Basic OpenMP and Profiling

4/29/2020

Agenda

10:00 – 10:45 am	Lecture 1: Introduction to OpenMP
10:45 – 11:00 am	Q&A; break
11:00 – 11:45 am	Lecture 2: Profiling OpenMP applications with Vtune
11:45 am – 12:00pm	Q&A

Instructor: Ying-Wai Li yingwaili@lanl.gov
(CCS-7 and Institutional Computing Applications Team)

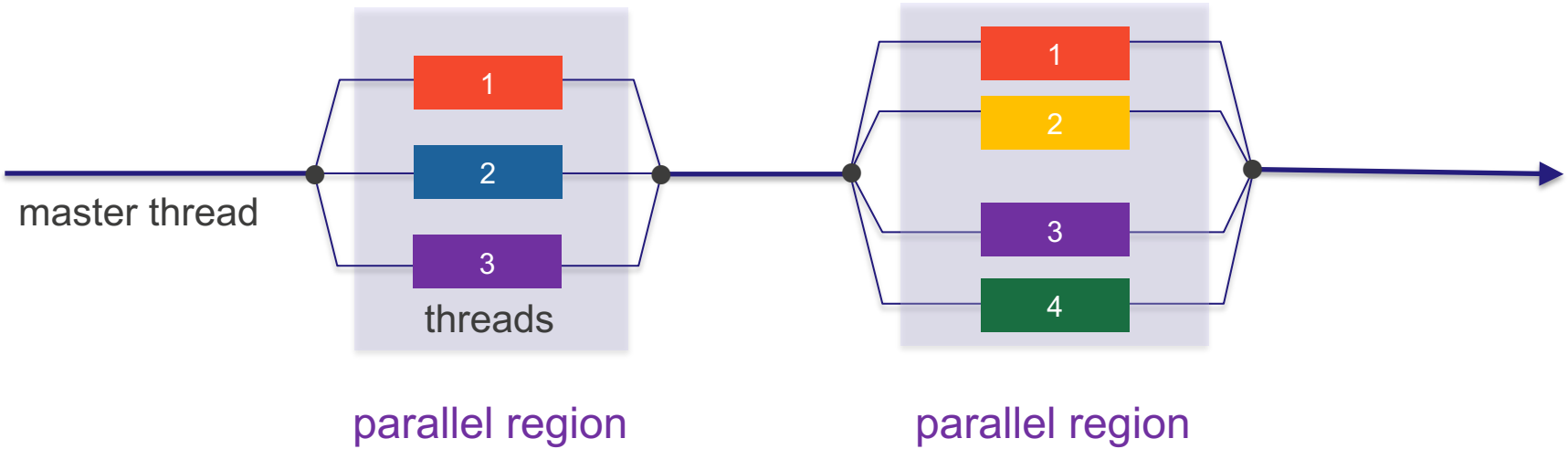
Lecture 1
Introduction to OpenMP

Parallel Computing Basics

- **Processing Element**: the part of the computer that computes. This can be a CPU or a component of a parallel processor such as a GPU.
- **Node**: a group of processing elements working together to do tasks.
- **Parallel computing:** the simultaneous execution of the same task, split up and specially adapted, on multiple processing elements in order to obtain faster results. The speed comes from doing more tasks at once.
- **Network**: the connection between nodes that allows communication between those nodes.
- **Communication:** communication between processing elements is required for most parallel programs.
- **Scaling:** how efficiently a program's ability to do tasks increases with increasing numbers of processing elements.

What is OpenMP?

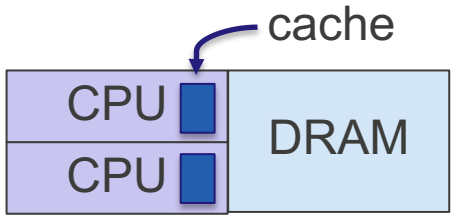
- A standard / library / application programming interface (API) for parallel programming
- Parallelism on single node or multicore CPU on a laptop
- Fork-join model



