Università della Svizzera italiana | Institute of Computational Science ICS

High-Performance Computing Lab **2020**

Student: Gabriele Berra Discussed with: -

Solution for Project 2

Due date: 20.10.2020 (midnight)

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP [10 points]

1.1. Parallel version with reduction clause and with parallel and critical clauses

The objective of this exercise was to parallelize the given dot product code by using the reduction clause and the parallel and critical clauses. As far as concerns the parallelization with the reduction clause, it has been done as follows: an additional variable named *parallel_alpha* has been introduced, so that every thread can compute its own part of the for loop and then all the intermediate results are summed up in the global variable.

Before starting the explanation of the parallelization with the clauses parallel and critical, I think is better to understand what *critical* is and what is its behaviour inside a parallel region. The critical clause permits at only one thread at a time to enter a specific part of the parallel code: when the thread inside the critical has finished to do its own computation, another thread can go in and so on. The major problem using this clause is that it can compromise the performance of the parallelization if it is not called in the right place and/or with the right change in the code. Thus, to avoid that the critical clause causes any problem, I created a private variable in which I store the result of the product between $a[i] * b[i]$ in the inner for loop. After that - outside of the inner loop - I put the critical section in which every thread sums its own copy of the private variable in the global variable *alpha_parallel*. My implementation is included here below.

```
// #include <omp.h>
#include <iostream>
#include "walltime.h"
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>

#define NUM_ITERATIONS 100 // era 100

// Example benchmarks
// 0.008s ~0.8MB
#define N 100000
// 0.1s ~8MB
// #define N 1000000
// 1.1s ~80MB
// #define N 10000000
// 13s ~800MB
// #define N 100000000
// 127s 16GB
//#define N 1000000000
#define EPSILON 0.1


using namespace std;

int main() {
    int myId, numTdreads;
    double time_serial, time_start = 0.0;
    long double dotProduct;
    double *a, *b;

    // Allocate memory for the vectors as 1-D arrays
    a = new double[N];
    b = new double[N];

    // Initialize the vectors with some values
    for (int i = 0; i < N; i++) {
        a[i] = i;
        b[i] = i / 10.0;
    }

    long double alpha = 0;
    // serial execution
    // Note that we do extra iterations to reduce relative timing
        overhead
    time_start = wall_time();
    for ( int iterations = 0; iterations < NUM_ITERATIONS; iterations
       ++) {
        alpha = 0.0;
    for ( int i = 0; i < N; i ++) {
        alpha += a[i] * b[i];
    }
}
time_serial = wall_time() - time_start;
cout << "Serial execution time = " << time_serial << " sec" << endl
   ;

long double alpha_parallel = 0;
double time_red = 0;
double time_critical = 0;


double time_start_red = 0.0;
time_start_red = wall_time();
#pragma omp parallel for reduction(+: alpha_parallel)
for ( int iterations = 0; iterations < NUM_ITERATIONS; iterations
   ++) {
    alpha_parallel = 0.0;
    for ( int i = 0; i < N; i ++) {
        alpha_parallel += a[i] * b[i];
    }
}
time_red = wall_time() - time_start_red;



long double alpha_private = 0;
double time_start_critical = 0.0;
#pragma omp barrier


time_start_critical = wall_time();
#pragma omp parallel firstprivate(alpha_private)
for ( int iterations = 0; iterations < NUM_ITERATIONS; iterations
   ++) {
    alpha_parallel = 0.0;
    alpha_private = 0.0;
    #pragma omp for
    for ( int i = 0; i < N; i ++) {
        alpha_private += a[i] * b[i];
    }
    #pragma omp critical
    alpha_parallel += alpha_private;
}

time_critical = wall_time() - time_start_critical;


if ( (fabs(alpha_parallel - alpha) / fabs(alpha_parallel)) >
   EPSILON) {
    cout << "parallel reduction: " << alpha_parallel << " serial :"
        << alpha << "\n";
        cerr << "Alpha not yet implemented correctly!\n";
        exit(1);
    }
    cout << "Parallel dot product = " << alpha_parallel
         << " time using reduction method = " << time_red
         << " sec, time using critical method " << time_critical
         << " sec" << endl;

// De-allocate memory
    delete [] a;
    delete [] b;

    return 0;
}
```

After explaining the modifications I have done to the code, we are interested in visualizing the results obtained against different numbers of threads, in order to analyze the scalability of the code. I was unsure of what graphical representation would be the best to visualize my results but, at the end, I decided to use the Matlab function *surf()*, which allows us to represent the data obtained in a three dimensional space (with axes given by number of threads, size of the vector and the metric considered). In figures 1 and 2 we can find the scaling of the code with reduction and with critical against different number of threads and against different sizes (number of elements N) of the vector for which we are computing the dot product.

