PHY1610 - High Performance Scientific Computing with OpenMP

Ramses van Zon, Marcelo Ponce

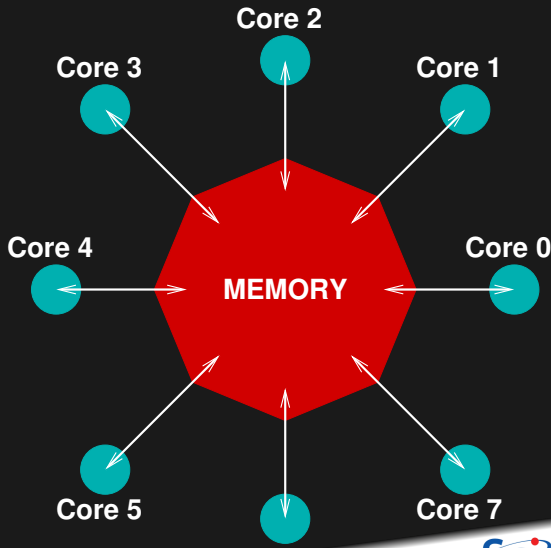
March 23, 2021

Section 1

Shared Memory Programming

Shared Memory

- 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.
- 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 how to use variables, where to run in parallel, ...
- 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 in g++, add the `-fopenmp` flag to the compilation and link commands.

When running

- The environment variable `OMP_NUM_THREADS` determines how many threads will be started in an OpenMP parallel block.

```
$ cd $SCRATCH
$ git clone /scinet/course/phy1610/omp
$ cd omp
$ source setup
$ make omp-hello-world
```

OpenMP example

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  std::cout << "At start of program\n";
  #pragma omp parallel
  {
     std::cout << "Hello world from thread "
       + std::to_string(omp_get_thread_num()) + "!\n";
  }
}
```

OpenMP example

```cpp
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  std::cout << "At start of program\n";
  #pragma omp parallel
  {
     std::cout << "Hello world from thread "
       + std::to_string(omp_get_thread_num()) + "!\n";
  }
}
```

```
$ g++ -std=c++14 -O2 -o omp-hello-world omp-hello-world.cc -fopenmp
$ #(or make omp-hello-world)
$ 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!
```

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 <iostream>
#include <omp.h>
#include <string>
int main() {
  std::cout << "At start of program\n";
  #pragma omp parallel
  {
     std::cout << "Hello world from thread "
      +std::to_string(omp_get_thread_num())+"!\n";
  }
}
```

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 <iostream>
#include <omp.h>
#include <string>
int main() {
  std::cout << "At start of program\n";
  #pragma omp parallel
  {
     std::cout << "Hello world from thread "
      +std::to_string(omp_get_thread_num())+"!\n";
  }
}
```

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

Running OpenMP batch jobs on the Teach cluster

If we all run our parallel codes on the Teach login node ("teach01"), we'll quickly slow down the node, as all cores because oversubscribed.

Scaling experiments (i.e., seeing how the runtime varies with the number of cores) are unreliable on the shared login node.

The Teach cluster has 40 other nodes, each with 16 cores, called the compute nodes.

For short interactive test, you can get access to compute nodes with the debugjob command, e.g., for 4 cores, do

```
debugjob -n 4
```

For larger runs or test, you must submit a jobscript to the scheduler.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=16
#SBATCH --time=1:00:00
#SBATCH --job-name openmp_job
#SBATCH --output=openmp_output_%j.txt
#SBATCH --mail-type=FAIL
module load gcc
OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./openmp_example
```

OpenMP: Language extension + a library

- `#pragma omp` give 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

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  std::cout << "At start of program\n";
  #pragma omp parallel
  std::cout << "Hello world from thread "
    + std::to_string(omp_get_thread_num()) + "!\n";
  std::cout << "There were "
    + std::to_string(omp_get_num_threads()) + " threads.\n";
}
```

New example

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  std::cout << "At start of program\n";
  #pragma omp parallel
  std::cout << "Hello world from thread "
    + std::to_string(omp_get_thread_num()) + "!\n";
  std::cout << "There were "
    + std::to_string(omp_get_num_threads()) + " threads.\n";
}
```

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

New example

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  std::cout << "At start of program\n";
  #pragma omp parallel
  std::cout << "Hello world from thread "
    + std::to_string(omp_get_thread_num()) + "!\n";
  std::cout << "There were "
    + std::to_string(omp_get_num_threads()) + " threads.\n";
}
```

```
$ 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.
```

New example

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  std::cout << "At start of program\n";
  #pragma omp parallel
  std::cout << "Hello world from thread "
    + std::to_string(omp_get_thread_num()) + "!\n";
  std::cout << "There were "
    + std::to_string(omp_get_num_threads()) + " threads.\n";
}
```

```
$ 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?

New example

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
  std::cout << "At start of program\n";
  #pragma omp parallel
  std::cout << "Hello world from thread "
    + std::to_string(omp_get_thread_num()) + "!\n";
  std::cout << "There were "
    + std::to_string(omp_get_num_threads()) + " threads.\n";
}
```

```
$ 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 in a parallel region inside said region.
- Let's try to store the result of `omp_get_num_threads` to a variable then.

Variables to the rescue!

- `omp_get_num_threads` only returns the number of threads in a parallel region inside said region.
- Let's try to store the result of `omp_get_num_threads` to a variable then.

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

Variables to the rescue!

- `omp_get_num_threads` only returns the number of threads in a parallel region inside said region.
- Let's try to store the result of `omp_get_num_threads` to a variable then.

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

- What are these extra clauses?
  - `shared`: read/write access to the variable for each thread
  - `private`: separate instance of the variable for each thread

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**.

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.

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.

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.

What happened?

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

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

