```c
//The answer from this computation should be 2.0.
#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP


/* Demo program for OpenMP: computes trapezoidal approximation to an integral*/
const double pi = 3.14159265358979323846264338307 9;

int main(int argc, char** argv) {
/* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a) / n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;

double f(double x);

/* parse command-line arg for number of threads */
if (argc > 1) {
threadcnt = atoi(argv[1]);
}

#ifdef _OPENMP
omp_set_num_threads( threadcnt );
printf("OMP defined, threadct = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif

integral = (f(a) + f(b))/2.0;
int i;

#pragma omp parallel for private(i) shared (a, n, h, integral)
for(i = 1; i < n; i++) {
integral += f(a+i*h);
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n", n);
printf("%f to %f is %f\n", a,b,integral);
}

double f(double x) {
return sin(x);
}
```

```c
//The answer from this computation should be 2.0.
#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP


/* Demo program for OpenMP: computes trapezoidal approximation to an integral*/

const double pi = 3.14159265358979323846264338307 9;

int main(int argc, char** argv) {
/* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a) / n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;

double f(double x);

/* parse command-line arg for number of threads */
if (argc > 1) {
threadcnt = atoi(argv[1]);
}

#ifdef _OPENMP
omp_set_num_threads( threadcnt );
printf("OMP defined, threadct = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif

integral = (f(a) + f(b))/2.0;
int i;

#pragma omp parallel for \
private(i) shared (a, n, h) reduction(+: integral)
for(i = 1; i < n; i++) {
integral += f(a+i*h);
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n", n);
printf("%f to %f is %f\n", a,b,integral);
}

double f(double x) {
return sin(x);
}
```

2.0 Integration using the trapezoidal rule

- nano trap-notworking.c
  - code is copied and pasted

Comments:

- "omp parallel" indicates this is an OpenMP pragma for parallelizing the following for loop. The OpenMP system will divide 2^20 (1048576) into the threadct segments, which will be executed in parallel on multiple cores.
- OpenMP clause
  - num_threads(threadct) specifies number of threads used in parallelization
- Clauses in the second line indicate whether the variables that appear in the for loop will be "private" (each thread gets its own copy, used by a single thread) or "shared" (all threads use the same copy of the variable in the memory, shared with other threads). Variable "i" is private and variables (a, n, h, integral) are shared.

- nano trap-working.c
  - code is copied and pasted

Comments

- A backslash character (\) before the end of the first line will cause the two lines to be treated as a single pragma.
- The variables listed are globally shared by all threads, and only the loop control variable "i" is local to each particular thread.
- "integral" is the accumulator variable uses the reduction clause
- The "+" in the reduction clause indicates the variable "integral" is computed by adding values together in the loop.

```
pi@raspberrypi:~ $ gcc trap-notworking.c -o trap-notworking -fopenmp
/tmp/cctJsHme.o: In function `f':
trap-notworking.c:(.text+0x17c): undefined reference to `sin'
collect2: error: ld returned 1 exit status
pi@raspberrypi:~ $ gcc trap-notworking.c -o trap-notworking -fopenmp -lm
pi@raspberrypi:~ $ gcc trap-working.c -o trap-working -fopenmp -lm
```

- gcc trap-notworking.c -o trap-notworking -fopenmp **-lm**
- gcc trap-working.c -o trap-working -fopenmp **-lm**
  - need to link with math library using (-lm) to make executable

2.2 DO THIS: Edit, Compile and Run the code

- attempt to compute a Calculus value, the "trapezoidal approximation of using 220 equal subdivisions."

```
pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.376029
pi@raspberrypi:~ $ ./trap-working 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
```

- ./trap-notworking 4 (4 threads)
  - OMP defined, threadct = 4 With 1048576 trapezoids, our estimate of the integral from 0.000000 to 3.141593 is 1.390723
- ./trap-working 4 (4 threads)
  - OMP defined, threadct = 4 With 1048576 trapezoids, our estimate of the integral from 0.000000 to 3.141593 is 2.000000

