );

    ...
  }

  omp_destroy_lock( & mutex_f );
}
```

```
void g ( int i ) {
  omp_lock_t     mutex_g;
  std::ofstream  out_g( "g.txt" );

  omp_init_lock( & mutex_g );

  #pragma omp parallel
  {
    ...

    log( out_g, "message g()", & mutex_g );

    ...
  }

  omp_destroy_lock( & mutex_g );
}
```

Now, both parallel teams can write to their corresponding output stream simultaneously but still only one thread per team may write at a time.

Mutexes

Example: Dot Product

The previous dot product compution implemented using `omp_lock_t`:

```
double dot ( const std::vector< double > & x, const std::vector< double > & y ) {
  double      sum = 0.0;
  omp_lock_t  mutex;

  omp_init_lock( & mutex );

  #pragma omp parallel
  {
    double s = 0.0;

    #pragma omp for
    for ( size_t i = 0; i < x.size(); ++i )
      s += x[i] * y[i];

    omp_set_lock( & mutex );
    sum += s;
    omp_unset_lock( & mutex );
  }

  omp_destroy_lock( & mutex );

  return sum;
}
```

Nested Mutexes

Locking a locked OpenMP mutex blocks the thread trying to aquire the lock. This may easily lead to deadlocks:

```
void f ( omp_lock_t * mutex ) {
  omp_set_lock( mutex );
  ...
  f();        // deadlock in recursive call
  ...
  omp_unset_lock( mutex );
}
```

For such cases, OpenMP provides *nested* mutexes, which can be locked *multiple* times by the *same* thread. Each time a nested mutex is locked, an internal counter is incremented. When unlocking a nested mutex this counter is decremented and the mutex is unlocked for other threads only when the counter is zero.

The data type for a nested lock in OpenMP is `omp_nest_lock_t` and the corresponding functions are

```
void omp_set_nest_lock     ( omp_nest_lock_t * mutex );
void omp_unset_nest_lock   ( omp_nest_lock_t * mutex );
void omp_init_nest_lock    ( omp_nest_lock_t * mutex );
void omp_destroy_nest_lock ( omp_nest_lock_t * mutex );
int  omp_test_nest_lock    ( omp_nest_lock_t * mutex );
```

Nested Mutexes

Remark

The function `omp_test_nest_lock` will return the new value of the internal counter of the lock was successfully aquired and 0 otherwise.

Using a nested lock, a recursion may proceed in the same thread, which has locked the mutex:

```
void f ( omp_nest_lock_t * mutex ) {
  omp_set_nest_lock( mutex );
  ...
  f();        // NO deadlock in recursive call
  ...
  omp_unset_nest_lock( mutex );
}
```

Mutex Overhead

Locking, unlocking or testing OpenMP mutexes involve read/write operations with respect to the the main memory, which induces some overhead. Furthermore, the different mutex types will perform different operations, and hence, may have different overhead.

The following table contains timings for some typical mutex operations on an Intel Xeon E5-2640:

Operation (20000000×)	Intel Compiler	GNU Compiler
`omp_lock_t`		
locking/unlocking	0.687s	0.313s
testing locked mutex	0.446s	0.202s
`omp_nest_lock_t`		
locking/unlocking	0.747s	0.466s
$n\times$locking/$n\times$unlocking	0.433s	0.174s
testing locked mutex	0.402s	0.063s

Mutexes and C++

Locked mutexes may pose a problem if a runtime *error* is detected in the program:

```
double sum_one_over_x ( std::vector< double > & x,          // shared
                        omp_lock_t &            mutex_x ) { // guards access to x
  double  sum = 0;

  omp_set_lock( & mutex );             // mutex locked
  for ( auto f : x ) {
    if ( f == 0.0 )
      throw "division by zero";       // mutex _not_ unlocked

    sum += 1.0 / f;
  }
  omp_unset_lock( & mutex );           // mutex unlocked

  return sum;
}
```

Before throwing the exception, the mutex has to be unlocked. Otherwise, the program may deadlock later.

This extra error handling is easily forgotten and hence, is a frequent source of errors.

Mutexes and C++

Fortunately, with C++, this can easily be remedied by wrapping mutexes in *scoped lockes*, which automatically locks associated mutexes during construction and unlocks them during destruction:

```
struct scoped_lock_t {
  omp_lock_t * mutex;               // associated mutex

  scoped_lock_t ( omp_lock_t & m ) : mutex(&m) {
    omp_set_lock( mutex );          // mutex locked in ctor
  }
  ~scoped_lock_t () {
    omp_unset_lock( mutex );        // mutex unlocked in dtor
  }
```

Using `scoped_lock_t`, the previous example can be safely rewritten as:

```
double sum_one_over_x ( std::vector< double > & x,          // shared
                        omp_lock_t &            mutex ) {   // guards access to x
  double        sum = 0;
  scoped_lock_t lock( mutex );

  for ( auto f : x ) {
    if ( f == 0.0 )
      throw "division by zero";
    sum += 1.0 / f;
  }
  return sum;
}
```

Now, when leaving the local scope, i.e. the function, the destructor of the variable `lock` is automatically invoked, thereby unlocking the mutex.

Barrier

Most OpenMP directives will have an implicit barrier at the end of their construct, which enforces a synchronisation between all threads of the current team.

With the `barrier` directive, an explicit barrier can be defined in the program:

```
#pragma omp barrier
```

The `barrier` directive has no associated construct.

All threads of the current team have to finish execution of all tasks *before* the `barrier` directive, e.g. all threads will wait until all other threads have reached the barrier:

```
#pragma omp parallel
{
  ...                           // executed _before_ barrier by _all_ threads

  #pragma omp barrier           // wait for _all_ other threads

  ...                           // executed _after_ barrier by _all_ threads
}
```

Barrier

Since the barrier affects all team threads, each thread must encounter the directive during execution or none at all:

```
#pragma omp parallel
{
  ...
  if ( omp_get_thread_num() != 0 ) {   // not by master
    #pragma omp barrier                // error
  }
  ...
}
```

Otherwise, depending on the implementation of the `barrier` directive, the program may block.

Furthermore, the sequence in which `for`, `sections` or `single` directives and the `barrier` directive are encountered during program execution must be *equal* for *all* threads of the team:

```
#pragma omp parallel
{
  if ( omp_get_thread_num() == 0 ) {   // master:
    #pragma omp for                    //    first loop
    ...
    #pragma omp barrier                //    then barrier
  } else {                             // rest:
    #pragma omp barrier                //    first barrier
    #pragma omp for                    //    then loop
    ...
  }
}
```

Barrier

There are also syntactical restrictions to the `barrier` directive.

If a `if`, `while`, `do` or `switch` is used, the `barrier` directive must be enclosed by a structured block, i.e. no standalone statement:

```
#pragma omp parallel
{
  if ( ... )
  #pragma omp barrier                // error

  if ( ... ) {
  #pragma omp barrier                // no error
  }
}
```

Atomic

Atomic operations are supported in OpenMP in the form of the `atomic` directive:

```
#pragma omp atomic [read | write | update | capture]
```

The `atomic` directive applies to the immediately following statement. Depending on the atomic operation, this statement is restricted to one of the following forms:

read:
```
y = x;
```

write:
```
x = expression;
```

update:
```
x++;   x--;   ++x;   --x;
x op= expression;
x = x op expression;
```

capture:
```
y = x++;   y = x--;   y = ++x;   y = --x;
y = x op= expression;
```

Here, `x` and `y` are both scalar types, e.g. `char`, `int` or `double`, and *expression* is a scalar expression. Furthermore, *op* is a standard arithmetic or bit operator, e.g. `+`, `-`, `*`, `/`, `^`, `&`, `|`, `<<` or `>>`.

Remark

The arithmetic and bit operators must not be overloaded.

Atomic

For the `capture` operation, also a structured block is permitted if it has one of the following forms:

```
{ v = x; x op= expression; }   { v = x; x = x op expression; }
{ x op= expression; v = x; }   { x = x op expression; v = x; }

{ v = x; x++; }   { v = x; ++x; }   { v = x; x--; }   { v = x; --x; }
{ x++; v = x; }   { ++x; v = x; }   { x--; v = x; }   { --x; v = x; }
```

Examples for atomic operations are:

```
double  pi = 355.0/113.0, x, y;

#pragma omp atomic read     // atomically read "pi"
x = pi;

#pragma omp atomic write    // atomically write "y"
y = 2.0 * pi;

#pragma omp atomic update   // atomically update "y"
y = y+1;

#pragma omp atomic capture  // atomically update "x" and write y
y = x++;

#pragma omp atomic capture
{ y = x; x--; }
```

Remark

The Intel Compiler does not support expressions of the form "`x = x op expression`", only "`x op= expression`".

Atomic

Atomic operations are performed with respect to memory addresses. All atomic operations affecting the *same* memory address will enforce a mutually exclusive access for *all* threads in the program.

```
double  x = 0;

int main () {
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      #pragma omp parallel  // team 1
      f();
    }
    #pragma omp section
    {
      #pragma omp parallel  // team 2
      g( 2.0 );
    }
  }
}
```

```
void f () {
  #pragma omp atomic update
  x += 1;
}
```

```
void g ( double y ) {
  #pragma omp atomic write
  x = y;
}
```

Since `x` is referenced in both atomic operations, both teams will be affected and the write and update operation will be performed only in *one* thread of the *whole* program at a time.

Atomic

When working with arrays, each array element has a different memory address and hence, will induce a different atomic operation:

```
#pragma omp parallel for
for ( size_t i = 0; i < n; ++i )
{
  #pragma omp atomic write
  x[index[i]] = f( i );
}
```

Here, write operations to different index positions form different atomic operations and may therefore be executed in parallel.

In such cases, `atomic` is a better alternative to `critical`:

```
#pragma omp parallel for
for ( size_t i = 0; i < n; ++i )
{
  #pragma omp critical
  x[index[i]] = f( i );
}
```

which would enforce sequential execution.

Atomic

Example: Dot Product

Implementing the dot product using `atomic`:

```
double dot ( const std::vector< double > & x, const std::vector< double > & y ) {
  double sum = 0.0;

  #pragma omp parallel
  {
    double s = 0.0;

    #pragma omp for
    for ( size_t i = 0; i < x.size(); ++i )
      s += x[i] * y[i];

    #pragma atomic update
    sum += s;
  }

  return sum;
}
```

Limits to Atomic

During an atomic `update` of a variable `x`, only reading and writing the corresponding memory position will be atomic, *not* the evaluation of the expression.

Due to this, the expression must *not* contain references to `x`, e.g.

```
#pragma omp atomic update
x += y*x+3;                 // race condition
```

Similarly, for an atomic `capture`, e.g.

```
#pragma omp atomic capture
y = x++;
```

or

```
#pragma omp atomic capture
{ ++x; y = x; }
```

only reading and writing corresponding to `x` is performed atomic. *Not* the evaluation of an expression or writing `y`.

As for `update`, any expression in the `capture` statement must not contain a reference to the captured variable.

```
#pragma omp atomic capture
{ x += 2*x+y; y = x; }      // race condition
```

Flush

OpenMP uses a memory model with *relaxed* consistency between several threads, i.e. changing a shared variable in one thread will not immediately change corresponding copies in other threads.

In the following example, the master thread changes the shared variable `x`, which is read in another thread:

```
double  x = 0;

#pragma omp parallel shared(x) num_threads(2)
{
  #pragma omp barrier

  if ( omp_get_thread_num() == 0 ) {
    x = 1;                              // written in first thread
  } else {
    std::cout << x << std::endl;        // consumed in second thread
  }
}
```

With an instant update of `x` at the second thread, it should always print out the value 1. Instead, during actual progam runs, also the initial value 0 is printed due to a delayed update of `x` at the second thread.

Remark

The barrier is used to reduce effects of task/thread scheduling.

Flush

Memory consistency with respect to shared variables can be enforced using the `flush` directive:

```
#pragma omp flush [(var1,var2,...)]
```

When writing to a variable, a following `flush` ensures, that the new value is written to memory:

```
x = y;
#pragma omp flush (x) // afterwards memory contains same value as x
```

Similarly, when reading a variable a proceeding `flush` ensures that the variable holds the same value as the corresponding memory position:

```
#pragma omp flush (x)
y = x;                // x has same value as in memory
```

Without a variable list *all* currently visible variables are affected by `flush`.

Flush

Memory consistency with respect to shared variables can be enforced using the `flush` directive:

```
#pragma omp flush [(var1,var2,...)]
```

When writing to a variable, a following `flush` ensures, that the new value is written to memory:

```
x = y;
#pragma omp flush (x) // afterwards memory contains same value as x
```

Similarly, when reading a variable a proceeding `flush` ensures that the variable holds the same value as the corresponding memory position:

```
#pragma omp flush (x)
y = x;                // x has same value as in memory
```

Without a variable list *all* currently visible variables are affected by `flush`.

Using `flush` in the above example, the expected value is always printed:

