CHAPTER 5

SHARED-MEMORY PROGRAMMING WITH OPENMP

Feel free to contact me for more information:

email: miguelsrochajr@gmail.com
LinkedIn: http://linkedin.com/in/miguelrochajr
YouTube: https://www.youtube.com/channel/UCYViw5AtRsUHMyoB_zljalA

Question 1.

If it's defined, the _OPENMP macro is a decimal int. Write a program that prints its value. What is the significance of the value?

Answer:

The _OPEMP macro is the date which the version of the OpenMP that is being used was released. The date format is *yyyymm*. In my case, the output was 201107 which stands for July of 2011.

Question2.

Download the omp_trap1.c from the book's website, and delete the critical directive. Now compile and run the program with more and more threads and larger and larger values of n. How many threads and how many trapezoids does it take before the result is incorrect?

Answer.

This may vary from system to system. In my case, I did the test with the number of threads ranging from 2 to 8192 and the number of trapezoids ranging from 32 to 8192. I made a slightly modification on the program so that it receives the parameters directly from the command line. The reason for that is that I have used a script to increment. The following script ran the omp_trap1.

```
#!/bin/bash

n=32 #starts with 32
thread_count=2 # minomum threads number
a=2
b=10
N_MAX=8192

echo "The excpected value is around 330.67"

while [[ $n -lt $N_MAX || $n -eq $N_MAX ]]; do
  thread_count=2
  echo "------------ FOR n=$n ----------------"
  while [[ $thread_count -lt n || $thread_count -eq $n ]]; do
    ./omp_trap1 $thread_count $a $b $n
    let "thread_count*=2"
  done
  let "n*=2"
done
```

Script to run omp_trap1

The modification I made, despite of removing the *omp critical* directive, was on on the parameters initialization. The new way is the following.

```
{...}
   thread_count = strtol(argv[1], NULL, 10); //convert string to long int
   a = (double)s
trtol(argv[2], NULL, 10);
   b = (double)strtol(argv[3], NULL, 10);
   n = strtol(argv[4], NULL, 10);
{...}
```

