Aston University

BIRMINGHAM UK

Module: Introduction to Parallel Programming Techniques
Module ID: EE4107
Student Name: Umid Narziev
Student ID: 200234832
Assignment Number: 6
Academic Year: 2020 - 2021

Contents

System Specification

Below, shown the system which was used to run and get the result of the simulation:

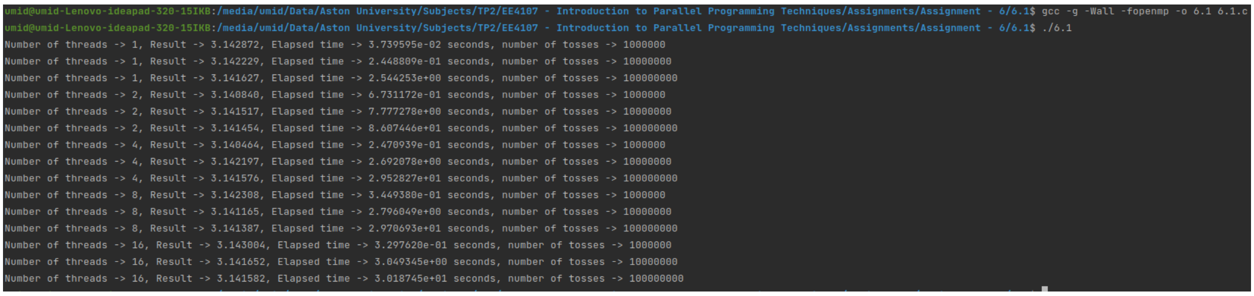
CPU:	Intel i5-7200U
Architecture:	Kaby Lake
Segment:	Mobile Processors
The number of cores:	2
Number of threads	4
Clock Frequency	2.50-3.10GHz (Turbo Boost)
Cache levels:	3
Cache level 1 size:	128KBytes
Cache level 2 size:	512Kbytes
Cache level 3 size:	3MBytes
RAM	12 GB
SSD:	250 GB
Operating System:	Ubuntu 20.04.2 LTS
Compiler:	Gcc and its libraries
IDE:	Clion (2020.03)

TASK 1

Code:

```
/*
 * To compile ->  gcc -g -Wall -fopenmp -o 6.1 6.1.c
 * To run     -> ./6.1
 */

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

long long int trap(int thread_count, long long int global_number_of_tosses);

int main(int argc, char *argv[]) {

   long long int global_number_in_circle;
   float pi_estimate;
   double start, finish;
   for (int ii = 2; ii <= 16; ii <<= 1) {
      for (int jj = 1000000; jj <= 100000000; jj *= 10) {
         int thread_count = ii;
         long long int number_of_tosses = jj;
         global_number_in_circle = 0;
         GET_TIME(start);
#pragma omp parallel  num_threads(thread_count) reduction(+: global_number_in_circle)
         {
            global_number_in_circle += trap(thread_count, number_of_tosses);
         }
         GET_TIME(finish);
         pi_estimate = (4.0 * (double) global_number_in_circle) / ((double) number_of_tosses);
         printf("Number of threads -> %d, Result -> %f, Elapsed time -> %e seconds, number of tosses ->
%lld\n",
              ii, pi_estimate, finish - start, number_of_tosses);
      }
   }


   return 0;
}

long long int trap(int thread_count, long long int global_number_of_tosses) {

   double x, y, distance_squared;
   long long int number_in_circle = 0;
   long long int number_of_tosses = global_number_of_tosses / thread_count;
   for (long long int toss = 0;
       toss < number_of_tosses; toss++) { // number of tosses should be equally divided among processor
      x = rand() / (RAND_MAX + 1.0) * (1 - (-1)) + (-1);
      y = rand() / (RAND_MAX + 1.0) * (1 - (-1)) + (-1);
      distance_squared = x * x + y * y;
      if (distance_squared <= 1) {
         number_in_circle++;
      }
   }
   return number_in_circle;
```