```
  if ( omp_get_thread_num() == 0 ) {
    x = 1;
    #pragma omp flush (x)
  } else {
    #pragma omp flush (x)
    std::cout << x << std::endl;
  }
```

Flush

In contrast to atomic operations, the flush operation only affects the thread encountering the `flush` directive and *not* other threads.

Hence, using `flush` to read a value from memory will *not* guarantee it to have the most recent value if another thread has not also used `flush` to write the local changes to the main memory:

```
if ( omp_get_thread_num() == 0 ) {
  x = 1;
  // no "write" flush
} else {
  #pragma omp flush (x)
  std::cout << x << std::endl;
}
```

Similarly, if a thread has issued a `flush` after changing a variable, but other threads did not enforce reading the value from memory, their data is still inconsistent:

```
if ( omp_get_thread_num() == 0 ) {
  x = 1;
  #pragma omp flush (x)
} else {
  // no "read" flush
  std::cout << x << std::endl;
}
```

Flush

Several other OpenMP directives perform an *implicit* flush operation with respect to all shared variables:

- during `barrier`,
- at entry and exit from `parallel`, `critical`, and `ordered`,
- at exit from `for` and `sections`, unless `nowait` was specified,
- during locking/unlocking `omp_lock_t` mutexes.

Furthermore, the `atomic` directive performs a flush corresponding to the referenced variable.

Flush

Several other OpenMP directives perform an *implicit* flush operation with respect to all shared variables:

- during `barrier`,
- at entry and exit from `parallel`, `critical`, and `ordered`,
- at exit from `for` and `sections`, unless `nowait` was specified,
- during locking/unlocking `omp_lock_t` mutexes.

Furthermore, the `atomic` directive performs a flush corresponding to the referenced variable.

In all other cases:

Flush *before* reading and *after* writing a shared variable!

Remark

Before OpenMP v2.5 locks did not perform a flush operation, hence, an explicit `flush` was necessary to obtain the most recent value of shared variables.

Flush

atomic vs. flush

Since atomic operations will always guarantee a consistent memory, they should be preferred over `flush` since they are usually *faster*:

```
if ( omp_get_thread_num() == 0 ) {
  x = 1;
  #pragma omp flush (x)
} else {
  #pragma omp flush (x)
  std::cout << x << std::endl;
}
```

```
if ( omp_get_thread_num() == 0 ) {
  #pragma omp atomic write
  x = 1;
} else {
  double y;
  #pragma omp atomic read
  y = x;
  std::cout << y << std::endl;
}
```

Flush

atomic vs. flush

Since atomic operations will always guarantee a consistent memory, they should be preferred over `flush` since they are usually *faster*:

```
if ( omp_get_thread_num() == 0 ) {
  x = 1;
  #pragma omp flush (x)
} else {
  #pragma omp flush (x)
  std::cout << x << std::endl;
}
```

```
if ( omp_get_thread_num() == 0 ) {
  #pragma omp atomic write
  x = 1;
} else {
  double y;
  #pragma omp atomic read
  y = x;
  std::cout << y << std::endl;
}
```

volatile

Variables declared `volatile` have an implicit flush before read and after write:

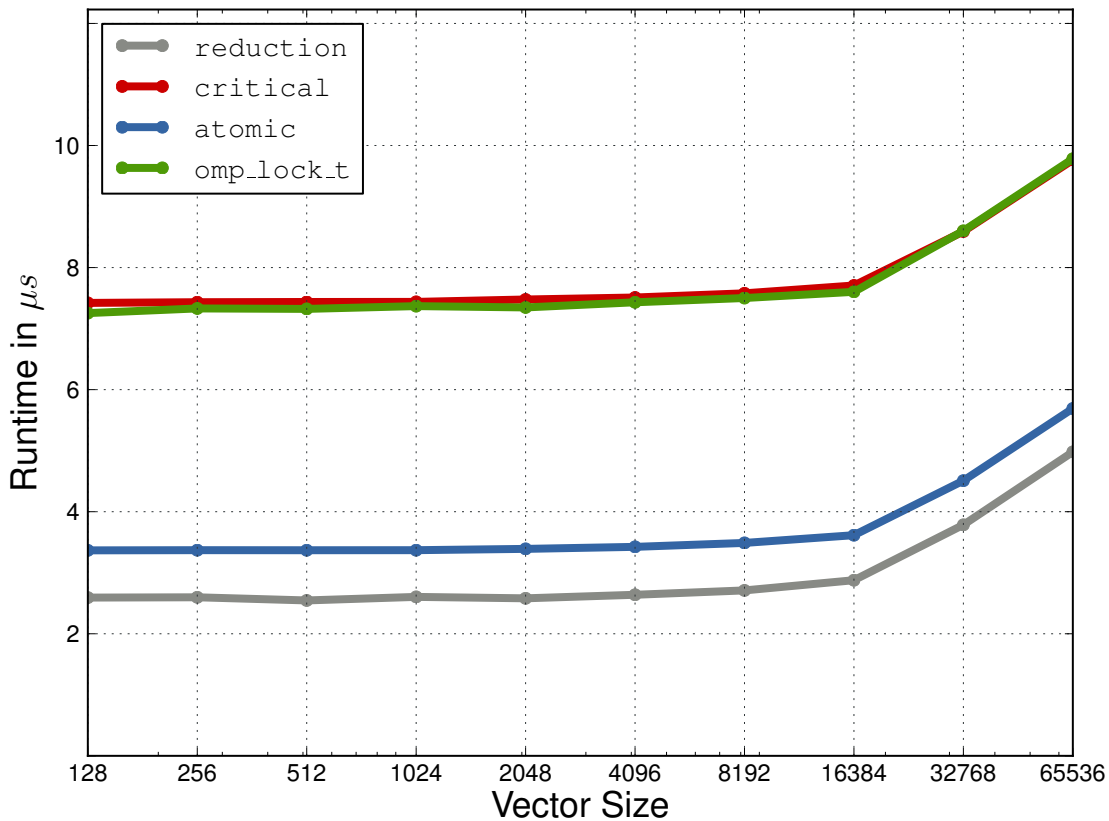
```
volatile double  x = 0;

#pragma omp parallel shared(x) num_threads(2)
{
  #pragma omp barrier
  if ( omp_get_thread_num() == 0 ) {
    x = 1;                          // implicit flush after
  } else {
    std::cout << x << std::endl;    // implicit flush before
  }
}
```

However, this solution usually leads to a worse performance since the compiler can *not* optimise volatile variables as much as non-volatile variables.

Comparison

For the dot product, four equivalent implementations were presented using the `reduction` clause, with the `critical` directive, the `omp_lock_t` locks and the `atomic` directive.

The following diagram shows the runtime of these implementations for different vector sizes on a 2-CPU Intel Xeon E5-2640:



Task based Computations

Task based Computations

Up to now, tasks where *implicitly* defined by the `for` directive, e.g. with

```
#pragma omp for
for ( size_t i = 0; i < n; ++i )
  x[i] = f(i);
```

p tasks are automatically defined by OpenMP, each of the form

```
for ( size_t i = n_lb; i < n_ub; ++i )
  x[i] = f(i);
```

Using one of the directives `sections`, `single` or `master`, tasks could also be defined *explicitly*, e.g.

