Exercises

Ex1 [creating threads]. We will start with a very simple *Hello World* program. New thing to include is *omp.h* library.

```
#include <stdio.h>
#include <stdlib.h>
#include “omp.h”

int main (int argc, char *argv[])
{
printf (“Hello World! \n”);
return 0;
}
```

For parallelization of this code the next step is to add OpenMP **directives**. The first directive we'll use is **`#pragma omp parallel`** which creates a team of threads. Each thread will execute the next block of code.

parallel_hello.c

```
int main (int argc, char *argv[])
{
printf (“Hello World! \n”);

#pragma omp parallel
{
     printf (“Hello from a thread! 'n”);
}

printf (“I am sequential now. \n”);
return 0;
}
```

We can compile this program with: `gcc –fopenmp –o par_hello parallel_hello.c`.
Other compilers have other options. In case we are using Intel compiler `icc`, then we have to include flag `–openmp`.
Task: What happens when we execute `par_hello`? What if we leave out the flag -fopenmp when compiling?

When we run the program, we want to have answers to next questions:

– How many threads are created?
– Will it be the same on your computer as it is on my computer?
– How much control do we have over the number of threads that are created?

Function that help us to find out how many threads are being created is

```
int omp_get_num_threads().
```

Another important OpenMP function is

```
int omp_get_thread_num(),
```

which returns the identification number of the current thread. To see how this function works we will put another statement in parallel region of `parallel_hello.c` using:

```
      printf (“Hello from thread %d. \n”, omp_get_thread_num());
```

When we run this new version of the program several times, we will likely see that the order in which the threads do the work is in non-deterministic way.

Tip: When we run these examples it is nice to know a trick that helps us to sort the output. We use sort command `./par_hello | sort`. This way we can check to make sure that we got everything we expected, without worrying about the order.

Ex2. Now will we start with small exercises. The aim of this exercises is to give an introduction to OpenMP programming. The test examples are all written in C. Goal of this part is to give you some familiarity to the OpenMP syntax.

Ex2-a : *Controlling a parallel section*

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "omp.h"

int main()
  {
  int i, j, ID, nthreads;

  printf ("\n   ----- Exercise 1 ------\n\n");

  #pragma omp parallel private(i, j)
   {
   for ( i = 0; i < 1000;  i++ )
    for ( j = 0; j < 1000; j++ )
     ID = omp_get_thread_num();

   printf ("Thread %d : My value of ID is %d .\n", omp_get_thread_num(), ID);
   }

  return 0;
  }
```

Execute the program several times. Sometimes output is incorrect. What is a problem, and why? Fix the parallelization directive so that the program always prints the expected output.

Ex2-b : *Data-sharing*
The default storage type in OpenMO is *shared.* In many instances it is very useful to add the clause `default(none)` to the directives. If you take this as a habit you will always be reminded of what variables are in use and not forget to decide if a variable is to be shared or private.
This exercise illustrates some of the data-sharing clauses in OpenMP.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "omp.h"
```

```
int main()
  {
  int i, j, ID, nthreads;
  int n;

  printf("\n   ----- Exercise 2b ------\n\n");

  n = 10;
#pragma omp parallel private(n)
  {
  n += omp_get_thread_num();
  printf("Thread %d : My value of n is %d \n", omp_get_thread_num(), n);
  }

  j = 100000;
#pragma omp parallel for private(n)
  for (i = 1; i <= j; i++)
    n = i;

  printf("\nAfter %d iterations the value of n is %d \n\n", j, n);

  return 0;
  }
```

Run the program and determine is the output correct. Try to make the program print the correct value for n, in case of both for-parallel sections.

Ex2-c: *Synchronization*
This exercise tries to illustrate the usage of rudimentary OpenMP synchronization constructs. Try to make all the threads end at the same time.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include "omp.h"

int main()
  {
  int i, j, ID, nthreads;
  int n;
  time_t ts;

  printf("\n   ----- Exercise 2c ------\n\n");

#pragma omp parallel private(ts, ID)
  {
  ID = omp_get_thread_num();
  time(&ts);

  printf("Thread %d spawned at:\t %s", ID, ctime(&ts));

  /*The sleep() function shall cause the calling thread to be suspended from
execution until either the number of realtime seconds specified by the argument
seconds has elapsed or a signal is delivered to the calling thread and its
action is to invoke a signal-catching function or to terminate the process. The
suspension time may be longer than requested due to the scheduling of other
activity by the system.
    Definition:
        unsigned sleep(unsigned seconds);
   */

  sleep(1);
  if(ID%2 == 0)
    sleep(2);

  time(&ts);
  printf("Thread %d done at:\t %s", ID, ctime(&ts));
  }

  return 0;
  }
```

Ex2-d: *Scheduling*
The workload distribution for a loop can be controlled by a scheduling clause in OpenMP. It can be controlled either by a directive or by an environment variable. In this exercise we will use the first approach. The scheduling policy is controlled by directive

