High Performance Scientific Computing with OpenMP

Ramses van Zon

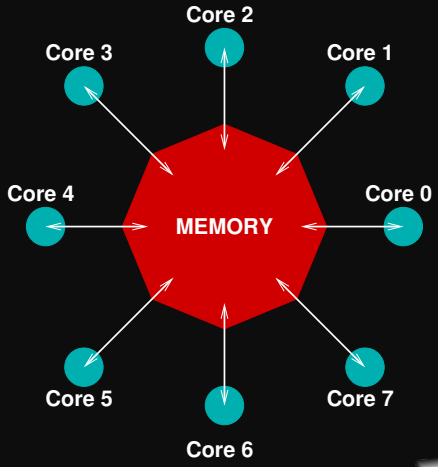
PHY1610 - Winter 2026

Shared Memory Programming

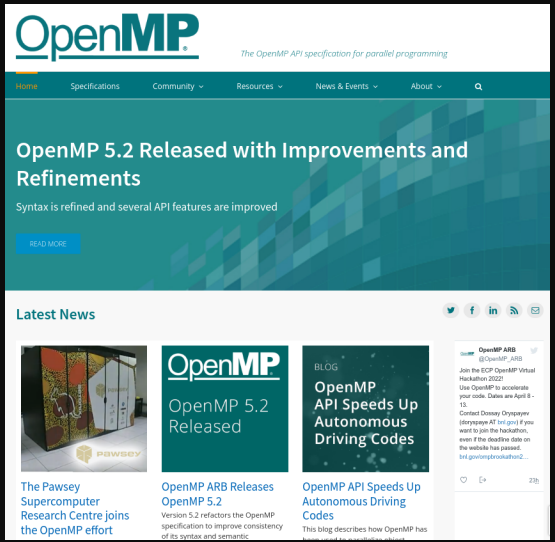
Shared Memory

Remember the paradigm:

- One large blob of memory, different computing cores acting on it. All 'see' the same data.
- Any coordination done through memory.
- Could use message passing, but no need.
- Each code is assigned a thread of execution of a single program that acts on the data.



OpenMP



- For on-node, performant, portable parallel code

  E.g. multi-core, shared memory systems. But also GPU offloading
- Add parallelism to functioning serial code.
- **https://openmp.org**
- Compiler, run-time environment does a lot of work for us (divides up work)
- But we have to tell it what to run in parallel and how to use variables.
- Works by adding compiler directives to C, C++, or Fortran code

OpenMP basic operations

In code:

- In C++, you add lines starting with `#pragma omp`
  This parallelizes the subsequent code block.

When compiling:

- To turn on OpenMP support, add `-fopenmp` to the compilation and link commands.

When running:

- The environment variable `OMP_NUM_THREADS` sets how many threads are to be used in parallel blocks.

```
$ git clone ~lcl_uotphy1610s1466/omp
$ cd omp
$ source setup
$ make omp-hello-world
```

OpenMP example

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

int main()
{
  std::println("At start of program.");

  #pragma omp parallel
  {
     std::println("Hello world from thread {}!",
                  omp_get_thread_num());
  }
}
```

```
$ g++ -std=c++23 -O3 -o omp-hello-world omp-hello-world.cpp -fopenmp
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
```

Output from OpenMP hello world

```
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
At start of program.
Hello world from thread 0!
```

```
$ export OMP_NUM_THREADS=8
$ ./omp-hello-world
At start of program.
Hello world from thread 0!
Hello world from thread 6!
Hello world from thread 3!
Hello world from thread 1!
Hello world from thread 7!
Hello world from thread 4!
Hello world from thread 5!
Hello world from thread 2!
```

What happened precisely?

```
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
At start of program.
Hello world from thread 0!
```

```
$ export OMP_NUM_THREADS=8
$ ./omp-hello-world
At start of program.
Hello world from thread 0!
Hello world from thread 6!
Hello world from thread 3!
Hello world from thread 1!
Hello world from thread 7!
Hello world from thread 4!
Hello world from thread 5!
Hello world from thread 2!
```

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

int main()
{
  std::println("At start of program.");

  #pragma omp parallel
  {
     std::println("Hello world from thread {}!",
                  omp_get_thread_num());
  }
}
```

- Threads were launched.
- Each prints `Hello, world ...`
- In seemingly random order.

Running OpenMP batch jobs on the Teach cluster

Running parallel codes on the login node will quickly cause its cores to be oversubscribed.

Scaling and timing experiments are unreliable on the shared login node.