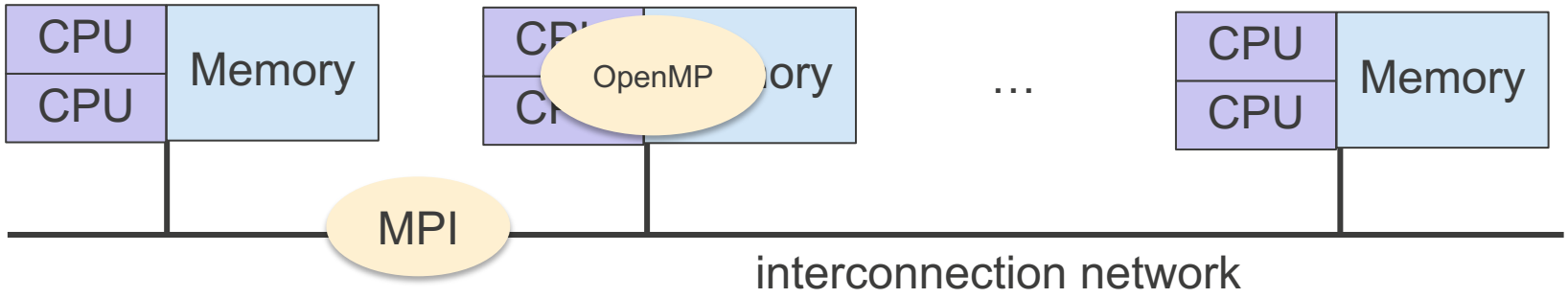
What is OpenMP?

- Shared-memory programming model:
  - Data in parallel regions is shared by default
  - All threads can access the shared data simultaneously

- Data scoping
  - Global variables in the heap (e.g. DRAM) are shared
  - Local variables in the stack (e.g. cache) are private



- Hybrid MPI-OpenMP parallelism:



A “Hello World!” serial C++ program

```
#include <cstdio>


int main ()
{
  printf( "Hello world !”);

  return 0;
}
```

First step to parallel programming:

Start from the serial code as long as it is possible

A “Hello World!” OpenMP program

```
#include <cstdio>
#include <omp.h>

int main ()
{
  #pragma omp parallel
  {
  int thread_id = omp_get_thread_num();
  int total_threads = omp_get_num_threads();
  printf( "Hello from thread %d of %d\n",
          thread_id, total_threads);
  }

  return 0;
}
```

Include OpenMP header file that contains the library’s API

Get the identifier of each thread

Get the total number of threads

Compiling the “Hello World!” OpenMP program

- Compile the program with: `g++ -fopenmp`
  (`f77` or `f90` for Fortran)
- Different compilers have different flags to activate OpenMP:
  - GNU: `g++/gfortran` `-fopenmp`
  - Intel: `icpc/ifort` `-qopenmp`
  - PGI: `pgCC/pgf90` `-mp`

```
> export OMP_NUM_THREADS=3
> g++ -fopenmp -o OpenMPHelloWorld OpenMPHelloWorld.cpp
>  ./OpenMPHelloWorld
hello from thread 1 of 3 !
hello from thread 0 of 3 !
hello from thread 2 of 3 !
```

Question:
What do you observe from the order of print-outs from different threads?

OpenMP API overview

The “Hello World” example shows the 3 fundamental OpenMP components (in order of precedence):

- Compiler directives
  `#pragma omp` (C++), `!$OMP` (Fortran)
- Runtime library routines
  e.g. `omp_get_thread_num(), omp_get_num_threads()`
- Environment variables
  e.g. `OMP_NUM_THREADS`

Exercise 1 What will you see in the output?

```
#include <cstdio>
#include <omp.h>

int main ()
{
  printf("Number of processors: %d \n", omp_get_num_procs());

  omp_set_num_threads(omp_get_num_procs());
  printf("Number of threads: %d \n", omp_get_num_threads());

  #pragma omp parallel
  {
    int thread_id = omp_get_thread_num();
    int total_threads = omp_get_num_threads();
    printf( "Hello from thread %d of %d\n", thread_id, total_threads);
  }

  return 0;
}
```

Parallelizing a for loop: the `for` directive

How to parallelize the following vector-vector addition example?

```
#include <cstdio>

int main ()
{
  int array_size = 100, sum = 0;
  int A[array_size], B[array_size], C[array_size];

  // Initialize A and B, add A and B and put the result in C
  for (int i=0; i<array_size; ++i) {
    A[i] = i;
    B[i] = i*2;
    C[i] = A[i] + B[i];
  }

  return 0;
}
```

2_vector_addition_3arrays.cpp

Parallelizing a for loop: the `for` directive

How to parallelize the following vector-vector addition example?

```
#include <cstdio>
#include <omp.h>

int main ()
{
  int array_size = 100, sum = 0;
  int A[array_size], B[array_size], C[array_size];

  // Initialize A and B, add A and B and put the result in C
  #pragma omp parallel for
  for (int i=0; i<array_size; ++i) {
    A[i] = i;
    B[i] = i*2;
    C[i] = A[i] + B[i];
  }

  return 0;
}
```