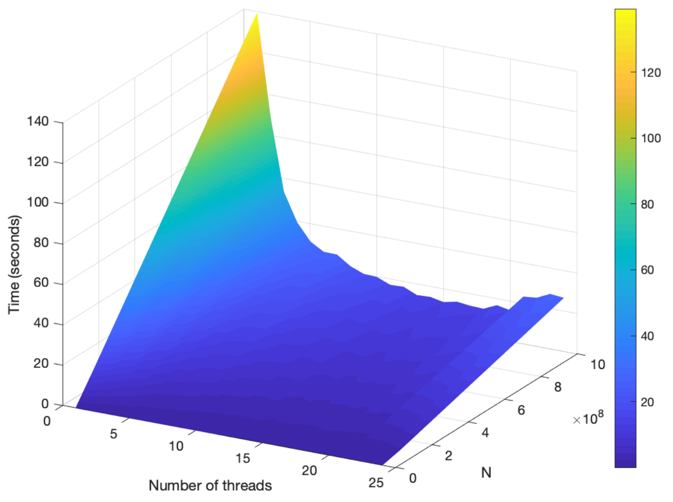

Figure 1: Scaling results by using reduction against different number of threads and number of elements N of the input vector.

In figures 3 and 5 we can find the parallel efficiency of the code with reduction and with critical against different number of threads and against different sizes (number of elements N) of the vector for which we are computing the dot product. The parallel efficiency has been computed simply as the ratio between the time obtained with a single thread and the time obtained for a higher number of threads (keeping N fixed).

According to the results I obtained, depicted in the figures above, it would be beneficial to use a multi-threaded version of the dot product for all the cases analysed, but - in particular - starting from a size $N = 10^7$, as can be noticed by observing the results with the times of the serial implementation.

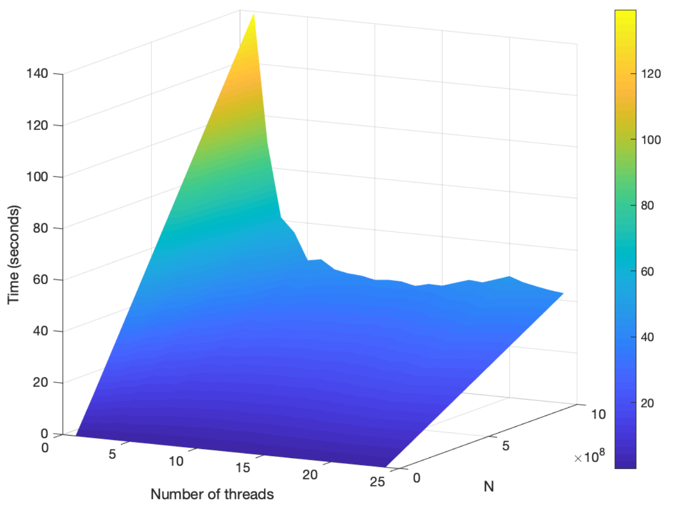

Figure 2: Scaling results by using critical against different number of threads and number of elements N of the input vector.

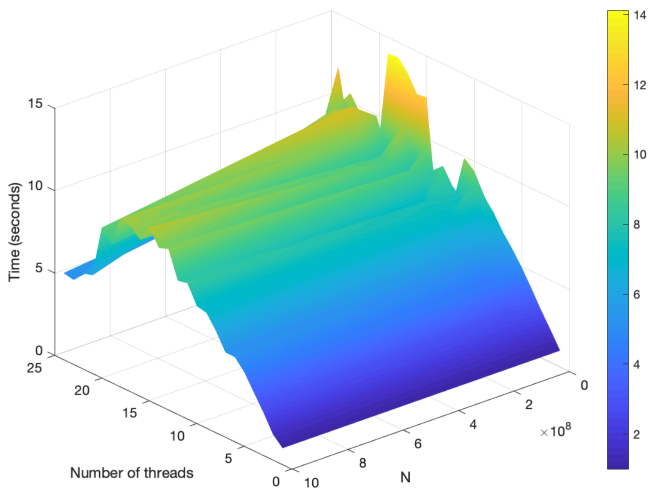

Figure 3: Parallelel efficiency of the implementation using reduction against different number of threads and number of elements N of the input vector.

2. The Mandelbrot set using OpenMP [30 points]

2.1. Mandelbrot set

In this second exercise regarding the Mandelbrot set I firstly understood the theory behind the so called ”most complex object in mathematics” (James Gleick, ”Chaos: Making a New Science”) and, secondly, how to plot it using C. We know that the Mandelbrot set is defined as a set of complex numbers for which the sequence $z, f_c(z), f_c^2(z), f_c^3(z)$... does not approach infinity (i.e., it remains bounded). In order to produce the plot we have to understand how the given *manedl_seq.c* code works and what are the interactions between the given variables. Mathematically we have:

$$\lim_{n\to\infty} \|z_{n+1} = z_n^2 + c\| \text{ does not approach } \infty$$

in which the norm considered is Euclidean and simply defined as:

$$\|z\| = \sqrt{x^2 + y^2}$$

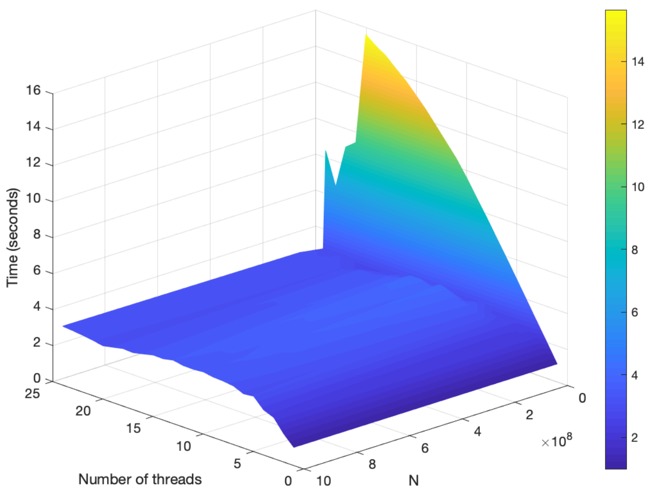

Figure 4: Parallelel efficiency of the implementation using critical against different number of threads and number of elements N of the input vector.

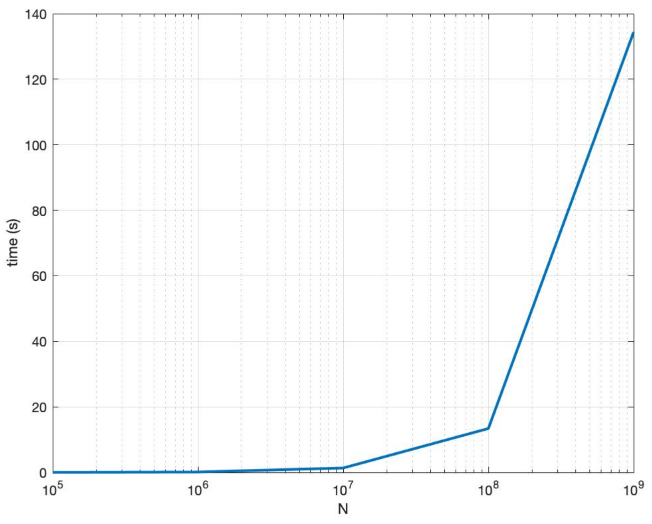

Figure 5: Time required by the serial implementation against different number of elements N of the input vector.

In the equation above, x indicates the real part, while y is the imaginary part. We can rewrite this equation as follows:

$$\|z\|^2 = x^2 + y^2.$$

After this brief recap of the theory we can proceed with the real C implementation. We know from the last equation and the theory that a given complex number belongs to the Mandelbrot set if $2 > \sqrt{x^2 + y^2}$ or $4 > x^2 + y^2$. In my code, I implemented a *while loop* which takes into consideration the maximum number of iterations and the boundary of the Mandelbrot set. In the loop I incremented the value of x and y so that, at every iteration, they are updated and ready to check if in the two constraint are satisfied and if the number belongs to the set.

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

```
#include <unistd.h>
#include <time.h>
#include <sys/time.h>

#include "pngwriter.h"
#include "consts.h"


unsigned long get_time () {
    struct timeval tp;
    gettimeofday (&tp, NULL);
    return tp.tv_sec * 1000000 + tp.tv_usec;
}

int main (int argc, char** argv) {
    png_data* pPng = png_create (IMAGE_WIDTH, IMAGE_HEIGHT);

    double x, y, x2, y2, cx, cy;
    cy = MIN_Y;

    double fDeltaX = (MAX_X - MIN_X) / (double) IMAGE_WIDTH;
    double fDeltaY = (MAX_Y - MIN_Y) / (double) IMAGE_HEIGHT;

    long nTotalIterationsCount = 0;
    unsigned long nTimeStart = get_time ();

    long i, j, n;

    n = 0;
    // do the calculation
    for (j = 0; j < IMAGE_HEIGHT; j++) {
        cx = MIN_X;
        for (i = 0; i < IMAGE_WIDTH; i++) {
            x = cx;
            y = cy;
            x2 = x * x;
            y2 = y * y;

            // compute the orbit z, f(z), f^2(z), f^3(z), ...
            // count the iterations until the orbit leaves the circle |z
                |=2.
            // stop if the number of iterations exceeds the bound
                MAX_ITERS.


            n = 0;
            while ((x2 + y2) < 4 && n < MAX_ITERS) {
                y = 2 * x * y + cy;
                x = x2 - y2 + cx;
                x2 = x * x;
                y2 = y * y;
                nTotalIterationsCount += 1;
                n += 1;
            }

            // n indicates if the point belongs to the mandelbrot set
            // plot the number of iterations at point (i, j)
            int c = ((long) n * 255) / MAX_ITERS;
            png_plot (pPng, i, j, c, c, c);
            cx += fDeltaX;
        }
        cy += fDeltaY;
    }
    unsigned long nTimeEnd = get_time ();

    // print benchmark data
    printf ("Total time:                 %g milliscconds\n", (nTimeEnd -
        nTimeStart) / 1000.0);
    printf ("Image size:                 %ld x %ld = %ld Pixels\n",
            (long) IMAGE_WIDTH, (long) IMAGE_HEIGHT, (long) (IMAGE_WIDTH
                * IMAGE_HEIGHT));
    printf ("Total number of iterations: %ld\n", nTotalIterationsCount);
    printf ("Avg. time per pixel:        %g microseconds\n", (nTimeEnd -
        nTimeStart) / (double) (IMAGE_WIDTH * IMAGE_HEIGHT));
    printf ("Avg. time per iteration:    %g microseconds\n", (nTimeEnd -
        nTimeStart) / (double) nTotalIterationsCount);
    printf ("Iterations/second:          %g\n", nTotalIterationsCount /
        (double) (nTimeEnd - nTimeStart) * 1e6);
    // assume there are 8 floating point operations per iteration
    printf ("MFlop/s:                    %g\n", nTotalIterationsCount *
        8.0 / (double) (nTimeEnd - nTimeStart));

    png_write (pPng, "mandel.png");
    return 0;
}
```

