OpenMP Applications and Practices

Widodo Samyono

Outline

1. Overview of OpenMP
   1.1 Threads vs Processes
   1.2. Shared Memory Concepts
   1.3 What is OpenMP?
   1.4 Compiler Directives
2. OpenMP Applications and Practices
   2.1 Implementation of OpenMP for calculating an area under a curve
   2.2 Scalability of OpenMP for calculating an area under a curve

Overview of OpenMP

What is OpenMP?

Threads vs Processes

Threads

- execution sequences that share a single memory area (“***address space***”)
- ***Multithreading***: parallelism via multiple **threads**
- Shared Memory Parallelism is concerned with **threads**
- **Use in OpenMP**

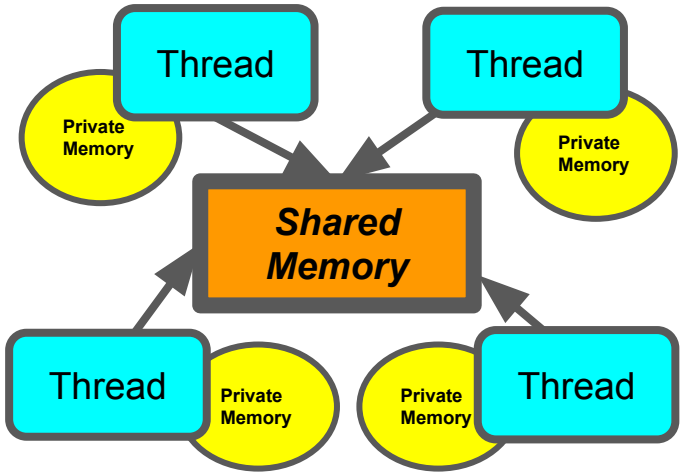
Processes

- execution sequences with their own independent, private memory areas.
- ***Multiprocessing***: parallelism via multiple **processes**
- Distributed Parallelism is concerned with **processes**
- **Use in MPI**.

Shared Memory Concept

Threads with their private memories use the shared (same) memory to process the same task.

Analogy: Painters with their own brushes share the same palette to draw on the same canvas.



What is OpenMP?

OpenMP is a standard API that can be used in shared memory programs, which are written in Fortran, C, and C++.

Main OpenMP components consist of ***compiler directives***, **runtime library routines (functions),** and **environment variables**.

OpenMP Requires a supportive compiler, such as: GNU, IBM, Oracle, Intel, HP, MS, Cray, Absoft Pro Fortran, Portland Group Compilers and Tools, Lahey/Fujitsu Fortran 95, Path Scale.

More details about OpenMP are in Petascale Website (www.shodor.org/petascale).

OpenMP main components:

Directives

- ★ Initializes parallel region.
- ★ Divides the work.
- ★ Synchronizes.
- ★ Data-Sharing Attributes.

Functions

- ★ Defines number of threads.
- ★ Gets thread ID.
- ★ Supports Nested Parallelism.

Environment Variables

- ★ Scheduling Type.
- ★ Number of Threads.
- ★ Dynamic Adjustment of Threads.
- ★ Maximum Number of Threads.

OpenMP Applications and Practices

How OpenMP Applications and Practices can be employed to create a new parallel code and to parallelize a serial code.

The parallel for Directive (C)

The parallel for directive tells the compiler that the for loop immediately after the directive should be executed in parallel; for example:

```
# pragma omp parallel for
   for (index = 0; index < length; index++) {
      array[index] = index * index;
 }
```

The iterations of the loop will be computed in parallel (note that they are independent of one another).

Chunks

By default, OpenMP splits the iterations of a loop into chunks of equal (or roughly equal) size, assigns each chunk to a thread, and lets each thread loop through its subset of the iterations.
So, for example, if you have 4 threads and 12 iterations, then each thread gets three iterations:
Thread 0: iterations 0, 1, 2
Thread 1: iterations 3, 4, 5
Thread 2: iterations 6, 7, 8
Thread 3: iterations 9, 10, 11
Notice that each thread performs its own chunk in deterministic order, but that the overall order is nondeterministic.

Private and Shared Data

Private data are data that are owned by, and only visible to, a single individual thread.

Shared data are data that are owned by and visible to all threads.