Global variables are shared and visible by all threads

2_vector_addition_3arrays_parallel2.cpp

Parallelizing a for loop: the `for` directive

How about this? How to parallelize it?

```
#include <cstdio>

int main ()
{
  int array_size = 100, sum = 0;
  int A[array_size], B[array_size];

  // Initialize A and B, add A and B and accumulate the results in sum
  for (int i=0; i<array_size; ++i) {
    A[i] = i;
    B[i] = i*2;
    sum += A[i] + B[i];
  }

  printf("The sum of the array is %d\n", sum);

  return 0;
}
```

2_vector_addition.cpp

Parallelizing a for loop: the `for` directive

Does it work? Why not? How to fix this?

```
#include <cstdio>
#include <omp.h>

int main ()
{
  int array_size = 100, sum = 0;
  int A[array_size], B[array_size];

  #pragma omp parallel for
  for (int i=0; i<array_size; ++i) {
    A[i] = i;
    B[i] = i*2;
    sum += A[i] + B[i];
  }
  printf("The sum of the array is %d\n", sum);
  return 0;
}
```

Race condition! `sum` is a global variable shared among all threads.

2_vector_addition_parallel_buggy.cpp

Solution 1: The `critical` directive

- `critical` specifies that the region of code must be executed by only one thread at a time

2_vector_addition_parallel_solution1.cpp

```
#include <cstdio>
#include <omp.h>

int main ()
{
  int array_size = 100, sum = 0;
  int A[array_size], B[array_size];

  #pragma omp parallel for
  for (int i=0; i<array_size; ++i) {
    A[i] = i;
    B[i] = i*2;
  #pragma omp critical
    sum += A[i] + B[i];
  }

  printf("The sum of the array is %d\n", sum);
  return 0;
}
```

Eliminate race condition, but serialize the computation that leads to poor performance

Solution 2: A better use of `critical` directive

2_vector_addition_parallel_solution2.cpp

```
#include <cstdio>
#include <omp.h>

int main ()
{
  int array_size = 100, sum = 0;
  int A[array_size], B[array_size];

  #pragma omp parallel
  {
    int sum_local = 0;
    #pragma omp for
    for (int i=0; i<array_size; ++i) {
      A[i] = i; B[i] = i*2;
      sum_local += A[i] + B[i];
    }
    #pragma omp critical
    sum += sum_local;
  }
  printf("The sum of the array is %d\n", sum);
  return 0;
}
```

Each thread has its own copy of `sum_local`

The mutually exclusive operations are brought outside the `for` loop

Solution 2.5: The `atomic` directive

2_vector_addition_parallel_solution2.cpp

```
#include <cstdio>
#include <omp.h>

int main ()
{
  int array_size = 100, sum = 0;
  int A[array_size], B[array_size];

  #pragma omp parallel
  {
    int sum_local = 0;
    #pragma omp for
    for (int i=0; i<array_size; ++i) {
      A[i] = i; B[i] = i*2;
      sum_local += A[i] + B[i];
    }
    #pragma omp atomic
    sum += sum_local;
  }
  printf("The sum of the array is %d\n", sum);
  return 0;
}
```

Solution 3: The `reduction` directive

- A private copy for the specified variable is created and initialized for each thread
- At the end of the loop, the reduction operation is carried out and the result is written to the global shared variable

2_vector_addition_parallel_solution3.cpp

```
#include <cstdio>
#include <omp.h>

int main ()
{
  int array_size = 100, sum = 0;
  int A[array_size], B[array_size];

  #pragma omp parallel for reduction(+:sum)
  for (int i=0; i<array_size; ++i) {
    A[i] = i;
    B[i] = i*2;
    sum += A[i] + B[i];
  }
  printf("The sum of the array is %d\n", sum);
  return 0;
}
```

Data scoping

- By default, all variables defined outside the parallel region are **shared**
- You can change it by:
  1. The `default` clause
     C++: `#pragma omp parallel default(shared/none)`
     Fortran: `!$omp parallel default(private/firstprivate/shared/none)`

  2. The `shared` clause to share a variable among threads
     `#pragma omp parallel shared(A,B)`

  3. The `private` clause to make a local copy for a variable in each thread
     `#pragma omp parallel private(sum)`
     Note: `private` does not initialize the variable

  4. Use the `firstprivate` clause instead initialize a local variable
     ```
     C = 1.0;
     #pragma omp parallel firstprivate(sum)
     ```

