INTRODUCTION TO OPENMP (PART II)

Hossein Pourreza

hossein.pourreza@umanitoba.ca

March 9, 2016

Acknowledgement: Some of examples used in this presentation are courtesy of SciNet.

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
    - E.g., putty or MobaXterm (http://mobaxterm.mobatek.net/download.html)

- If you do not see `openmp-wg-pII-2016` folder under your home directory, please run the following command:
  - `cp -r /global/scratch/workshop/openmp-wg-pII-2016 .`
  - Please do not forget the . (dot) at the end

Logistics of this webinar

- Once you are in the `openmp-wg-pII-2016` 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 Jasper
  - If you want to use an editor with a graphical user interface, you may experience some slowness over SSH connection
  - Copying codes/commands from slides may result in unexpected error messages

Review of the previous webinar

- Modern CPUs come with multiple processing units (cores) and to harness their power we need to do multi threading/processing

- OpenMP is a de-facto industry standard API to create parallel regions in a code
  - All threads will run the same code in a parallel region
  - Work sharing is possible through other directives

- OpenMP includes compiler directives, library routines, and environment variables

Loops in OpenMP

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

- OpenMP has specific loop parallelization directives:
  - C/C++
    - `#pragma omp parallel for`
    - `#pragma omp for`
  - Fortran
    - `!$OMP PARALLEL DO … !OMP END PARALLEL DO`
    - `!$OMP DO … !OMP END DO`
  - Loop iterations should be independent

- Try modifying `omp_loop_template.c` or `.f90`

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`) clauses 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 clause to break up work of arbitrary blocks of code with `omp task`

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

computation

initialization

```
#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);
}
```

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

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

```
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.
```

OpenMP reduction

- Aggregating values from different threads is a common operation that OpenMP has a special *reduction* clause
- Reduction variables can support several types of *associative* operations: + - *

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

2.89 times speedup

```
$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.
```

Nested loops

- When there is a rectangular loop, you can use the `collapse` clause:

```
#pragma omp parallel for collapse(2)
    for (i = 0;i < N;i++){
        for(j = 0;j < M;j++){
      …
        }
    }
```

- Argument is the number of loops to collapse and will form a single loop of NxM iterations

Thread synchronization

- To ensure the correctness of computation, threads sometimes need to be synchronized
  - Critical region and atomic update
  - At the end of parallel region and the loop construct
- To override the default behavior, `nowait` can be used:

```
#pragma omp for nowait
!$OMP DO
!$OMP END DO nowait
```

- If there are multiple independent loops in a parallel region, having `nowait` could improve the performance
  - Some considerations when reduction is used

Conditional threading

- Sometimes creating threads takes longer than doing the calculation by one thread

- OpenMP, using `if(condition)` clause, supports conditional threading to avoid extra overhead for small size problems.
  - The `if(condition)` clause can be added to `omp parallel`

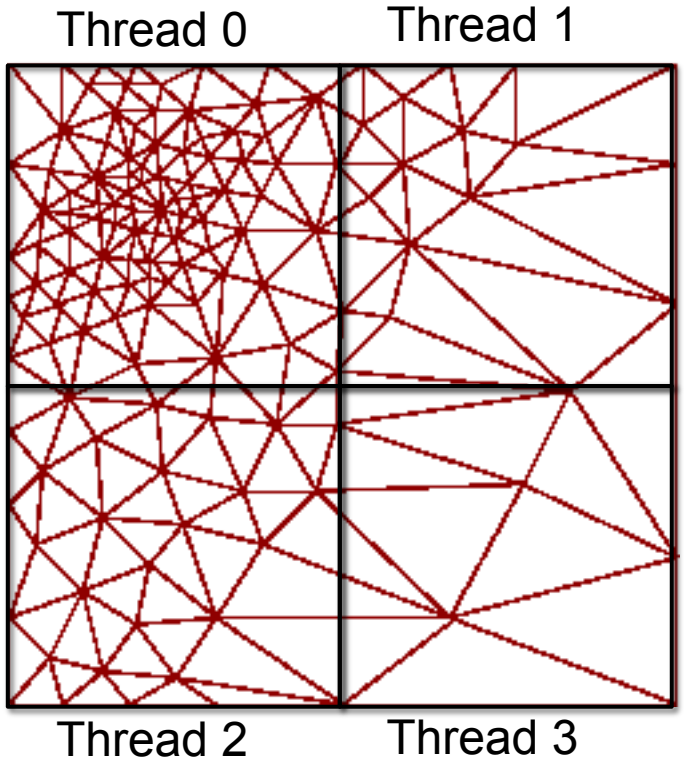
Conditional threading

```
//omp_condition.c
#include <omp.h>
#include <stdio.h>
int main(int args,char *argv[]){
  int N = atoi(argv[1]); int i = 0;
  #pragma omp parallel for if(N>10)
  for(i = 0;i < N;i++)
    printf("Thread %d gets i = %d
      \n",omp_get_thread_num(),i);
}
```

