Shared memory programming model – OpenMP

TMA4280—Introduction to Supercomputing

NTNU, IMF

February 16. 2018

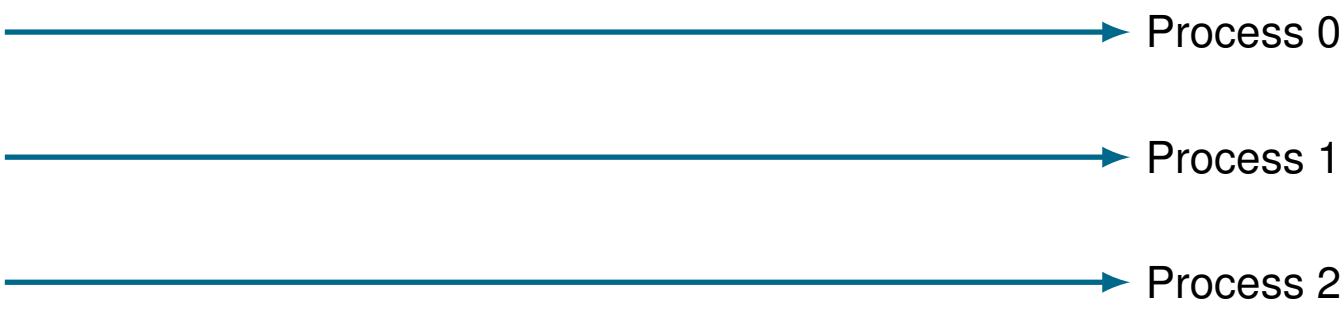
Recap: Distributed memory programming model

Parallelism with MPI.
An MPI execution is started on a set of processes $\mathcal{P}$ with:

```
mpirun -n N_P ./exe
```

with $N_P = card(\mathcal{P})$ the number of processes.

A program is executed in parallel on each process:



where each process in $\mathcal{P} = \{P0, P1, P2\}$ has a access to data in its memory space: remote data exchanged through interconnect.

An ordered set of processes defines a **Group** (`MPI_Group`): the inital group (`WORLD`) consists of **all** processes.

Recap: Distributed memory programming model

Two motivations for leveraging another type of parallelism:

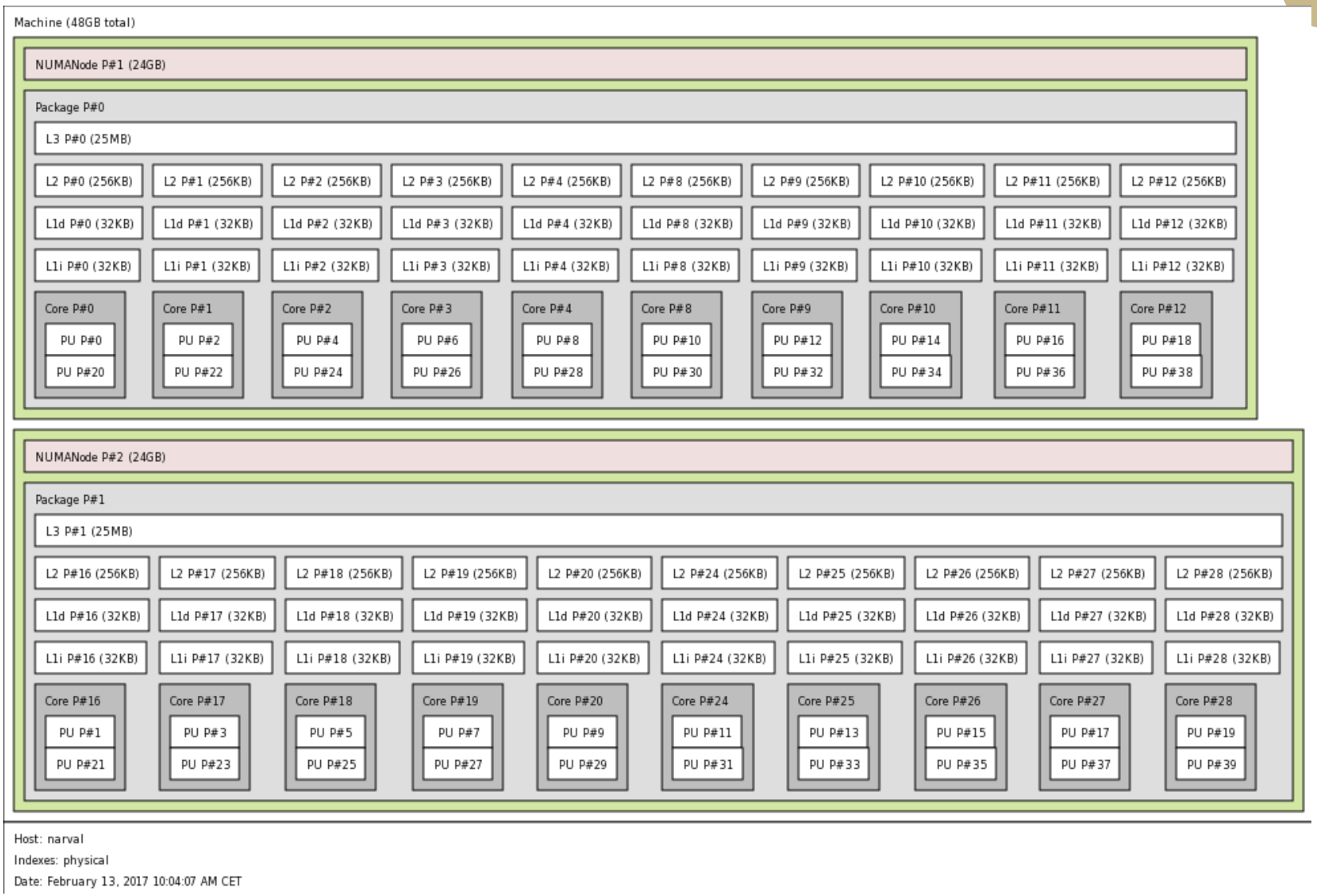
1. Parallelization with MPI requires substantial changes:
   - explicit programming
   - using an external library
   - adding functions to exchange data, perform collective operations

   The programmer needs to define data structures to divide work then handle data communication: it may alter heavily his serial implementation, intrusive.

2. Single core performance has reached its limits (see ILP):
   - Typical multiprocessor machine is now multicore,
   - NUMA architecture: a machine consists of multiple NUMA nodes (memory locality) with cache coherency,
   - Message Passing is not necessary,
   - Finer parallelism can be achieved.

Recap: Shared memory architecture

Figure: Example of NUMA plotted by `lstopo` utility of `hwloc`



Shared memory programming model

In a shared memory model, any worker can take a memory reference.

Some implementations of the shared memory programming model:

- Native threads (UNIX System V, POSIX, Solaris, Linux)
- Java threads (Virtual Machine)
- OpenMP
- Apple Grand Central Dispatch (GCD)
- Intel Thread-Building Blocks (TBB)
- ...

Initially developed for HPC systems and now widely used on the desktop, for example in multimedia applications.

OpenMP

OpenMP, portable standard for shared-memory parallel programming:

OpenMP 1.0	Fortran (1997) C/C++ (1998)
OpenMP 1.1	Fortran (1999)
OpenMP 2.5	Fortran/C/C++ (2005)
OpenMP 3.0	Fortran/C/C++ (2008)
OpenMP 3.1	Fortran/C/C++ (2011)
OpenMP 4.0	Fortran/C/C++ (2013)
OpenMP 4.5	Fortran/C/C++ (2015)

Latest specifications introduce support *offloading* to a SIMD unit, see:

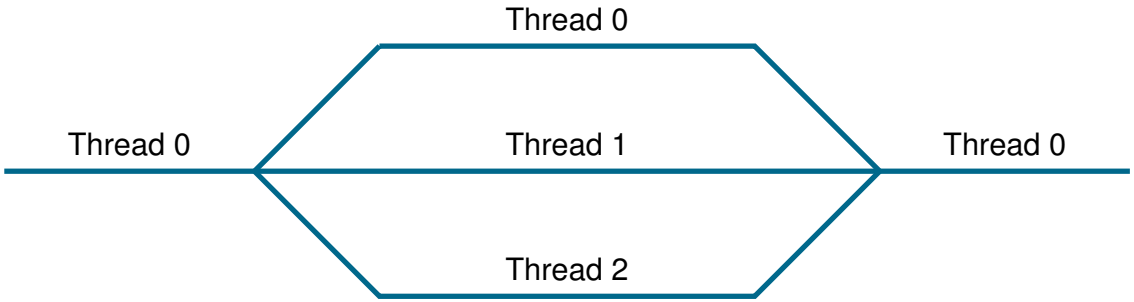
http://www.openmp.org/specifications/

OpenMP

1. Thread-based parallelism
    - Parallelism through concurrent use of threads
    - Thread of execution: minimum unit of processing which can be scheduled on a system
    - Thread exist with the resource of a process
    - Number of executing threads is defined by the user or the application

2. Explicit parallelism:
    - Explicit, non-automatic programming model offering full control on parallelism
    - Does not require an external library but compiler support
    - Expressivity through compiler directives but also more complexe program structures through a runtime.

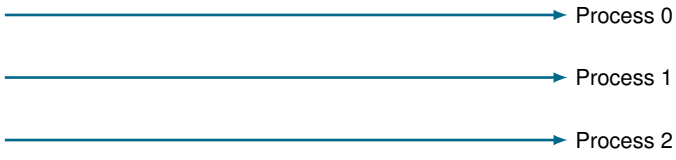
OpenMP: Program flow

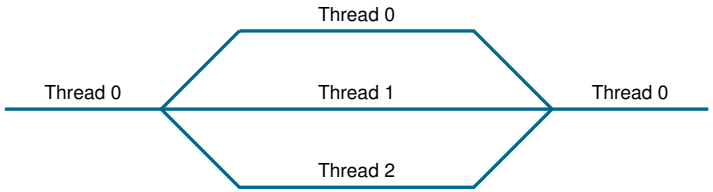


Fork/Join model:

- The main program flow happens on one processor.
- One master thread executes sequentially until a parallel section is encountered.
- Fork: master thread creates a **team** of parallel threads executing the task enclosed in the parallel section.
- Parallel execution: each thread contributes to processing,
- Join: when the parallel section completes, spawned threads synchronize and terminate: only the master threads remains.
- No explicit data division is performed.

Comparison MPI vs. OpenMP





1. Parallel execution of programs
   vs. Concurrent execution of threads from a single execution thread.

2. Each MPI process as *private data* in its address space
   vs. All OpenMP threads may share data.

Pitfalls: reading/writing the same memory location concurrently
$\rightarrow$ separate working buffers, control of resource access.

OpenMP characteristics

— **Expressivity.**
Compiler-based directives #pragma processed by the compiler and embedded in the source code

— **Nested parallelism.**
Placing parallel regions within parallel sections

— **Dynamic Threads.**
Runtime may change the number of threads needed to parallel sections

— **I/O**
OpenMP does not provide such specification

— **Memory management.**
A model of *relaxed consistency*: threads can cache the data and not enforce consistency with main memory.

Operations suitable for OpenMP

- Moderately large computation intensive operations. Since thread-dispatchment always has some costs attached, spawning multiple threads easily costs more time than what you gain by doing the calculations in parallel if the operations are too small.
- Decoupled problems. Since the threads are sharing resources, we have to protect resources using mutual exclusion locks if several threads need to read/write to the same resources concurrently. This adds code complexity and in most cases severe loss of performance. OpenMP shines when you are able to decouple problems in large, independent chunks.