Other useful OpenMP functions and concepts

- **Thread placement**
  - How an OpenMP thread is mapped onto hardware core
    `#pragma omp for schedule(static/dynamic/runtime/auto)`
- **Synchronization**
  - Ensuring all the threads arrive at the same point before moving on
    `#pragma omp barrier`
  - The opposite:
    `#pragma omp [for/single/…] nowait`
    (Note: by default, there is a barrier at the end of a parallel region/construct)
- **Loop flattening** (for nested for loops)
  `#pragma omp for collapse(level)`
- **Sections and Tasks** (the `sections` or `task` constructs)
- **Offloading to accelerators (e.g. GPUs)** (the `target` constructs)

Some useful resources

- Youtube video series “Introduction to OpenMP” by Tim Mattson (Intel) (slides and exercise sample codes are available)
- Lawrence Livermore National Laboratory’s OpenMP tutorial https://computing.llnl.gov/tutorials/openMP/
- OpenMP’s official website https://www.openmp.org/ (the examples in the Specifications are particularly useful)
- “Using OpenMP – Portable Shared Memory Parallel Programming” by B. Chapman, G. Jost, and R. van der Pas, The MIT Press.

Lecture 2
Profiling OpenMP applications with Vtune

Profiling and debugging tools available on IC machines

Tools	Module	Kodiak	Badger/Grizzly	Capulin
gprof	---	✓	✓	✓
Valgrind	valgrind	✓	✓	✓
TotalView	totalview	✓	✓	✓
ARM Forge (DDT, MAP)	forge	✓	✓	✓ ddt / ddt-memdebug
PAPI	papi	✓	✓	✓
HPCToolkit	hpctoolkit	✓	✓	
Score-P	scorep	✓	✓	
Tau	tau	✓	✓	
NVIDIA tools • nvprof • NSIGHT	cudatoolkit	✓		
Intel tools	intel-advisor intel-inspector intel-trace-analyzer intel-vtune-amplifier	✓	✓	

Profiling and debugging tools available on IC machines

Tools	Module	Kodiak	Badger/Grizzly	Capulin
CrayPAT	perftools			✓
Stack Trace Analysis Tool	stat			✓
Cray's line mode debugger	gdb4hpc			✓
Cray's abnormal termination processing	atp			✓
Arm Allinea Perf-Report Tool	arm-perf-reports			✓

Loading modules and compiling codes on Badger

- Load a compiler
  `module load gcc` or `module load intel`
- Load Intel's Vtune Amplifier module (for generating profiles)
  `module load intel-vtune-amplifier`
- Compile the code with the debugging flag
  `icpc -qopenmp -g -o 1_hello_world 1_hello_world.cpp`
- Set the environment variable to determine the number of threads:
  `export OMP_NUM_THREADS=36`
- Run the executable with Vtune Amplifier:
  `srun -n 1 amplxe-cl -collect threading -r output ./1_hello_world`
- An output directory named `output.xxx.xxx/` will be resulted

Analyzing VTune's profiling output files

- If X11 works fine, visualize the results on the compute node:
  `amplxe-gui output.xxx.xxx/`