Modified parameter receivement

The script was ran with the command "./script.sh > result.txt" which takes the output and writes onto the file result.txt. In the case the file does not exist, the command creates it. The result was the following.

```
The excpected value is around 330.67
------------ FOR n=32 ----------------
With thread_count = 2 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30750000000000e+02
With thread_count = 4 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30750000000000e+02
With thread_count = 8 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30750000000000e+02
With thread_count = 16 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30750000000000e+02
With thread_count = 32 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30750000000000e+02
------------ FOR n=64 ----------------
With thread_count = 2 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30687500000000e+02
With thread_count = 4 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30687500000000e+02
With thread_count = 8 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30687500000000e+02
With thread_count = 16 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30687500000000e+02
With thread_count = 32 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30687500000000e+02
With thread_count = 64 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30687500000000e+02
------------ FOR n=128 ----------------
With thread_count = 2 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30671875000000e+02
With thread_count = 4 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30671875000000e+02
With thread_count = 8 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30671875000000e+02
With thread_count = 16 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30671875000000e+02
With thread_count = 32 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30671875000000e+02
With thread_count = 64 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30671875000000e+02
With thread_count = 128 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30671875000000e+02
------------ FOR n=256 ----------------
With thread_count = 2 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30667968750000e+02
With thread_count = 4 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30667968750000e+02
With thread_count = 8 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30667968750000e+02
With thread_count = 16 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30667968750000e+02
With thread_count = 32 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30667968750000e+02
With thread_count = 64 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30667968750000e+02
With thread_count = 128 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30667968750000e+02
With thread_count = 256 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30667968750000e+02
------------ FOR n=512 ----------------
With thread_count = 2 threads, our estimate of the integral from 2.000000 to 10.000000 =
```

```
3.30666992187500e+02
With thread_count = 4 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666992187500e+02
With thread_count = 8 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666992187500e+02
With thread_count = 16 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666992187500e+02
With thread_count = 32 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666992187500e+02
With thread_count = 64 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666992187500e+02
With thread_count = 128 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666992187500e+02
With thread_count = 256 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666992187500e+02
With thread_count = 512 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666992187500e+02
------------ FOR n=1024 ----------------
With thread_count = 2 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666748046875e+02
With thread_count = 4 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666748046875e+02
With thread_count = 8 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666748046875e+02
With thread_count = 16 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666748046875e+02
With thread_count = 32 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666748046875e+02
With thread_count = 64 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666748046875e+02
With thread_count = 128 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666748046875e+02
With thread_count = 256 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666748046875e+02
With thread_count = 512 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666748046875e+02
With thread_count = 1024 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.30666748046875e+02
------------ FOR n=2048 ----------------
With thread_count = 2 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666687011719e+02
With thread_count = 4 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666687011719e+02
With thread_count = 8 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666687011719e+02
With thread_count = 16 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666687011719e+02
With thread_count = 32 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666687011719e+02
With thread_count = 64 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666687011719e+02
With thread_count = 128 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666687011719e+02
With thread_count = 256 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666687011719e+02
With thread_count = 512 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666687011719e+02
With thread_count = 1024 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.28463060498238e+02
With thread_count = 2048 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.30666687011719e+02
------------ FOR n=4096 ----------------
With thread_count = 2 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666671752930e+02
With thread_count = 4 threads, our estimate of the integral from 2.000000 to 10.000000 =
```

```
3.30666671752930e+02
With thread_count = 8 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666671752930e+02
With thread_count = 16 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666671752930e+02
With thread_count = 32 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666671752930e+02
With thread_count = 64 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666671752930e+02
With thread_count = 128 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666671752930e+02
With thread_count = 256 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666671752930e+02
With thread_count = 512 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666671752930e+02
With thread_count = 1024 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.30522945702076e+02
With thread_count = 2048 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.28323772065341e+02
With thread_count = 4096 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.30666671752930e+02
------------ FOR n=8192 ----------------
With thread_count = 2 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666667938232e+02
With thread_count = 4 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666667938232e+02
With thread_count = 8 threads, our estimate of the integral from 2.000000 to 10.000000 =
3.30666667938232e+02
With thread_count = 16 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666667938232e+02
With thread_count = 32 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666667938232e+02
With thread_count = 64 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666667938232e+02
With thread_count = 128 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666667938232e+02
With thread_count = 256 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.30666667938232e+02
With thread_count = 512 threads, our estimate of the integral from 2.000000 to 10.000000
= 3.28401535026729e+02
With thread_count = 1024 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.27348529487848e+02
With thread_count = 2048 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.30169853605330e+02
With thread_count = 4096 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.30666667938232e+02
With thread_count = 8192 threads, our estimate of the integral from 2.000000 to
10.000000 = 3.30666667938232e+02
```

With the following values, still with that, I could not achieve an unpredictable behavior.

Question 3

Modify the omp_trap1.c so that.

a. It uses the first block of code on page 222, and

b. the time used by the parallel block is timed using the OpenMP function *omp_get_wtime().* The syntax is.

double omp_get_wtime(void)

It returns the number of seconds that have passed since some time in the past. For details on taking timings, see Section 2.6.4. Also recall that OpenMP has a *barrier* directive:

pragma omp barrier

Now find a system with at least two cores and time the program with.

c. one thread and a large value of n, and

d. two threads and the same value of n.

What happens? Download *omp_trap_2.c* from the book's website. How does its performance compare? Explain your answers.

Answer.

a.) First, the modification was made to the function prototype. The new one at the is the following.

```
double Local_Trap(double a, double b, int n);
```

The modification in the code was done so that the part where the old Trap function was being called now has the following code, based on the one found on the page 222.

```
{...}
#  pragma omp parallel num_threads(thread_count)
  {
#   pragma omp critical
    global_result +=Local_Trap(a, b, n);
  }
{...}
```

The last modification made was at the return value of the Trap function. The previous one was a void function and this returns the my_result, which is a double function.

```
{...}
  for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

   return my_result;
}  /* Trap */
{...}
```

b.)

Now the program will be timed at the parallel part. The following code shows how the time is being taken.

```
   start_time = omp_get_wtime();
#  pragma omp parallel num_threads(thread_count)
  {
#   pragma omp critical
    global_result +=Local_Trap(a, b, n);
  }
   finish_time = omp_get_wtime();
```

c.)

With 1 thread and $n = 10^7$ the time taken was 1.405839 seconds.

```
The time taken by the parallel: 0.020990
msrj@ubuntu:~/Documents/UFRN/ParallelProgramming/BookExercises/Chapter5/Question3$ ./omp_trap1 1
Enter a, b, and n
2 10 100000000
With n = 100000000 trapezoids, our estimate
of the integral from 2.000000 to 10.000000 = 3.30666666666750e+02
The time taken by the parallel: 1.405839
```

d.)

