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 23, 2020

Slightly adapted from Georg Stadler's lectures.

Logistics

- We meet via Zoom for the rest of the semester
- Zoom meeting link is available in the Google Docs and NYU classes
- Please be active participants despite online teaching
  - If you have a question/comment unmute you and you can speak to the class
  - Make sure you mute again after you spoke
  - Alternatively, you can post your question in the chat window
- Link to Zoom meeting for office hour is in the Google Docs document
  - I might be in other meetings during the office hour: best is to send an email if Zoom says I am in another meeting.
  - Be aware that other people can join the office hour while we speak
  - If you want to talk about confidential things, let me know at the beginning of the meeting and we use a “breakout room”
- Homework submission/final project will proceed as before

Logistics

- We meet via Zoom for the rest of the semester
- Zoom meeting link is available in the Google Docs and NYU classes
- Please be active participants despite online teaching
    - If you have a question/comment unmute you and you can speak to the class
    - Make sure you mute again after you spoke
    - Alternatively, you can post your question in the chat window
- Link to Zoom meeting for office hour is in the Google Docs document
    - I might be in other meetings during the office hour: best is to send an email if Zoom says I am in another meeting.
    - Be aware that other people can join the office hour while we speak
    - If you want to talk about confidential things, let me know at the beginning of the meeting and we use a “breakout room”
- Homework submission/final project will proceed as before
- Any questions/comments?

Today

Last lecture

- OpenMP: II

Today

- Finish OpenMP III
- Libraries

Announcements and upcoming

- Homework 3 posted (due April 6)
- You should have a rough idea about your final project by now - talk to me about it and finalize a project description.

Final project

Summarize your current plan for the final project in a PDF document and send to me and Melody via email.

- We assume you have already talked to us about your project ideas when this homework is due and when you have sent the project description via email.
- Detail *what* you are planning to do, and with *whom* you will be cooperating.
- The preferred size of final project teams is two, but if this makes sense in terms of the size of the project, teams of three or doing a project by yourself is fine as well.
- Each team is expected to give a 10 minute presentation about the problem they have worked on and their results and experience during the final week (likely May 4) and hand in a report as well as their code in a repo.
- However, it is important that you call out 4-5 concrete tasks in your project description. We will request frequent updates during the rest of the semester on the progress you are making on these tasks.

Review: 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
```

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.

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`.

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.

Dot product implementations

Reduction

```
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;
}
```

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;
}
```

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;
}
```

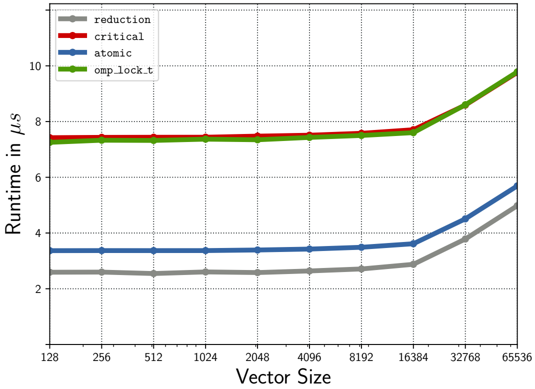
Dot product computation can also be implemented using `critical`:

```
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;
}
```

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:



Memory

DEMO: omp16.cpp

What will the output of this program be?

```
double  x = 0;
double  y = 0;

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

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

Memory

DEMO: omp16.cpp

What will the output of this program be?

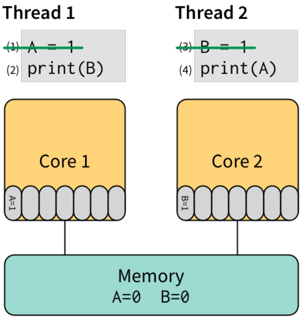
```
double  x = 0;
double  y = 0;

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

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

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.

→ it is possible that "00" is printed

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.



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.
[Figure: Bornholt]

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`.

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
```