(Note: In distributed parallelism, all data are private, as we'll see next time.)

A Private Variable (C)

Consider this loop:

```
#pragma omp parallel for …
    for (iteration = 0;
          iteration < number_of_threads; iteration++) {
       this_thread = omp_get_thread_num();
       printf("Iteration %d, thread %d: Hello, world!\n",
            iteration, this_thread);
}
```

Notice that, if the iterations of the loop are executed concurrently, then the loop index variable named iteration will be wrong for all but one of the threads.

Each thread should get its own copy of the variable named iteration.

Another Private Variable (C)

```
#pragma omp parallel for …
    for (iteration = 0;
         iteration < number_of_threads; iteration++) {
       this_thread = omp_get_thread_num();
       printf("Iteration %d, thread %d: Hello, world!\n",
            iteration, this_thread);
}
```

Notice that, if the iterations of the loop are executed concurrently, then this_thread will be wrong for all but one of the threads.
Each thread should get its own copy of the variable named this_thread.

A race condition is a situation in which multiple processes can change the value of a variable at the same time.

A Shared Variable (C)

```
#pragma omp parallel for …
    for (iteration = 0;
          iteration < number_of_threads; iteration++) {
       this_thread = omp_get_thread_num();
       printf("Iteration %d, thread %d: Hello, world!\n",
            iteration, thread);
}
```

Notice that, regardless of whether the iterations of the loop are executed serially or in parallel, number_of_threads will be correct for all of the threads.

All threads should share a single instance of number_of_threads.

Creating a simple parallel code in OpenMP

1) Create a bare bone code as in C with #include<omp.h> ←- function call OpenMP Library routines
2) Create multiple threads. #pragma omp parallel ←- parallel compiler directive
3) Specify the number of threads (n) can be created to run the program. omp_set_num_threads(n) ←- OpenMP function call
4) Create private and shared variables to avoid the race condition. #pragma omp parallel private(var1, var2, ….) shared(var1, var2, ….) ←- OpenMP environment private variables
5) Get individual thread numbers inside the parallel section. omp_get_thread_num()
6) Get the total number of threads OpenMP created in the run. int omp_get_num_threads(int num)
7) mmm

We can see that all the main components of OpenMP included. We may apply all these components in the next slide.

A First OpenMP Program hello_world.c

```
#include <stdio.h>
#include <omp.h>  /* Including OpenMp Library routines */
int main ()
{
   int number_of_threads, this_thread, iteration;
   int omp_get_max_threads(), omp_get_thread_num();
   number_of_threads = omp_get_max_threads();
   fprintf(stderr, "%2d threads\n", number_of_threads);
# pragma omp parallel for default(private) shared(number_of_threads)
   for (iteration = 0;
         iteration < number_of_threads; iteration++) {
      this_thread = omp_get_thread_num();
      fprintf(stderr, "Iteration %2d, thread %2d: Hello, world!\n",
            iteration, this_thread);
   }
}
```

Running hello_world.c

```
% setenv OMP_NUM_THREADS 4
% hello_world
4 threads
Iteration 0, thread 0: Hello, world!
Iteration 1, thread 1: Hello, world!
Iteration 3, thread 3: Hello, world!
Iteration 2, thread 2: Hello, world!
% hello_world
4 threads
Iteration 2, thread 2: Hello, world!
Iteration 1, thread 1: Hello, world!
Iteration 0, thread 0: Hello, world!
Iteration 3, thread 3: Hello, world!
```

Create a second simple program helloWorld.c in OpenMP

The following code will tell thread zero to display how many threads there are and leave the other threads to say hello world. See the output in the next slide.

```
#include <stdio.h>
#include <omp.h>  /* Including OpenMp Library routines */
   int main(){
     omp_set_num_threads(5);  /* 5 threads specified by OpenMP call function */
     int numThreads, tNum;
    /*  Start of Parallel Section. Set tNum as private variable to avoid race condition.  */
     #pragma omp parallel private(tNum)
     {
          tNum = omp_get_thread_num(); /* function call to get the individual thread numbers */
          if(tNum == 0) {
             numThreads = omp_get_num_threads(); /* function call to get the total number of threads */
             printf("Hello World! I am thread %d. There are %d threads.\n", tNum,numThreads);
          }
             else {
             printf("Hello World from thread %d.\n", tNum);
          }
     }
     /*  End of Parallel Section */
return (0);
}
```

helloWorld.c output

Hello World! I am thread 0. There are 5 threads.
Hello World from thread 1.
Hello World from thread 2.
Hello World from thread 3.
Hello World from thread 4.
Hello World from thread 5.