```
    #pragma omp for schedule ([clause],[chunk_size])
```

where `clause` can be any of `static, dynamic, guided` and `chunk_size` is an optional argument controlling the granularity of the workload distribution.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include "omp.h"

int main()
  {
  int i, j, ID, nthreads, *buffer;
  int n;
  FILE *fp;

  printf("\n   ----- Exercise 2d ------\n\n");

  n = 1000;
  buffer = (int*) malloc(n * sizeof(int));

  printf("Computing...");
  fflush(stdout);
#pragma omp parallel for schedule(runtime)
  for (i = 0; i < n; i++)
    {
    buffer[i] = omp_get_thread_num();
    usleep(random()%2000);
      /*The usleep() function suspends execution of the calling process until
either microseconds microseconds have elapsed or a signal is delivered to the
process and its action is to invoke a signal-catching function or to terminate
the process. System activity may lengthen the sleep by an indeterminate amount.
      */
    }
  printf("Done\n");
  fp = fopen("schedule.dat","w");
  for (i = 0; i < n; i++)
    fprintf(fp, "%d\t%d\n", i, buffer[i]);

  fclose(fp);

  printf("\nNow, run 'gnuplot schedule.gp' to visualize the scheduling
policy!\n\n");
  free(buffer);

  return 0;
  }
```

This exercise illustrates the different scheduling algorithms in OpenMP. Run the program several times, with different values for *schedule clause* (try *schedule (static)*).
After each run, execute `gnuplot schedule.gp`, and look at “schedule.png” to see how the schedule assigned parts of the iteration to different threads.

Content of `schedule.gp`:

```
set terminal png
set autoscale
set output "schedule.png"
set xlabel "Iteration"
set ylabel "Thread ID"
unset key
plot "schedule.dat" using 1:2 ls 4
```

Ex3. In this section we need to think of a way to parallelize codes that are being presented. First application creates three matrices according to a recursive secret formula. Second application then multiplies the matrices with each other a few times.

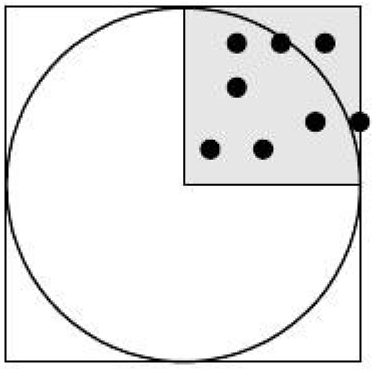
Ex3-a: Parallel matrix generation
The code below generates three matrices. Try to think of a way in which this can be made parallel in any way. Make sure that the printed output x is correct in your parallel version.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include <sys/time.h>

#define MATSIZE 900

double gettime(void) {
  struct timeval tv;
  gettimeofday(&tv,NULL);
  return tv.tv_sec + 1e-6*tv.tv_usec;
}

double check_sum(double **mat) {
  int i, j;
  double sum = 0.0;

  for(i = 0; i < MATSIZE; i++)
    for(j = 0; j < MATSIZE; j++)
```

```
      sum += mat[i][j];
  return sum;
}

/*
  This program generates some vectors and matrices
  It manipulates them and finally computes some global
  things that are printed
*/

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

  int i,j, k, iptr;
  double **mat_a, **mat_b, **mat_c, **mat_d;
  double  x, scal;
  double dtime;

  mat_a = (double **) malloc(MATSIZE * sizeof(double *));
  mat_b = (double **) malloc(MATSIZE * sizeof(double *));
  mat_c = (double **) malloc(MATSIZE * sizeof(double *));
  mat_d = (double **) malloc(MATSIZE * sizeof(double *));

  mat_a[0] = (double *) malloc(MATSIZE * MATSIZE * sizeof(double));
  mat_b[0] = (double *) malloc(MATSIZE * MATSIZE * sizeof(double));
  mat_c[0] = (double *) malloc(MATSIZE * MATSIZE * sizeof(double));
  mat_d[0] = (double *) malloc(MATSIZE * MATSIZE * sizeof(double));

  for (i = 0; i < MATSIZE; i++) {
    mat_a[i] = mat_a[0] + i*MATSIZE;
    mat_b[i] = mat_b[0] + i*MATSIZE;
    mat_c[i] = mat_c[0] + i*MATSIZE;
    mat_d[i] = mat_d[0] + i*MATSIZE;
  }
  /*----------------------------------------------------------------------*/
  printf("\n   ----- Exercise 1 ------\n\n");

  dtime = gettime();

  x = 0.35;
  for (j = 0; j < MATSIZE; j++) {
    for (i = 0; i < MATSIZE; i++) {
      x = 1 - frexp(sin(x), &iptr);
      mat_a[i][j] = x;
    }
  }

  x = 0.68;
    for (j = 0; j < MATSIZE; j++) {
      for (i = 0; i < MATSIZE; i++) {
      x = 1 - frexp(sin(x), &iptr);
      mat_b[i][j] = x;
    }
  }