Reads a number from the command line and if it is bigger than 10 (in this example) creates the parallel region otherwise executes serially

Load balancing

- So far iterations of a loop had the same amount of work
  - But this is not always the case
- Working on a mesh with different granularities
  - A finer mesh requires more computations
- The default work sharing divides N iterations equally among nthreads
  - Some threads will finish their jobs sooner and sit idle



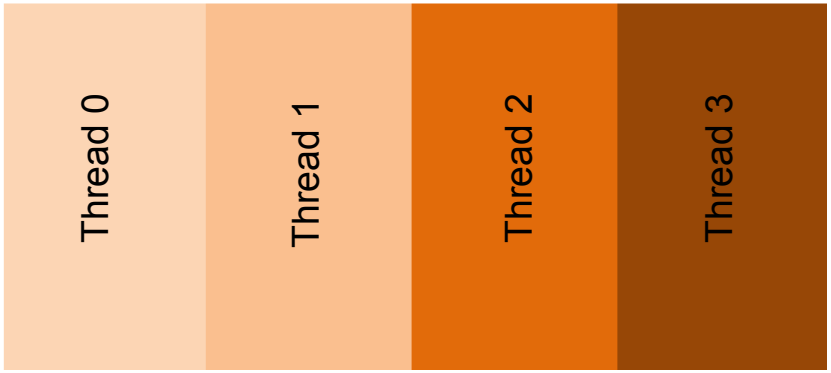
The image was taken from http://courses.teresco.org/cs341_f07/assignments/quadtree_jacobi/

```
//omp_imbalance.c
#include "ticktock.h"
int runloop(int i) {
    int sum=0,j;
    for (j = 0;j < i;j++)
        sum+=i^j;
    return sum;
}
int main() {
    int i = 0, N = 4e4;
    int arr[N];
    tick_tock tt;
    tick(&tt);
    //imbalance loop iterations
    for (i = 0;i < N;i++) {
      arr[i]=runloop(i);
    }
    tock(&tt);
    return 0;
}
```

computation

OpenMP implementation

```
int main() {
    int i = 0, N = 4e4;
    int arr[N];
    tick_tock tt;
    tick(&tt);
    #pragma omp parallel for
    for (i = 0;i < N;i++) {
      arr[i] = runloop(i);
    }
    tock(&tt);
    return 0;
}
```



Each thread gets `N/nthreads` iterations of the loop but the amount of work is not equal

```
Serial: 1.96 sec
Parallel with 4 threads: 0.88 sec
Speedup: 2.23
Efficiency: 56%
```

Scheduling clause in OpenMP

- The default work sharing in OpenMP assigns N/nthreads consecutive iterations of a loop to each thread
  - N is the maximum number of loop iterations
- Often it is better to assign smaller interleaved chunks
- The `schedule(type[,chunk])` clause can change the default behavior

```
#pragma omp parallel for schedule(static,m)
```