Result:



```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.1$ gcc -g -Wall -fopenmp -o 6.1 6.1.c
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.1$ ./6.1
Number of threads -> 1, Result -> 3.142872, Elapsed time -> 3.739595e-02 seconds, number of tosses -> 1000000
Number of threads -> 1, Result -> 3.142229, Elapsed time -> 2.448809e-01 seconds, number of tosses -> 10000000
Number of threads -> 1, Result -> 3.141627, Elapsed time -> 2.544253e+00 seconds, number of tosses -> 100000000
Number of threads -> 2, Result -> 3.140840, Elapsed time -> 6.731172e-01 seconds, number of tosses -> 1000000
Number of threads -> 2, Result -> 3.141517, Elapsed time -> 7.777278e+00 seconds, number of tosses -> 10000000
Number of threads -> 2, Result -> 3.141454, Elapsed time -> 8.607446e+01 seconds, number of tosses -> 100000000
Number of threads -> 4, Result -> 3.140464, Elapsed time -> 2.470939e-01 seconds, number of tosses -> 1000000
Number of threads -> 4, Result -> 3.142197, Elapsed time -> 2.692078e+00 seconds, number of tosses -> 10000000
Number of threads -> 4, Result -> 3.141576, Elapsed time -> 2.952827e+01 seconds, number of tosses -> 100000000
Number of threads -> 8, Result -> 3.142308, Elapsed time -> 3.449380e-01 seconds, number of tosses -> 1000000
Number of threads -> 8, Result -> 3.141165, Elapsed time -> 2.796049e+00 seconds, number of tosses -> 10000000
Number of threads -> 8, Result -> 3.141387, Elapsed time -> 2.970693e+01 seconds, number of tosses -> 100000000
Number of threads -> 16, Result -> 3.143004, Elapsed time -> 3.297620e-01 seconds, number of tosses -> 1000000
Number of threads -> 16, Result -> 3.141652, Elapsed time -> 3.049345e+00 seconds, number of tosses -> 10000000
Number of threads -> 16, Result -> 3.141582, Elapsed time -> 3.018745e+01 seconds, number of tosses -> 100000000
```

Figure 1: Simulation Result

Table 1: Table of result

Result	Elapsed time	Number of tosses	Correctness of PI	Speed-UP	Efficiency	Number of threads
3,142872	3,74E-02	1000000	99,96%	1,00	100%	1
3,142229	2,45E-01	10000000	99,98%	1,00	100%	1
3,141627	2,54E+00	100000000	100,00%	1,00	100%	1
3,140840	6,73E-01	1000000	99,98%	0,94	47%	2
3,141517	7,78E+00	10000000	100,00%	1,00	50%	2
3,141454	8,61E+01	100000000	100,00%	1,00	50%	2
3,140464	2,47E-01	1000000	99,96%	0,85	21%	4
3,142197	2,69E+00	10000000	99,98%	0,99	25%	4
3,141576	2,95E+01	100000000	100,00%	1,00	25%	4
3,142308	3,45E-01	1000000	99,98%	0,89	11%	8
3,141165	2,80E+00	10000000	99,99%	0,99	12%	8
3,141387	2,97E+01	100000000	99,99%	1,00	12%	8
3,143004	3,30E-01	1000000	99,96%	0,89	6%	16
3,141652	3,05E+00	10000000	100,00%	0,99	6%	16
3,141582	3,02E+01	100000000	100,00%	1,00	6%	16

Conclusion:

- Simulation is done with 1-16 threads, and the number of tosses is in a range of $10^6 – 10^9$.
- Prediction of pi increases with increase in number of tosses.
- However, as we see the program is perfectly serialized with parallel section. The reason for that is that it has critical section, which worsen the performance of the program with larger number of threads.