  x = 0.24;
  for (j = 0; j < MATSIZE; j++) {
    for (i = 0; i < MATSIZE; i++) {
      x = 1 - frexp(sin(x), &iptr);
      mat_c[i][j] = x;
    }
  }
  dtime = gettime() - dtime;
  printf("The sum of the matrices evaluates to:\n"
       "Matrix A:%12.4f\t Matrix B:%12.4f\t Matrix C:%12.4f \n",
       check_sum(mat_a), check_sum(mat_b), check_sum(mat_c));
  printf("Time for the exercise: %9.5f seconds\n", dtime);

  free(mat_d[0]);
  free(mat_c[0]);
  free(mat_b[0]);
  free(mat_a[0]);

  free(mat_d);
  free(mat_c);
  free(mat_b);
  free(mat_a);

  return 0;
}
```

Ex4. [*Parallelization of more complex problem*] Monte Carlo Simulation - MCS

The following example represents a way of finding Π (Pi), based on a Monte Carlo simulation. The model is fairly simple; if we take a square with two units side length and inscribe that by a circle with unit radius, then we can throw a dart at the drawing multiple times. Each time when the dart falls within the circle, we increase a counter. We know that the area of the square is four unit-squared and the area of the circle is Π; thus the ratio of darts that fall within the circle is the ratio of four unit-squared that Π represents. For simplicity we reduce the problem to the upper right quadrant of the circle, e.g. one fourth of the original problem.



Monte Carlo methods are quite valuable tools in various fields of science and the simple model represented here may be replaced by a set of “real” problems within physics, chemistry and biology. The sequential code is straightforward:

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

#include "rand-generator.h"
#define P 100000000

int main()
  {
  double x, y;
  int i,j, count = 0;
  unsigned int seed;
  double r;
  time_t ctime;

  time ( &ctime );
  seed = ctime;

  for ( i = 0 ; i < P; i++ )
    {
    x = simple_random ( &seed );
    y = simple_random ( &seed );

    if ( ( x*x+y*y ) <= 1.0  )
      count++;
    }

  r = (double) 4.0 * count / P;
  printf ("PI is %lf \n", r);

  return 0;
  }
```

Content of `rand-generator.h`:

```
/*
 * Function for simple RNG to generate doubles.
 * Algorithm is LCG using constants from Wikipedia page.
 */
static unsigned int A = 1664525;
static unsigned int B = 1013904223;
static unsigned int SIMPLE_RANDMAX = 0xffffffff;

double simple_random(unsigned int * current) {
    unsigned int val = *current;
    val = ( (val * A) + B ) & SIMPLE_RANDMAX;
    *current = val;
    return ((double) val) / ((double) SIMPLE_RANDMAX);
}
```

Performance

Running the code with the time commands returns:

```
time ./monte
PI is 3.141500

real    0m3.324s
user    0m3.310s
sys     0m0.000s
```

It takes 3.4 seconds to perform the hundred million simulated dart throws, and returns Π with only three correct decimals. There are of course much better ways of calculating Π but the Monte Carlo simulation technique is in fact a useful mechanism for other problems, so speeding up the simulation is of real interest.

Parallelization
Parallelizing the code is straightforward. Instead of one simulated dart and one target, we simply can make more. For example, the two darts can be thrown in parallel without any problem since the result of one dart does not influence the result of other darts.
Task: What is that we need to take care about to get correct result?

You can also compare with MPI version of MCS for calculating Π. With MPI we would use MPI_Send and MPI_Recieve functions. Instead of this, in OpenMP we are using shared and private variables that allow us communication between threads (shared) and autonomy (private) during executions. Only using this paradigm we can parallelize our Monte Carlo simulation for calculating value of Π.

There are different ways to do parallelization of MCS for calculating number Π. In the first part of this exercises you can choose one of the approaches:

1. define chunks that each thread will execute and be careful about the size of the chunks while changing number of threads;
2. define chunks for each thread, regardless of the number of threads we use.

Using OpenMP functionality
In the second part of parallelization of MCS you will use OpenMP directives that frees us from defining chunks. The idea is that instead of chunks we use the `#pragma omp for` directive. To understand different types of for-directive, as part of the exercise, we will cover three types of schedule clause:

- static,
- dynamic,
- guided.

- *Static schedule*
  The iteration space is broken into chunks of approximately size N/nthreads of threads. Then these chunks are assigned to the threads in a Round-Robin fashion.
- *Static, N schedule (Interleaved)*
  The iteration space is broken in chunks of size N. Then these chunks are assigned to the threads in a Round-Robin fashion.
  Characteristics of static schedules:
  - Low overhead
  - Good locality (usually)
  - Can have load imbalance problems
- *Dynamic, N schedule*
  Variant of **dynamic**. The size of the chunks deceases as the threads grab iterations, but it is at least of size N. If no chunk is specified, then N = 1.
  Characteristics of dynamic schedules:
  - Higher overhead
  - Not very good locality (usually)
  - Can solve imbalance problems