OpenMP implementations

Compilers provide support for OpenMP:

OpenMP	2.5	3.0	3.1	4.0	4.5
GCC	4.2	4.7	4.7	4.9	6.1
Intel		11 .0	12.1	16.0	
Clang			3.7		
Solaris Studio		12.1	12.3	12.4	

Source code generation translator exists: ROSE (compiler framework) Lawrence Livermore

Auto-parallelization can be provided by the compiler to distribute loops independently of OpenMP directives (`-xautopar` in Solaris Studio, `-ftree-parallelize-loops` in GCC), do not mix them!

OpenMP implementations

The specification defines:

1. Compiler Directives.
2. Runtime Library Routines.
3. Environment Variables.

OpenMP “hello world”.

```
#include <omp.h>

#pragma omp parallel
{
   int thread = omp_get_thread_num();
   printf("Hello world from thread number %d\n",
          thread);
}
```

The code inside the block is executed by each participating thread.

Compiler Directives

Processed if flag passed to the compiler, and ignored otherwise.

```
sentinel directive-name [clause,...]
```

Applies to the next structured block of code or OpenMP construct to:

- Create a parallel section
- Distribute loops across threads
- Divide blocks of code between threads
- Serialize a section of code
- Synchronize the work
- Offload data to a SIMD device (OpenMP 4.5)

Runtime Library Routines

Compilation requires including a header and linking to a runtime library:

```
#include <omp.h>
```

Set of functions for use in the code to:

- Set or query the number of threads
- Query thread attributes
- Set or query dynamic thread features
- Query if the section if parallel and the level
- Set or query nested parallelism
- Set, initialize or terminate locks
- Query wall-time and resolution

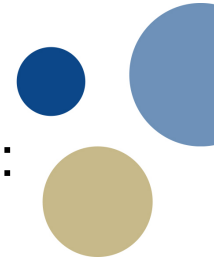
Environment Variables

```
OMP_NUM_THREADS=4 ./exe
```

Specified at execution to:

- Set the number of threads
- Specify how loop iterations are divided
- Bind threads to processors
- Enable or disable nested parallelism, set nesting level
- Set thread stack size
- Set thread wait policy

Directive and clauses



Each directive can be followed by clauses pertaining to its category:

- Parallel construct: `parallel`
- Work-sharing construct: `for`, `sections`, `single`
- Parallel Work-sharing construct: `parallel for`, `task` (OpenMP 3.0)
- Coordination and Synchronization: `master`, `critical`, `barrier`, `taskwait`, `atomic`, `flush`, `ordered`, `threadprivate`

Clauses can be categorized as well:

1. Data sharing clauses
2. Data copying clauses
3. Map clauses (OpenMP 4.0)
4. SIMD clauses (OpenMP 4.0)

Loop distribution: fixed cost

Loop distribution is an instance of Data Parallelism.
$\rightarrow$ scalability w.r.t the data.

```
for (int i = 0; i < 100; i++)
  do_something(i);
```

— We assume that `do_something` does not depend on any global resources: each call is independent of any other call.

— To split this loop among several threads, you can simply do

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < 100; i++)
  do_something(i);
```

Loop distribution: fixed cost

```
#pragma omp parallel for schedule(static)
```

The pragma has three parts:

- `#pragma omp`: all OpenMP directives start with this.
- `parallel for`: tells the compiler that the following for-loop should be parallelized.
- `schedule(static)`: tells the compiler to give each thread approximately the same number of iterations up-front.

Loop distribution: varying cost

If each call to `do_something(i)` takes a different amount of time, depending perhaps on `i`, we can use dynamic scheduling:

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < 100; i++)
  do_something(i);
```

This instructs the compiler not to divide the work up-front, but instead leave one thread as a "broker". Initially, each thread is assigned a single iteration. Upon completing this, they ask the broker for more work. The broker keeps handing out work until each iteration has been assigned.

Loop distribution: varying cost

— This works well if there are many iterations, and each of them are suitably expensive.

— If the iterations are cheap, then execution time will be dominated by negotiating with the broker thread, since this can only happen serially.

— We can minimize this problem by specifying a *chunk size*,

```
#pragma omp parallel for schedule(dynamic, 5)
for (int i = 0; i < 100; i++)
  do_something(i);
```

— Now, the broker hands out assignments in chunks of five. Each chunk is now expensive enough that the broker is not overrun.

