INTRODUCTION TO OPENMP

Hossein Pourreza

hossein.pourreza@umanitoba.ca

March 26, 2015

Acknowledgement: Examples used in this presentation are courtesy of SciNet.

WestGrid

compute • calcul CANADA

What is High Performance Computing (HPC)

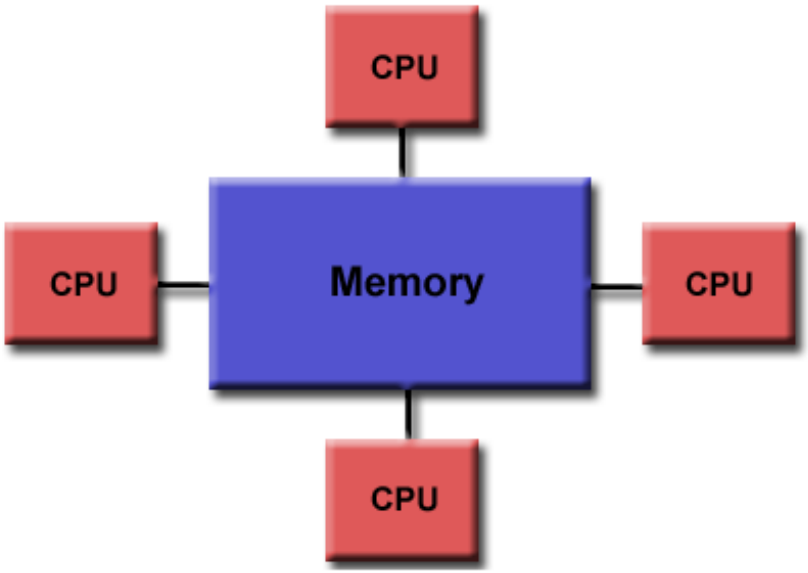
- A way to take advantage of aggregate power of conventional computers to run larger programs

- To run multiple programs or to find parts of a program that can be run concurrently

Why HPC

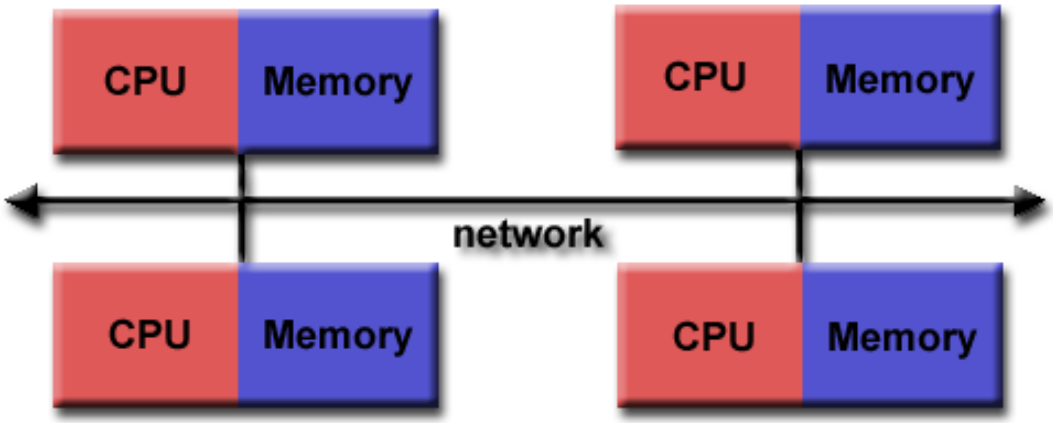
- A single computer has a limited compute power
  - CPU, memory, storage, etc.
- Newer computers add more cores
  - As opposed to previous model of increasing clock speed
  - New opportunities to modify legacy codes to run faster
- New problems and lots of data to be processed

Parallel computer architectures



Shared Memory

Distributed Memory



Parallel programming models

- Multi-threading
  - To be used on shared memory computers
  - Easier to program but not very scalable
  - Different ways to do multi-threading but OpenMP is an industry standard and provides a high level programming interface

- Multi-processing
  - To be used on distributed memory computers
  - Harder to program but scalable
  - Mostly MPI-based