```
#pragma omp sections
{
  #pragma omp section   // task 1
  { ... }
  #pragma omp section   // task 2
  { ... }
  #pragma omp section   // task 3
  { ... }
}
```

or

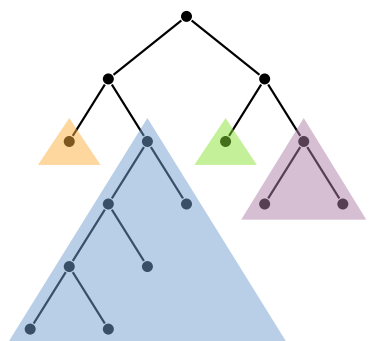
```
#pragma omp parallel
{
  f();                  // part of all tasks

  #pragma omp single    // only part of _one_ task
  { ... }
}
```

Task based Computations

However, such constructs may easily induce additional overhead due to many `parallel` constructs, e.g.:

```
void traverse_tree ( node_t * p ) {
  #pragma omp parallel sections     // new parallel block for each node
  {
    #pragma omp section
    {
      traverse_tree( p->left );
    }
    #pragma omp section
    {
      traverse_tree( p->right );
    }
  }
  compute( p );
}
```

Furthermore, there is a direct relationship between tasks and threads, which may lead to poor load balancing, e.g. if traversing an unbalanced tree with the above function:



Task mapping in `traverse_tree` on 4 processors

Task based Computations

Since v3.0, OpenMP also supports explicit task creation without thread binding:

```
#pragma omp task [clause1 [[,] clause2, ...]]
```

Each task consists of code to execute and a *data environment*. The code is defined by the immediately following block:

```
#pragma omp task
{
  ...           // code executed in task
}
```

All code reachable by code in the construct defines the *task region*.

An important difference between a `task` and a `section` is the time at which it may be scheduled for execution.

`section`: task is executed when associated thread will encounter the directive,

`task`: task may be executed *after* thread encounters directive.

Tasks will only be executed by threads of the current team, e.g. to which also the thread encountering the `task` directive belongs. Furthermore, the encountering thread is *not* neccessarily the executing thread of the task.

Task based Computations

All threads, which encounter a `task` directive, will generate a new task:

```
#pragma omp parallel
{
  for ( int i = 0; i < 4; ++i ) {
    #pragma omp task
    {
      ...
    }
  }
}
```

Here, each of the p threads will execute the `parallel` construct and hence, generate 4 tasks for a total of 16 tasks.

If tasks should be generated only *once* for an algorithm, the `single` or `master` directive may be used:

```
#pragma omp parallel
{
  #pragma omp single
  for ( int i = 0; i < 4; ++i ) {
    #pragma omp task
    {
      ...
    }
  }
}
```

Now, 4 tasks are generated by only *one* thread of the team. However, *all* team threads will execute the generated tasks.

Data Environment

The *data environment* of a task is the set of all variables which are in the scope of the task region. The definition of the data environment depends on the data clauses used before and during the definition of the task.

shared

Shared variables will refer to the memory address available at task construction. This address must be *valid* until the task has finished execution:

```
void f () {
  double  x = produce_x();    // lifetime restricted to f, not the task

  #pragma omp task shared(x)
  { consume_x( x ); }         // error: x may no longer exist
}
```

firstprivate

The variable will be defined with the value at task construction. Since task execution is not immediate, this value has to be stored, i.e. the variable is allocated and packaged together with the task code.

```
void f () {
  double  x = produce_x();

  #pragma omp task firstprivate(x)
  { consume_x( x ); }         // ok: copy of x is packaged with task
}
```

Data Environment

`private`

Private variables will be uninitialised and hence, only allocated when the task is scheduled for execution.

Variables referenced in the task construct with no explicit data sharing rules are shared, if the variables are shared by all implicit tasks in the enclosing region. Otherwise, such variables are `firstprivate`:

```
void f () {
  double  x1 = 1.0;
  double  x2 = 2.0;

  #pragma omp parallel firstprivate(x2)
  {
    double  x3 = 3.0;     // private to each implicit task due to scope

    #pragma omp task
    {
      double  x4 = 4.0;   // private due to scope

      // x1 : shared        ( shared by all implicit tasks )
      // x2 : firstprivate  ( due to “firstprivate(x2)” )
      // x3 : firstprivate  ( not shared by all implicit tasks )
    }
  }
}
```

Remark

Use the clause “`default(none)`” to prevent undefined behaviour.

Task Synchronisation

A task encountering a `task` directive becomes the *parent* task to the newly created *child* task.

Remark

All code in the region of a `parallel` directive is executed in an implicit task.

Code in the construct of a child task is *not* part of the task region of the parent task:

```
#pragma omp task
{
  ...                  // part of parent task

  #pragma omp task
  {
    ...                // part of child task, not of parent task
  }

  ...                  // part of parent task
}
```

After creating the child task, the parent may immediatly proceed with the execution of it's task region.

Task Synchronisation

To synchronise with the end of child tasks, OpenMP provides the directive

```
#pragma omp taskwait
```

Now, the parent task will block until all child tasks have finished execution:

```
void fib ( size_t n ) {
  size_t  i, j;

  #pragma omp task shared(i)
  i = fib( n-1 );

  #pragma omp task shared(j)
  j = fib( n-2 );

  #pragma omp taskwait

  return i+j;
}
```

Task Synchronisation

To synchronise with the end of child tasks, OpenMP provides the directive

```
#pragma omp taskwait
```

Now, the parent task will block until all child tasks have finished execution:

```
void fib ( size_t n ) {
  size_t  i, j;

  #pragma omp task shared(i)
  i = fib( n-1 );

  #pragma omp task shared(j)
  j = fib( n-2 );

  #pragma omp taskwait

  return i+j;
}
```

Placement Restriction

After an `if`, `while`, `do` or `switch`, the `taskwait` must be enclosed by a structured block.