TASK 2

Code:

```
/*
 * File:    6.2.c
 * Purpose: Estimate definite integral (or area under curve) using the
 *         trapezoidal rule.  This version uses a parallel for directive
 *
 * Input:   a, b, n
 * Output:  estimate of integral from a to b of f(x)
 *         using n trapezoids.
 *
 * Compile: gcc -g -Wall -fopenmp -o 6.2 6.2.c
 * Usage:   ./omp_trap3 <number of threads>
 * export OMP_SCHEDULE="guided, 500"
 *
 * Notes:
 *   1.  The function f(x) is hardwired.
 *   2.  In this version, it's not necessary for n to be
 *       evenly divisible by thread_count.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "timer.h"
void Usage(char* prog_name);
double f(double x);    /* Function we're integrating */
double Trap(double a, double b, int n, int thread_count);
void Print_iters(int iterations[], long n);
int*    iterations;
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;
   iterations = malloc((n+1)*sizeof(int));
   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);
   global_result = Trap(a, b, n, thread_count);
   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
        a, b, global_result);

   Print_iters(iterations, n);

   return 0;
}  /* main */

/*--------------------------------------------------------------------
 * Function:    Usage
 * Purpose:     Print command line for function and terminate
 * In arg:      prog_name
 */
void Usage(char* prog_name) {
```

```
   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   exit(0);
}  /* Usage */

/*--------------------------------------------------------------------
 * Function:    f
 * Purpose:     Compute value of function to be integrated
 * Input arg:   x
 * Return val:  f(x)
 */
double f(double x) {
   double return_val;

   return_val = x*x;
   return return_val;
}  /* f */
/*--------------------------------------------------------------------
 * Function:    Trap
 * Purpose:     Use trapezoidal rule to estimate definite integral
 * Input args:
 *    a: left endpoint
 *    b: right endpoint
 *    n: number of trapezoids
 * Return val:
 *    approx:  estimate of integral from a to b of f(x)
 */
double Trap(double a, double b, int n, int thread_count) {
   double  h, approx;
   int  i;

   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
     reduction(+: approx) schedule(runtime)
   for (i = 1; i <= n-1; i++){
      approx += f(a + i*h);
      iterations[i] = omp_get_thread_num();
   }

   approx = h*approx;
   return approx;
}  /* Trap */
void Print_iters(int iterations[], long n) {
   int i, start_iter, stop_iter, which_thread;

   printf("\n");
   printf("Thread\t\tIterations\n");
   printf("------\t\t----------\n");
   which_thread = iterations[0];
   start_iter = stop_iter = 0;
   for (i = 0; i <= n; i++)
      if (iterations[i] == which_thread)
         stop_iter = i;
      else {
         printf("%4d  \t\t%d -- %d\n", which_thread, start_iter, stop_iter);
         which_thread = iterations[i];
         start_iter = stop_iter = i;
      }
   printf("%4d  \t\t%d -- %d\n", which_thread, start_iter, stop_iter);
}
```

Result:

1. Default scheduler:

```
:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.2$ ./6.2 2
Enter a, b, and n
0 3 100
With n = 100 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.00045000000000e+00

Thread          Iterations
------          ----------
   0            0 -- 1
   1            2 -- 2
   0            3 -- 3
   1            4 -- 4
   0            5 -- 5
   1            6 -- 6
   0            7 -- 8
   1            9 -- 9
   0            10 -- 10
   1            11 -- 11
   0            12 -- 12
   1            13 -- 13
   0            14 -- 15
   1            16 -- 16
   0            17 -- 17
   1            18 -- 19
   0            20 -- 20
   1            21 -- 21
   0            22 -- 22
   1            23 -- 23
   0            24 -- 24
   1            25 -- 25
   0            26 -- 26
   1            27 -- 28
   0            29 -- 29
   1            30 -- 30
   0            31 -- 31
   1            32 -- 32
   0            33 -- 33
   1            34 -- 34
   0            35 -- 36
```

2. Dynamic, 8:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.2$ export OMP_SCHEDULE="dynamic, 8"
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.2$ ./6.2 2
Enter a, b, and n
0 3 100
With n = 100 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.00045000000000e+00

Thread          Iterations
------          ----------
   0            0 -- 16
   1            17 -- 24
   0            25 -- 32
   1            33 -- 40
   0            41 -- 48
   1            49 -- 56
   0            57 -- 64
   1            65 -- 72
   0            73 -- 80
   1            81 -- 88
   0            89 -- 96
   1            97 -- 99
   0            100 -- 100
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.2$
```

3. Guided, 8:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.2$ ./6.2 2
Enter a, b, and n
0 3 102
With n = 102 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.00043252595156e+00

Thread          Iterations
------          ----------
   0            0 -- 0
   1            1 -- 51
   0            52 -- 89
   1            90 -- 101
   0            102 -- 102
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.2$
```

4. Static, 8

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.2$ export OMP_SCHEDULE="static, 8"
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.2$ ./6.2 2
Enter a, b, and n
0 3 100
With n = 100 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.00045000000000e+00