Mini introduction to WestGrid

- You need an account and an SSH client
  - Assuming you have the Compute Canada account
- www.westgrid.ca has lots of information about different systems, installed software, etc.
- Send us email at support@westgrid.ca if you need any help

Logistics of this webinar

- Login to Grex using your WestGrid username/password
  - ssh -Y username@grex.westgrid.ca if you are using Linux or Mac OR
  - Use the SSH client of your choice on Windows

- If you do not see openmp-wg-2015 folder under your home directory, please run the following command:
  - `cp -r /global/scratch/workshop/openmp-wg-2015 .`
- Once you are in the openmp-wg-2015 folder, run the following command to access a compute node :
  - `sh omp_node.sh`

- Please use editors such as vim, nano, etc. to edit the files
  - Please refrain from transferring the files to Windows, editing, and transferring them back to Grex

OpenMP

- A library (de facto standard) to divide up work in your program and add parallelism to a serial code
  - http://www.openmp.org
  - Consists of compiler directives, a runtime library, and environment variables
- It is supported by major compilers such as Intel and GCC
- You can use within C, C++, and FORTRAN programs

Basics

- In C/C++ all OpenMP directives start with `#pragma omp`
  - These lines will be skipped by non-OpenMP compilers
    - You may get a warning message but it should be OK

```
#pragma omp parallel
{
   …
}
```

- In Fortran all OpenMP directives start with `!$OMP`

```
!$OMP PARALLEL
…
!$OMP END PARALLEL
```

Basics (Cont'd)

- When compiling a code with OpenMP directives, you add `-fopenmp` flag to C, C++, or Fortran compilers
  - The preferred flag for the Intel compilers is `-openmp` but they also accept `-fopenmp`

- The `OMP_NUM_THREADS` environment variable determines how many threads will be started
  - If not defined, OpenMP will spawn one thread per hardware thread
  - `export OMP_NUM_THREADS=value (in bash)`
  - `setenv OMP_NUM_THREADS value (in tcsh)`

OpenMP example

- If you are not in the openmp-wg-2015 folder, please change your directory using the following command:
  - `cd /home/$USER/openmp-wg-2015`

- Once you are there, you can run the commands in the following slides and see the results for yourself

OpenMP example: omp_helloworld.c

```
#include <stdio.h>
#include <omp.h>
int main(){
    printf("At the start of program\n");
    #pragma omp parallel
    {
      printf("Hello from thread %d!\n",
             omp_get_thread_num());
    }
    return 0;
}
```

OpenMP example: omp_helloworld.f90

```
program hello
  use omp_lib
  write(*,"(A)") "At the start of program.”
!$omp parallel
  write(*,"(A,I3,A)") "Hello from thread ", &
      omp_get_thread_num(),"!”
!$omp end parallel
end program hello
```

OpenMP examples

- Compiling the code:

```
$ gcc -fopenmp omp_helloworld.c -o omp_helloworld
$ icc -fopenmp omp_helloworld.c -o omp_helloworld
$ gfortran -fopenmp omp_helloworld.f90 -o
  omp_helloworld
```

- Running the code:

```
$ export OMP_NUM_THREAD=4
$ ./omp_helloworld
…
$ export OMP_NUM_THREAD=1
$ ./omp_helloworld
```

OpenMP examples

- Output for `OMP_NUM_THREADS=4`:

```
At the start of program
Hello from thread 0!
Hello from thread 2!
Hello from thread 1!
Hello from thread 3!
```

OpenMP example: Under the hood

```
#include <stdio.h>
#include <omp.h>
int main(){
    printf("At the start of program\n");
    #pragma omp parallel
    {
       printf("Hello from thread %d!\n",omp_get_thread_num());
    }
    return 0;
}
```

Program starts normally with one thread

OpenMP example: Under the hood

```
#include <stdio.h>
#include <omp.h>
int main(){
    printf("At the start of program\n");
    #pragma omp parallel
    {
       printf("Hello from thread %d!\n",omp_get_thread_num());
    }
    return 0;
}
```

