Introduction to using OpenMP the Cluster

Cynthia Cornelius

Center for Computational Research
University at Buffalo, SUNY
701 Ellicott St
Buffalo, NY 14203
Phone: 716-881-8959
Office: B1-109
cdc at ccr.buffalo.edu

June 2011
This tutorial is an introduction to using OpenMP on the cluster.

Familiarity with using a cluster and submitting jobs to a scheduler will be helpful.

Refer to the Guidelines to Running Computations on the Cluster.

- Login
- Basic Unix commands
- Submitting and monitoring jobs
OpenMP

- OpenMP (Open Multi-Processing)
- Parallel computing using multi-threads and shared memory.
- OpenMP uses Fork and Join. The master process forks creating multiple threads. These threads (include the master) run in parallel. When the tasks complete, the threads join again. Leaving only the master thread.
- The master thread persists throughout the computation.
- Each thread has a unique id.
OpenMP

- Directives are added to the code. Typically, these instructions make use of existing loops to parallelize the code with threads.
  - A directive for C/C++ code is a pragma.
- Code with OpenMP directives must be compiled with the appropriate flag, otherwise the code will run single threaded.
  - GNU C: -fopenmp
  - Intel C: -openmp
OpenMP

- The scope of variables is important.
  - Some variables should be shared (global), available to all threads.
  - Others should be private to each thread.
    - The default is shared.
- A reduction is an operation that takes a value from each thread and produces a result.
  - Examples of reduction operations are addition, minimum, and maximum.
  - The master thread has the result.
OpenMP

- Basic parallel construct
  - `#pragma omp parallel shared (list) private (list)`

- Parallel construct with for loop
  - `#pragma omp for shared (list) private (list)`

- Parallel construct with reduction
  - `#pragma omp for shared (list) private (list) reduction (operator: list)`
Hello World with OpenMP

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

int main( int argc, char **argv)
{
    int numthreads, myid;
    #pragma omp parallel private(myid)
    {
        myid = omp_get_thread_num();
        if (myid == 0)
        {
            numthreads = omp_get_max_threads();
            printf("total number of threads = %d\n", numthreads);
        }
        printf("hello from thread %d\n", myid);
    }
}
```
Annotated Hello World with OpenMP

```c
#include <stdio.h>

#include <omp.h>

int main(int argc, char **argv)
{
    int numthreads, myid;

    #pragma omp parallel private(myid)
    {
        int numthreads, myid;

        myid = omp_get_thread_num();
    }
```
Annotated Hello World with OpenMP

- Check if thread is Master Thread 0
  ```c
  if (myid == 0)
  {
  }
  ```

- Get total number of threads
  ```c
  numthreads = omp_get_max_threads();
  printf("total number of threads = %d\n", numthreads);
  }
  ```

- All threads print thread id
  ```c
  printf("hello from thread %d\n", myid);
  ```
Compiling Hello World with the GNU gcc compiler.

gcc -o exec -fopenmp code.c

The gnu compilers are in the default path.

[cdc@edge d_openmp]$ which gcc
/usr/bin/gcc

[cdc@edge d_openmp]$ gcc -o hello-omp-gnu -fopenmp hello-omp.c

[cdc@edge d_openmp]$ file hello-omp-gnu
hello-omp-gnu: ELF 64-bit LSB executable, AMD x86-64, version 1 (SYSV), for GNU/Linux 2.6.9, dynamically linked (uses shared libs), for GNU/Linux 2.6.9, not stripped

[cdc@edge d_openmp]$
Compiling Hello World with the Intel icc compiler.

```bash
icc -o exec -openmp code.c
```

Load the module for the Intel compilers.

```bash
[cdc@edge d_openmp]$ module load intel/11.1
[cdc@edge d_openmp]$ which icc
/util/intel/Compiler/11.1/059/bin/intel64/icc
```

Compile with icc

```bash
[cdc@edge d_openmp]$ icc -o hello-omp-intel -openmp hello-omp.c
```
Running Hello World

Hello World runs on all 8 cores of the machine.