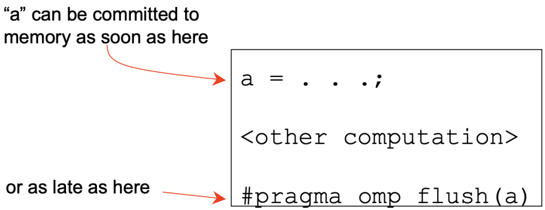
Similarly, when reading a variable a proceeding `flush` ensures that the variable holds the same value as the corresponding memory position:

```
#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;                              // written in first thread
        #pragma omp flush (x,y)
        std::cout << y;
    } else {
        y = 1;
        #pragma omp flush (y,x)
        std::cout << x;       // consumed in second thread
    }
```

Temporary view allows hiding memory latency



```
a = . . .;

<other computation>

#pragma omp flush(a)
```

Re-ordering example

```
a = ...;  //(1)
b = ...;  //(2)
c = ...;  //(3)

#pragma omp flush(c)    //(4)
#pragma omp flush(a,b)  //(5)

. . . a . . . b . . .;  //(6)
. . . c . . .;          //(7)
```

(1) and (2) may not be moved after (5).

(6) may not be moved before (5).

(4) and (5) may be interchanged at will.

[Figures: Vivek Sarkar]

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;
}
```

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.

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!

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;
}
```

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.

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
  { ... }
}
```

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.

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
}
```

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 synchronization

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.

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;
}
```

DEMO: omp21.cpp

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.

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
}
```

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 processor, this behavior favors *cache locality* of task private data and, to some degree, limits access to remote memory.

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, a task-parallel matrix multiplication will 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 Factorization

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

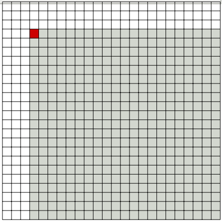
$$A = L \cdot U$$

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);
}
```

After handling the diagonal element,



Example: LU Factorization

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

$$A = L \cdot U$$

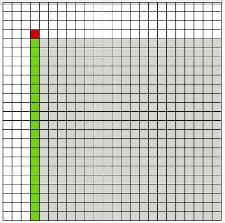
is sought with a (unit-diagonal) lower triangular matrix $L \in \mathbb{R}^{n \times n}$ and an upper triangular matrix $U \in \mathbb{R}^{n \times n}$.

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);
}
```

After handling the diagonal element,

1. all coefficients in the current column are solved and



Example: LU Factorization

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

$$A = L \cdot U$$

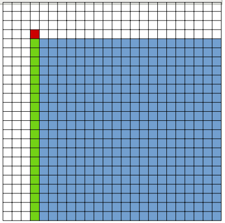
is sought with a (unit-diagonal) lower triangular matrix $L \in \mathbb{R}^{n \times n}$ and an upper triangular matrix $U \in \mathbb{R}^{n \times n}$.

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);
}
```

After handling the diagonal element,

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



Parallelization?

```
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);
}
```

Parallelization?

```
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);
}
```

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

Parallelization?

```
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);
}
```

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

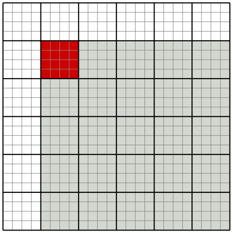
Remark

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

Parallel code $\rightarrow$ DEMO omp17a.cpp

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

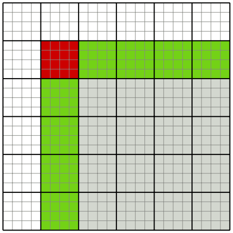
```
for ( size_t i = 0; i < n/N; ++i ) {
  load_block( A, A_ii, i, i );
  lu( A_ii );                                // factorise A_ii
  store_block( A, A_ii, i, i );

  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.

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

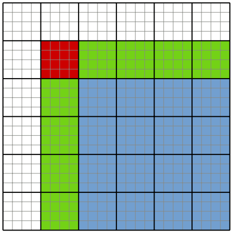
```
for ( size_t i = 0; i < n/N; ++i ) {
  load_block( A, A_ii, i , i );
  lu( A_ii );                                  // factorise A_ii
  store_block( A, A_ii, i , i );

  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.

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.

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
}
```

DEMO: omp17.cpp

Thread private data

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

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

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

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

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.