The Teach cluster has 8 other nodes, each with 40 cores, called the compute nodes.

For short interactive test, get access to compute nodes using `debugjob`, e.g., for 8 cores:

```
debugjob -n 8
```

For larger runs or test, you must submit a jobscript to the scheduler with `sbatch`:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --time=1:00:00
#SBATCH --output=openmp_output_%j.txt
module load gcc/14
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./omp-hello-world
```

OpenMP: Language extension + a library

- `#pragma omp` gives the language extensions
- `#include <omp.h>` give access to library functions, such as

```
int omp_get_num_threads();          // number of threads currently running

int omp_get_thread_num();           // index of the current threads (starts at 0)

void omp_set_num_threads(int n);    // number of threads to be used at the next parallel section

int omp_get_num_procs();            // get maximum number of processors
```

New example

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

int main()
{
  std::println("At start of program.");
  #pragma omp parallel
  {
    std::println("Hello world from thread {}!",
                 omp_get_thread_num());
  }
  std::println("There were {} threads.",
               omp_get_num_threads());
}
```

```
$ make omp-num-threads2
$ export OMP_NUM_THREADS=3
$ ./omp-num-threads2
```

```
At start of program.
Hello world from thread 0!
Hello world from thread 1!
Hello world from thread 2!
There were 1 threads.
```

Strange, says: 'There were 1 threads.'. Why?

Because that is true outside the parallel region!

Variables to the rescue!

- omp_get_num_threads only returns the number of threads inside current region.
- Let's try to store the result of omp_get_num_threads to a variable.

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

int main()
{
  int nthreads, t;
  #pragma omp parallel default(none) \
              shared(nthreads) private(t)
  {
    t = omp_get_thread_num();
    if (t == 0)
      nthreads = omp_get_num_threads();
  }
  std::print("There were {} threads.", nthreads);
}
```

```
$ make omp-num-threads3
$ export OMP_NUM_THREADS=3
$ ./omp-num-threads3
```

```
There were 3 threads.
```

- What are these extra shared and private clauses?

Shared and Private Variables

Shared Variables

- A variable designated as `shared` can be accessed by all threads.
- For reading variable values, this is very convenient.
- For assigning to variables, this introduces potential **race conditions**.

Private Variables

- If a variable is designated as `private`, each thread gets its own separate version of the variable.
- Different threads cannot see other threads' versions.
- Thread-private versions do not have the value of the variable outside the parallel loop.
- The thread-private versions cease to exists after the parallel region.

default(none)

If a variable is not designated as either `shared` or `private`, the compiler chooses.

- That may seem like a nice feature, but try not to rely on this!
- With `default(none)`, compilation fails if undesignated variables are used in parallel regions.
- This is a good thing; it tells you that you have not thought about the role of your variables inside the parallel region.

So what happened now?

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

int main()
{
  int nthreads, t;
  #pragma omp parallel default(none) \
              shared(nthreads) private(t)
  {
    t = omp_get_thread_num();
    if (t == 0)
      nthreads = omp_get_num_threads();
  }
  std::print("There were {} threads.", nthreads);
}
```

```
$ make omp-num-threads3
$ export OMP_NUM_THREADS=3
$ ./omp-num-threads3
There were 3 threads.
```

- Program runs, lauches threads.
- Each thread gets copy of t.
- Only thread 0 writes to nthreads.

Tip: Declare private variables, such as t, as local variables.

So what happened now?

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

int main()
{
  int nthreads;
  #pragma omp parallel default(none) \
              shared(nthreads)
  {
    int t = omp_get_thread_num();
    if (t == 0)
      nthreads = omp_get_num_threads();
  }
  std::print("There were {} threads.", nthreads);
}
```

```
$ make omp-num-threads4
$ export OMP_NUM_THREADS=4
$ ./omp-num-threads4
There were 3 threads.
```

- Program runs, lauches threads.
- Each thread gets copy of t.
- Only thread 0 writes to nthreads.

Tip: Declare private variables, such as t, as local variables.

Single Execution

- We do not care which thread sets `nthreads`.
- Might as well be the first thread that gets to it.
- OpenMP has a construct for this:

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

int main()
{
  int nthreads;
  #pragma omp parallel default(none) shared(nthreads)
  {
    #pragma omp single
    {
      nthreads = omp_get_num_threads();
    }
  }
  std::println("There were {} threads.", nthreads);
}
```

```
$ make omp-num-threads5
$ export OMP_NUM_THREADS=3
$ ./omp-num-threads5
There were 3 threads.
```