2.2. Mandelbrot set: parallelization

With the aim of parallelizing the Mandelbrot set I used the *reduction clause* in a *parallel for* section. This approach produced the correct result in almost every pixel, despite a few one which were suppose to be black instead of white and vice versa. This error is called a *race condition* and the understanding of this problem is crucial to arrive at the correct result. Basically when we write a simple code without parallelization we use only one thread which performs all the calculations. In contrast, in a "parallel code", the *master thread* - i.e. the thread that performs the calculations in the non-parallelized section - whenever it hits a parallel section, splits itself and creates a *thread team* that works together to perform the computations. In the parallel section, sometimes, two threads work with the same variable and it could happen that the second overwrites the value of the first one: this situation is the so-called *race condition.* In order to avoid it, I had to set all the variable as private: in this way, every thread has its own copy of the variable and, thus, the race condition is no longer possible.

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

#include <unistd.h>
#include <time.h>
```

```
#include <sys/time.h>

#include "pngwriter.h"
#include "consts.h"
#include <omp.h>


unsigned long get_time () {
   struct timeval tp;
   gettimeofday (&tp, NULL);
   return tp.tv_sec * 1000000 + tp.tv_usec;
}

int main (int argc, char** argv) {
   png_data* pPng = png_create (IMAGE_WIDTH, IMAGE_HEIGHT);

   double x, y, x2, y2, cx, cy;   // x real, y imaginary
   cy = MIN_Y;

   double fDeltaX = (MAX_X - MIN_X) / (double) IMAGE_WIDTH;
   double fDeltaY = (MAX_Y - MIN_Y) / (double) IMAGE_HEIGHT;

   long nTotalIterationsCount = 0;
   unsigned long nTimeStart = get_time ();

   long i, j, n;

   n = 0;
   // do the calculation
   #pragma omp parallel firstprivate(x,x2,y,y2,cx,fDeltaY,fDeltaX,i,j,n
      )
   {
      #pragma omp for reduction(+:nTotalIterationsCount) schedule(
         dynamic)
      for (j = 0; j < IMAGE_HEIGHT; j++) {
         cx = MIN_X;
         for (i = 0; i < IMAGE_WIDTH; i++) {
            x = cx;
            y = cy;
            x2 = x * x;
            y2 = y * y;

            // compute the orbit z, f(z), f^2(z), f^3(z), ...
            // count the iterations until the orbit leaves the circle |
               z|=2.
            // stop if the number of iterations exceeds the bound
               MAX_ITERS.

            n = 0;
            while ((x2 + y2) < 4 && n < MAX_ITERS) {
               y = 2 * x * y + cy;
               x = x2 - y2 + cx;
               x2 = x * x;
```

```
                y2 = y * y;
                n += 1;
            }
            nTotalIterationsCount += n;
            // n indicates if the point belongs to the mandelbrot set
            // plot the number of iterations at point (i, j)

            int c = ((long) n * 255) / MAX_ITERS;
            png_plot (pPng, i, j, c, c, c);
            cx += fDeltaX;


        }
        cy += fDeltaY;
    }
}


unsigned long nTimeEnd = get_time ();