Thread          Iterations
------          ----------
   0            0 -- 8
   1            9 -- 16
   0            17 -- 24
   1            25 -- 32
   0            33 -- 40
   1            41 -- 48
   0            49 -- 56
   1            57 -- 64
   0            65 -- 72
   1            73 -- 80
   0            81 -- 88
   1            89 -- 96
   0            97 -- 100
```

Conclusion:

- The default scheduler is static, the chunk size is 8.
- The dynamic scheduler assigns 8 elements from the remaining array for each iteration.
- In a guided scheduler, iterations are assigned in progressive order, where the next iteration is found as the remaining elements/number of threads. It is clearly shown in the result.

TASK 3

Code:

```
/*
 * To compile -> gcc -g -Wall -fopenmp -o 6.3 6.3.c
 * TO run     -> ./6.3
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include <stdbool.h>
#include "omp.h"

#define DEBUG 0
#define THRESHOLD (5e-3)
#define innerMost /*
 * 1. outterMost -> a
 * 2. innerMost -> b
 *
 */
int thread_count;

void init_phi(double *phi, int n);

void compute_outter(double *cur, double *next, int n);

void compute_inner(double *cur, double *next, int n);

int converged(double *cur, double *next, int n);

int compute_conv(double *cur, double *next, int n);

int compute_single_region(double *cur, double *next, int n);

int main(int argc, char *argv[]) {
   int n, niters, conv, n_array[3] = {100, 500, 1000};
   double *phi_cur, *phi_next, *tmp;
   double start, end, elapsed;
//    if (argc != 2)
//    {
//        fprintf( stderr, "Usage: %s <n>\n", argv[0] );
//        exit( -1 );
//    }


//    n = atoi(argv[1]);

//------------------------------compute_outter------------------------------------------------------------------------
   for (int i = 0; i < 3; ++i) {
      for (int j = 1; j < 16; j <<= 1) {

         n = n_array[i];
         phi_cur = (double *) malloc(n * n * sizeof(double));
         phi_next = (double *) malloc(n * n * sizeof(double));
         thread_count = j;
         init_phi(phi_cur, n);
```

```
        init_phi(phi_next, n);
        niters = 0;
        start = omp_get_wtime();
        while (1) {
          niters++;
#if DEBUG
          printf("Iteration %d\n", niters );
          int i,j;
          for ( i = 0; i < n; i++ )
          {
            printf("[" );
            for ( j = 0; j < n; j++ )
              printf(" %10.6f ", phi_cur[j*n + i] );
            printf("]\n" );
          }
          sleep(1);
#endif
          // Compute next (new) phi from current (old) phi
          compute_outter(phi_cur, phi_next, n);
          // If converged, we are done
          conv = converged(phi_cur, phi_next, n);
          if (conv)
            break;
          // Otherwise, swap pointers and continue
          tmp = phi_cur;
          phi_cur = phi_next;
          phi_next = tmp;
        }
        end = omp_get_wtime();
        free(phi_cur);
        free(phi_next);
        elapsed = end - start;
        printf("[compute_outter] Converged after %d iterations, elapsed time -> %lf, thread_number -> %d n->
%d \n",
               niters, elapsed,
               thread_count, n);
      }
    }

//------------------------------compute_inner--------------------------------------------------------------------------------
    for (int i = 0; i < 3; ++i) {
      for (int j = 1; j < 16; j <<= 1) {

        n = n_array[i];
        phi_cur = (double *) malloc(n * n * sizeof(double));
        phi_next = (double *) malloc(n * n * sizeof(double));
        thread_count = j;
        init_phi(phi_cur, n);
        init_phi(phi_next, n);
        niters = 0;
        start = omp_get_wtime();
        while (1) {
          niters++;
#if DEBUG
          printf("Iteration %d\n", niters );
          int i,j;
          for ( i = 0; i < n; i++ )
          {
            printf("[" );
            for ( j = 0; j < n; j++ )
              printf(" %10.6f ", phi_cur[j*n + i] );
            printf("]\n" );
```