```
if ( ... )
#pragma omp taskwait                // error

if ( ... ) {
#pragma omp taskwait                // no error
}
```

Task Scheduling

The execution of a task is usually deferred to some later point.

Remark

A simplified model uses a FIFO work queue to whish tasks are inserted and requested by idle threads (see "Work Pool Model" or "Online Scheduline").

Furthermore, the execution of a task may be suspended at `task scheduling points`. At such points, the task scheduler may perform a task switch and proceed with the execution of other tasks.

Task scheduling points are implicitly defined

- immediately after creating an explicit task,
- after the last instruction of a task,
- at a `taskwait` directive and
- at implicit and explicit barriers.

Task switching will only be performed with respect to the local thread team and tasks created within.

Task Scheduling

OpenMP also provides the directive `taskyield` to explicitly define a task scheduling point:

```
#pragma omp taskyield
```

If a `taskyield` directive is encountered during the execution of a task, this task may be suspended in favor of another task.

A typical example is a I/O routine, e.g. network communication, which initiates the I/O operation and then waits for it to be finished. During this time, the task may *yield* the executing thread to another task:

```
#pragma omp task
{
  req = start_recv();           // initiate network operation

  while ( ! is_finished( req ) ) {
    #pragma omp taskyield       // switch to computing task
  }

  finish_recv();                // finish network operation
}
```

Remark

The same syntactical restrictions as for `taskwait` apply to `taskyield`.

Task Scheduling

If a task was scheduled to be executed by a thread, this task is *tied* to this thread until the task has finished execution, even if it was suspended in between.

Assuming that the thread is also tied to a specific processer, this behaviour favors *cache locality* of task private data and, to some degree, limits access to remote memory.

Example: Matrix Multiplication with Cache Locality

Exploiting processor caches with a blocked matrix mult. (block size N):

```
Matrix  A(n,n), B(n,n), C(n,n);

for ( size_t i = 0; i < n/N; ++i ) {
  for ( size_t j = 0; j < n/N; ++j ) {
    #pragma omp task
    {
      Matrix  TA(N,N), TB(N,N), TC(N,N);    // cache copies of A,B and C of size NxN

      for ( size_t k = 0; k < n/N; ++k ) {
        load_block( A, TA, i, k );          // read block of A into cache copy
        load_block( B, TB, k, j );          // read block of B into cache copy
        multiply_add( TA, TB, TC );         // pointwise multiplication
      }
      store_block( C, TC, i, j );           // store cache copy in C
    }
  }
}
```

When switching the execution thread for a single task, the local copies for A, B and C may no longer reside at the local processor cache.

Task Scheduling

On the other hand, other tasks may be tied to the same thread, competing for execution time, potentially leading to a load imbalance between team threads.

For such situations, the task may be created using the `untied` clause:

```
#pragma omp task untied
```

This allows other threads to continue task execution if it was suspended at a task scheduling point.

Remark

A consequence of untied tasks is, that the thread id may change during the execution of a task.

Final Clause

Generating and scheduling tasks induces an overhead, which may be more costly, than the execution of the task region. In such cases, it is more efficient to execute the tasks directly by the encountering task.

For this, OpenMP provides the *final* clause:

```
#pragma omp task final(expression)
```

If the expressions of the `final` clause evaluates to `true`, the current task is suspended and the child task is executed immediately by the thread executing the encountering task.

Furthermore, the `final` clause applies to all other tasks descending from the task, i.e. all tasks created in the task region are immediatly executed.

In the following example, the previous task-parallel matrix multiplication will actually use tasks only if $n > N$:

```
Matrix  A(n,n), B(n,n), C(n,n);

#pragma omp task final(n>N)
mat_mul( A, B, C);                          // mat_mul only parallel if n > N
```

Example: LU Factorisation

For a matrix $A \in \mathbb{R}^{n \times n}$ a factorisation

$$A = L \cdot U$$

is sought with a (unit-diagonal) lower triangular matrix $L \in \mathbb{R}^{n \times n}$ and an upper triangular matrix $U \in \mathbb{R}^{n \times n}$.

A sequential implementation is given by:

```
for ( size_t i = 0; i < n; ++i ) {
  // compute column
  for ( size_t j = i+1; j < n; ++j )
    A(j,i) = A(j,i) / A(i,i);