OMP_NUM_THREADS threads will be launched and execute this line

OpenMP example: Under the hood

```
#include <stdio.h>
#include <omp.h>
int main(){
    printf("At the start of program\n");
    #pragma omp parallel
    {
       printf("Hello from thread %d!\n",omp_get_thread_num());
    }
    return 0;
}
```

A function from omp.h to find the number/rank of current thread (starting from 0)

OpenMP example: Under the hood

```
#include <stdio.h>
#include <omp.h>
int main(){
    printf("At the start of program\n");
    #pragma omp parallel
    {
       printf("Hello from thread %d!\n",omp_get_thread_num());
    }
    return 0;
}
```

Threads join at the end of parallel section and execution continues serially

OpenMP example: Variables

```
#include <stdio.h>
#include <omp.h>
int main(){
    printf("At the start of program\n");
    #pragma omp parallel
    {
      printf("Hello from thread %d of %d!\n",
                  omp_get_thread_num());
    }
    printf("There were %d threads.\n",
      omp_get_num_threads());
    return 0;
}
```

OpenMP example: Variables

- Running the code in the previous slide prints 1 as the number of threads
- It is correct but not what we want!!
- We should call `omp_get_num_threads()` within the parallel region

OpenMP example: Variables

```
//omp_helloworld_variable.c
#include <stdio.h>
#include <omp.h>
int main(){
  int nthreads = 0;
  printf("At the start of program\n");
  #pragma omp parallel default(none) shared(nthreads)
  {
    int my_thread = omp_get_thread_num();
    printf("Hello from thread %d!\n",my_thread);
    if(my_thread == 0)
      nthreads = omp_get_num_threads();
  }
  printf("There were %d threads.\n",nthreads);
  return 0;
}
```

OpenMP example: Variables

```
!omp_helloworld_variable.f90
program hello
   use omp_lib
   integer :: nthreads = 0,my_thread = 0
   write(*,"(A)") "At the start of program."
!$omp parallel default(none) share(nthreads) private(my_thread)
   my_thread = omp_get_thread_num()
   write(*,"(A,I4,A)") "Hello from thread",my_thread, "!"
   if (my_thread == 0) then
      nthreads = omp_get_num_threads()
   end if
!$omp end parallel
   write(*,"(A,I4,A)") "There were",nthreads," threads."
end program hello
```

OpenMP example: Variables

- `default(none)` can save you hours of debugging
- `shared` means each thread can see and modify it
- `private` means each thread will have its own copy
- We define `my_thread` locally instead of making it `private`

OpenMP example: Variables

```
#include <stdio.h>
#include <omp.h>
int main(){
    int nthreads;
    printf("At the start of program\n");
    #pragma omp parallel default(none) shared(nthreads)
    {
        int my_thread = omp_get_thread_num();
        printf("Hello from thread %d!\n",my_thread);
        if(my_thread == 0)
          nthreads = omp_get_num_threads();
    }
    printf("There were %d threads.\n",nthreads);
    return 0;
}
```

We do not care which thread updates the `nthreads` variable

OpenMP example: Variables

```
//omp_helloworld_single.c or f90
#include <stdio.h>
#include <omp.h>
int main(){
    int nthreads;
    printf("At the start of program\n");
    #pragma omp parallel default(none) shared(nthreads)
    {
        int my_thread = omp_get_thread_num();
        printf("Hello from thread %d!\n",my_thread);
        #pragma omp single
        nthreads = omp_get_num_threads();
    }
    printf("There were %d threads.\n",nthreads);
    return 0;
}
```

Only one thread will execute. Do not care which one!!