```
            }
            sleep(1);
#endif
            // Compute next (new) phi from current (old) phi
            compute_inner(phi_cur, phi_next, n);
            // If converged, we are done
            conv = converged(phi_cur, phi_next, n);
            if (conv)
               break;
            // Otherwise, swap pointers and continue
            tmp = phi_cur;
            phi_cur = phi_next;
            phi_next = tmp;
         }
         end = omp_get_wtime();
         free(phi_cur);
         free(phi_next);
         elapsed = end - start;
         printf("[compute_inner] Converged after %d iterations, elapsed time -> %lf, thread_number -> %d n->
%d \n",
                niters, elapsed,
                thread_count, n);
      }
   }

//------------------------------compute_outter & compute_conv-----------------------------------------------------
   for (int i = 0; i < 3; ++i) {
      for (int j = 1; j < 16; j <<= 1) {

         n = n_array[i];
         phi_cur = (double *) malloc(n * n * sizeof(double));
         phi_next = (double *) malloc(n * n * sizeof(double));
         thread_count = j;
         init_phi(phi_cur, n);
         init_phi(phi_next, n);
         niters = 0;
         start = omp_get_wtime();
         while (1) {
            niters++;
#if DEBUG
            printf("Iteration %d\n", niters );
            int i,j;
            for ( i = 0; i < n; i++ )
            {
               printf("[" );
               for ( j = 0; j < n; j++ )
                  printf(" %10.6f ", phi_cur[j*n + i] );
               printf("]\n" );
            }
            sleep(1);
#endif
            // Compute next (new) phi from current (old) phi
            compute_outter(phi_cur, phi_next, n);
            // If converged, we are done
            conv = compute_conv(phi_cur, phi_next, n);
            if (conv)
               break;
            // Otherwise, swap pointers and continue
            tmp = phi_cur;
            phi_cur = phi_next;
            phi_next = tmp;
         }
```

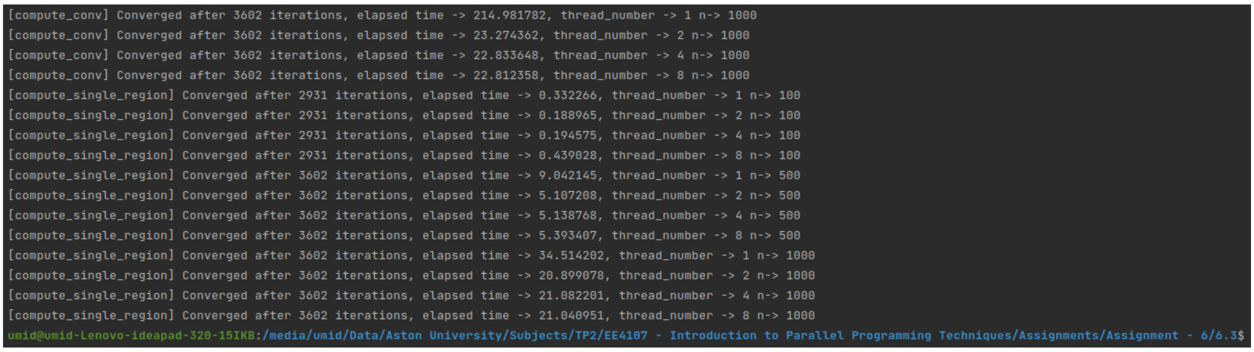
```c
        end = omp_get_wtime();
        free(phi_cur);
        free(phi_next);
        elapsed = end - start;
        printf("[compute_conv] Converged after %d iterations, elapsed time -> %lf, thread_number -> %d n->
%d \n",
              niters, elapsed,
              thread_count, n);
      }
   }
//------------------------------compute_outter & compute_single_region-------------------------------------------------
---------------------
   for (int i = 0; i < 3; ++i) {
      for (int j = 1; j < 16; j <<= 1) {
        n = n_array[i];
        phi_cur = (double *) malloc(n * n * sizeof(double));
        phi_next = (double *) malloc(n * n * sizeof(double));
        thread_count = j;
        init_phi(phi_cur, n);
        init_phi(phi_next, n);
        niters = 0;
        start = omp_get_wtime();
        while (1) {
           niters++;
#if DEBUG
           printf("Iteration %d\n", niters );
           int i,j;
           for ( i = 0; i < n; i++ )
           {
              printf("[" );
              for ( j = 0; j < n; j++ )
                 printf(" %10.6f ", phi_cur[j*n + i] );
              printf("]\n" );
           }
           sleep(1);
#endif
           // Compute next (new) phi from current (old) phi
           compute_outter(phi_cur, phi_next, n);
           // If converged, we are done
           conv = compute_single_region(phi_cur, phi_next, n);
           if (conv)
              break;
           // Otherwise, swap pointers and continue
           tmp = phi_cur;
           phi_cur = phi_next;
           phi_next = tmp;
        }
        end = omp_get_wtime();
        free(phi_cur);
        free(phi_next);
        elapsed = end - start;
        printf("[compute_single_region] Converged after %d iterations, elapsed time -> %lf, thread_number ->
%d n-> %d \n",
              niters, elapsed,
              thread_count, n);