```
#include <iostream>
#include <omp.h>
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::cout<<"There were "<<nthreads<<" threads.\n";
}
```

What happened?

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

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

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

```
#include <iostream>
#include <omp.h>
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::cout<<"There were "<<nthreads<<" threads.\n";
}
```

What happened?

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

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

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

```
#include <iostream>
#include <omp.h>
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::cout<<"There were "<<nthreads<<" threads.\n";
}
```

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:

```
#include <iostream>
#include <omp.h>
int main()
{
    int nthreads;
    #pragma omp parallel default(none) shared(nthreads)
    #pragma omp single
    nthreads = omp_get_num_threads();
    std::cout << "There were " << nthreads << " threads.\n";
}
```

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

Section 2

Loops

Loops in OpenMP

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

```
#include <iostream>
#include <omp.h>
#include <string>
int main() {
 int i, t;
 #pragma omp parallel default(none) private(i,t)\
                      shared(std::cout)
 {
  t = omp_get_thread_num();
  for (i=0; i<16; i++)
   std::cout << "Thread " + std::to_string(t)
             + " gets i=" + std::to_string(i) + "\n";
 }
}
```

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
Thread 1 gets i=9
Thread 0 gets i=12
Thread 1 gets i=10
Thread 0 gets i=13
Thread 1 gets i=11
```

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
Thread 1 gets i=9
Thread 0 gets i=12
Thread 1 gets i=10
Thread 0 gets i=13
Thread 1 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.

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.

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 <iostream>
#include <omp.h>
#include <string>
int main() {
  #pragma omp parallel default(none) shared(std::cout)
  {
   int t = omp_get_thread_num();
   #pragma omp for
   for (int i=0; i<16; i++)
     std::cout << "Thread " + std::to_string(t)
               + " gets i=" + std::to_string(i) + "\n";
  }
}
```

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(rarray<double,1>& x, rarray<double,1>& y);

void mydaxpy(double a, const rarray<double,1>& x,
             const rarray<double,1>& y, rarray<double,1>& z);

int main()
{
   int n = 10*1000*1000;
   rarray<double,1> x(n), y(n), z(n);
   double a = 5./3.;
   TickTock tt;
   tt.tick();
   init(x,y);
   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(rarray<double,1>& x, rarray<double,1>& y) {
   int n = std::min(x.size(), y.size());
   for (int i=0; i<n; i++) {
      x[i] = double(i)*double(i);
      y[i] = double(i+1)*double(i-1);
   }
}
// Add a*x+y to z.  x, y, and z are arrays and a is a scalar.
void mydaxpy(double a, const rarray<double,1>& x,
             const rarray<double,1>& y, rarray<double,1>& 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 this?

Parallelizing the loops

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(rarray<double,1>& x, rarray<double,1>& y) {
   int n = std::min(x.size(), y.size());
   #pragma omp parallel default(none) shared(x,y,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);
      }
  }
}
void mydaxpy(double a, const rarray<double,1>& x,
             const rarray<double,1>& y, rarray<double,1>& 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

- Constants are forced to be automatically shared
- `#pragma omp parallel` and `#pragma omp for` may be combined to

  `#pragma omp parallel for`

Parallel DAXPY, simplifications

```
void init(rarray<double,1>& x, rarray<double,1>& y) {
   int n = std::min(x.size(), y.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);
      }
  }
}
void mydaxpy(double a, const rarray<double,1>& x,
             const rarray<double,1>& y, rarray<double,1>& 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(rarray<double,1>& x, rarray<double,1>& y) {
   const int n = std::min(x.size(), y.size());
   #pragma omp parallel default(none) shared(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);
      }
  }
}
void mydaxpy(double a, const rarray<double,1>& x,
             const rarray<double,1>& y, rarray<double,1>& z);
{
   const int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel default(none) shared(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(rarray<double,1>& x, rarray<double,1>& y) {
   const int n = std::min(x.size(), y.size());
   #pragma omp parallel for default(none) shared(x,y)


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

}
void mydaxpy(double a, const rarray<double,1>& x,
             const rarray<double,1>& y, rarray<double,1>& z);
{
   const int n = std::min(x.size(), std::min(y.size(),z.size()));
   #pragma omp parallel for default(none) shared(x,y,a,z)


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

}
```

Parallel DAXPY, simplifications

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

Parallel DAXPY, simplifications

```
$ make mydaxpy
$ ./mydaxpy
Tock registers 0.3936 sec
$ make mydaxpy-parallel
$ export OMP_NUM_THREADS=16
$ ./mydaxpy-parallel
Tock registers 0.07156 sec
```

5.5 times faster!

Getting reliable timing: get your own cores!

To get reliable timings, on the teach01 node, first grab a compute node for your self, i.e, do:

```
$ debugjob -n 8   # from teach.scinet.utoronto.ca i.e. teach01
$ cd $SCRATCH/omp
$ source setup
```

If you leave out `-n 8` you only get one core, so you can't do parallelism.

(In contrast, on Niagara, leaving out `-n 8` gives you a 40 core node)