Loops in OpenMP

- Most of the scientific codes have a few loops where bulk of the computation happens there

- OpenMP has specific directives:
  - C/C++
    - `#pragma omp parallel for`
    - `#pragma omp for`
  - Fortran
    - `!$OMP PARALLEL DO … !OMP END PARALLEL DO`
    - `!$OMP DO … !OMP END DO`

Loops in OpenMP

```
//omp_loop.c or f90
#include <stdio.h>
#include <omp.h>
int main(){
    #pragma omp parallel default(none)
    {
      int i;
      int my_thread = omp_get_thread_num();
      #pragma omp for
      for(i=0;i<16;i++)
        printf("Thread %d gets i = %d\n",my_thread,i);
    }
    return 0;
}
```

Loops in OpenMP

- The output of running the previous program with `OMP_NUM_THREADS=4` will look like:

```
Thread 3 gets i = 12
Thread 3 gets i = 13
Thread 3 gets i = 14
Thread 3 gets i = 15
Thread 0 gets i = 0
Thread 0 gets i = 1
…
```

Loops in OpenMP

- The `omp for` (and `omp do`) constructs break up the iterations by number of threads
- If it cannot divide evenly, it does as best as it can
- There is a more advanced construct to break up work of arbitrary blocks of code with `omp task`

More advanced example

- Multiply a vector by a scalar

$$Z=aX+Y$$

- First implement serially then with OpenMP

- The serial implementation is in `daxpy.c/daxpy.f90`

- Warning: This example is for illustration only and you should use BLAS implementation in your real applications

```
#include <stdio.h>
#include "ticktock.h"
void daxpy(int n, double a, double *x, double *y, double *z)
{
  int i;
  for (i=0; i<n; i++){ //initialize vectors
    x[i] = (double)i*(double)i;
    y[i] = (i+1.)*(i-1.);
  }
  for (i=0; i<n; i++)
    z[i] += a * x[i] + y[i];
}//end of daxpy
int main(){
  int n=1e7; double *x = malloc(sizeof(double)*n);
  double *y = malloc(sizeof(double)*n);
  double *z = malloc(sizeof(double)*n);
  double a = 5./3.;
  tick_tock tt;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(x);
  free(y);
  free(z);
}
```

```
#include <stdio.h>
#include "ticktock.h"
void daxpy(int n, double a, double *x, double *y, double *z)
{
  int i;
  for (i=0; i<n; i++){ //initialize vectors
    x[i] = (double)i*(double)i;
    y[i] = (i+1.)*(i-1.);
  }
  for (i=0; i<n; i++)
    z[i] += a * x[i] + y[i];
}//end of daxpy
int main(){
  int n=1e7; double *x = malloc(sizeof(double)*n);
  double *y = malloc(sizeof(double)*n);
  double *z = malloc(sizeof(double)*n);
  double a = 5./3.;
  tick_tock tt;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(x);
  free(y);
  free(z);
}
```

Utilities for this course

```
#include <stdio.h>
#include "ticktock.h"
void daxpy(int n, double a, double *x, double *y, double *z)
{
  int i;
  for (i=0; i<n; i++){ //initialize vectors
    x[i] = (double)i*(double)i;
    y[i] = (i+1.)*(i-1.);
  }
  for (i=0; i<n; i++)
    z[i] += a * x[i] + y[i];
}//end of daxpy
int main(){
  int n=1e7;double *x = malloc(sizeof(double)*n);
  double *y = malloc(sizeof(double)*n);
  double *z = malloc(sizeof(double)*n);
  double a = 5./3.;
  tick_tock tt;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(x);
  free(y);
  free(z);
}
```

Initialization

computation

```
#include <stdio.h>
#include "ticktock.h"
void daxpy(int n, double a, double *x, double *y, double *z)
{
  int i;
  for (i=0; i<n; i++){ //initialize vectors
    x[i] = (double)i*(double)i;
    y[i] = (i+1.)*(i-1.);
  }
  for (i=0; i<n; i++)
    z[i] += a * x[i] + y[i];
}//end of daxpy
int main(){
  int n=1e7;double *x = malloc(sizeof(double)*n);
  double *y = malloc(sizeof(double)*n);
  double *z = malloc(sizeof(double)*n);
  double a = 5./3.;
  tick_tock tt;
  tick(&tt);
  daxpy(n,a,x,y,z);
  tock(&tt);
  free(x);
  free(y);
  free(z);
}
```

Initialization

computation

Setup, call, timing

```
$ gcc daxpy.c ticktock.c -o daxpy
$./daxpy
Tock registers 0.2403 seconds.
```