  // update trailing matrix
  for ( size_t j = i+1; j < n; ++j )
    for ( size_t k = i+1; k < n; ++k )
      A(j,k) = A(j,k) - A(j,i) * A(i,k);
}
```

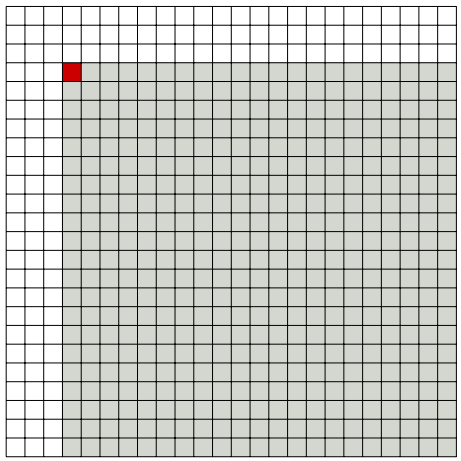
Remark

For simplicity, the LU factorisation is performed without pivoting, although this might result in a breakdown or a less acurate factorisation.

Example: LU Factorisation

The outer loop of the LU factorisation can *not* be parallelised, since the factorisation enforces sequential execution with respect to the diagonal.

After handling the diagonal element,

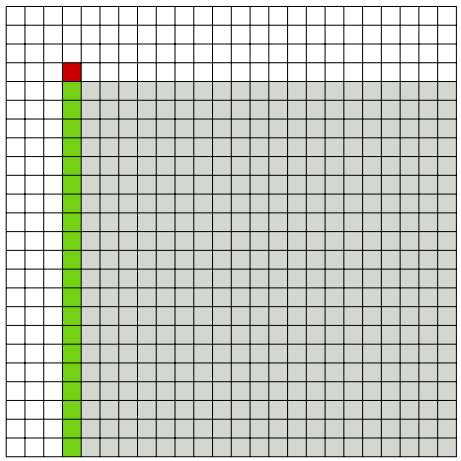


Example: LU Factorisation

The outer loop of the LU factorisation can *not* be parallelised, since the factorisation enforces sequential execution with respect to the diagonal.

After handling the diagonal element,

1. all coefficients in the current column are solved and

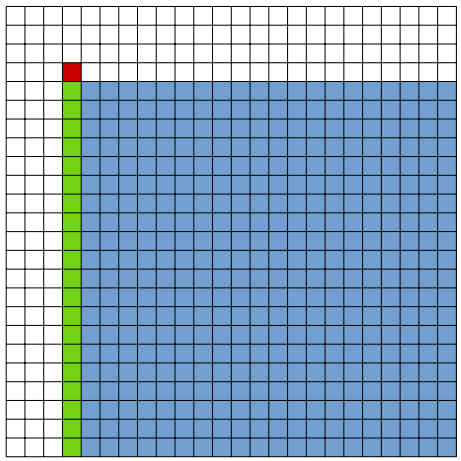


Example: LU Factorisation

The outer loop of the LU factorisation can *not* be parallelised, since the factorisation enforces sequential execution with respect to the diagonal.

After handling the diagonal element,

1. all coefficients in the current column are solved and
2. all coefficients in the trailing submatrix are updated



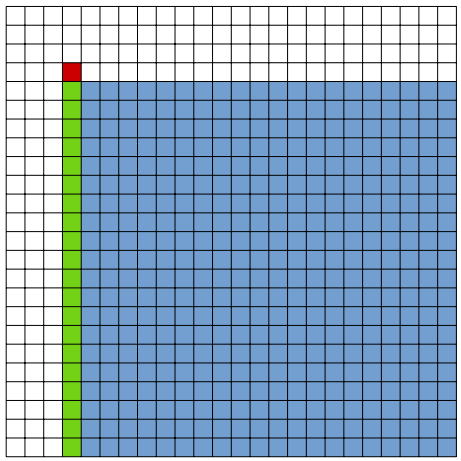
Example: LU Factorisation

The outer loop of the LU factorisation can *not* be parallelised, since the factorisation enforces sequential execution with respect to the diagonal.

After handling the diagonal element,

1. all coefficients in the current column are solved and
2. all coefficients in the trailing submatrix are updated

Both steps perform independent operations and can be parallelised:



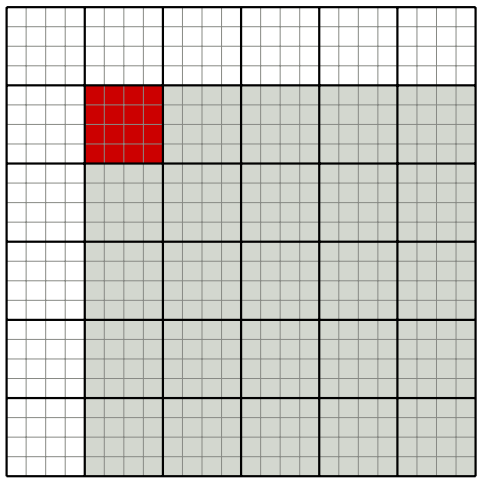
```
for ( size_t i = 0; i < n; ++i ) {
  #pragma omp parallel
  {
    #pragma omp for
    for ( size_t j = i+1; j < n; ++j )
      A(j,i) = A(j,i) / A(i,i);

    #pragma omp for collapse(2)
    for ( size_t j = i+1; j < n; ++j )
      for ( size_t k = i+1; k < n; ++k )
        A(j,k) = A(j,k) - A(j,i) * A(i,k);
} }
```

The update phase involves $\mathcal{O}\left(n^2\right)$ independent operations and hence, may use many processors simultaneously. However, each sub operation only handles a single coefficient.

Example: LU Factorisation

As with the matrix multiplication, LU factorisation will benefit from cache locality. A sequential implementation of the blocked LU factorisation is given by

```
for ( size_t i = 0; i < n/N; ++i ) {
  load_block( A, A_ii, i, i );
  lu( A_ii );                              // factorise A_ii
  store_block( A, A_ii, i, i );

  for ( size_t j = i+1; j < n/N; ++j ) {  // solve
    load_block(  A,    TA, j, i );
    solve_upper( TA,   A_ii );            // L_ji U_ii = A_ji
    store_block( A,    TA, j, i );

    load_block(  A,    TA, i, j );
    solve_lower( A_ii, TA );              // U_ij L_ii = A_ij
    store_block( A,    TA, i, j );
  }

  for ( size_t j = i+1; j < n/N; ++j ) {
    for ( size_t k = i+1; k < n/N; ++k ) {
      load_block( A, L_ji, j, i );
      load_block( A, U_ik, i, k );
      load_block( A, TA, j, k );          // A_jk -= L_ji*U_ik
      multiply_sub( L_ji, U_ik, TA );
      store_block( A, TA, j, k );
} } }
```

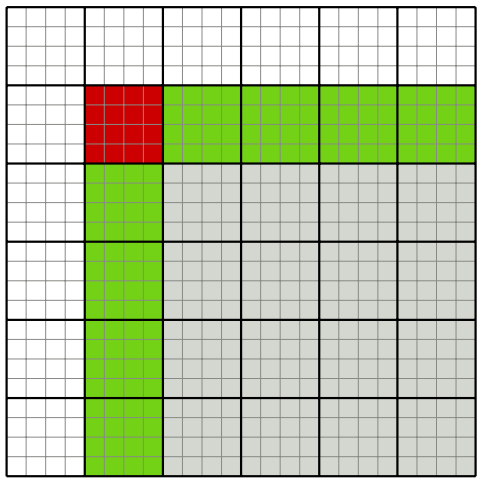


Again, the outer loop has to be handled in strict sequential order. Solving is now also performed for the current row. Furthermore, the number of independent operations is now reduced by a factor if N for solving and N^2 for the matrix updates.

Example: LU Factorisation

As with the matrix multiplication, LU factorisation will benefit from cache locality. A sequential implementation of the blocked LU factorisation is given by

```
for ( size_t i = 0; i < n/N; ++i ) {
  load_block( A, A_ii, i, i );
  lu( A_ii );                              // factorise A_ii
  store_block( A, A_ii, i, i );

  for ( size_t j = i+1; j < n/N; ++j ) {  // solve
    load_block(  A,    TA, j, i );
    solve_upper( TA,   A_ii );             // L_ji U_ii = A_ji
    store_block( A,    TA, j, i );

    load_block(  A,    TA, i, j );
    solve_lower( A_ii, TA );               // U_ij L_ii = A_ij
    store_block( A,    TA, i, j );
  }