With two threads and the same number of trapezoids, the time taken was 1.410747 seconds.

```
msrj@ubuntu:~/Documents/UFRN/ParallelProgramming/BookExercises/Chapter5/Question3$ ./omp_trap1 2
Enter a, b, and n
2 10 100000000
With n = 100000000 trapezoids, our estimate
of the integral from 2.000000 to 10.000000 = 3.30666666666653e+02
The time taken by the parallel: 1.410747
```

The times were pretty much the same but when the number of threads was increased there was also a increase in time. The reason here is simple: the code inserted is obligating the sequential execution. And with more threads we have a increase in time of overhead.

In the case of the *omp_trap2.c*, the one used here was the omp_*trap2a.c* which contains the correction of the above problem, as mentioned on page 222 of the text book. The correction is simple, it creates a private variable for each of the threads and just sums up the local value of them as the critical section. The following chunk shows the corrected part together with the calls to time the code execution.

```
   start_time = omp_get_wtime();
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;
      my_result += Local_trap(a, b, n);
#     pragma omp critical
      global_result += my_result;
   }
   end_time = omp_get_wtime();
```

The omp_trap2a.c with 1 threads, got the following result. With 1.440586 seconds. Which is not that different from the previous.

```
msrj@ubuntu:~/Documents/UFRN/ParallelProgramming/BookExercises/Chapter5/Question3$ ./omp_trap2a 1
Enter a, b, and n
2 10 100000000
With n = 100000000 trapezoids, our estimate of the integral from 2.000000 to 10.000000 = 3.30666666666750e+02
The time taken by the parallel: 1.440586
```

Now, with two threads we have the following results.

```
msrj@ubuntu:~/Documents/UFRN/ParallelProgramming/BookExercises/Chapter5/Question3$ ./omp_trap2a 2
Enter a, b, and n
2 10 100000000
With n = 100000000 trapezoids, our estimate of the integral from 2.000000 to 10.000000 = 3.30666666666653e+02
The time taken by the parallel: 0.735200
```

With 0.735200 seconds, the program takes pretty much half of the time. All this is due to the implementation shown above. With this, the program is running in actual parallel mode while the previous was being forced to be executed as sequential.

Question 4

Recall that OpenMP creates private variables for reduction variables, and these private variables are initialized to the identity value for the reduction operator. For example, if the operator is addition, the private variables are initialized to 0, while if the operation is multiplication, the private variables are initialized to 1. What are the identity values for the operators &&, ||, &, | and ^?

Answer.

The private variables are initialized such that the operation is not changed. So, for example, the multiplicative identity is 1 and the sum is zero. The following table shows the ones asked.

&&	1
||	0
& (bit to bit operator)	$1.....1111_2$
| (bit to bit operator)	$0....000_2$
^ (Binary XOR Operator)	0

Question 5

Suppose that on the amazing Bleeblon computer, variables with types *float* can store three decimal digits. Also suppose that the Bleebon's floating point operation, the result is rounded to four decimal digits before being stored. Now suppose a C program declare an array *a* as follows:

$$float\ a[]\ =\ \{4.0,\ 3.0,\ 1000.0\};$$

a.) What is the output of the following block of code if it's run on the Bleebon? (sequential code)

b.) Now consider the following code: (parallel code)

Answer.

The way floating points are stored in the computer follows the scientific notation. Then, our array *a* is in the memory as

a[] = { 4.00e+00, 3.0e+00, 3.0e+00, 1.00e+03 }

Note here that we used 3 digits to store the base number, as described by the Bleebon's computer.

When the values are stored in the floating points registers, an additional digit is added. Such that the representation is the following.

a[] = { 4.000e+00, 3.000e+00, 3.000e+00, 1.000e+03 }

a.)

In the case of the sequential for loop, we will have four iterations as described on the following table.

i = 0	sum = 4.00e+00
i = 1	sum = 7.00e+00
i = 2	sum = 1.00e+01
i = 3	sum = 1.01e+03

An important thing to notice here is when i=3. We have 10 + 1000, which is 1010 in decimal digits. When the variables are used during computation, the computer stores the value as 1.010e+03. However, when finishes the calculation, the value is truncated to 3 digits and the output of the program is

sum = 1010.0

The formatting is due to the %4.1, which stands for print at least four digits before the '.' and with 1 decimal digit. In the case of a variable with 3 digits before the dot, there will be a padding to complement the formatting. Source: http://www.cplusplus.com/reference/cstdio/printf/

b.)