OpenMP version

- Try to insert proper OpenMP directives in `omp_daxpy-template.c` or `omp_daxpy-template.f90`
- Compile your code and set `OMP_NUM_THREAD` to 2
- Run your code

OpenMP version: omp_daxpy.c

```
void daxpy(int n, double a, double *x, double *y, double *z){
  #pragma omp parallel default(none) shared(n,x,y,a,z)
  {
    int i;
    #pragma omp for
    for (i=0; i<n; i++){
      x[i] = (double)i*(double)i;
      y[i] = (i+1.)*(i-1.);
    }
    #pragma omp for
    for (i=0; i<n; i++)
      z[i] += a * x[i] + y[i];
  }
}
```

Running parallel daxpy code

```
$gcc -fopenmp omp_daxpy.c ticktock.c -o omp_daxpy

$export OMP_NUM_THREADS=2
$./omp_daxpy
Tock registers 0.1459 seconds.
```
1.65x speedup, 83% efficiency

```
$export OMP_NUM_THREADS=4
$./omp_daxpy
Tock registers 0.0855 seconds.
```
2.81x speedup, 70% efficiency

```
$export OMP_NUM_THREADS=8
$./omp_daxpy
Tock registers 0.0538 seconds.
```
4.67x speedup, 58% efficiency

Dot product

- Dot product of two vectors

$$n = \vec{x}.\vec{y}$$
$$= \sum_i x_i y_i$$

- Start with a serial code then add OpenMP directives
  - Serial code is located in the `ndot.c` and `ndot.f90` files

```
#include <stdio.h>
#include "ticktock.h"
double ndot(int n, double *x, double *y){
  double tot = 0; int i;
  for (i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
int main(){
  int n=1e7; int i;
  double *x = malloc(sizeof(double)*n);
  double *y = malloc(sizeof(double)*n);
  for (i=0; i<n; i++)
    x[i] = y[i] = (double)i;
  double ans=(n-1.)*n*(2.*n-1.)/6.0;
  tick_tock tt;
  tick(&tt);
  double dot=ndot(n,x,y);
  printf("Dot product: %8.4e (vs %8.4e) for n=%d\n",
       dot, ans, n);
  free(x);free(y);
}
```

computation

initialization

Dot product

- Compile and running:

```
$gcc ndot.c ticktock.c -o ndot
$./ndot
Dot product: 3.3333e+20 (vs 3.3333e+20) for n = 10000000
Tock registers  0.0453 seconds.
```

Towards a parallel solution

- Try to insert proper OpenMP directives in `omp_ndot_race-template.c` or `omp_ndot_race-template.f90`

- Compile your code and set `OMP_NUM_THREAD` to 4

- Run your code

Towards a parallel solution

- Use `#pragma omp parallel for` in the `ndot` function
- We need the sum from everyone

```
double ndot(int n, double *x, double *y) {
  double tot = 0;
  int i = 0;
  #pragma omp parallel for default(none) shared(tot,n,x,y)
  for (i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

Answer is wrong and slower than serial version

```
$gcc -fopenmp omp_ndot_race.c
ticktock.c -o omp_ndot_race
$export OMP_NUM_THREADS=4
$./omp_ndot_race
Dot product: 2.2725e+20 (vs
3.3333e+20) for n = 10000000
Tock registers   0.1319 seconds.
```

Race condition

- Threads try to update the `tot` variable at the same time
- Your program could run correctly for small runs
- Primarily a problem with shared memory

`tot = 0`

Thread 0	Thread 1
read `tot`(=0) into register	
`reg` = `reg + 1`	Read `tot`(=0) into register
Store `reg`(=1) into tot	`reg` = `reg + 1`
	Store `reg`(=1) into tot

`tot = 1`

OpenMP critical construct

- `#pragma omp critical`
- Creates a *critical* region where only one thread can be operating at a time

```
double ndot(int n, double *x, double *y) {
  double tot = 0; int i;
  #pragma omp parallel for default(none) shared(tot,n,x,y)
  for (i=0; i<n; i++)
    #pragma omp critical
    tot += x[i] * y[i];
  return tot;
}
```

Answer is correct but 50x slower than serial version