      }
   }
   return 0;
}
```

```c
void init_phi(double *phi, int n) {
   int i, j;

   // Interior points initialized to 50 degrees
   for (i = 1; i < n - 1; i++)
      for (j = 1; j < n - 1; j++)
         phi[j * n + i] = 50.0;

   // Top, left, and right boundaries fixed at 100 degrees
   for (i = 0; i < n; i++) {
      phi[0 * n + i] = 100.0;
      phi[(n - 1) * n + i] = 100.0;
      phi[i * n + 0] = 100.0;
   }
   // Bottom boundary fixed at 0 degrees
   for (i = 0; i < n; i++)
      phi[i * n + (n - 1)] = 0.0;
}

void compute_outter(double *cur, double *next, int n) {
   int i, j;
   double temp;

#pragma omp parallel for default(none) num_threads(thread_count) shared(n, cur, next) private(j, i, temp)
   for (j = 1; j < n - 1; j++) {
      for (i = 1; i < n - 1; i++) {
         next[j * n + i] =
               (cur[(j - 1) * n + i] + cur[j * n + (i - 1)] + cur[j * n + (i + 1)] + cur[(j + 1) * n + i]) / 4;
//#pragma omp critical
//          next[j * n + i] = temp;
      }
   }
}

void compute_inner(double *cur, double *next, int n) {
   int i, j;
   double temp;

   for (j = 1; j < n - 1; j++) {
#pragma omp parallel for  default(none) num_threads(thread_count) shared(n, cur, next, j) private(i, temp)
      for (i = 1; i < n - 1; i++) {
         next[j * n + i] =
               (cur[(j - 1) * n + i] + cur[j * n + (i - 1)] + cur[j * n + (i + 1)] + cur[(j + 1) * n + i]) / 4;
//#pragma omp critical
//          next[j * n + i] = temp;
      }
   }


}

int converged(double *cur, double *next, int n) {
   int i, j;

   for (j = 1; j < n - 1; j++)
      for (i = 1; i < n - 1; i++)
         if (fabs(next[j * n + i] - cur[j * n + i]) > THRESHOLD)
            return 0;
   return 1;
}
```

```c
int compute_conv(double *cur, double *next, int n) {
   int i, j;
   bool flag = true;
#pragma omp parallel for default(none) num_threads(thread_count) shared(n, cur, next, flag) private(i, j)
   for (j = 1; j < n - 1; j++) {
#pragma omp parallel for
      for (i = 1; i < n - 1; i++)
         if ((fabs(next[j * n + i] - cur[j * n + i]) > THRESHOLD) && flag == true) {
#pragma omp critical
            flag = false;
         }

   }

   if (flag == false) {
      return 0;
   } else {
      return 1;
   }

}

int compute_single_region(double *cur, double *next, int n) {
   int i, j;
   bool flag = true;
#pragma omp parallel for default(none) num_threads(thread_count) shared(n, cur, flag, next) private(i, j)
   for (j = 1; j < n - 1; j++) {
      for (i = 1; i < n - 1; i++)
         if (fabs(next[j * n + i] - cur[j * n + i]) > THRESHOLD && flag == true)
            flag = false;
   }