An application of OpenMP for finding an area under a curve

Using everything learned in this module, let's try to find the area under the x^2 curve, in the domain X[0, 1].
Below we have the serial code that we rewrite to the parallel code in OpenMP by simply adding one directive in the line above of the for loop.
#pragma omp parallel for private(var) reduction(operation:var)
(note: The directives should be all in one line)

The two new directives on this line are:

for: This part of the directive tells the compiler that the next line is a for loop and then it divides the loop into equal parts for each thread, but the order that the loop is performed is ***nondeterministic***, in other words, it ***does not*** perform the loop in ***chronological order***.

reduction: Creates a private copy of each variable for each thread. After the parallel section has ended, all the copies are combined using the operator specified, such as "+"(addition), "-"(subtraction), "*"(multiplication), etc.

An application of OpenMP for finding an area under a curve

i = 0,1,2, …., N-1. dx = 1.0/(N-1). x = i*dx.

y=x*x. area(i) = y*x=y*i*dx.

RS=area(0)+area(1)+area(2)+……+area(N-1)

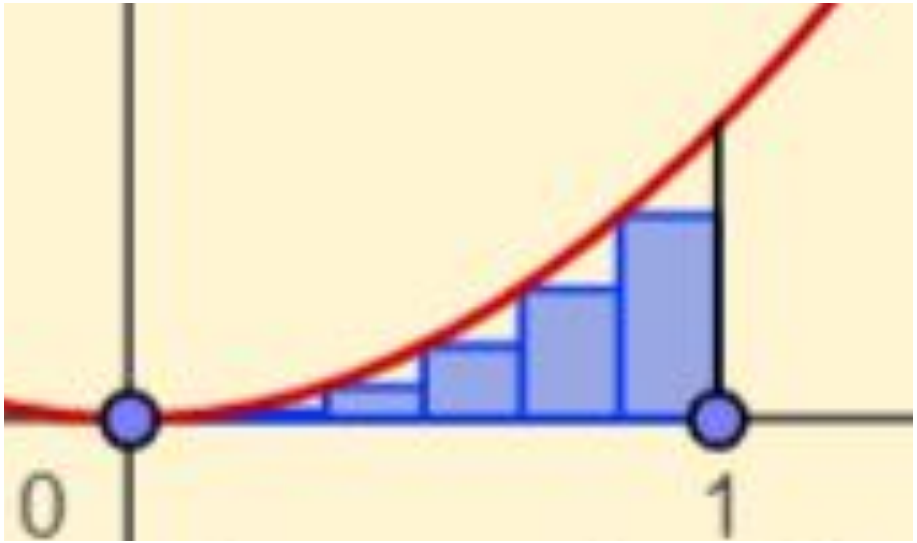


Fig. 1. Using 6 rectangles to find a Left Riemann sum

The Serial Code (integration.c)

```
#include <stdio.h>

float f(float x) {
return (x*x);
}
  int main() {
     int i, SECTIONS = 1000;
     float area = 0.0, y = 0.0, x = 0.0;
     float dx = 1.0/(float)SECTIONS;
          for( i = 0; i < SECTIONS; i++){
               x = i*dx;
               y = f(x);
               area += y*dx;
           }
  printf("Area under the curve is %f\n",area);
  return (0);
  }
```