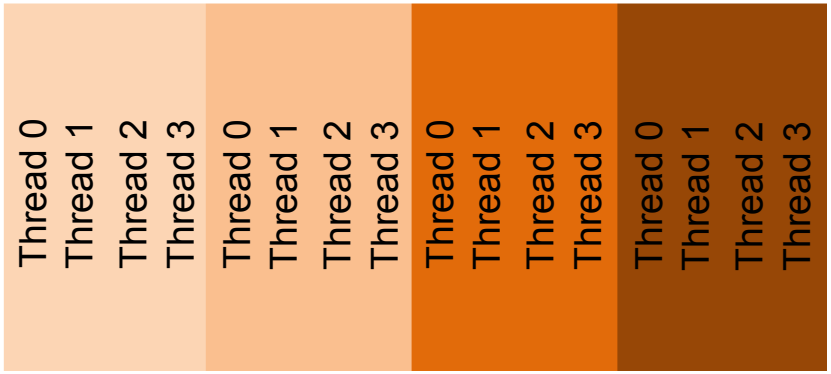
  - Chunks of size m will be assigned to each thread in a round robin order

```
#pragma omp parallel for schedule(dynamic,m)
```

  - Each thread will get more work after finishing m iterations of the loop

Second try with schedule clause

```
int main() {
  int i = 0, N = 4e4;
  int arr[N];
  tick_tock tt;
  tick(&tt);
  #pragma omp parallel for schedule(static,2500)
  for (i=0;i<N;i++) {
    arr[i]=runloop(i);
  }
  tock(&tt);
  return 0;
}
```

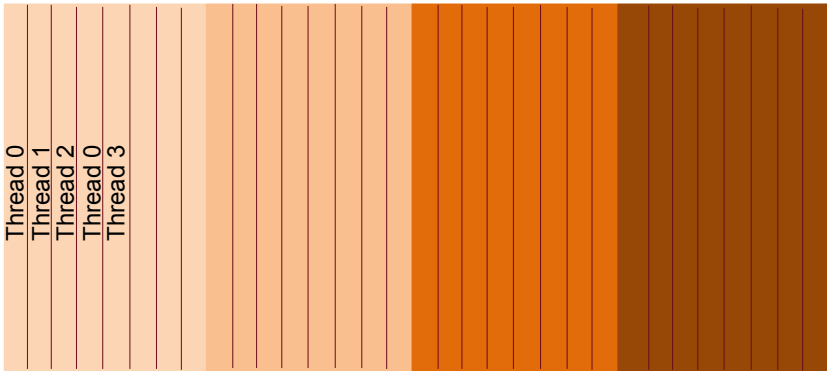


Each thread gets chunks of 2500 iterations of the loop

```
Serial: 1.96 sec
Parallel with 4 threads: 0.62 sec
Speedup: 3.16
Efficiency: 79%
```

Third try with dynamic scheduling

```
int main() {
  int i = 0, N = 4e4;
  int arr[N];
  tick_tock tt;
  tick(&tt);
  #pragma omp parallel for schedule(dynamic)
  for (i=0;i<N;i++) {
    arr[i]=runloop(i);
  }
  tock(&tt);
  return 0;
}
```



Work is divided into smaller pieces and assigned to threads when they are ready

```
Serial: 1.96 sec
Parallel with 4 threads: 0.51 sec
Speedup: 3.84
Efficiency: 96%
```

Tuning

- If you have a loop with imbalance workload, start with `schedule(static,m)` or `schedule(dynamic)`

- You may have to try different chunk sizes to get the best possible performance

Non-loop parallelism

- `#pragma omp sections` defines a block containing N sub-blocks that my be executed by N threads
  - `#pragma omp section` defines each of those sub-blocks

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      something to do
    #pragma omp section
      other thing to do at the same time
  }
}
```

Non-loop parallelism

```
//omp_section.c
#include <stdio.h>
int main(){
  #pragma omp parallel sections
  {
   #pragma omp section
   printf("Hello 1 from thread %d\n",
      omp_get_thread_num());
   #pragma omp section
   printf("Hello 2 from thread %d\n",
      omp_get_thread_num());
  }
}
```