  for ( size_t j = i+1; j < n/N; ++j ) {
    for ( size_t k = i+1; k < n/N; ++k ) {
      load_block( A, L_ji, j, i );
      load_block( A, U_ik, i, k );
      load_block( A, TA, j, k );           // A_jk -= L_ji*U_ik
      multiply_sub( L_ji, U_ik, TA );
      store_block( A, TA, j, k );
} } }
```

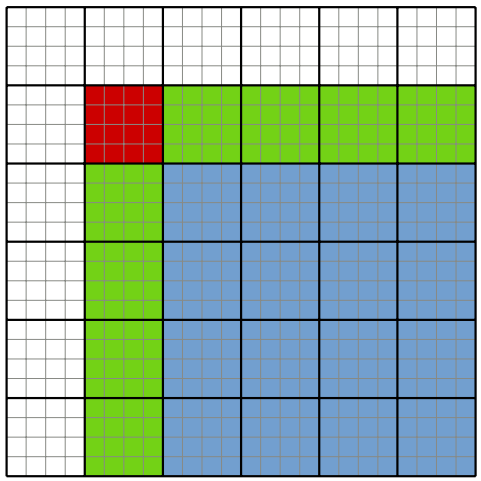


Again, the outer loop has to be handled in strict sequential order. Solving is now also performed for the current row. Furthermore, the number of independent operations is now reduced by a factor if N for solving and N^2 for the matrix updates.

Example: LU Factorisation

As with the matrix multiplication, LU factorisation will benefit from cache locality. A sequential implementation of the blocked LU factorisation is given by

```
for ( size_t i = 0; i < n/N; ++i ) {
  load_block( A, A_ii, i, i );
  lu( A_ii );                              // factorise A_ii
  store_block( A, A_ii, i, i );

  for ( size_t j = i+1; j < n/N; ++j ) {  // solve
    load_block(  A,    TA, j, i );
    solve_upper( TA,   A_ii );             // L_ji U_ii = A_ji
    store_block( A,    TA, j, i );

    load_block(  A,    TA, i, j );
    solve_lower( A_ii, TA );               // U_ij L_ii = A_ij
    store_block( A,    TA, i, j );
  }

  for ( size_t j = i+1; j < n/N; ++j ) {
    for ( size_t k = i+1; k < n/N; ++k ) {
      load_block( A, L_ji, j, i );
      load_block( A, U_ik, i, k );
      load_block( A, TA, j, k );           // A_jk -= L_ji*U_ik
      multiply_sub( L_ji, U_ik, TA );
      store_block( A, TA, j, k );
} } }
```



Again, the outer loop has to be handled in strict sequential order. Solving is now also performed for the current row. Furthermore, the number of independent operations is now reduced by a factor if N for solving and N^2 for the matrix updates.

Example: LU Factorisation

Using tasks, each solve or update operation may be performed as a single task:

```
for ( size_t i = 0; i < n/N; ++i ) {
  load_block( A, A_ii, i, i );
  lu( A_ii );                                        // factorise A_ii
  store_block( A, A_ii, i, i );

  for ( size_t j = i+1; j < n/N; ++j ) {
    #pragma omp task firstprivate(j) shared(A,A_ii)
    { Matrix A_ji(N,N);                              // task private

      load_block(  A,  A_ji, j, i );
      solve_upper( TA, A_ii );                       // L_ji U_ii = A_ji
      store_block( A,  A_ji, j, i );
    }
    #pragma omp task firstprivate(j) shared(A,A_ii)
    { Matrix A_ij(N,N);                              // task private

      load_block(  A,    A_ij, i, j );
      solve_lower( A_ii, A_ij );                     // U_ij L_ii = A_ij
      store_block( A,    A_ij, i, j );
  } }
  #pragma omp taskwait                               // wait for all solves
  for ( size_t j = i+1; j < n/N; ++j ) {
    for ( size_t k = i+1; k < n/N; ++k ) {
      #pragma omp task firstprivate(j,k) shared(A)
      { Matrix A_jk(N,N), L_ji(N,N), U_ik(N,N);      // task private

        load_block( A, L_ji, j, i );
        load_block( A, U_ik, i, k );
        load_block( A, A_jk, j, k );
        multiply_sub( L_ji, U_ik, A_jk );            // A_jk -= L_ji*U_ik
        store_block( A, A_jk, j, k );
  } } }
  #pragma omp taskwait                               // wait for all updates
}
```

Example: LU Factorisation

Finally, the following speedups are achieved by the different versions of the LU factorisation for $n = 8192$ and $N = 64$:

	Xeon X5650 (24 threads)	Xeon E5-2640 (24 threads)	XeonPhi 5110P (240 threads)
block vs. point wise	1.20x	1.05x	1.42x
parallel			
point wise	1.11x	1.57x	0.24x
block wise	9.43x	12.44x	0.59x

The difference between point and block wise in the sequential case is not as large as for matrix multiplication.

However, parallelising the point wise LU factorisation is very inefficient, due to the bad ratio of task computation and overhead.

The better approach is the parallel block wise factorisation, which achieves (nearly) optimal speedup. Except on the MIC architecture, which currently seems to have a problem with tasks.

Example: LU Factorisation

Using the `for` directive to parallelise the block wise LU factorization resulted in the following, only slightly worse speedup:

	Xeon X5650 (24 threads)	Xeon E5-2640 (24 threads)	XeonPhi 5110P (240 threads)
parallel block wise			
`task`	9.43x	12.44x	0.59x
`for`	9.28x	12.20x	118.21x

Miscellanea

Thread Private Data

Up to now, data is associated with implicitly or explicitly defined tasks:

```
#pragma omp parallel shared(x,n) private(y)    // p implicit tasks
{
  ...                                          // access to x,n,y

  #pragma omp for                              // n/p implicit tasks
  for ( size_t i = 0; i < n; ++i )
  {
    ...                                        // access to x,n,y,i
  }

  #pragma omp single
  for ( int j = 0; j < 4; ++j )
  {
    #pragma omp task firstprivate(j,x)         // 4 explicit tasks
    {
      ...                                      // access to j,x
    }
  }
}
```

For implict tasks, task and thread are tightly coupled and may be used interchangeably.

However, a thread is the software unit which executes tasks and does *not* define them. This distinction is more obvious for explicit tasks, especially *untied* tasks.

Thread Private Data

In OpenMP, data can also be coupled with threads. Such data is called *thread-private* and is declared by the directive

```
#pragma omp threadprivate (var1,var2,...)
```

The lifetime of a `threadprivate` variable must *not* depend on the scope of a function. Hence, thread-private variables must either be in the scope of files or namespaces:

```
double x;                              // file scope
#pragma omp threadprivate(x)

void f () { ... }

namespace X {
  double y;                            // namespace scope
  #pragma omp threadprivate(y)
  ...
}
```

or declared *static*:

```
static double x;
#pragma omp threadprivate(x)

void f () {
  static double y;
  #pragma omp threadprivate(y)
  ...
}
```

```
struct A {
  static double z;
  #pragma omp threadprivate(z)
  ...
}
```

Thread Private Data

The inital value of thread-private variables equals the value at the declaration.