// print benchmark data
printf ("Total time:                 %g millisconds\n", (nTimeEnd -
    nTimeStart) / 1000.0);
printf ("Image size:                 %ld x %ld = %ld Pixels\n",
        (long) IMAGE_WIDTH, (long) IMAGE_HEIGHT, (long) (IMAGE_WIDTH
            * IMAGE_HEIGHT));
printf ("Total number of iterations: %ld\n", nTotalIterationsCount);
printf ("Avg. time per pixel:        %g microseconds\n", (nTimeEnd -
     nTimeStart) / (double) (IMAGE_WIDTH * IMAGE_HEIGHT));
printf ("Avg. time per iteration:    %g microseconds\n", (nTimeEnd -
     nTimeStart) / (double) nTotalIterationsCount);
printf ("Iterations/second:          %g\n", nTotalIterationsCount /
    (double) (nTimeEnd - nTimeStart) * 1e6);
// assume there are 8 floating point operations per iteration
printf ("MFlop/s:                    %g\n", nTotalIterationsCount *
    8.0 / (double) (nTimeEnd - nTimeStart));

png_write (pPng, "mandel.png");
return 0;
}
```

To conclude this section, I want to present the results obtained by running the serial version of the Mandelbrot set against the parallelized version which I implemented. As we can see from figure 6, the time (in milliseconds) required by the serial version quickly increases when the size of the picture increases. On the contrary, the parallelized version performs way better and allows us to obtain the same output (i.e., the graphical representation o the Mandelbrot set) in a significantly lower time.

3. Bug hunt [20 points]

3.1. omp_bug1.c

In the first script, the error is that the *pragma omp parallel for* has to be declared just before the for loop. In this case, in order to solve the problem, I suggest to move the omp declaration just

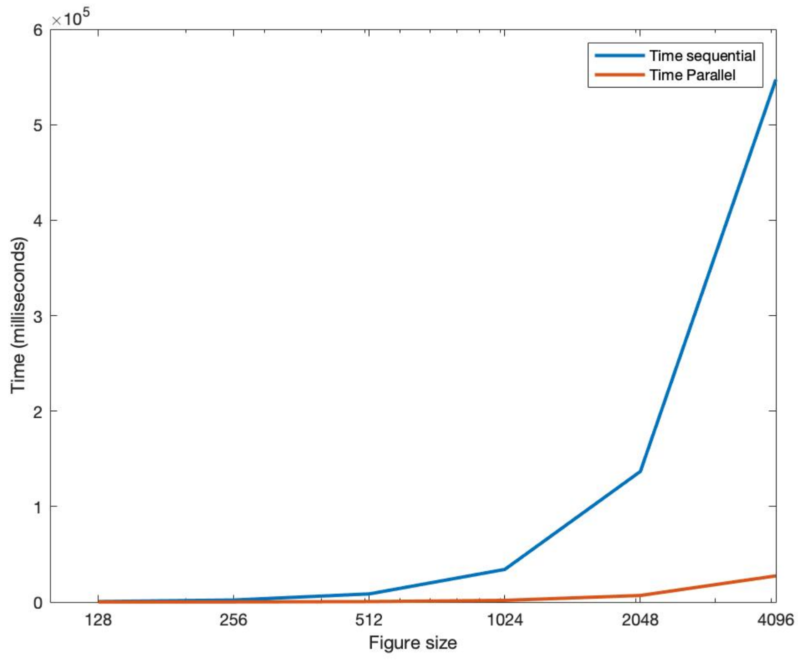

Figure 6: Serial versus parallelized Mandelbrot against different image sizes.

before the for loop and also to remove the curly brackets because they give an error in function main, line 27. With the removal of the curly brackets, the code works as expected.

```
/* FILE: omp_bug1.c
* DESCRIPTION:
*   This example attempts to show use of the parallel for construct.
    However
*   it will generate errors at compile time.  Try to determine what is
    causing
*   the error.
*****************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N        50
#define CHUNKSIZE    5

int main (int argc, char *argv[]) {
  int i, chunk, tid;
  float a[N], b[N], c[N];

  /* Some initializations */
  for (i = 0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;


  tid = omp_get_thread_num();
  #pragma omp parallel for      \
  shared(a,b,c,chunk)           \
  private(i,tid)                \
  schedule(static,chunk)
  for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
  }
  /* end of parallel for construct */

}
```

3.2. omp_bug2.c

The second script should print, for every thread, both when it begins to do its job and when it has finished to do its computations. The bug is that only the last thread is printed at the end of the computation. The solution is to privatize the variable *tid* and *total*, so that every thread has its own copy of the variable.

```
/* FILE: omp_bug2.c
* DESCRIPTION:
*   Another OpenMP program with a bug.
******************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  int nthreads, i, tid;
  float total;

  /*** Spawn parallel region ***/
  #pragma omp parallel firstprivate(tid, total)
  {
    /* Obtain thread number */
    tid = omp_get_thread_num();
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d is starting...\n", tid);

    #pragma omp barrier

    /* do some work */
    total = 0.0;
    #pragma omp for schedule(dynamic,10)
    for (i = 0; i < 1000000; i++)
      total = total + i * 1.0;

    printf ("Thread %d is done! Total= %e\n", tid, total);

  } /*** End of parallel region ***/
}
```

3.3. omp_bug3.c

In this third code, we can easily notice that the program never terminates its execution. We know from the theory that the barrier clause is used to "stop" the threads and wait until all of them have finished their work: afterwards, once they have been all synchronised, they can continue with the computations in the parallel section. In this specific code, the problem is relative to the number of barriers that the threads hit before the end of the program. At the end of the program, N-2 threads hit a barrier less than the others and, thus, we reach a situation of stall in which they keep waiting for the other two threads, that will however never come. In order to solve this bug we just have to remove the *#pragma omp barrier* in the *print_results().* By removing this last barrier, every thread hits - during the computations - the same number of barriers, thus solving the bug.

