High Performance Scientific Computing with OpenMP

Ramses van Zon

PHY1610 - Winter 2024

Using HPC clusters

Login and compute nodes

So far, you've been running short computations on the Teach login node.

But most clusters (incl. Teach) consists of at least **two functionality different nodes**.

- Login nodes where you develop, edit, compile, prepare and submit jobs. You share its resources with other students and courses.
- Compute nodes where real computations should run.
- The login nodes and compute nodes have the same architecture, operating system, and software stack.
- To run on the compute nodes, you must submit a batch job.
- When running this way, your job get dedicated resources: no contention, less timing fluctuations.

Storage Systems and Locations on SciNet systems

You have a home and scratch directory on the system, whose locations will be given by

$HOME=/home/g/groupname/username

$SCRATCH=/scratch/g/groupname/username

Use these convenient environment variables!

```
teach01:~$ pwd
/home/s/scinet/rzon
teach01:~$ echo $HOME
/home/s/scinet/rzon
teach01:~$ cd $SCRATCH
teach01:~$ pwd
/scratch/s/scinet/rzon
```

Testing

You should test your code before you submit it to the cluster to know if your code is correct and what kind of resources you need.

- Small test jobs can be run on the login nodes.

  Rule of thumb: couple of minutes, taking at most about 1-2GB of memory, couple of cores.

- Short tests that do not fit on a login node, or for which you need a dedicated node or set of cores, request an interactive debug job with the debugjob command

```
teach01:~$ debugjob -n 2
debugjob: Requesting 1 nodes with 2 tasks for 240 minutes and 0 seconds
SALLOC: Granted job allocation 202753
SALLOC: Waiting for resource configuration
SALLOC: Nodes teach35 are ready for job
teach35:~$ nproc
2
teach35:~$
```

The Scheduler

- The scheduler is a program that organizes the work load on the cluster.
- You submit a request to the scheduler, and it will find the right moment for your request to run.
- It does that by looking at the resources available, your priority, times and resources requested, . . .
- Even when we run interactively, we are requesting resources to the scheduler

Submitting jobs

- Teach (as well as all other SciNet and Alliance systems) uses SLURM as its job scheduler.
- You submit jobs from a login node by passing a job script to the sbatch command:

```
teach01:~$ cd $SCRATCH
teach01:scratch$ sbatch jobscript.sh
```

- This puts the job in the queue. It will run on the compute nodes in due course.

Restrictions

You cannot submit arbitrary jobs:

- Maximum walltime is 24 hours.
- home is read-only on compute nodes:
  Jobs must write to your scratch directory
- Compute nodes have no internet access:
  Download data you need beforehand on a login node.
- Different clusters have different restrictions.

Example submission script

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

module load gcc/13

./serial_code 1
```

```
teach01:scratch$ sbatch serial_job.sh
```

- First line indicates that this is a bash script.
- Lines starting with `#SBATCH` go to SLURM.
- sbatch reads these lines as a job request.
- In this case, SLURM looks for one core to be used for one task, for 1 hour.
- Once SLURM finds a node with an unused core, the script is run there in a clean environment.

Monitoring jobs

Once the job is incorporated into the queue, there are some command you can use to monitor its progress.

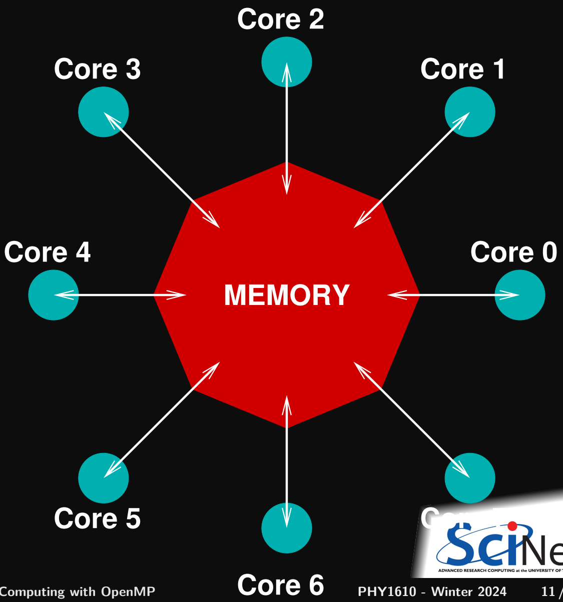
- "`squeue --me`" to show your jobs in the job queue.
- "`squeue --start -j JOBID`" to get an estimate for when a job will run.
- "`jobperf JOBID`" to get an instantaneous view of the cpu+memory usage of a running job's nodes.
- "`scancel JOBID`" to cancel the job.
- "`sacct`" to get information on your recent jobs.

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

The OpenMP API specification for parallel programming

OpenMP 5.2 Released with Improvements and Refinements

Syntax is refined and several API features are improved

Latest News

The Pawsey Supercomputer Research Centre joins the OpenMP effort

OpenMP ARB Releases OpenMP 5.2

Version 5.2 refactors the OpenMP specification to improve consistency of its syntax and semantic

BLOG OpenMP API Speeds Up Autonomous Driving Codes

OpenMP API Speeds Up Autonomous Driving Codes

This blog describes how OpenMP has

OpenMP ARB
@OpenMP_ARE

Join the ECP OpenMP V
Hackathon 2022!
Use OpenMP to acceler
your code. Dates are Ap
13.
Contact Dossay Oryspa
(doryspaye AT bnl.gov) i
want to join the hackath
even if the deadline date
the website has passed.
bnl.gov/ompbrookathon

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.

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

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

OpenMP example

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

```
$ g++ -std=c++17 -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!
```

```
$ 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>
using namespace std;
int main() {
 cout << "At start of program\n";
 #pragma omp parallel
 {
   cout << "Hello world from thread "
     + 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

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 40 other nodes, each with 16 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/13
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./openmp_example
```

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

```
#include <iostream>
#include <omp.h>
#include <string>
using namespace std;
int main()
{
  cout << "At start of program\n";

  #pragma omp parallel
  {
    cout << "Hello world from thread "
      + to_string(omp_get_thread_num())
      + "!\n";
  }

  cout << "There were "
    + 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 inside current region.
- Let's try to store the result of omp_get_num_threads to a variable.

```
#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 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.

What happens now?

```cpp
#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";
}
```

```
$ 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.

What happens now?

```
#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";
}
```

```
$ 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.

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.
```

More. . .

There is much more to be said about OpenMP.

Next lecture we also be about OpenMP and we will look at:

- Loops and work-sharing
- Reductions
- Load balancing