```
int  n = 0;
#pragma omp threadprivate(n)

int main () {
  #pragma omp parallel
  {
    ...                                 // n == 0
} }
```

Changes after declaration will only affect the private variable of each thread:

```
int  n = 0;
#pragma omp threadprivate(n)

int main () {
  #pragma omp parallel
  {
    n = 2 * omp_get_thread_num();   // n == 2*i in thread i
} }
```

Since the `main` function or the function of the master thread is already executed by a thread, this also applies to changes therein:

```
int  n = 0;
#pragma omp threadprivate(n)

int main () {
  n = 10;                               // change n of main/master thread

  #pragma omp parallel
  {                                     // n == 10 in master thread, n == 0 in all other threads
    n = 2 * omp_get_thread_num();   // n == 2*i in thread i
} }
```

Copyin Clause

To update the value of thread private variables when entering a `parallel` region, OpenMP provides the clause

```
#pragma omp parallel copyin (var1,var2,...)
```

The master thread value of a variable in the list of the `copyin` clause is then copied to all thread-private variables of the newly created team thread:

```
int  n = 0;
#pragma omp threadprivate(n)

int main () {
  n = 10;                              // change n of main/master thread

  #pragma omp parallel copyin(n)
  {                                    // n == 10 in all threads
    n = 2 * omp_get_thread_num();      // n == 2*i in thread i
} }
```

Remark

The copy assigment operator is used for `copyin` variables. Arrays are copied elementwise. For classes, the corresponding operator has to be accessible.

Restrictions

Beside the limitation to file or namespace scope and static variables, further restrictions apply to `threadprivate` variables:

- Thread private variables may only be used in `copyin`, `copyprivate`, `schedule` or `num_threads` clauses:

```
int  n = 0;
#pragma omp threadprivate(n)
...
#pragma omp parallel shared(n)     // error: invalid clause
```

- A reference to a thread private variable must not appear *before* the `threadprivate` declaration:

```
void f ( int & i ) {
  static int  n = 0;
  ...
  j = i * n;                       // error: reference before declaration
  #pragma omp threadprivate(n)
}
```

- Thread private variables of class type must have an accessible default and copy constructor and a constructor for a given intialisation:

```
struct A {
  A ();
private:
  A ( const A & );                 // error: not accessible
};
static A a = double(2);            // error: no constructor
#pragma omp threadprivate(a)
```

Thread Scheduling

OpenMP itself only provides very limited control over the scheduling of the threads on to processors.

The following two environment variables are available

OMP_PROC_BIND: If set to `true`, the OpenMP threads will not be moved between processes, i.e. once spawned, they are bound to a specific processor.

OMP_DYNAMIC: If set to `false`, OpenMP threads will *not* be created dynamically, e.g. a fixed set of threads is created upon program start (or at the first `parallel` directive).

For both variables, the default value depends on the OpenMP implementation, e.g. the compiler used.

In case of `OMP_DYNAMIC`, the GNU and Intel compiler both default to `false`.

Thread Scheduling

The Intel and the GNU compiler also provide special environment variables to explicitly define the mapping of threads to processors using the processor affinity map of the Linux operating system.

Intel Compiler

The processor affinity of threads is controlled by the *KMP_AFFINITY* environment variable.

Two different levels of control are provided. The high level gives general control over the assignment of threads with the values *compact* and *scatter*:

KMP_AFFINITY=compact	KMP_AFFINITY=scatter
Place thread $i+1$ as close as possible to thread i, e.g. first fill all cores of same processor.	Distribute threads as evenly as possible over system, e.g. assign thread to not yet used processor.

(Example with 8 threads on four dual-core CPUs.)

Thread Scheduling

The low level interface uses an explicit processor list. For each thread the corresponding processor is specified.

For this, `KMP_AFFINITY` has to be set to *explicit* with the additional *proclist* argument:

```
KMP_AFFINITY="explicit,proclist=[0 2 4 6]"
```

Here, the thread 0 is assigned to processor 0, thread 1 to processor 2, etc..

GNU Compiler

The GNU compiler only supports the low level interface for thread affinity. The corresponding environment variable is *GOMP_CPU_AFFINITY*:

```
GOMP_CPU_AFFINITY="0 2 4 6"
```

The general format for `GOMP_CPU_AFFINITY` is $N - M : S$, with start processor N, end processor M and optional stride S, e.g.:

```
GOMP_CPU_AFFINITY="0-6:2"
```

is identical to the above statement.

If Clause

For the `parallel` and the `task` directive, OpenMP allows the conditional parallel execution of the corresponding constructs in the form of the *if* clause:

```
#pragma omp parallel if(expression)
```

or

```
#pragma omp task if(expression)
```

In case of the `parallel` directive, if the `if` clause evaluates to `false`, no parallel team is formed and the construct is executed sequentially by the master thread:

```
double dot ( const size_t n, double * x, double * y ) {
  double f = 0;

  #pragma omp parallel for reduction(+:f) if(n>=1000)
  for ( size_t i = 0; i < n; ++i )
    f += x[i] * y[i];

  return f;
}
```

Here, parallel execution is only started if enough work per thread is available and the overhead of task generation and task scheduling can be neglected.

For the `task` directive, if the expression of the `if` clause is `false`, the encountering task is suspended and the new task is immediately executed.

C++ Exceptions

When throwing exceptions in C++, OpenMP enforces a strict handling of these exceptions:

> *All exceptions thrown within a region of an OpenMP directive have to be catched with the same region by the same thread (or task) throwing the exception.*

This applies to all directives generating explicit or implicit tasks, i.e. `parallel`, `for`, `section`, `single`, `task`, `master`, `critical` and `ordered`.

Hence, the following handling is not allowed:

```
try {
  #pragma omp parallel for
  for ( size_t i = 0; i < n; ++i ) {
    if ( x[i] == 0 )
      throw "division by zero";
    else
      x[i] = 1.0 / x[i];
  }
}
catch ( const char * e ) {            // error: exception not catched within same thread
  std::cout << e << std::endl;
}
```

Literature

"*OpenMP Application Program Interface, Version 3.1*". http://www.openmp.org.

"*OpenMP Application Program Interface, Version 4.0 RC2*". http://www.openmp.org.

Vladimirov, A. and V. Karpusenko. "*Test-driving Intel Xeon Phi coprocessors with a basic N-body simulation*". http://research.colfaxinternational.com/.