In the case of the parallel for loop with two threads, we must first recall that the run-time system will create private variables for the threads. So, in this case, if we call the private variable of the thread 0 **local_sum0** and **local_sum1** the private one for the thread 1, after each thread finish their jobs, we would have the following for the thread 0.

Thread 0
local_sum0 = 7.00e+00

The thread 1 will have a little bit different behavior. Recall that, according to the question, after each operation, the result is rounded with 3 digits to be stored in the floating main memory. During the calculations, the result of the **local_sum1** is 1003, which the computer stores in the registers as the following.

Thread 1
local_sum1 = 1.003e+10

Then, after the operation, the value must be store to the main memory which can handle only 3 digits. This will cause the 3 to be rounded down and the final result to be as the following.

```
local_sum1 = 1.00e+10
```

At the end, the two private variables are added together and the output of the program will be:

sum = 1010.0

Question 6

Write an OpenMP program that determines the default scheduling of parallel for loops. Its input should be the number of iterations, the output might be:

Thread 0: Iterations 0 - 1
Thread 1: Iterations 2 - 3

Answer.

The core of the algorithm is on the vector *min* and *max*. They represent the min and max iterations for each thread. The thread is identified by ids ID. So, in the case of *n=4* and *thead_count=2* we would have the following result.

Vector	Index (thread rank)	Value
min	0	0
	1	2
max	0	1
	1	3

This means that the thread 0 has the iterations 0 - 1 and thread 1 has those from 2 to 3.

The function that does this is the Schedule function presented below.

```
void Schedule(int thread_count, int n, int max[], int min[]) {
   int  i;
   for (i = 0; i < thread_count; i++) {
      min[i] = n;
      max[i] = 0;
   }
#  pragma omp parallel num_threads(thread_count) \
      default(none) private(i) shared(min, max, n)
   {
   int my_rank = omp_get_thread_num();
#     pragma omp for
      for (i = 0; i < n; i++) {
#        ifdef DEBUG
         printf("Th %d > iteration %d\n", my_rank, i);
#        endif
         if (i < min[my_rank])
```

```
            min[my_rank] = i;
         if (i > max[my_rank])
            max[my_rank] = i;
      }
   }
```

To breakdown the code above, let's first understand the first pragma. First, It creates a number of *thread_count* threads. Right below, the default(none) keyword makes the compiler require that we specify the scope of each variable we use in the block and that has been declared outside of it. This is a good OpenMP programming practice. And, because of that, we declare that each thread has its own private copy of *i* with the clause *private(i)*. Also, the shared(min, max, n) says that each thread will have the reference to the variable min, max and n from the program's stack and not from each thread stack. This is completely fine since each thread will only change certains positions of the vector, making the access of them mutually exclusive.

The if and else statements aim to get which iterations will be stored on the vectors max and min.

After that, the vectors are printed onto the screen with a function called PrintResult. The whole code is shown below.

```
#include <iostream>
 #include <omp.h>
 #include <cstdlib>
 #include <stdio.h>
 #include <stdlib.h>

 void Usage(char prog_name[]);
 void Get_args(char* argv[], int* thread_count, int* n);
 void Schedule(int thread_count, int n, int max[], int min[]);
 void PrintResult(int max[], int min[], int thread_count, int n);

 int main(int argc, char* argv[]) {
    if (argc != 3) Usage(argv[0]);

    int thread_count, n;
    Get_args(argv, &thread_count, &n); // Initialize the values above

    int max[thread_count];
    int min[thread_count];

    Schedule(thread_count, n, max, min);
    PrintResult(max,min, thread_count, n);

    return 0;
 }

 void PrintResult(int max[], int min[], int thread_count, int n){
   for (int i = 0; i < thread_count; i++) {
      if (min[i] == n && max[i] == 0) /* This can happen when thread_count>n */
         printf("Thread %d: No iterations\n", i);
      else if (min[i] != max[i])
         printf("Thread %d: Iterations %d - %d\n", i, min[i], max[i]);
      else
         printf("Th %d > Iteration  %d\n", i, min[i]);
   }
 }
```

```
void Usage(char prog_name[]) {
   fprintf(stderr, "usage: %s <thread_count> <n>\n", prog_name);
   exit(0);
}

void Get_args(char* argv[], int* thread_count, int* n) {
   *thread_count = strtol(argv[1], NULL, 10);
   *n = strtol(argv[2], NULL, 10);
}

void Schedule(int thread_count, int n, int max[], int min[]) {
   int  i;

   for (i = 0; i < thread_count; i++) {
      min[i] = n;
      max[i] = 0;
   }

#  pragma omp parallel num_threads(thread_count) \
      default(none) private(i) shared(min, max, n)
   {
   int my_rank = omp_get_thread_num();
#     pragma omp for
      for (i = 0; i < n; i++) {
#        ifdef DEBUG
         printf("Th %d > iteration %d\n", my_rank, i);
#        endif
         if (i < min[my_rank])
            min[my_rank] = i;
         if (i > max[my_rank])
            max[my_rank] = i;
      }
   }
}
```