Single Execution

- We do not care which thread sets `nthreads`.
- Might as well be the first thread that gets to it.
- OpenMP has a construct for this:

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

int main()
{
  int nthreads;
  #pragma omp parallel default(none) shared(nthreads)
  #pragma omp single
  nthreads = omp_get_num_threads();
  std::println("There were {} threads.", nthreads);
}
```

```
$ make omp-num-threads5
$ export OMP_NUM_THREADS=3
$ ./omp-num-threads5
There were 3 threads.
```

C++ has native threads!

This is true: C++ can spawn threads natively. But this is very, very verbose and manual:

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

int main()
{
  std::println("At start of program.");

  #pragma omp parallel
  {
     std::println("Hello world from thread {}!",
                  omp_get_thread_num());
  }
}
```

```cpp
#include <thread>
#include <vector>
#include <print>
#include <cstdlib>
#include <string>
int get_num_threads() {
 if (char* env = std::getenv("OMP_NUM_THREADS"))
  return std::stoi(env);
 else
  return std::thread::hardware_concurrency();
}
int main() {
 std::println("At start of program.");
 int nthreads = get_num_threads();
 std::vector<std::thread> threads;
 for (int i = 0; i < nthreads; ++i)
  threads.emplace_back([i]() {
   std::println("Hello world from thread {}!",i);
   });
 for (auto& t : threads)
  t.join();
}
```

Loops


SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Loops in OpenMP

Lots of loops in scientific code. Let's add a senseless loop:

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

int main()
{
  #pragma omp parallel default(none)
  {
    int mythread = omp_get_thread_num();
    for (int i=0; i<16; i++) {
      std::println("Thread {} gets i={}", mythread, i);
    }
 }
}
```

What would you expect this to do with e.g. 2 threads?

This is what it does:

```
$ make omp-loop1
$ export OMP_NUM_THREADS=2
$ ./omp-loop1
Thread 0 gets i=0
Thread 0 gets i=1
Thread 0 gets i=2
Thread 1 gets i=0
Thread 0 gets i=3
Thread 1 gets i=1
Thread 0 gets i=4
Thread 1 gets i=2
Thread 0 gets i=5
Thread 1 gets i=3
Thread 0 gets i=6
Thread 1 gets i=4
Thread 0 gets i=7
Thread 1 gets i=5
Thread 0 gets i=8
Thread 1 gets i=6
Thread 0 gets i=9
Thread 1 gets i=7
Thread 0 gets i=10
Thread 1 gets i=8
Thread 0 gets i=11
```

- Every thread executes all 16 cases!
- Probably not what we want.

Worksharing in OpenMP

- We don't generally want tasks to do exactly the same thing.
- Want to divide a problem into pieces that threads works on.
- OpenMP has a worksharing construct: omp for.

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

int main() {
  #pragma omp parallel default(none)
  {
    int mythread = omp_get_thread_num();
    #pragma omp for
    for (int i=0; i<16; i++) {
        std::println("Thread {} get i={}", mythread, i);
    }
  }
}
```

Worksharing constructs in OpenMP

- `omp for` construct breaks up the iterations by thread.
- If doesn't divide evenly, does the best it can.
- Allows easy breaking up of work!
- Code need not know how many threads there are; OpenMP does the work division for you.

```
$ make omp_loop2
$ export OMP_NUM_THREADS=2
$ ./omp_loop2
Thread 0 gets i=0
Thread 0 gets i=1
Thread 0 gets i=2
Thread 1 gets i=8
Thread 0 gets i=3
Thread 1 gets i=9
Thread 0 gets i=4
Thread 0 gets i=5
Thread 0 gets i=6
Thread 0 gets i=7
Thread 1 gets i=10
Thread 1 gets i=11
Thread 1 gets i=12
Thread 1 gets i=13
Thread 1 gets i=14
Thread 1 gets i=15
```

Less trivial example: DAXPY

```
#include <rarray>
#include "ticktock.h"

void init(rvector<double>& x, rvector<double>& y, rvector<double>& z);

void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z);

int main()
{
   int n = 10'000'1000;
   rvector<double> x(n), y(n), z(n);
   double a = 5./3.;
   TickTock tt;
   tt.tick();
   init(x,y,z);
   mydaxpy(a,x,y,z);
   tt.tock("Tock registers");
}
```

DAXPY - Function definitions

```
#include <algorithm>

// Initialize arrays x and y with i^2 and i^2-1, respectively
void init(rvector<double>&x,rvector<double>&y,rvector<double>&z){
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   for (int i=0; i<n; i++) {
      x[i] = double(i)*double(i);
      y[i] = double(i+1)*double(i-1);
      z[i] = 0.0;
   }
}
```

```
// Add a*x+y to z.  x, y, and z are arrays and a is a scalar.
void mydaxpy(double a, const rvector<double>& x,
          const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   for (int i=0; i<n; i++)
      z[i] += a * x[i] + y[i];
}
```

How would you OpenMP-parallelize these?

