CS 596: Introduction to Parallel Computing
Lecture 19: Shared Memory Programming with OpenMP

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 11/06/14
Last Update: 12/01/14
1 Introduction to OpenMP
   • Hello World Example
2 Trapezoidal Rule with OpenMP
3 Variable Scope
4 Reduction Clause
5 Next Time
What is OpenMP?

- An API for shared-memory parallel programming.
- MP = multiprocessing
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPUs, all of which have access to main memory
**What is OpenMP?**

- OpenMP grew out of the need to standardize different vendor specific directives related to parallelism.
- Pthreads not scaleable to large systems and does not support incremental parallelism very well.
- Correlates with evolution of hybrid architectures: shared memory and multi PE architectures being developed in early '90s.
- Structured around parallel loops and was meant to handle dense numerical applications.

Source: https://computing.llnl.gov/tutorials/openMP
OpenMP is an implementation of *multithreading*:
- Method of parallelizing where a master thread forks a specified number of slave threads
- Tasks are divided among them.
- Threads run concurrently.

Shared memory architecture
Introduction to OpenMP

Features & Challenges (1)

- Portable, threaded, shared-memory programming specification with light syntax
- Exact behavior depends on OpenMP implementation!
- Requires compiler support (C or Fortran)
- Allows programmer to define and separate serial and parallel regions
- Does not "detect" parallel dependencies or guarantee speedup
Features & Challenges (2)

- **Features**
  - Can use OpenMP to parallelize many serial for loops with only small changes to the source code.
  - Task parallelism.
  - Explicit thread synchronization.
  - Standard problems in shared-memory programming

- **Challenges**
  - Cache memories
  - dealing with serial libraries
  - thread safety
#pragma

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that are not part of the basic C specification.
- OpenMP determines how many threads to create, how to synchronize and destroy.
- Compilers that don't support the pragmas ignore them.
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4

compiling

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

possible outcomes

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4

running with 4 threads
running OpenMP Hello World

[mthomas]$
[mthomas@tuckoo ch5]$ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
OR
[mthomas@tuckoo ch5]$ mpicc -g -Wall -fopenmp -o omp_hello omp_hello.c

[mthomas@tuckoo ch5]$ ./omp_hello 10
Hello from thread 6 of 10
Hello from thread 4 of 10
Hello from thread 5 of 10
Hello from thread 0 of 10
Hello from thread 1 of 10
Hello from thread 7 of 10
Hello from thread 2 of 10
Hello from thread 3 of 10
Hello from thread 9 of 10
Hello from thread 8 of 10
OpenMP pragmas

- # pragma omp parallel
  - Most basic parallel directive.
  - The number of threads that run the following structured block of code is determined by the run-time system.
Running OpenMP Hello World

- `#pragma` is first OpenMP directive.
- Used to start threads running thread function `Hello`.
- `num_threads(thread_count)` is an OpenMP clause similar to the Pthread command:
  
  ```c
  pthread_create(&thread_handles[i], NULL, Thread_work, (void*) i);
  ```

- Pthreads required much more work

```c
#pragma omp parallel num_threads(thread_count)
Hello();
```
```bash
gcc -g -Wall -fopenmp -oomp_helloomp_hello.c
../omp_hello 4
```

- compiling
- running with 4 threads

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```
OpenMP pragmas

- `# pragma omp parallel`
  - Most basic parallel directive.
  - The number of threads that run the following structured block of code is determined by the run-time system.
OpenMP pragma directives

`#pragma omp atomic` Identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.

`#pragma omp parallel` Defines a parallel region to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive.

`#pragma omp for` Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel. `#pragma omp parallel for` Shortcut combination of `omp parallel` and `omp for` pragma directives, used to define a parallel region containing a single for directive.

`#pragma omp ordered` Work-sharing construct identifying a structured block of code that must be executed in sequential order.

`#pragma omp section, #pragma omp sections` Work-sharing construct identifying a non-iterative section of code containing one or more subsections of code that should be run in parallel.

`#pragma omp parallel sections` Shortcut combination of `omp parallel` and `omp sections` pragma directives, used to define a parallel region containing a single sections directive.

`#pragma omp single` Work-sharing construct identifying a section of code that must be run by a single available thread.

`#pragma omp master` Synchronization construct identifying a section of code that must be run only by the master thread.

`#pragma omp critical` Synchronization construct identifying a statement block that must be executed by a single thread at a time.

`#pragma omp barrier` Synchronizes all the threads in a parallel region.

`#pragma omp flush` Synchronization construct identifying a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.

`#pragma omp threadprivate` Defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.
A process forking and joining two threads

Diagram showing a process divided into two threads.
Hello World Example
Of note...

- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn’t guarantee that this will actually start thread_count threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we’re trying to start a lot of threads, we will almost always get the desired number of threads.
Some Notes

- In OpenMP parlance the collection of threads executing the parallel block, the original thread and the new threads, is called a *team*.
- The original thread is called the *master*.
- Additional threads are called *slaves*.
- The master starts p-1 new threads.
- Implicit barrier: formed after the hello thread – all threads must return to this point in the code.
- All threads share STDIO.
In case the compiler doesn’t support OpenMP

```c
#include <omp.h>
#ifdef _OPENMP
#include <omp.h>
#endif
```
In case the compiler doesn’t support OpenMP

```c
#ifdef _OPENMP
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
#else
    int my_rank = 0;
    int thread_count = 1;
#endif
```

Use `get_thread` and `get_num_threads` to get ranks and number of PE’s
THE TRAPEZOIDAL RULE
The trapezoidal rule
Scope in OpenMP

- A variable that can be accessed by all the threads in the team has shared scope.