   if (flag == false) {
      return 0;
   } else {
      return 1;
   }

}
```

Result:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.3$ ./6.3
[compute_outter] Converged after 2931 iterations, elapsed time -> 0.206806, thread_number -> 1 n-> 100
[compute_outter] Converged after 2931 iterations, elapsed time -> 0.104518, thread_number -> 2 n-> 100
[compute_outter] Converged after 2931 iterations, elapsed time -> 0.112249, thread_number -> 4 n-> 100
[compute_outter] Converged after 2931 iterations, elapsed time -> 0.234922, thread_number -> 8 n-> 100
[compute_outter] Converged after 3602 iterations, elapsed time -> 5.993196, thread_number -> 1 n-> 500
[compute_outter] Converged after 3602 iterations, elapsed time -> 3.103796, thread_number -> 2 n-> 500
[compute_outter] Converged after 3602 iterations, elapsed time -> 3.503462, thread_number -> 4 n-> 500
[compute_outter] Converged after 3602 iterations, elapsed time -> 3.840441, thread_number -> 8 n-> 500
[compute_outter] Converged after 3602 iterations, elapsed time -> 26.619609, thread_number -> 1 n-> 1000
[compute_outter] Converged after 3602 iterations, elapsed time -> 13.478866, thread_number -> 2 n-> 1000
[compute_outter] Converged after 3602 iterations, elapsed time -> 13.842169, thread_number -> 4 n-> 1000
[compute_outter] Converged after 3602 iterations, elapsed time -> 13.663856, thread_number -> 8 n-> 1000
[compute_inner] Converged after 2931 iterations, elapsed time -> 0.432172, thread_number -> 1 n-> 100
[compute_inner] Converged after 2931 iterations, elapsed time -> 0.533935, thread_number -> 2 n-> 100
[compute_inner] Converged after 2931 iterations, elapsed time -> 0.599224, thread_number -> 4 n-> 100
[compute_inner] Converged after 2931 iterations, elapsed time -> 9.904789, thread_number -> 8 n-> 100
[compute_inner] Converged after 3602 iterations, elapsed time -> 7.269447, thread_number -> 1 n-> 500
[compute_inner] Converged after 3602 iterations, elapsed time -> 5.683205, thread_number -> 2 n-> 500
[compute_inner] Converged after 3602 iterations, elapsed time -> 7.224843, thread_number -> 4 n-> 500
[compute_inner] Converged after 3602 iterations, elapsed time -> 68.655859, thread_number -> 8 n-> 500
[compute_inner] Converged after 3602 iterations, elapsed time -> 27.998511, thread_number -> 1 n-> 1000
[compute_inner] Converged after 3602 iterations, elapsed time -> 17.935279, thread_number -> 2 n-> 1000
[compute_inner] Converged after 3602 iterations, elapsed time -> 18.195418, thread_number -> 4 n-> 1000
[compute_inner] Converged after 3602 iterations, elapsed time -> 151.713768, thread_number -> 8 n-> 1000
[compute_conv] Converged after 2931 iterations, elapsed time -> 15.204088, thread_number -> 1 n-> 100
[compute_conv] Converged after 2931 iterations, elapsed time -> 0.322312, thread_number -> 2 n-> 100
[compute_conv] Converged after 2931 iterations, elapsed time -> 0.300696, thread_number -> 4 n-> 100
[compute_conv] Converged after 2931 iterations, elapsed time -> 0.640585, thread_number -> 8 n-> 100
[compute_conv] Converged after 3602 iterations, elapsed time -> 102.203678, thread_number -> 1 n-> 500
[compute_conv] Converged after 3602 iterations, elapsed time -> 6.196394, thread_number -> 2 n-> 500
[compute_conv] Converged after 3602 iterations, elapsed time -> 6.273762, thread_number -> 4 n-> 500
[compute_conv] Converged after 3602 iterations, elapsed time -> 6.659824, thread_number -> 8 n-> 500
```



```
[compute_conv] Converged after 3602 iterations, elapsed time -> 214.981782, thread_number -> 1 n-> 1000
[compute_conv] Converged after 3602 iterations, elapsed time -> 23.274362, thread_number -> 2 n-> 1000
[compute_conv] Converged after 3602 iterations, elapsed time -> 22.833648, thread_number -> 4 n-> 1000
[compute_conv] Converged after 3602 iterations, elapsed time -> 22.812358, thread_number -> 8 n-> 1000
[compute_single_region] Converged after 2931 iterations, elapsed time -> 0.332266, thread_number -> 1 n-> 100
[compute_single_region] Converged after 2931 iterations, elapsed time -> 0.188965, thread_number -> 2 n-> 100
[compute_single_region] Converged after 2931 iterations, elapsed time -> 0.194575, thread_number -> 4 n-> 100
[compute_single_region] Converged after 2931 iterations, elapsed time -> 0.439028, thread_number -> 8 n-> 100
[compute_single_region] Converged after 3602 iterations, elapsed time -> 9.042145, thread_number -> 1 n-> 500
[compute_single_region] Converged after 3602 iterations, elapsed time -> 5.107208, thread_number -> 2 n-> 500
[compute_single_region] Converged after 3602 iterations, elapsed time -> 5.138768, thread_number -> 4 n-> 500
[compute_single_region] Converged after 3602 iterations, elapsed time -> 5.393407, thread_number -> 8 n-> 500
[compute_single_region] Converged after 3602 iterations, elapsed time -> 34.514202, thread_number -> 1 n-> 1000
[compute_single_region] Converged after 3602 iterations, elapsed time -> 20.899078, thread_number -> 2 n-> 1000
[compute_single_region] Converged after 3602 iterations, elapsed time -> 21.082201, thread_number -> 4 n-> 1000
[compute_single_region] Converged after 3602 iterations, elapsed time -> 21.040951, thread_number -> 8 n-> 1000
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 6/6.3$
```

