3.1. OpenMP

Exercises are located at Exercises/5_OMP/
Solutions are located at Exercises/5_OMP/***/***_solution.cpp
To compile and run exercise programs use the line given in the head-comments in the code.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

OpenMP Introduction

OpenMP is an API, which consist of a set of compiler directives, library routines and environment variables that are used to create multithreaded applications on multiprocessor systems with a shared memory. It defines a simple interface that allows to parallelise tasks between cores of the CPU.

The OpenMP framework supports most processor architectures and operating systems, including Linux, Unix, AIX, Solaris, OS X, and Microsoft Windows platforms.

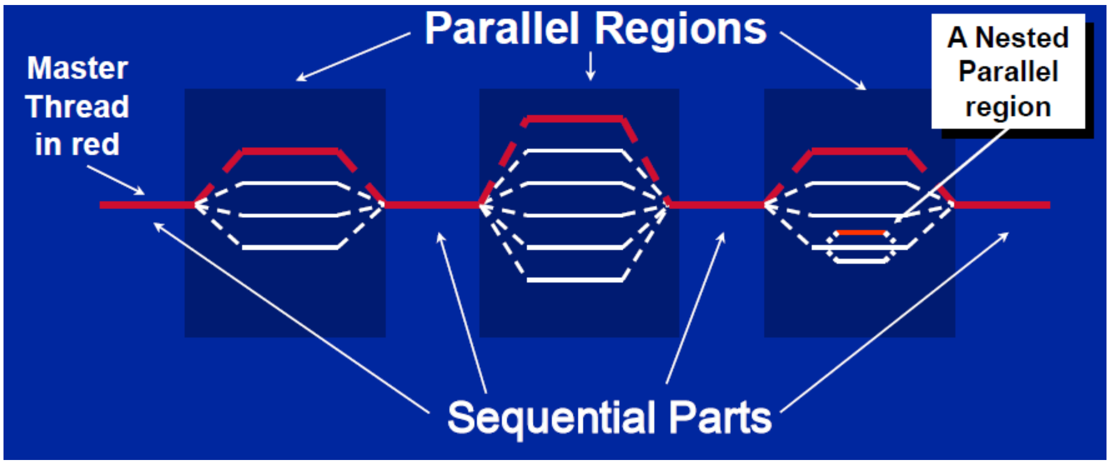

Fig. 1. Fork-join parallelism model of OpenMP.

The programming model of OpenMP is the fork-join parallelism: the master thread is created, when a program starts and when the calculation should be parallelised additional threads are created and the task is distributed between them (see Fig. 1). The nested parallelism is also possible within the OpenMP programming model. The parts of the code, which should be executed in parallel are marked with the directives of the compiler preprocessor:

#pragma omp parallel.

Threads in the OpenMP programming model are communicated by sharing variables. Each variable can be private or shared. The shared variables are seen by all threads. If the variable is declared as private a local copy of it is created by each thread.

Most of the constructs in OpenMP are compiler directives:

#pragma omp <construct> [clause [clause]...].

The implementation of programs using OpenMP constructs allow to keep the code unchangeable with respect to the single core version. In this case only an appropriate flag should be added during the compilation to enable the OpenMP directives. OpenMP includes such constructs, for example, as **omp for**, **omp reduction(op:list), omp num_threads(), omp scheduler [clause]**, etc. These directives allow to control the parallel regions, to collect the data from different threads, to synchronise threads, setting of the scheduler.

omp for construct provides a simple and easy to use interface for loop parallelisation. Let us compare two approaches of parallelisation - with worksharing construct and without:

Without worksharing:	With worksharing:

`#pragma omp parallel`
`{`
`  int id, i, Nthrds, istart, iend;`
`  id = omp_get_thread_num();`
`  Nthrds = omp_get_num_threads();`
`  istart= id * N / Nthrds;`
`  iend= (id+1) * N / Nthrds;`
`  if (id == Nthrds-1)`
`    iend= N;`
`  for(i=istart;I<iend;i++) { a[i] = a[i] + b[i];}`
`}`
	`#pragma omp parallel`
	`#pragma omp for`
	`   for(i=0;i<N;i++) { a[i] = a[i] + b[i];}`
	
	`Or:`
	
	`#pragma omp parallel for`
	`   for(i=0;i<N;i++) { a[i] = a[i] + b[i];}`

Also it is important to keep iteration completely independent using **omp for**. Otherwise the result will be incorrect.

omp num_threads() clause set the number of threads for the current parallel region.