- A variable that can only be accessed by a single thread has private scope.

- The default scope for variables declared before a parallel block is shared.
THE REDUCTION CLAUSE
Serial algorithm

/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
A First OpenMP Version

1) We identified two types of tasks:
   a) computation of the areas of individual trapezoids, and
   b) adding the areas of trapezoids.

2) There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1b.
A First OpenMP Version

3) We assumed that there would be many more trapezoids than cores.

- So we aggregated tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).
Assignment of trapezoids to threads
### Trapezoidal Rule with OpenMP

Results in a race condition

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>global_result = 0 to register</td>
<td>finish my_result</td>
</tr>
<tr>
<td>1</td>
<td>my_result = 1 to register</td>
<td>global_result = 0 to register</td>
</tr>
<tr>
<td>2</td>
<td>add my_result to global_result</td>
<td>my_result = 2 to register</td>
</tr>
<tr>
<td>3</td>
<td>store global_result = 1</td>
<td>add my_result to global_result</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>store global_result = 2</td>
</tr>
</tbody>
</table>

Unpredictable results when two (or more) threads attempt to simultaneously execute:

```
global_result += my_result;
```
Mutual exclusion

```c
#pragma omp critical
global_result += my_result;
```

only one thread can execute
the following structured block at a time

critical directive tells compiler that system needs to provide
mutually exclusive access control for the block of code.
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double global_result = 0.0; /* Store result in global_result */
    double a, b; /* Left and right endpoints */
    int n; /* Total number of trapezoids */
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
    #pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
    return 0;
} /* main */
```
void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    # pragma omp critical
    *global_result_p += my_result;
}    /* Trap */
Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.
Scope in OpenMP

- A variable that can be accessed by all the threads in the team has **shared** scope.

- A variable that can only be accessed by a single thread has **private** scope.

- The default scope for variables declared before a parallel block is **shared**.
for C, variables defined in *main* have global; variables defined in a *function* have function scope.

for OpenMP: the scope of a variable is associated with the set of threads that can access the variable in a parallel block.

**shared scope:**
- the default scope for variables defined outside a parallel block
- e.g. *global_results* was declared in *main*, so it is shared by all threads

**private scope:**
- a variable that can only be accessed by a single thread
- The default scope for variables declared inside a parallel block is private (e.g. all vars in defined in Trap).
int main(int argc, char* argv[]) {
    /* Store result in global_result */
    double global_result = 0.0;
    /* Left and right endpoints */
    double a, b;
    int n; /* Total number of trapezoids*/
    int thread_count;
    if (argc != 2) Usage(argv[0]);
    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
    if (n % thread_count != 0) Usage(argv[0]);
    # pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
    return 0;
} /* main */

* Function: Trap
* Purpose: Use trapezoidal rule to
* estimate definite integral
* Input args:
*   a: left endpoint
*   b: right endpoint
*   n: number of trapezoids
*   global_result_p: pointer to global trap sum
* Output arg:
*   integral: estimate of integral from a to b of f(x)
*/
void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;
    # pragma omp critical
    *global_result_p += my_result;
} /* Trap */
Reduction Clause

OpenMP: Reduction Clause

We need this more complex version to add each thread’s local calculation to get global_result.

```c
void Trap(double a, double b, int n, double* global_result_p);
```

Although we’d prefer this.

```c
double Trap(double a, double b, int n);
```

global_result = Trap(a, b, n);
If we use this, there’s no critical section!

```c
double Local_trap(double a, double b, int n);
```

If we fix it like this...

```c
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
    #pragma omp critical
    global_result += Local_trap(double a, double b, int n);
}
```

...we force the threads to execute sequentially.

**Local Trap does not have reference to the global variable global_result**
We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

```c
    global_result = 0.0;
    # pragma omp parallel num_threads(thread_count)
    {
        double my_result = 0.0; /* private */
        my_result += Local_trap(double a, double b, int n);
        # pragma omp critical
        global_result += my_result;
    }
```

Notes: the call to Local_Tramp is inside the parallel block, but outside critical section; my_result is private to each thread
Reduction operators

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.
A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```

```
global_result = 0.0;
#pragma omp parallel num_threads(thread_count) 
  reduction(+: global_result)
global_result += Local_trap(double a, double b, int n);
```
A few comments

- OpenMP (1) creates private thread variable, (2) stores result for thread, and (3) creates critical section block.
- Subtraction ops are not guaranteed (not associative or commutative):
  
  $\text{result} = 0$
  
  $\text{for } (i = 1; \quad i \leq 4; \quad i++)$
  
  $\text{result} -= i$

- Floating point arithmetic is not associative, so results are not guaranteed:
  
  $a + (b + c)$ may not equal $(a + b) + c$
Next Time

- Next class: 11/13/14
- Quiz 2, Part 2 (MPI) Due: 11/13/14.
- HW #4 (Pthreads) Due: 11/25/14
- Quiz 3: Delayed