```
$gcc -fopenmp omp_ndot_critical.c
ticktock.c -o omp_ndot_critical
$export OMP_NUM_THREADS=4
$./omp_ndot_critical
Dot product: 3.3333e+20 (vs
3.3333e+20) for n = 10000000
Tock registers 2.0842 seconds.
```

OpenMP atomic construct

- `#pragma omp atomic`
- Most hardware has support for indivisible instructions (load/add/store as one instruction)
- Lower overhead than critical

```
double ndot(int n, double *x, double *y) {
  double tot = 0; int i;
  #pragma omp parallel for default(none) shared(tot,n,x,y)
  for (i=0; i<n; i++)
    #pragma omp atomic
    tot += x[i] * y[i];
  return tot;
}
```

```
$gcc -fopenmp omp_ndot_atomic.c
ticktock.c -o omp_ndot_atomic
$export OMP_NUM_THREADS=4
$./omp_ndot_atomic
Dot product: 3.3333e+20 (vs
3.3333e+20) for n = 10000000
Tock registers 0.5638 seconds.
```

Answer is correct and 12x slower than serial version

Fixing the slowness

- Try to add proper OpenMP directives in either `omp_ndot_local-template.c` or `omp_ndot_local-template.f90`

- Use a local variable for each thread to calculate the partial sum and a shared variable to calculate the total sum

- Use `atomic` directive to calculate the total sum

Fixing the slowness

- We can use local sums and only add those sums to `tot`
  - Each thread will do a local sum
  - Only P (number of threads) additions with atomic or critical vs 10^7

```
double ndot(int n, double *x, double *y) {
  double tot = 0;
  #pragma omp parallel default(none) shared(tot,n,x,y)
  {
    double mytot = 0; int i;
    #pragma omp for
    for (i=0; i<n; i++)
      mytot += x[i] * y[i];
    #pragma omp atomic
    tot += mytot;
  }
  return tot;
}
```

```
$gcc -fopenmp omp_ndot_local.c
ticktock.c -o omp_ndot_local
$export OMP_NUM_THREADS=4
$./omp_ndot_local
Dot product: 3.3333e+20 (vs
3.3333e+20) for n = 10000000
Tock registers 0.0159 seconds.
```

Answer is correct and 2.85x speedup!!

OpenMP reduction

- Aggregating values from different threads is a common operation that OpenMP has a special *reduction* variable
  - Similar to private and shared
- Reduction variables can support several types of operations: + - *

```
double ndot(int n, double *x, double *y) {
  double tot = 0;
  #pragma omp parallel for shared(n,x,y) reduction(+:tot)
  int i;
  for (i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
```

```
$gcc -fopenmp omp_ndot_reduction.c
ticktock.c -o omp_ndot_reduction
$export OMP_NUM_THREADS=4
$./omp_ndot_reduction
Dot product: 3.3333e+20 (vs
3.3333e+20) for n = 10000000
Tock registers 0.0157 seconds.
```

Same speed as local but simpler code

Performance of dot product

- Increasing the number of threads from 4 to 8 does not give us any noticeable speedup

- Sometimes we are limited by how fast we can feed CPUs
  - Not by the CPU speed/power

- In the dot product problem, we had 10^7 double vectors, with 2 numbers of 8 bytes long flowing through in 0.0159 seconds giving about 9 GB/s memory bandwidth which is what you would expect from this architecture

Conclusion

- New computers come with increased number of cores and the best way to take full advantage of them is using parallel programming
- OpenMP is an easy and quick way to make a serial job run faster on multi-core machines
- Getting a good speedup from a parallel program is a challenging task

Conclusion

- WestGrid's Tier 2 and Tier 3 support are available to help you with your parallel programming needs

- Some useful links:
  - http://www.openmp.org
  - https://www.westgrid.ca/support/programming
  - https://www.westgrid.ca/events/intro_westgrid_tips_choosing_system_and_running_jobs
  - https://www.cac.cornell.edu/VW/OpenMP/default.aspx?id=xup_guest
  - http://www.openmp.org/mp-documents/OpenMP3.0-FortranCard.pdf