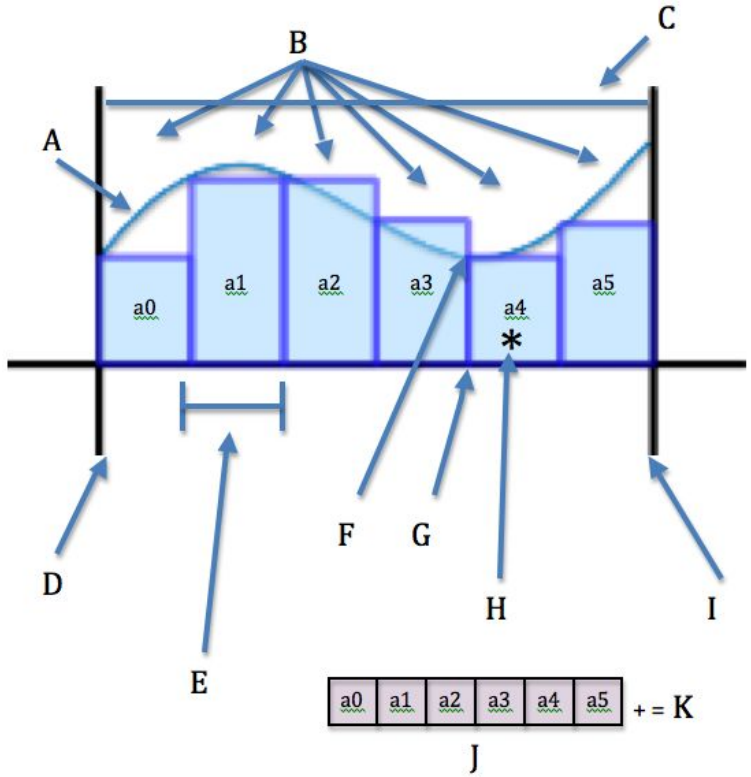


Fig. 2 A picture for the Left Riemann Sum Algorithm

The parallel Code (integration_omp.c)

```
#include <stdio.h>
#include <omp.h>  /* Including OpenMp Library routines */

float f(float x) {
return (x*x);
}
  int main() {
     int i, SECTIONS = 1000; ;
     float area = 0.0, y = 0.0, x = 0.0;
     float dx = 1.0/(float) SECTIONS;;
    /* Start of the parallel section */
     #pragma omp parallel for private(x, y)  reduction(+: area)
          for( i = 0; i < SECTIONS; i++){
               x = i*dx;
               y = f(x);
               area += y*dx;
             }
   /* End of  the parallel section */
  printf("Area under the curve is %f\n",area);
  return (0);
  }
```

Scalability of OpenMP for calculating an area under a curve

Why do parallelism?

- **Speedup** – solve a problem faster.
- **Accuracy** – solve a problem better.
- **Scaling** – solve a bigger problem.
- **Strong scaling** – increasing the number of processes but keeping the problem size constant.
- **Weak scaling** – increasing the problem size as the number of processes increases.

As exercises, the students need to compile, run, and check the scalability of the parallel OpenMP code integration_omp.c compare to the serial code integration.c.

Scalability of OpenMP for calculating an area under a curve

In order to do the scaling, at the beginning of the code the students need to set the timer for measuring the time by adding this line:
double start_time = omp_get_wtime();

At the end of the code, the students need to get the elapsed time by adding this line:
double elapsed_time = omp_get_wtime() - start_time;
printf("%d threads with execution time: %1e seconds. \n",omp_get_thread_num(),elapsed_time);

Let T_N be the time to completion using N threads. The strong scaling efficiency is measured by

Strong efficiency = $T1/(N*T_N)$.

Let T1 be the time to solve a problem of size M on a single thread and T_N be the time to solve a problem of size NM using N threads, the weak scaling is measured by

Week efficiency = $T1/T_N$.

Acknowledgements

This module adapted these slides:

1. Henry Neeman, Supercomputing in Plain English: Shared Memory Multithreading, Presentation #5: Shared Memory Multithreading
2. Tiago Sommer Damasceno, Parallel Programming Using OpenMP. http://shodor.org/media/content//petascale/materials/UPModules/openMP/openMP_Module_pdf.pdf
3. Parallelization: Area under a curve. http://www.shodor.org/petascale/materials/UPModules/AreaUnderCurve/

More about OpenMP

To learn more about OpenMP, here is a list of website that can be useful use to learn more about OpenMP.

http://openmp.org/

http://en.wikipedia.org/wiki/OpenMP

https://computing.llnl.gov/tutorials/openMP/

These sites include good summaries of all OpenMP's directives, functions, environment variables.

http://www.openmp.org/mpdocuments/OpenMP3.0-SummarySpec.pdf

http://www.openmp.org/mpdocuments/OpenMP3.0-FortranCard.pdf