Question 7

In our first attempt to parallelize the program for estimating the value of Pi, our program was incorrect. In fact, we used the result of the program when it was run with one thread as evidence that the program run two with two threads was incorrect. Explain why we could "trust" the result of the program when it was run with one thread.

Answer

The reason that it was correct when only one thread was used is that in this situation there is a lack of competition between the threads because it only exists one. The behavior is the same as the procedural approach. Note that before the code was corrected with private *factor*, there was no assignment of the scope of the variable *factor*, which by default makes it shared among the threads. Thus, some thread can change the value of the variable while another is using it, generating an inconsistency on the result.

Question 8

Consider the loop

```
a[0] = 0;
for (i = 1; i < n; i++ ){
  a[i] = a[i-1] + i;
}
```

There is clearly a loop-carried dependency, as the value of *a[i]* can't be computed without the value of *a[i-1]*. Can you see a way to eliminate this dependecy and parallelize the loop?

Answer

First, let's take a look at the implementation results iteration by iteration for *n*=5.

```
a[0] = 0                                    = 0*(0+1)/2
a[1] = a[0] + 1 = 0 + 1 = 1                 = 1*(1+1)/2
a[2] = a[1] + 2 = 0 + 1 + 2 = 3             = 2*(2+1)/2
a[3] = a[2] + 3 = 0 + 1 + 2 + 3 = 6         = 3*(3+1)/2
a[4] = a[3] + 4 = 0 + 1 + 2 + 3 + 4 = 10 = 4*(4+1)/2
```

$$a[i] = a[i-1] + i = \sum_{j=0}^{i} j = \frac{i(i+1)}{2}$$

So, since we now see a direct equation to compute the value of `a[i]`, the block of code could be rewritten as follows.

```
for(i=0; i<n; i++)
  a[i] = i*(i+1)/2;
```

The main thing here in this loop is that the value is the `a[i]` is not reused later on the code, which eliminates the dependency we had before. Thus, the code can be parallelized using a **parallel for** directive as follows.

```
# pragma omp parallel for num_threads(thread_count) \
      default(none) private(i) shared(a,n)
    for(i=0; i<n; i++){
```

```
    a[i] = i*(i+1)/2;
}
```

Note that the **i** was declared just before the for directive. This is the C way to do it. Under C++ you could just leave the initialization inside the **for loop** arguments.

Question 9

Modify the trapezoidal rule program that uses a `paralell` for directive (omp_trap_3.c) so that the parallel for is modified by a schedule(runtime) clause. Run the program with various assignments to the environment variable OMP_SCHEDULE and determine which iterations are assigned to which thread. This can be done by allocating an array **iterations** of *n* ints and in the Trap function of the *for loop*.

Answer

When there is no `schedule` clause is used, the default assignment of iterations of my system is the default of most OpenMP implementations, approximately a block partition. As described in the textbook on page 236, at the beginning of the section 5.7, if there are ***n*** iterations in the serial loop, then in the parallel loop the first ***n/thread_count*** are assigned to thread 0, the next ***n/thread_count*** are assigned to thread 1 and so on. For some reason, the thread 0 receives one more thread and the thread *`thread_count-1`* receives one less. The following table shows my results when running the program with different number of threads and fixed parameters.

