Lecture 28: OpenMP: Trap Function, Var Scope, Reduction

Mary Thomas

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

Presented: 04/06/16
Updated: 04/06/16
Table of Contents

1. Trapezoidal Rule with OpenMP
2. Variable Scope
3. Reduction Clause
The Trapezoid Rule for Numerical Integration

Solve the Integral: \( \int_{a}^{b} F(x) \, dx \)

The Trapezoidal Rule

Where \( F(x) \) can be any function of \( x \): \( f(x^2), f(x^3) \).
See Pacheco IPP (2011), Ch3.
Trapezoid Equations

Integral: \[ \int_{a}^{b} f(x) \, dx \]

Area of 1 trapezoid: \[ \frac{h}{2} \left[ f(x_i) + f(x_{i+1}) \right] \]

Base: \[ h = \frac{b-a}{n} \]

Endpoints: \[ x_0 = a, \quad x_1 = a + h, \quad x_2 = a + 2h, \ldots, \quad x_{n-1} = a + (n - 1) h, \quad x_c = b \]

Sum of Areas: \[ \text{Area} = h \left[ \frac{f(x_0)}{2} + f(x_{i+1}) + f(x_{i+1}) + \ldots + f(x_{n-1}) \frac{f(x_n)}{2} \right] \]
Trapezoid Problem: Serial Algorithm

/* Input: a, b, n */
h = (b-a)/n;
approx = (F(a) + F(b))/2.0
for (i=0; i<= n-1; i++) {
    x_i = a + i*H;
    approx += f(x_i);
}
approx = h* approx
Parallelizing the Trapezoidal Rule

**PCAM Approach**

- Partition problem solution into tasks.
- Identify communication channels between tasks.
- Aggregate tasks into composite tasks.
- Map composite tasks to cores.
Two types of tasks:
- Compute area of 1 trapezoid
- Compute area sums
First OpenMP Version of the Trap Alg.

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.

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

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

```c
global_result += my_result;
```

Results in a race condition
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 = \%le\n", 
            a, b, global_result);
    return 0;
} /* main */
```
```c
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 */

#pragma omp parallel num_threads(thread_count)
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 */
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);
```

```c
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
  #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_Trap 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.

\[ \text{reduction}(<\text{operator}>: <\text{variable list}>) \]

\[
\begin{align*}
global\_result & = 0.0; \\
\# \text{pragma omp parallel num_threads(thread\_count) } \\
& \quad \text{reduction}(+: global\_result) \\
global\_result & += \text{Local\_trap}(\text{double } a, \text{double } b, \text{int } n);
\end{align*}
\]
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):
  
  \[
  \text{result} = 0;
  \]
  
  \[
  \text{for } (i = 1; i \leq 4; i++)
  \]
  
  \[
  \text{result} -= i;
  \]

- Floating point arithmetic is not associative, so results are not guaranteed:

  \[
  a + (b + c) \text{ may not equal } (a + b) + c
  \]