```
/******************************************************************
* FILE: omp_bug3.c
* DESCRIPTION:
*   Run time error
******************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N     50

int main (int argc, char *argv[]) {
  int i, nthreads, tid, section;
  float a[N], b[N], c[N];
  void print_results(float array[N], int tid, int section);

  /* Some initializations */
  for (i = 0; i < N; i++)
    a[i] = b[i] = i * 1.0;

  #pragma omp parallel private(c,i,tid,section)
  {
    tid = omp_get_thread_num();
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }

    /*** Use barriers for clean output ***/
    #pragma omp barrier
    printf("Thread %d starting...\n", tid);
    #pragma omp barrier

    #pragma omp sections nowait
    {
      #pragma omp section
      {
        section = 1;
        for (i = 0; i < N; i++)
          c[i] = a[i] * b[i];
        print_results(c, tid, section);
      }

      #pragma omp section
      {
        section = 2;
        for (i = 0; i < N; i++)
          c[i] = a[i] + b[i];
        print_results(c, tid, section);
      }

    } /* end of sections */

    /*** Use barrier for clean output ***/
    #pragma omp barrier
    printf("Thread %d exiting...\n", tid);

  } /* end of parallel section */
}


void print_results(float array[N], int tid, int section) {
  int i, j;

  j = 1;
  /*** use critical for clean output ***/
  #pragma omp critical
  {
    printf("\nThread %d did section %d. The results are:\n", tid,
        section);
    for (i = 0; i < N; i++) {
      printf("%e  ", array[i]);
      j++;
      if (j == 6) {
        printf("\n");
        j = 1;
      }
    }
    printf("\n");
  } /*** end of critical ***/


  printf("Thread %d done and synchronized.\n", tid);

}
```

3.4. omp_bug4.c

If we run this fourth code, we encounter a *Segmentation fault (core dumped)*. After doing some research, I found that the maximum stack of one ICS cluster node is 8192 kbytes, which means that, in the code, I cannot allocate more than the 8192 kbytes of data. The variable a is an NxN matrix of doubles (8 bytes each) and N is 1048., thus leading to:

$$1048 * 1048 * 8 = 9786432 \text{ bytes or } 8580.5 \text{ kbytes}$$

which exceeds the maximum of the ICS cluster. In order to solve this bug we have to set N to the maximum value allowed for the given stack size, i.e. we have to set $N = 1023$.

Remark: If we try to run the code with $N = 1024$ the result is a segmentation fault of the same type of the previous one. This is because the weight of the matrix in terms of kbytes is exactly 8192, but since we have to save not only the matrix (but also, e.g., the indices of the matrix), 8192 kbytes are not enough.

```
/******************************************************************
* FILE: omp_bug4.c
* DESCRIPTION:
*   This very simple program causes a segmentation fault.
******************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1023 //48

int main (int argc, char *argv[]) {
  int nthreads, tid, i, j;
  double a[N][N];

  /* Fork a team of threads with explicit variable scoping */
  #pragma omp parallel shared(nthreads) private(i,j,tid)
  {

    /* Obtain/print thread info */
    tid = omp_get_thread_num();
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);

    /* Each thread works on its own private copy of the array */
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        a[i][j] = tid + i + j;

    /* For confirmation */
    printf("Thread %d done. Last element= %f\n", tid, a[N - 1][N - 1]);

  }  /* All threads join master thread and disband */

}
```

3.5. omp_bug5.c

Finally the last bugged code is a neverending program. This problem is caused by an overlap of two open locks which cannot permit to the threads to continue with their computation. This problem can be easily solved by separating the locks and not nesting one inside the other as follows:

```
/******************************************************************
* FILE: omp_bug5.c
* DESCRIPTION:
*   Using SECTIONS, two threads initialize their own array and then add
*   it to the other's array, however a deadlock occurs.
******************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
#define PI 3.1415926535
#define DELTA .01415926535

int main (int argc, char *argv[]) {
  int nthreads, tid, i;
  float a[N], b[N];
  omp_lock_t locka, lockb;

  /* Initialize the locks */
  omp_init_lock(&locka);
  omp_init_lock(&lockb);

  /* Fork a team of threads giving them their own copies of variables
     */
  #pragma omp parallel shared(a, b, nthreads, locka, lockb) private(tid
     )
  {

    /* Obtain thread number and number of threads */
    tid = omp_get_thread_num();
    #pragma omp master
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);
    #pragma omp barrier