Loop distribution: varying cost

- By using a fixed chunk size, we may end up in a situation where all the remaining work is assigned to a single thread. (Poor load balancing.)
- OpenMP offers a third scheduling mode to help fix this problem.

```
#pragma omp parallel for schedule(guided, 5)
for (int i = 0; i < 100; i++)
  do_something(i);
```

Loop distribution: scheduling policies

Scheduling policies aim at devising an optimal division of work.

Several policies are available in OpenMP:

Name	Policy
`static`	*a priori* division of work based on loop
`dynamic`	load-balancing of work during runtime by one thread
`dynamic, N`	same as dynamic but with hint about data chunk size
`guided, N`	same as previous but with decrease of chunk size policy
`auto, N`	automatically determined at runtime
`runtime, N`	specified at runtime by environment variable

Chosing one of them depends on the nature of the work (fixed or varying cost) and the granularity of the work (minimum amount of work as compared to overhead).

Task distribution: sections

Task distribution is an instance of Task/Function Parallelism.
$\rightarrow$ scalability w.r.t logical division into subtasks.

Consider the serial snippet

```
do_job_1();
do_job_2();
do_job_3();
```

Again we assume that the jobs are independent and share no resources, but they are different enough in nature that it doesn't make sense to write this in the form of a loop.

Task distribution: sections

This can be parallelized as such.

```
#pragma omp parallel sections
{
  #pragma omp parallel section
  {
    do_job_1();
  }
  #pragma omp parallel section
  {
    do_job_2();
  }
  #pragma omp parallel section
  {
    do_job_3();
  }
}
```

Shared and private variables

As a rule,

- variables that are declared *outside* the parallel construct are shared between threads. Extra care must be taken when writing to these variables, and it should preferably be avoided if at all possible.
- variables that are declared *inside* are private: each thread has its own copy of it. Since a parallel block defines a scope, these variables do not leak out to the surrounding serial scope.

It is possible to override this using `private(...)` and `shared(...)` directives in OpenMP pragmas, where in each case the `...` is a list of variables.

Example: summing a vector

```
double vec[N] = { ... };
double sum = 0.0;

#pragma omp parallel for
for (size_t i = 0; i < N; i++)
  sum += vec[i];
```

Here, `i` is private and `sum`, `vec` and `N` are shared.

We are writing to a shared variable. How do we solve this?

Example: summing a vector

OpenMP supports the *reduction* directive for precisely this kind of situation.

```
double vec[N] = { ... };
double sum = 0.0;

#pragma omp parallel for reduction(+:sum)
for (size_t i = 0; i < N; i++)
  sum += vec[i];
```

Now, each thread gets its private variable `sum`, and after the loop, each of these variables are summed into the shared one.