Fixed parameters: a = 2 b = 10 n = 128

Number of threads	Thread rank	Iterations
2	0	0 - 64
	1	65 - 128
4	0	0- 32
	1	33 - 64
	2	65 - 96
	3	97 - 127
	0	0 - 16
	1	17 - 32

8	**2**	**33 - 48**
	3	**49 - 64**
	4	**65 - 80**
	5	**81 - 96**
	6	**97 - 112**
	7	**113 - 127**

When the `schedule(runtime)` clause is included but the environment variable OMP_SCHEDULE is not defined, the assignment of iterations gets very weird and goes all to the thread 0.

When the OMP_SCHEDULE variable is defined as guided, or the directive is set to `schedule(guided)`, the amount of threads is roughly the number of itertions remaining divided by the number of threads. In order to compare it to the Table 5.3 at page 240, the program was ran with n=10,000 and two threads.

Thread	Iterations
0	0 -- 5000
1	5001 -- 9687
0	9688 -- 9843
1	9844 -- 9960
0	9961 -- 9980
1	9981 -- 9995
0	9996 -- 9997
1	9998 -- 9998
0	9999 -- 9999

Question 10

Answer

First let's understand the use of the `atomic` directive. This is similar to the `atomic` directive but with a major difference: it can only protect critical sectoin that consist of a single C assignment statement. The following is the accepted form.

```
x <op> = <expression>;
```

Ok, then. My implementation to check if the `atomic` are treated as one ow different critical regions is the following.

```
int main(int argc, char* argv[]) {
   int thread_count, n;
   double start, finish;

   if (argc != 3) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   n = strtol(argv[2], NULL, 10);

   start = omp_get_wtime();
#  pragma omp parallel num_threads(thread_count)
   {
      int i;
      double my_sum = 0.0;

      for(i = 0; i < n; i++) {
#        pragma omp atomic
         my_sum += sin(i);
      }
   }
   finish = omp_get_wtime();

   printf("Thread_count = %d, n = %d, Time = %e seconds\n",
         thread_count, n, finish-start);
   return 0;
}
```

Knowing that my system has 2 available cores, I ran the program with 1, 2 and 4 threads for a large value of n. The results were the following.

```
msrj@ubuntu:~/Documents/UFRN/PCD/ParallelProgramming/BookExercises/Chapter5/Question10$ ./q10 1 100000000
Thread_count = 1, n = 100000000, Time = 3.713299e+00 seconds
msrj@ubuntu:~/Documents/UFRN/PCD/ParallelProgramming/BookExercises/Chapter5/Question10$ ./q10 2 100000000
Thread_count = 2, n = 100000000, Time = 3.835380e+00 seconds
msrj@ubuntu:~/Documents/UFRN/PCD/ParallelProgramming/BookExercises/Chapter5/Question10$ ./q10 4 100000000
Thread_count = 4, n = 100000000, Time = 7.663669e+00 seconds
msrj@ubuntu:~/Documents/UFRN/PCD/ParallelProgramming/BookExercises/Chapter5/Question10$ ./q10 8 100000000
Thread_count = 8, n = 100000000, Time = 1.539942e+01 seconds
```

As the question said, if the atomic directive is treated as different critical sections, the performance would improve as long as the number of threads is less than the number of threads. Since my system has 2 threads, that's what happened.

In my implementation, the variable can be updated on different threads. Also, when *thread_counts>2*, we have a program that starts to lose performance drastically. Thus, this indicates that the OpenMP implementation used here treats all these omp atomic directives as the same critical section when in the same core. That means that if I have 2 cores and 2 threads, each thread will run in a single core. However, if we have 2 cores and 4 threads, two threads are going to core 0 and the other two to core 1. Inside of these cores, the atomic directive is treated as the same critical section.

Question 11

Recall that in C, a function that takes a 2D array argument must specify the number of columns in the argument list, so it is quite common for C programmers to only use one-dimensional arrays, and to write explicit code for converting pairs of subscripts into a single dimension. Modify the OpenMP matrix-vector multiplication so that it uses a one dimensional array for the matrix.

Answer.

First, let's understand what the row-major matrix representation is. A simple NxM with N=2 and M=3 as follows.

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[2][2]

If we we separate this in a row-major manner, we would have the following representation.

A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[2][2]

Which the black background represents the first row and the white the second row. So, now that we now what is the row-major representation. How would we access each value? The approach is simple and is based on two tasks: select the row and select the column.