    #pragma omp sections nowait
    {
      #pragma omp section
      {
        printf("Thread %d initializing a[]\n", tid);
        omp_set_lock(&locka);
        for (i = 0; i < N; i++)
          a[i] = i * DELTA;
        omp_unset_lock(&locka);

        omp_set_lock(&lockb);
        printf("Thread %d adding a[] to b[]\n", tid);
        for (i = 0; i < N; i++)
          b[i] += a[i];
        omp_unset_lock(&lockb);
      }

      #pragma omp section
      {
        printf("Thread %d initializing b[]\n", tid);
        omp_set_lock(&lockb);
        for (i = 0; i < N; i++)
          b[i] = i * PI;
        omp_set_lock(&locka);

        omp_unset_lock(&lockb);
        printf("Thread %d adding b[] to a[]\n", tid);
        for (i = 0; i < N; i++)
          a[i] += b[i];
        omp_unset_lock(&locka);
      }
    }  /* end of sections */
  }  /* end of parallel region */

}
```

4. Parallel histogram calculation using OpenMP [20 points]

In the parallelization of the histogram calculation I used a reduction clause that allowed me to reduce the time for the computations. In table 1 we can find the results of the parallelization with up to 10 threads (please notice that the serial execution of the original implementation is MANCA). In figure 7 we can observe a graphical representation of the scaling results for up to 10 threads: in particular, we can observe that we gain a significant reduction of the computational till up to 5 threads, while afterwards the gains start to decrease. The latter behaviour could be explained as a consequence of the relative small size of our input: in other words, initializing too many threads offsets somehow the advantages of parallelizing our application, given the size of the problem.

```
#include <iostream>
#include "walltime.h"
#include <stdlib.h>
#include <random>
#include <omp.h>


#define VEC_SIZE 1000000000
#define BINS 16

using namespace std;

int main() {
    double time_start, time_end;

    // Initialize random number generator
    unsigned int seed = 123;
    float mean = BINS / 2.0;
    float sigma = BINS / 12.0;
    std::default_random_engine generator(seed);
    std::normal_distribution<float> distribution (mean, sigma);
    // Generate random sequence
    // Note: normal distribution is on interval [-inf; inf]
    //       we want [0; BINS-1]
    int *vec = new int[VEC_SIZE];
    for (long i = 0; i < VEC_SIZE; ++i) {
        vec[i] = int(distribution(generator));
        if (vec[i] < 0)
            vec[i] = 0;
        if (vec[i] > BINS - 1)
            vec[i] = BINS - 1;
    }

    // Initialize histogram
    // Set all bins to zero
    long dist[BINS];
    for (int i = 0; i < BINS; ++i) {
        dist[i] = 0;
    }

    time_start = wall_time();

    // Scaling 1 to 11 threads
    #pragma omp parallel for reduction(+:dist)
    for (long i = 0; i < VEC_SIZE; ++i) {
        dist[vec[i]] = dist[vec[i]] + 1;
    }

    time_end = wall_time();

    // Write results
    for (int i = 0; i < BINS; ++i) {
        cout << "dist[" << i << "]=" << dist[i] << endl;
    }
    cout << "Time: " << time_end - time_start << " sec" << endl;

    delete [] vec;

    return 0;
}
```

5. Parallel loop dependencies with OpenMP [20 points]

We know from the theory that, in some cases, it is not trivial to parallelize a for loop, especially in the case of a loop-dependant variable. In other words, a possible scenario is that in the $n+1$ iteration we need a variable that has been computed in the previous iteration n. In these cases, the first thing that we have to do is solving the dependence between the variables. and then start the parallelization. In our case the code creates a pointer vector (opt) in which:

$$opt[n+1] = opt[n] * up$$

In order to solve this problem, my first attempt consisted in simply using the *pow()* function, in order to compute - at every iteration - the exact value at position n of the pointer vector. By doing

# Threads	Time(s)
1	0.826919
2	0.438337
3	0.291285
4	0.219064
5	0.17477
6	0.144787
7	0.124644
8	0.10845
9	0.0976775
10	0.0879112

Table 1: Time required for the histogram computation with different numbers of threads.

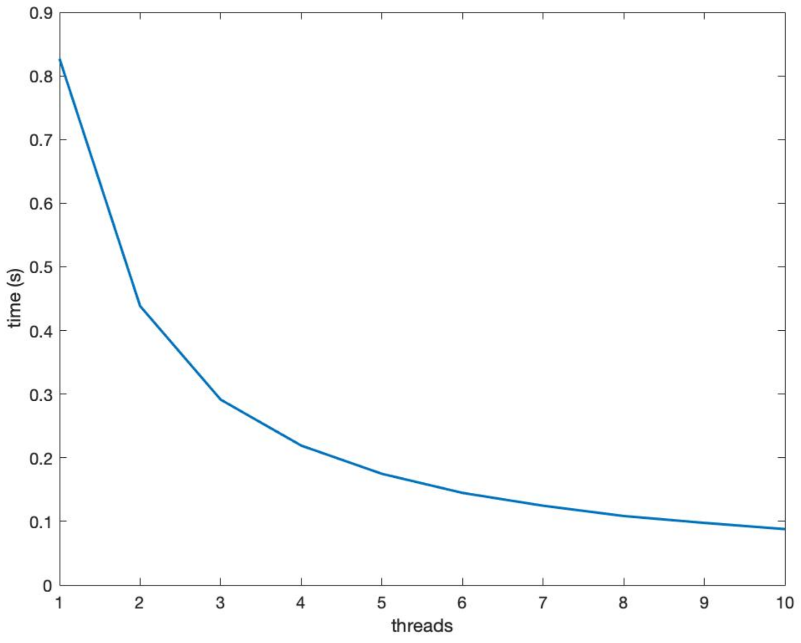

Figure 7: Scaling results for the histogram computation with different number of threads.

so, I - however - realized that the repeated calls to the function *pow()* were significantly expensive and my code was actually performing worse than the serial version, despite the parallelization.

In the second version of my code, I decided then to minimize the utilisation of the *pow()* function by adding an *if ... else* construct in which only the first computation of each thread would use the function and, after that, the computations would be carried over with the regular $Sn * up$ defined above. The results obtained for a number of threads ranging from 1 to 18 are summarized in table 2. In figure 8 we can find a graphical representation of the results obtained: as happened in the previous exercise, we can notice a significant scaling of our parallelized code up to 5 threads, after which number the performance starts to decrease (and what is gained from adding an additional thread is only a small speed up).