```
[cdc@edge d_openmp]$ ./hello-omp-gnu
hello from thread 1
hello from thread 4
hello from thread 5
hello from thread 6
hello from thread 7
total number of threads = 8
hello from thread 0
hello from thread 3
hello from thread 2
[cdc@edge d_openmp]$
```
OpenMP Hello World

- Running Hello World
- Make sure that intel module is loaded.

```bash
[cdc@edge d_openmp]$ ./hello-omp-intel
hello from thread 3
total number of threads = 8
hello from thread 0
hello from thread 2
hello from thread 5
hello from thread 4
hello from thread 6
hello from thread 1
hello from thread 7
[cdc@edge d_openmp]$
```
Controlling processes

- The default number of threads is the number or processors or cores on the machine.
- Set the number of threads in the environment.
  - OMP_NUM_THREAD variable
    - export OMP_NUM_THREAD=3
- Set the number of threads in the code.
  - The omp_set_num_threads function is used in the program.
    - omp_set_num_threads(3)
Example with reduction.

```c
#include <stdio.h>
#include <omp.h>
/* Base on example from Lawrence Livermore National Laboratory webpage */
main () {

int i, limit, myid, chunksize;
float answer, a[80], b[80];
limit = 80;
chunksize = 10;
answer = 0.0;
for (i=0; i < limit; i++)
{
    a[i] = i * 1.0;
    b[i] = a[i] + 1;
}
```
Example with reduction.

```
#pragma omp parallel for default(shared) private(i,myid) \
schedule(static,chunksize) reduction(+:answer)
for (i=0; i < limit; i++) {
    answer = answer + (a[i] * b[i]);
    myid = omp_get_thread_num();
    /* printf("thread = %d index = %d answer from this thread = %f\n", myid, i, answer); */
}
printf("Answer = %f\n",answer);
```

Running the code

```
[cdc@edge d_openmp]$ ./for-test
Answer = 170640.000000
[cdc@edge d_openmp]$ 
```
Annotated example with reduction.

```c
#include <stdio.h>

#include <omp.h>
/* Base on example from Lawrence Livermore National Laboratory webpage */

main () {

int i, limit, myid, chunksize;
float answer, a[80], b[80];
limit = 80;

Define the size of work for each thread
chunksize = 10;
answer = 0.0;

```
Parallel For with Reduction

- Annotated example with reduction.
  ```c
  for (i=0; i < limit; i++)
  {
    a[i] = i * 1.0;
    b[i] = a[i] + 1;
  }
  ```

- OpenMP Pragma to create threads
  - The index i and thread id myid are private to each thread
  - Each thread will work on a chucksize of the problem
  - The result from each thread is added to the final answer

```
#pragma omp parallel for default(shared) private(i,myid) \
schedule(static,chunksize) reduction(+:answer)
```
Parallel For with Reduction

- Annotated example with reduction.
  ```c
  for (i=0; i < limit; i++) {
      answer = answer + (a[i] * b[i]);
  }
  ``

- Get the unique thread id
  ```c
  myid = omp_get_thread_num();
  ``

- Debug information
  ```c
  /* printf("thread = %d index = %d answer from this thread = %f\n", myid, i, answer); */
  }
  ``

- The resulting answer from a summation of the terms computed by each thread.
  ```c
  printf("Answer = %f\n",answer);
  ```
Timing for OpenMP

- **OMP_GET_WTICK**
  - The number of seconds between clock ticks.
  - Used to determine the precision of the timing measurement.
  - Double precision floating point

- **OMP_GET_WTIME**
  - The number of elapsed seconds.
  - Used the before/after pairs.
  - per thread
  - Double precision floating point
Additional Information

- OpenMP
  - http://openmp.org/wp/
- OpenMP tutorial
  - https://computing.llnl.gov/tutorials/openMP/
- CCR training webpage
  - http://www.ccr.buffalo.edu/display/WEB/Training
- Getting help with using the cluster
  - email ccr-help at ccr.buffalo.edu
    http://www.ccr.buffalo.edu/display/WEB/Getting+Started
    http://www.ccr.buffalo.edu/display/WEB/Training