To select the first element of each row, we just need to multiply the number of the row we want by the number of columns that we already now. Here, we have that the number of columns is M and let's say the row we want is called `wanted_row`. So, the access to the first element on the `wanted_row`, is:

A [`wanted_row*M`]

In order to complete the second task, select the column, we must sum the number of the `wanted_column` to `wanted_row*M`. Then, the access to the position A[`wanted_row`][`wanted_column`] is:

A [`wanted_row*M + wanted_column`]

So, the matrix multiplication is implemented by the following code. Please note that the `wanted_row and wanted_column` variables are represented as i and j, respectively.

```
# pragma omp parallel for num_threads(thread_count) \
  default(none) private(i, j) shared(A, x, y, M, N)
  // default(none) - obligates all variables to have its scope determined
  // private(i, j) - each thread has its own copy of the iterators
  // shared(A, x, y, m, n) - Threads share the A[MxN] matrix and x vector.
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
      /* y[i] += A[i][j]*x[j]; // Old way */
      y[i] += A[i*n + j]*x[j];
    }
  }
```

Question 12

Download the source file **omp_mat_vect_rand_split.c** from the book's web-site. Find a program that does cache profiling (e.g.,Valgrind[49]) and compile the program according to the instructions in the cache profiler documentation. (For example, with Valgrind you will want a symbol table and full optimization. (With **gcc** use, **gcc -g -02 …)**. Now run the program according to the instructions in the cache profiler documentation, using input $k \times (k \cdot 10^6)$, $(k \cdot 10^3) \times (k \cdot 10^3)$ and $(k \cdot 10^6) \times k.$ Choose k so large that the number of level 2 cache misses is of the order 10^6 for at least one of the input sets of data.

A. How many level 1 cache write-misses occur with each of the three inputs?
B. How many level 2 cache write-misses occur with each of the three inputs?
C. Where do most of the write-misses occur? For which input data does the program have the most write-misses? Can you explain why?
D. How many level 1 cache read-misses occur with each of the three inputs?
E. How many level 2 cache read-misses occur with each of the three inputs?
F. Where do most of the read-misses occur? For which input data does the program have the most read-misses? Can you explain why?
G. Run the program with each of the three inputs, but without the cache profiler. With which input is the program fastest? With which input is the program the slowest? Can your observations about cache misses help explain the differences? How?

Answer

Choosing k = 8, we have the following result for two threads.

Matrix	Data cache misses					
	I1 - Writes	I1 - Read	D1-Write	D1-Read	LL-Write	LL-Read
8 x 8 M						
8K x 8K						
8M x 8						

Question 13

Recall the matrix-vector multiplication example with the 8000x8000 input. Suppose that thread 0 and thread 2 are assigned to different processors. If a cache line contains 64 bytes or 8 doubles, is it possible for false sharing between threads 0 and 2 to occur for any part of the vector y? Why? What about if thread 0 and thread 3 are assigned to different processors; is it possible for false sharing to occur between them for any part of y?

Answer.

With the input matrix having 8000x8000 elements, the y vector will have 8000 elements. Thus, with four threads, assuming a default scheduling of 8000/t, where t is the thread_count, the elements of the output vector will be divided approximately as follows:

$8000/4 = 2000$ elements per thread

Thread	Elements of y			
0	y[0]	y[1]	. . . .	y[1999]
1	y[2000]	y[2001]	. . . .	y[3999]
2	y[4000]	y[4001]	. . . .	y[5999]
3	y[6000]	y[6001]	. . . .	y[7999]

The false sharing occurs when some element store on a cache belongs to the same cache line as another element that is stored in other cache line. So, for example, if the Thread 0 has a value in its cache that belongs to a line that an element of this line is stored into the cache of Thread 1, when the Thread 0 updates its variable the line will be invalidated and the Thread 1 will have to request a new load from memory if uses the dirty line.

In order for false-sharing to occur between thread 0 and thread 2, there must be elements of y that belong to the same cache line, but are assigned to different threads. Assuming a row-major storage in memory, that all values are doubles and the 64bytes cache line can only store 8 values, this will only occur on the "edge of columns". So, the first value of thread 1 which belongs to a line that includes a value from thread two will be index 1993. And the highest index from thread 0 that will still include an element from itself is the 1999. All the scenarios are presented below.