Figure 2: Simulation result

Table 2: Simulation result

Function	Thread number	N	Elapsed Time	Speed-up	Efficiency
Compute outter	1	100	0,21	1,00	100%
	2	100	0,10	1,98	99%
	4	100	0,11	1,84	46%
	8	100	0,23	0,88	11%
	1	500	5,99	1,00	100%
	2	500	3,10	1,93	97%
	4	500	3,50	1,71	43%
	8	500	3,84	1,56	20%
	1	1000	26,62	1,00	100%
	2	1000	13,48	1,97	99%
	4	1000	13,84	1,92	48%
	8	1000	13,66	1,95	24%
Compute inner	1	100	0,43	1,00	100%
	2	100	0,53	0,81	40%
	4	100	0,60	0,72	18%
	8	100	9,90	0,04	1%
	1	500	7,27	1,00	100%
	2	500	5,68	1,28	64%
	4	500	7,22	1,01	25%
	8	500	68,66	0,11	1%
	1	1000	28,00	1,00	100%
	2	1000	17,94	1,56	78%
	4	1000	18,20	1,54	38%
	8	1000	151,73	0,18	2%
Compute conv	1	100	15,20	1,00	100%
	2	100	0,32	47,17	2359%
	4	100	0,30	50,56	1264%
	8	100	0,64	23,73	297%
	1	500	102,20	1,00	100%

	2	500	6,20	16,49	825%
	4	500	6,27	16,29	407%
	8	500	6,66	15,35	192%
	1	1000	214,98	1,00	100%
	2	1000	23,27	9,24	462%
	4	1000	22,83	9,42	235%
	8	1000	22,81	9,42	118%
Compute inner	1	100	0,33	1,00	100%
	2	100	0,19	1,76	88%
	4	100	0,19	1,71	43%
	8	100	0,44	0,76	9%
	1	500	9,04	1,00	100%
	2	500	5,11	1,77	89%
	4	500	5,14	1,76	44%
	8	500	5,39	1,68	21%
	1	1000	34,51	1,00	100%
	2	1000	20,90	1,65	83%
	4	1000	21,08	1,64	41%
	8	1000	21,04	1,64	21%

Conclusion:

- The outermost loop parallelism will have higher efficiency and it is shown in the table speed-up is much higher than inner loop parallelism. The reason behind it is, in the inner loop parallelism need to fork/join for each iteration and which adds extra overhead to computation.
- Also from the result, it is seen that parallelizing the convergence function, with two for loop give much higher speed-up and efficiency than with one loop parallelising. The efficiency staggering 2359%,)).
- It should be noted, as my CPU has two core, so for my pc the optimal thread is between 2-4, with 8 processors it gives lower efficiency.
- Note, the critical section should be implemented but I did not implement it as it increased simulation result for days. But gave the same output as original version.