Architecture of Parallel Computers

Program 2: OpenMP Programming

1. Run the following commands, describe what you observe and explain why it happens:

./trap-omp

```
OMP defined, threadct = 1
With n = 1048576 trapezoids, our estimate of the integral from 0 to 3.14159 is 2
```

./trap-omp 2

```
OMP defined, threadct = 2
With n = 1048576 trapezoids, our estimate of the integral from 0 to 3.14159 is 2
```

./trap-omp 4

```
OMP defined, threadct = 4
With n = 1048576 trapezoids, our estimate of the integral from 0 to 3.14159 is 2
```

./trap-omp 8

```
OMP defined, threadct = 8
With n = 1048576 trapezoids, our estimate of the integral from 0 to 3.14159 is 2
```

./trap-omp 16

```
OMP defined, threadct = 16
With n = 1048576 trapezoids, our estimate of the integral from 0 to 3.14159 is 2
```

Observations:

Output irrespective of the number of threads used while computation ***is equal to 2.***

Why does this happen?

The result is consistent irrespective of the number of threads being read as the input. This output is however, not due to the successful parallel execution because the code as it stands is actually sequential. No OpenMP parallel directives are applied to the loop that calculates the integral. The loop calculating the integral runs in a sequential manner in every case, regardless of the thread count specified. This is because the updated value of the '***threadct=X'*** variable is not being read anywhere and is only being updated and the code runs the same way every time.

2. Add the following preprocessor pragma directive, just before the **for** loop, which starts at line 33. Run the commands in part 1 again, describe what you observe and explain why it happens:

```
#pragma omp parallel for num_threads(threadct) shared (a, n, h, integral) private(i)
```

./trap-omp

```
OMP defined, threadct = 1
With n = 1048576 trapezoids, our estimate of the integral from 0 to 3.14159 is 2
```

./trap-omp 2

```
OMP defined, threadct = 2
With  n  =  1048576  trapezoids,  our  estimate  of  the  integral  from  0  to  3.14159  is
1.73905
```

./trap-omp 4

```
OMP defined, threadct = 4
With  n  =  1048576  trapezoids,  our  estimate  of  the  integral  from  0  to  3.14159  is
1.05078
```

./trap-omp 8

```
OMP defined, threadct = 8
With  n  =  1048576  trapezoids,  our  estimate  of  the  integral  from  0  to  3.14159  is
0.45028
```

./trap-omp 16

```
OMP defined, threadct = 16
With  n  =  1048576  trapezoids,  our  estimate  of  the  integral  from  0  to  3.14159  is
0.239156
```

Observations:

The output of the integral is only calculated correctly when the program is run with a single thread. For the rest of the inputs the output seems to progressively deviate from the actual value of the integral. Another observation is that the output/estimated value of the integral is decreasing as we increase the number of threads for execution.

Why does it happen?

Race condition. The discrepancy for the output values for different thread counts occurs due to a race condition caused by the incorrect handling of the shared variable **'integral'** in the parallel section of the code.

The **‘integral’** variable is shared among all threads due to the **‘shared’** clause in the OpenMP directive. This shared access leads to a situation where multiple threads are trying to update the **‘integral’** variable simultaneously without proper synchronization. When this happens, some updates are lost and one thread’s update can overwrite another’s before it is added to the total sum resulting the incorrect result being computed. And this condition only seems to worsen as the number of threads increases.

---

3. Try to fix trap-omp.c using three kinds of OpenMP primitives, viz., reduction, atomic and critical. Here is a reference you can turn to: https://computing.llnl.gov/tutorials/openMP/ Put your fixed trap-omp.c into its corresponding folder.

Code updated in:

```
reduction/trap-omp.c
atomic/trap-omp.c
critical/trap-omp.c
```

How to run the code?

```
In the code directory

make
make run
```

---

4. For each strategy, you are asked to measure and compare the performance. The performance can be measured in the following way. Put the `omp_get_wtime` calls in the indicated places:

```
// Starting the time measurement
double start = omp_get_wtime();

// Computations to be measured
...
// Measuring the elapsed time double
end = omp_get_wtime();

// Time calculation (in seconds)
double elapsed_time = end – start;

// Print out the elapsed_time
```

Put the elapsed time into a table after running with various numbers of threads. Which one is the fastest? Explain why it produces the best performance.

Number of Threads	**Reduction**	**Atomic**	**Critical**
2	*0.0288152*	0.0778597	0.187493
4	*0.0204421*	0.0910039	0.242074
8	*0.0140793*	0.105711	0.565866
16	*0.00828635*	0.112208	0.624393

Reduction is the fastest. In all the four cases for different number of threads being used in execution the elapsed time used in **Reduction** is the least always and is thus producing the best performance in this scenario.

Comparison by Elapsed Time (lowest to highest): ***Reduction, Atomic, Critical***

Why?

The performance difference between using the *reduction, atomic and critical OpenMP primitives* can attributed to how each approach manages concurrent access and updates the shared resources.

In, (i) Reduction, each thread gets its own private copy of the ***integral*** variable. The threads perform their calculations independently and without any need for synchronization or waiting for access to shared resources. OpenMP combines these independent results into a single value only after all threads have completed their calculations. This approach is effective because it minimizes any overhead that is related to thread synchronization and avoids any contention in between the threads. All of these factors combined are why the reduction method performs the best across different number of threads.

Why the atomic and critical primitives fail to deliver?

In, (ii) Atomic, each thread must wait for their turn to perform an update operation which introduces delays as the number of threads increases. The Atomic directive performs badly because of the waiting time and also because the of the overhead of ensuring atomicity.

In, (iii) Critical, the primitive makes sure that only one thread can execute the enclosed code block at a time. While preventing any race conditions, it does increase the cost of parallel efficiency, forcing threads to queue up one after the other and update the value one by one.

As a comparison, Reduction performs the best because it allows for maximum parallelism with minimal overhead. Specifically in cases where calculating the sum across many iterations is required, Reduction will generally perform the best as shown by the results above.