	Thread 1			Thread 0			
	y indexes						
Cache line	1993	1994	1995	1996	1997	1998	1999	2000
	1994	1995	1996	1997	1998	1999	2000	2001
	1995	1996	1997	1998	1999	2000	2001	2002
	1996	1997	1998	1999	2000	2001	2002	2003
	1997	1998	1999	2000	2001	2002	2003	2004
	1998	1999	2000	2001	2002	2003	2004	2005
	1999	2000	2001	2002	2003	2004	2005	2006

Following the same approach, since the least index of an element of y assigned to thread 2 is 4000, there can't possibly be a cache line that has elements belonging to both thread 0 and thread 2. The same approach applies to threads 0 and 3, which have their least elements as 0 and 6000.

Question 14

Recall the matrix-vector multiplication example with an 8x8,000,000 matrix. Suppose that doubles use 8 bytes of memory and that a cache line is 64 bytes. Also suppose that our system consists of two dual-core processors.

A. What is the minimum number of cache lines that are needed to store the vector y?
B. What is the maximum number of lines that are needed to store the vector y?
C. If the boundaries if cache lines always coincide with the boundaries of 8-byte doubles, in how many different ways can the components of y be assigned to cache lines/

D. If we only consider which pairs of threads share a processor, in how many different way can four threads be assigned to the processors in our computer? Here, we're assuming that cores on the same processor share cache.
E. Is there an assignment of components to cache lines and threads to processors that will result in no false-sharing in our example? In other words, is it possible that the threads assigned to the other processor will have their components in a different cache line?
F. How many assignments of components to cache lines and threads to processors are there?
G. Of these assignments, how many will result in no false sharing?

Answer.

A. From the matrix multiplication, we know that for an input of this dimension, we have ah y output with 8 elements. So, since they are all doubles that take 8 bytes each and our cache line has 64 bytes, the y vector fits into a single cache line.

	64 bytes							
Cache line	y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]
	8 bytes	8 bytes	8 bytes	8 bytes	8 bytes	8 bytes	8 bytes	8 bytes

B. The worst case scenario for the cache is if the y[0] was stored into the last cache element, as following:

1st line	-	-	-	-	-	-	-	y[0]
2nd line	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	-

Thus, the maximum number of cache lines that are needed to store the vector y is 2.

C. There are eight ways the doubles can be assigned to the cache line: they do correspond to the eight different possible locations for y[0].

D.

Assuming, as said on the example, 4 cores split in two dual-core processors, we can choose two threads and assign them to one of the processors (each one is fine). The following shows the possibilities for one processor.

Core 0		Core 1	
Thread 0	Thread 1	Thread 2	Thread 3
Thread 0	Thread 2	Thread 1	Thread 3
Thread 0	Thread 3	Thread 1	Thread 2

Note that the table above covers all the possibilities. If we are thinking about the Core 0, for example, choosing 0 and 2 for it automatically means that Core 1 has threads 1 and 3. Thus, the total number of possibilities is three.

E.

Yes, there is a possibility. The problem with false-sharing is that different variables are part of the same cache line. So, if part of the variables could be assigned to a different line, we would have an assignment with no false-sharing. Let's imagine that Core 0 has the threads 0 and 1, so core 1 has threads 2 and 3. Here, if y[0] through y[3] are occupying the last 4 slots on the first cache line the resto would go to the second line. The following table illustrates this.

1st line	-	-	-	-	y[0]	y[1]	y[2]	y[3]
2nd line	y[4]	y[5]	y[6]	y[7]		-	-	-

Then, when an element of the first line is updated, only the first line (which is stored in the core 0 cache) is invalidated. Similarly, any writing done by threads 2 and 3 on in core 1 would have no effect on core 0 because they will only mark the second line as invalid.

F.

From each of the three assignments of threads to processes found at question 14-D, there are 8 ways the y vector can be stored into the cache line. So, the total possibilities are: $8 \times 3 = 24$ possibilities

G.

There are a couple of cases that will result in no false sharing. If the threads 0 and 1 are assigned to different processors, then any assignment of y that puts y[1]

and y[2] in the same cache line will cause false sharing between 0 and 1. The only way this this can fail to happen is if y[0] and y[1] are in one line and the remainder of y is another, like the following example.

1st line	-	-	-	-	-	-	y[0]	y[1]
2nd line	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	-	-

However, this case will still fail when we analyse threads 2 and 3. Their values are still sharing a line, which will cause a false sharing between them.

Then, threads 0 and 1 must be assigned to the same processor. Furthermore, if any component of y with index >3 is assigned to this processor or if any component with index < 4 is assigned to the other processor, there will be false sharing. Thus, the assignment from 14-E is the only case that the false sharing will not occur.