```
$gcc -fopenmp omp_sections.c -o
omp_sections
$export OMP_NUM_THREADS=4
$./omp_sections
Hello 2 from thread 2
Hello 1 from thread 3
```

Tasks in OpenMP

- Introduced in OpenMP 3.0 to enable non-loop or unbounded loop parallelism (e.g., a `while` loop)

```
#pragma omp task
```

- A flexible alternative to the `sections` directive
  - Tasks can be created dynamically whereas the number of sub-blocks in `sections` was static

- Tasks must be created within the parallel region and should usually be enclosed by `#pragma omp single` directive

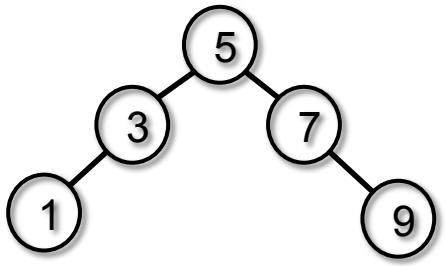
Tasks in OpenMP

```
//omp_task.c
int main(){
  #pragma omp parallel
  #pragma omp single
  {
    printf(”Thread %d is creating tasks\n”,
       omp_get_thread_num());
    #pragma omp task
      printf("Hello 1 from thread %d\n",omp_get_thread_num());
    #pragma omp task
      printf("Hello 2 from thread %d\n",omp_get_thread_num());
  }
}
```

- Try commenting out the `pragma omp single` directive to see what happens

Tasks in OpenMP

- Tasks are independent units of work
- A thread is assigned to perform a task



- Some hard to parallelize problems can be done using tasks
  - Problems such as parallel traversing of linked lists or trees
- Compared to the loop parallelization, tasks have substantial overhead

Tasks in OpenMP

- Write a program to randomly print either “A race car” or “A car race”
  - Use two threads

- Start modifying the `omp_task_fun_template.c` or `.f90` file

Tasks in OpenMP: A more advanced code

- A team of threads is created at the `omp parallel` clause
- A single thread (T) will execute the while loop
- Thread T operates the while loop, creates tasks, and updates p
  - `firstprivate` means p will be initialized with the last value of p before task gets started.
- Each time T crosses `omp task` a new task is created
- Each task runs in its own thread
- The moment of execution of a task is up to the runtime system

```
#pragma omp parallel
#pragma omp single
{
 node *p = head_of_list;
 while(p){
 #pragma omp task firstprivate(p)
    process(p);
  p=p->next;
 }
}
```

Conclusion

- OpenMP is an easy and quick way to make a serial job run faster on multi-core machines

- In addition to loop parallelization, OpenMP offers various clauses such as tasks and sections for hard to parallelize problems

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.westgrid.ca/events/introduction_openmp
  - https://www.cac.cornell.edu/VW/OpenMP/default.aspx?id=xup_guest
  - http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf
  - http://www.openmp.org/mp-documents/OpenMP3.0-FortranCard.pdf
  - http://archer.ac.uk/training/course-material/2014/11/ShMem_OpenMP_Durham/

Sample PBS script for OpenMP jobs

```
#!/bin/bash
#PBS -S /bin/bash
# Sample script for running an OpenMP-based parallel program, openmp_diffuse.
#PBS -l nodes=1:ppn=4
#PBS -l mem=2000m
#PBS -l walltime=24:00:00

#load compiler module that you compiled your code with

cd $PBS_O_WORKDIR
echo "Current working directory is `pwd`"

export OMP_NUM_THREADS=$PBS_NUM_PPN

# On systems where $PBS_NUM_PPN is not available, one could use:
#CORES=`/bin/awk 'END {print NR}' $PBS_NODEFILE`
#export OMP_NUM_THREADS=$CORES

./openmp_diffuse < openmp_diffuse.in
echo "Program openmp_diffuse finished at: `date`"
```