Advanced Topics in Numerical Analysis: High Performance Computing

Cross-listed as MATH-GA.2012-001 and CSCI-GA 2945.001

Benjamin Peherstorfer
Courant Institute, NYU
pehersto@cims.nyu.edu

Spring 2020, Monday, 1:25–3:15PM, WWH #512

March 9, 2020

Slightly adapted from Georg Stadler's lectures.

Today

Last lecture

- OpenMP: I

Today

- OpenMP II

Announcements and upcoming

- Homework 2 due Fri, March 14 midnight

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.

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 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.

Nested loops

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

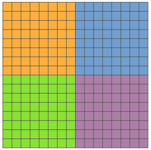
In the matrix multiplication algorithm, the two outer loops apply to independent parts of the destination matrix C. Hence, these should be natural candidates for nested parallel loops:

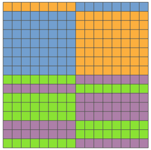
```
void mat_mul ( const size_t n, const Matrix & A, const Matrix & B, Matrix & C ) {
  #pragma omp parallel num_threads(2)
  #pragma omp for
  for ( size_t  i = 0; i < n; ++i ) {
    #pragma omp parallel num_threads(2)  // nested, parallel loop
    #pragma omp for
    for ( size_t  j = 0; j < n; ++j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

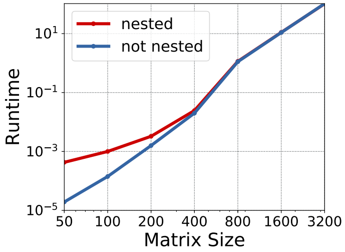
DEMO: omp08.cpp

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.



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

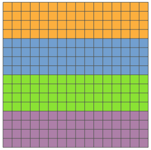
```
#pragma omp for collapse(2)
for ( size_t i = 0; i < n; ++i ) {
  for ( size_t j = 0; j < i; ++j ) {          // loop dependence
    ...
} }
```

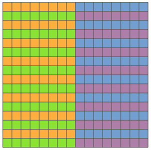
are not allowed.

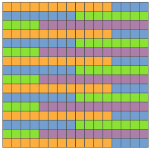
For the matrix multiplication, the outer loops are perfectly nested and independent, hence, they may be collapsed as an alternative to nested parallelism:

```
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;
} } }
```

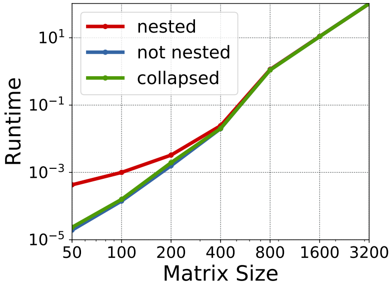
The resulting mapping depends on the specified scheduling and especially the chunk size:


collapse(2)


collapse(2) schedule(static,8)


collapse(2) schedule(static,12)

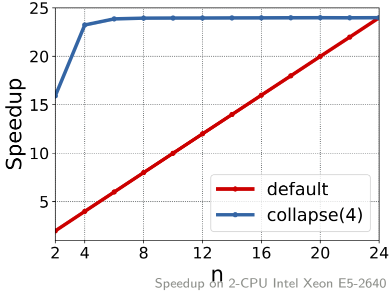
As the scheduling is now applied to all rows and columns simultaneously, the additional overhead of the nested parallelism is eliminated:



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.

DEMO: omp09.cpp

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.

Supported operators are:

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

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
```

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.

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 synchronization

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.

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 serialization

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.

DEMO: omp10.cpp

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.

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.

Any problem swith the following code?

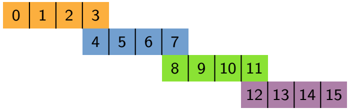
```
#pragma omp for schedule(static) ordered
for ( size_t i = 0; i < 16; ++i )
{
  double  x = f();

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

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*.

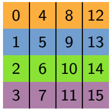
For example, the following loop is executed almost sequentially while using 4 threads:

```
#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;
}
```



N-body problem

N-body problem:
DEMO: omp11.cpp

N-body problem

N-body problem:
DEMO: omp11.cpp

Jacobi iterations:
DEMO: omp14.cpp

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.

Remark

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

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
}
```

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`:	enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section
`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.

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.

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.

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
}
```

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
}
```

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 synchronization

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        // tea
      for ( int i = 0; i < n; ++i )
        f( i );
    }
    #pragma omp section
    {
      #pragma omp parallel for        // tea
      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.

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          // tea
      for ( int i = 0; i < n; ++i )
        f( i );
    }
    #pragma omp section
    {
      #pragma omp parallel for          // tea
      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
  }
  ...
}
```

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          // tea
      for ( int i = 0; i < n; ++i )
        f( i );
    }
    #pragma omp section
    {
      #pragma omp parallel for          // tea
      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
  }
  ...
}
```

DEMO: omp12.cpp

DEMO: omp12.cpp

Remark

Recursive function calls *within* critical sections may lead to *deadlocks*:

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

DEMO: omp12.cpp

Remark

Recursive function calls *within* critical sections may lead to *deadlocks*:

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

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

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`.

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`.

Dot Product

The previous dot product computation 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;
}
```

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 (20000000x)	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

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
}
```

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. +, −, ∗, /, ˆ, &, |, << or >>.

Remark

The arithmetic and bit operators must not be overloaded.

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++;
```

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 vs critical

Which ones would you prefer? Why? → DEMO omp15.cpp

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

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

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.

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 of 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
```

1. Evaluate expression $y * x + 3$ (potential interrupts during evaluation of expression)
2. Atomic: Read-x, add value of expression, write-x

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

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

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