Parallelizing the loops

```
#include <algorithm>

// Initialize arrays x and y with i^2 and i^2-1, respectively
void init(rvector<double>&x,rvector<double>&y,rvector<double>&z){
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   for (int i=0; i<n; i++) {
      x[i] = double(i)*double(i);
      y[i] = double(i+1)*double(i-1);
      z[i] = 0.0;
   }
}
```

```
// Add a*x+y to z.  x, y, and z are arrays and a is a scalar.
void mydaxpy(double a, const rvector<double>& x,
           const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   for (int i=0; i<n; i++)
      z[i] += a * x[i] + y[i];
}
```

Things to consider when parallelizing:

- Where is the concurrency?

  I.e. what loops have independent iterations, so they may be done in parallel?

- If we divide the work over threads, which variables do the threads need to know about?

- Which ones are `shared`, which ones are to be `private`?

Parallel DAXPY

```
void init(rvector<double>& x, rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel default(none) shared(x,y,z,n)
   {
      #pragma omp for
      for (int i=0; i<n; i++) {
         x[i] = double(i)*double(i);
         y[i] = double(i+1)*double(i-1);
         z[i] = 0.0;
      }
  }
```

```
void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel default(none) shared(x,y,a,z,n)
   {
      #pragma omp for
      for (int i=0; i<n; i++)
         z[i] += a * x[i] + y[i];
   }
}
```

For your convenience

short-hand/combined pragmas

`#pragma omp parallel`

and

`#pragma omp for`

may be combined to

`#pragma omp parallel for`

Also note that instead of a code block with curly braces, a single line or a single loop with a single lines can be a parallel region.

Parallel DAXPY, simplifications

```
void init(rvector<double>& x, rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel default(none) shared(n,x,y)
   {
      #pragma omp for
      for (int i=0; i<n; i++) {
         x[i] = double(i)*double(i);
         y[i] = double(i+1)*double(i-1);
         z[i] = 0.0;
      }
  }
}
void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel default(none) shared(n,x,y,a,z)
   {
      #pragma omp for
      for (int i=0; i<n; i++)
         z[i] += a * x[i] + y[i];
   }
}
```

Parallel DAXPY, simplifications

```
void init(rvector<double>& x, rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel for default(none) shared(n,x,y)


      for (int i=0; i<n; i++) {
         x[i] = double(i)*double(i);
         y[i] = double(i+1)*double(i-1);
         z[i] = 0.0;
      }

}
void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel for default(none) shared(n,x,y,a,z)


      for (int i=0; i<n; i++)
         z[i] += a * x[i] + y[i];

}
```

Parallel DAXPY, simplifications

```
void init(rvector<double>& x, rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel for default(none) shared(n,x,y)
   for (int i=0; i<n; i++) {
         x[i] = double(i)*double(i);
         y[i] = double(i+1)*double(i-1);
         z[i] = 0.0;
   }
}
void mydaxpy(double a, const rvector<double>& x,
             const rvector<double>& y, rvector<double>& z) {
   int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel for default(none) shared(n,x,y,a,z)
   for (int i=0; i<n; i++)
         z[i] += a * x[i] + y[i];
}
```

Looks just like the serial code, except or the `#pragma omp`.

"Incremental parallelism!"

Parallel DAXPY performance

```
$ debugjob -n 16  # if on the Teach cluster
$ module load gcc/14 rarray
$ make daxpy
$ ./daxpy
Tock registers 0.2353 sec
$ make daxpy-parallel
$ export OMP_NUM_THREADS=16
$ ./daxpy-parallel
Tock registers 0.03174 sec
```

7.4 times faster!

Submitting OpenMP jobs to the scheduler

- OpenMP uses shared memory, so you need to stay on one node.
- The application is a single process, so one task.
- That application needs multiple CPUs for its threads.
- But that application still needs to be told how many threads openmp should use.
- You probably want to know how long it took.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=16
#SBATCH --time=1:00:00
#SBATCH --output=openmp_output_%j.txt
#SBATCH --mail-type=FAIL

module load gcc/14 rarray

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

time ./daxpy-parallel
```