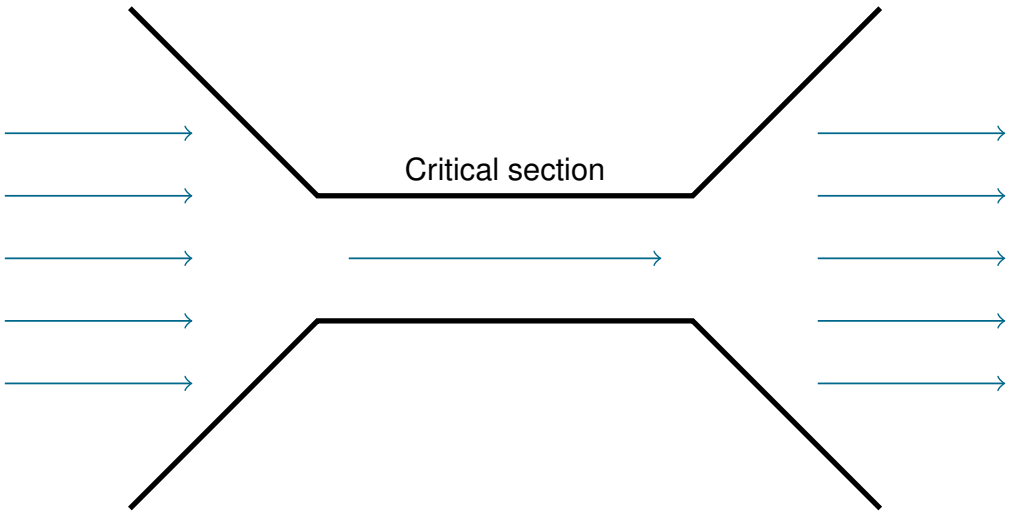
Critical sections

Case: several threads need access to the same resource. $\rightarrow$ allow access to one thread **only**: resource locking.

Critical section is a program section which can only be executed by one thread at any time.

Managing unique acces to a resource can be achieved by a *mutual exclusion* mechanism, also called *mutex*.

Critical sections



A critical section is a part of the code that, for whatever reason, should only be accessed by one thread at a time. Typically this happens whenever you need to read or write from or to shared resources.

Critical sections: explicit locks

You can use mutexes (*mutual exclusion locks*) for this.

```
omp_lock_t my_lock;
omp_init_lock(&my_lock);

#pragma omp parallel for
for (int i = 0; i < 100; i++) {
  do_something(i);

  omp_set_lock(&my_lock);
  // critical section
  omp_unset_lock(&my_lock);
}

omp_destroy_lock(&my_lock);
```

Critical sections: pragmas

OpenMP has built-in language support for critical sections.

```
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
  do_something(i);

  #pragma omp critical
  {
    // critical section
  }
}
```

Critical sections: pragmas

You can even have named critical sections.

```
#pragma omp parallel for
for (int i = 0; i < 100; i++) {
  do_something(i);

  #pragma omp critical updateLog
  {
    // update the log file
  }

  do_something_else(i);

  #pragma omp critical updateLog
  {
    // update the log file
  }
}
```

Pitfall: A critical section suffers then from *serialization*.
$\rightarrow$ Crucial for efficiency, remember Amdahl's law!

How to enable

- With GNU compilers (`gcc`, `g++`, `gfortran`), use the compiler flag `-fopenmp`.
- With Intel compilers (`icc`, `icpc`, `ifort`), use the compiler flag `-openmp`.
- Without this flag, GNU compilers will silently ignore pragmas, while Intel compilers will give a warning. In either case, the resulting binary is equivalent to serial code.
- In CMake you can do this

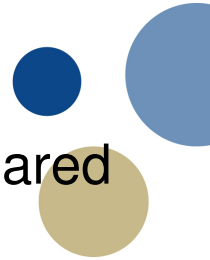
```
find_package(OpenMP)
if(OPENMP_FOUND)
  set(CMAKE_C_FLAGS
      "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}$")
endif()
```

Number of threads

The number of threads spawned is determined in the following order.

1. Programs can call `omp_set_num_threads(int)` to request a number of threads at runtime. This function is defined in `omp.h`.
2. The environment variable `OMP_NUM_THREADS`, if it exists. You can run your programs in most shells as `OMP_NUM_THREADS=8 ./myprogram`.
3. A sensible default value which depends on your system hardware and the data. Typically this will be the number of logical cores.

Conclusions



- OpenMP offers easy exploitation of computing resources on shared memory machines.
- The parallel code is very close to the serial code - it usually only differs by some pragmas which you can tell a compiler to ignore.
- The fork-join programming model often is harder to grasp than the distributed model, in particular if several threads need to access the same resources.
- Best results are usually achieved if you combine OpenMP and MPI. This is also the model that maps the best to modern hardware, where you typically have a few handfuls of CPUs sharing memory, while using more than that requires a distributed memory programming model.

Example code: using OpenMP for the parallel assembly of the linear system in a C++ finite element code. Notice that `for` loops should be made explicit!

More information

The official page for the OpenMP specification can be found at

http://openmp.org including very helpful summary documents!

Some very instructive tutorials can be found at

http://computing.llnl.gov/tutorials/openMP