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. For example:

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

void f () { ... }
```

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)
  ...
}
```

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 thread
    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.

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`.

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.

DEMO: omp20.cpp

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)
```

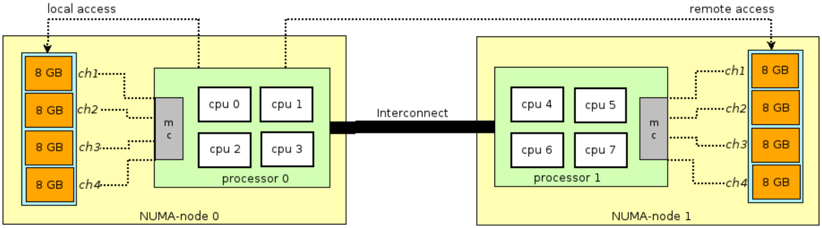
In case of the `parallel` directive, if the `if` clause evaluates to `false`, no parallel team is formed and the construct is executed sequentially by the master thread:

```
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.

Cache Coherent Non-uniform Memory Access

- **Cores:** individual processing units.
- **Sockets:** collection of cores on the same silicon die.
- Each sockets connected to its own DRAM.
- Sockets interconnected using a network: QPI (Intel), HT (AMD).



*

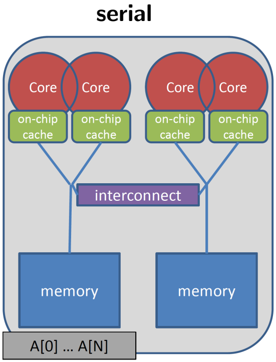
DEMO: omp18.cpp

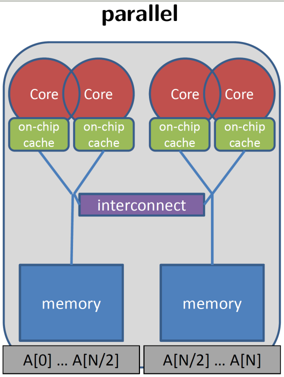
*figure from: https://www.boost.org

NUMA: A simple "solution"

Parallelize initialization → first-touch policy

```
double reduce(long n, double* A) {
  double sum = 0;
  #pragma omp parallel for schedule(static) reduction(+:sum)
  for (long i = 0; i < n; i++) sum += A[i];
  return sum;
}

#pragma omp parallel for schedule(static)
for (long i = 0; i < n; i++) A[i] = i+1;

double sum = reduce(n, A);
```





[Figures: Schmidl, Terboven]

Basic Linear Algebra Subprograms (BLAS)

Specification for basic linear algebra operations

- Level 1: vector-vector operations (dot-product, norm, axpy)
- Level 2: matrix-vector operations (generalized matrix-vector multiplication: gemv)
- Level 3: matrix-matrix operations (generalized matrix-matrix multiplication: gemm)

Many implementations available

- Open-source: OpenBLAS, ATLAS (Automatically Tuned Linear Algebra Software)
- Intel Math Kernel Library (MKL)

GEMM (sgemm, dgemm)

Function declaration:

```
extern "C" { // from C++
void dgemm_(char* TRANSA, char* TRANSB, int* M, int*
    N, int* K, double* ALPHA, double* A, int* LDA,
    double* B, int* LDB, double* BETA, double* C,
    int* LDC);
}
```

Linking:

```
g++ MMult.cpp -lblas
icpc MMult.cpp -mkl
```

(for linking to MKL using non-Intel compilers:
https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor)

DEMO: omp19.cpp

Other Useful Libraries

Linear Algebra Package (LAPACK)

- Linear solvers, LU factorization, QR factorization, singular value decomposition (SVD)
- Built on top of BLAS
- Many implementations available (also included in MKL)

Fastest Fourier Transform in the West (FFTW) http://www.fftw.org

- Optimized FFT implementation, open source.