2.3 Data Decomposition: "equal chunks"

- There was no explicit static nor dynamic clause on the pragma lines (37-38). This cause the behavior of the code to perform consecutive iterations of the loop that are assigned equal amounts of work. Since there are four threads, thread 0 will do the first 2^20 (1048576)/4 trapezoids, thread 1 will do the next 2^20 (1048576)/4 trapezoids, and so on.

```c
/* barrier.c
 * ... illustrates the use of the OpenMP barrier command,
 *      using the commandline to control the number of threads...
 *
 * Joel Adams, Calvin College, May 2013.
 *
 * Usage: ./barrier [numThreads]
 *
 * Exercise:
 * - Compile & run several times, noting interleaving of outputs.
 * - Uncomment the barrier directive, recompile, rerun,
 *    and note the change in the outputs.
 */

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    printf("\n");
    if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
    }

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        printf("Thread %d of %d is BEFORE the barrier.\n", id, numThreads);

//        #pragma omp barrier

        printf("Thread %d of %d is AFTER the barrier.\n", id, numThreads);
    }

    printf("\n");
    return 0;
}
```

3.0 Coordination: Synchronization with a Barrier

- nano barrier.c
  - ensures all threads completes a parallel section of code before the execution goes on. May be necessary when threads generate computed data that needs to be completed for use in another computation.

- gcc barrier.c -o barrier -fopenmp
  - Makes the program executable

```
pi@raspberrypi:~ $ ./barrier

Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.

pi@raspberrypi:~ $ ./barrier

Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.

pi@raspberrypi:~ $ ./barrier

Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.

pi@raspberrypi:~ $ ./barrier

Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
```

3.1 DO THIS: Look at some code

- ./barrier
  - Note: with the commented pragma omp barrrier the threads are in pairs for their before and after barrier execution

```
pi@raspberrypi:~ $ ./barrier

Thread 0 of 4 is BEFORE the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 2 of 4 is AFTER the barrier.

pi@raspberrypi:~ $ ./barrier

Thread 0 of 4 is BEFORE the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.

pi@raspberrypi:~ $ ./barrier

Thread 1 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 0 of 4 is AFTER the barrier.

pi@raspberrypi:~ $ ./barrier

Thread 2 of 4 is BEFORE the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 0 of 4 is AFTER the barrier.
```

- ./barrier
  - Note: without the commented pragma omp barrrier the threads are complete a parallel section of code for “before the barrier” before completing the “after the barrier”.

```c
/* masterWorker.c
 * ... illustrates the master-worker pattern in OpenMP
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./masterWorker
 *
 * Exercise:
 * - Compile and run as is.
 * - Uncomment #pragma directive, re-compile and re-run
 * - Compare and trace the different executions.
 */

#include <stdio.h>   // printf()
#include <stdlib.h>  // atoi()
#include <omp.h>     // OpenMP

int main(int argc, char** argv) {
    printf("\n");
    if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
    }

//    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();

        if ( id == 0 ) {  // thread with ID 0 is master
            printf("Greetings from the master, # %d of %d threads\n",
                            id, numThreads);
        } else {          // threads with IDs > 0 are workers
            printf("Greetings from a worker, # %d of %d threads\n",
                            id, numThreads);
        }
    }

    printf("\n");

    return 0;
}
```

```
pi@raspberrypi:~ $ ./masterWorker

Greetings from the master, # 0 of 1 threads
```

```
pi@raspberrypi:~ $ ./masterWorker

Greetings from a worker, # 3 of 4 threads
Greetings from a worker, # 2 of 4 threads
Greetings from a worker, # 1 of 4 threads
Greetings from the master, # 0 of 4 threads
```

4.0 Program Structure: The Master-Worker Implementation Strategy

- nano masterWorker.c
  - One thread (master) that executes a block of code when it forks, while of the rest of the threads (workers) execute a different block of code when they fork.

4.1 DO THIS: Look at some code

- ./masterWorker
  (thread number not specified)
- With commented pragma:
  Greetings from the master, # 0 of 1 threads
- Results in one thread (master) to execute the one block of code (when thread ID is 0)

- ./masterWorker
  (thread number not specified)
- Without commented pragma:
  Greetings from a worker, # 3 of 4 threads
  Greetings from a worker, # 2 of 4 threads
  Greetings from a worker, # 1 of 4 threads
  Greetings from the master, # 0 of 4 threads
- Note: With the number of threads not specified, default thread count is 4. whereas with the pragma, the rest of the threads (workers) execute a different block of code

which is executed before the master to execute the one block of code (once thread ID is 0)