```
# include <stdlib.h>
# include <math.h>
# include "walltime.h"
# include <omp.h>
```

```
int main ( int argc, char *argv[] ) {
    int N = 2000000000;
    double up = 1.00000001;
    double Sn = 1.00000001;
    double Sn_loop = 1.00000001;
    int n, n_new;
    n_new = -2;

    /* allocate memory for the recursion */
    double* opt = (double*) malloc ((N + 1) * sizeof(double));


    if (opt == NULL)  die ("failed␣to␣allocate␣problem␣size");

    double time_start = wall_time();


    #pragma omp parallel private(n) firstprivate(n_new)
    {
        #pragma omp for lastprivate(Sn)
        for (n = 0; n <= N; ++n) {
            if (n_new != n - 1) {
                Sn = Sn_loop * pow(up, n);
            } else {
                Sn *= up;
            }
            opt[n] = Sn;
            n_new = n;
        }

    }
    Sn *= up;

    printf("Parallel␣RunTime␣␣␣:␣␣%f␣seconds\n", wall_time() -
       time_start);
    printf("Final␣Result␣Sn␣␣␣␣:␣␣%.17g␣\n", Sn);


    double temp = 0.0;
    for (n = 0; n <= N; ++n) {
        temp +=  opt[n] * opt[n];
    }
    printf("Result␣||opt||^2_2␣:␣␣%f\n", temp / (double) N);
    printf ( "\n" );

    return 0;
}
```

# Threads	**Time(s)**
1	6.715699
2	3.377089
4	1.690796
6	1.132449
8	0.857513
10	0.724234
12	0.597950
14	0.569803
16	0.521026
18	0.517610

Table 2: Time required for the recursion code with different numbers of threads.

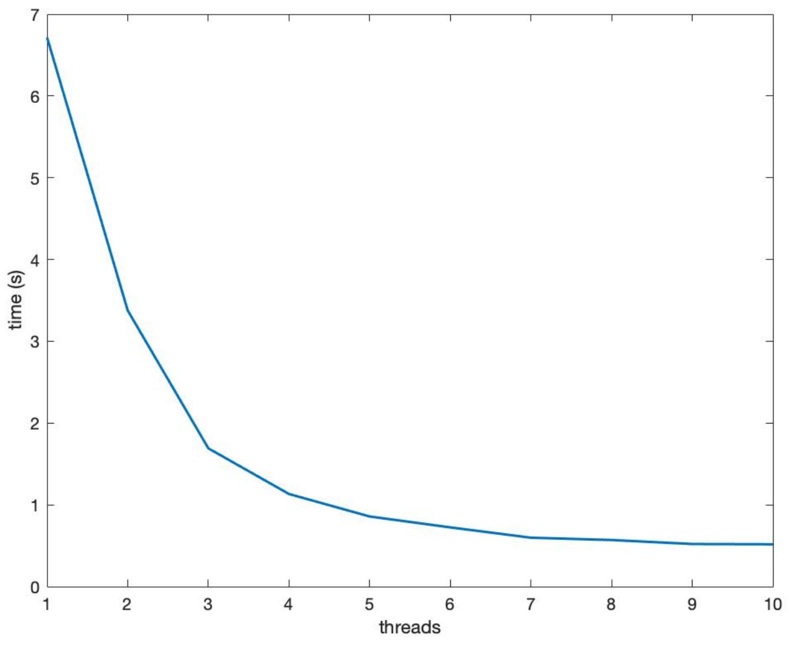

Figure 8: Scaling results for the recursion code with different number of threads.