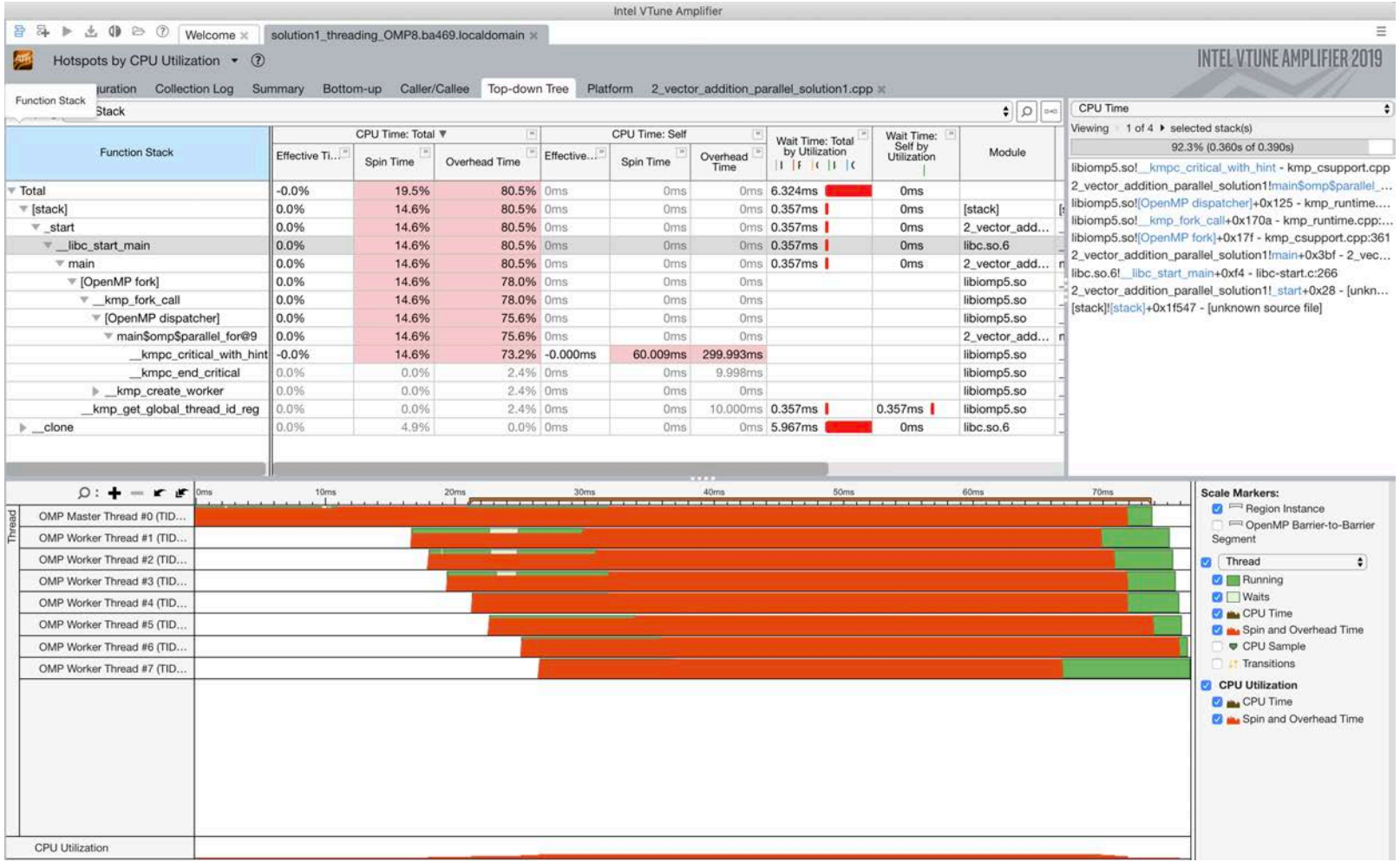
- If you are comfortable with command line:
  `amplxe-cl -report summary -r output.xxx.xxx/`

- If working remotely, copy the output directory to our computer:
  `scp -r [usr]@wtrw.lanl.gov:ba-fe:[path_to_output_directory] ./`

  Then analyze with a local Intel VTune Amplifier client (demo)

- Tips: use `amplxe-cl -help` to explore different functionalities of Amplifier

Analyzing VTune's profiling output files



Detecting data racing problem using Intel's Inspector

- Compile the code with the debugging flag:
  `icpc -qopenmp -g -o 2_vector_addition_parallel_buggy.cpp`

- Load Intel's Vtune Amplifier module (for generating profiles)
  `module load intel-inspector`

- Run the executable with Inspector:
  ```
  export OMP_NUM_THREADS=36
  srun -n 1 inspxe-cl -collect=ti3 -r inspector_out ./2_vector_addition_parallel_buggy
  ```

  Other analysis types you may do (`-collect=<flag>`):

flag(s)	Analysis type
`mi1`	Detect memory leaks
`mi2, mi3`	Detect/locate memory problems
`ti1`	Detect deadlocks
`ti2, ti3`	Locate deadlocks and data races

- An output directory named `inspector_out.xxx.xxx/` will be resulted

Detecting data racing problem using Intel's Inspector

- Viewing the report (command line only):
  `inspxe-cl -report=summary -r inspector_out.xxx.xxx/`
  `inspxe-cl -report=problems -report-all -r inspector_out.xxx.xxx/`

```
P1: Error: Data race: New
 P1.37: Error: Data race: New
  /turquoise/users/ywl/openmp_Apr20/2_vector_addition_parallel_buggy.cpp(16):
  Error X73: Write: Function main: Module /turquoise/users/ywl/openmp_Apr20/2_vector_addition_parallel_buggy
  Code snippet:
   14      A[i] = i;
   15      B[i] = i*2;
  >16      sum += A[i] + B[i];
   17    }
   18

  Stack (1 of 1 instance(s))
  >2_vector_addition_parallel_buggy!main - /turquoise/users/ywl/openmp_Apr20/2_vector_addition_parallel_buggy.cpp:16
```

Data race found!

- Tips: use `inspxe-cl -help` to explore different functionalities of Inspector
- Exercise: find another problem with the demo code using Inspector ;)