Using OpenMP clauses it is possible to define if the variable shared or private:

- **shared(list)** - the variable is shared between threads, can not be applied to omp for loop;
- **private(list)** - creates a new copy of a variable for each thread, value is uninitialised;
- **firstprivate(list)** - like private, but copies are initialised with the value of the master thread;
- **lastprivate(list)** - like private, but the last value will be saved to a copy of the variable in the master thread;
- **default (private or none)** - sets the default clause for all variables;

By default all variables are shared.

omp reduction(op:list) specifies the list of variables, that should be reduced at the end of the parallel region. Each thread creates a local copy of the variable, when the parallel region is finished the master thread collects values stored in the local copies into it's variable, which is initialised according to the table (depending on the specified operation):

Reduction operands	
Operator	Initial value
+	0
*	1
-	0
&	0
|	0
^	0
&&	1
||	0

omp schedule [clause] defines the way, how the data is distributed between threads:

- **schedule(static [,chunk])** - blocks of iterations of size "chunk" are gradually distributed to each thread;
- **schedule(dynamic[,chunk])** - each thread grabs "chunk" iterations off a queue, when it finished the previous work, until all iterations have been handled;
- **schedule(guided[,chunk])** - threads dynamically grab blocks of iterations, the size of the block starts large and shrinks down to size "chunk" as the calculation proceeds;
- **schedule(runtime)** - schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).

In order to control the race conditions the framework contains a set of tools for a thread synchronisation including high level directives, like **omp critical**, **omp atomic**, **omp barrier**, and a set of simple lock functionality for a low level control:

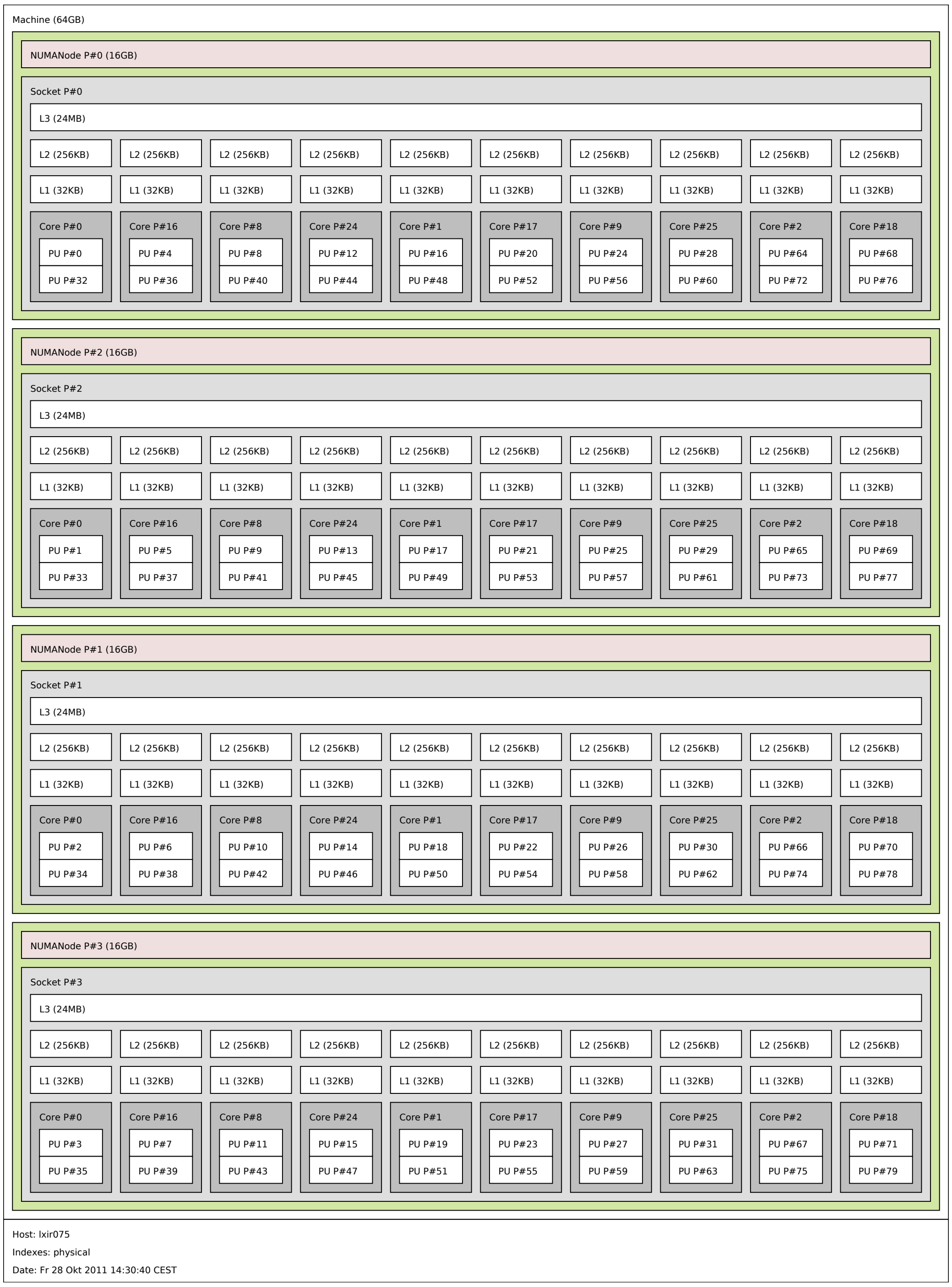
Machine (64GB)
NUMANode P#0 (16GB)
Socket P#0
L3 (24MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB)
Core P#0 Core P#16 Core P#8 Core P#24 Core P#1 Core P#17 Core P#9 Core P#25 Core P#2 Core P#18
PU P#0 PU P#4 PU P#8 PU P#12 PU P#16 PU P#20 PU P#24 PU P#28 PU P#64 PU P#68
PU P#32 PU P#36 PU P#40 PU P#44 PU P#48 PU P#52 PU P#56 PU P#60 PU P#72 PU P#76
NUMANode P#2 (16GB)
Socket P#2
L3 (24MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB)
Core P#0 Core P#16 Core P#8 Core P#24 Core P#1 Core P#17 Core P#9 Core P#25 Core P#2 Core P#18
PU P#1 PU P#5 PU P#9 PU P#13 PU P#17 PU P#21 PU P#25 PU P#29 PU P#65 PU P#69
PU P#33 PU P#37 PU P#41 PU P#45 PU P#49 PU P#53 PU P#57 PU P#61 PU P#73 PU P#77
NUMANode P#1 (16GB)
Socket P#1
L3 (24MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB)
Core P#0 Core P#16 Core P#8 Core P#24 Core P#1 Core P#17 Core P#9 Core P#25 Core P#2 Core P#18
PU P#2 PU P#6 PU P#10 PU P#14 PU P#18 PU P#22 PU P#26 PU P#30 PU P#66 PU P#70
PU P#34 PU P#38 PU P#42 PU P#46 PU P#50 PU P#54 PU P#58 PU P#62 PU P#74 PU P#78
NUMANode P#3 (16GB)
Socket P#3
L3 (24MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB) L1 (32KB)
Core P#0 Core P#16 Core P#8 Core P#24 Core P#1 Core P#17 Core P#9 Core P#25 Core P#2 Core P#18
PU P#3 PU P#7 PU P#11 PU P#15 PU P#19 PU P#23 PU P#27 PU P#31 PU P#67 PU P#71
PU P#35 PU P#39 PU P#43 PU P#47 PU P#51 PU P#55 PU P#59 PU P#63 PU P#75 PU P#79
Host: lxir075
Indexes: physical
Date: Fr 28 Okt 2011 14:30:40 CEST

Fig. 2. The structure of the lxir075 server.

- **critical** - only one thread at a time can enter a critical region;
- **atomic** - only one thread can operate with a memory location marked by atomic;
- **barrier** - each thread waits at the barrier until all threads arrive;
- **ordered** - threads execute in the sequential order;
- **flush** - forces data to be updated in memory so other threads see the most recent value;
- **locks** (both simple and nested) - allows to lock the part of the code, only one thread can enter the locked region; are useful, for example, if some elements of the array shared between threads should be locked.

OpenMP contains also a set of runtime library routines which allow, for instance, setting and to checking the number of threads, checking the maximum number of threads, getting the number of a current thread, checking the number of cores in a computer, to control the nested parallelism, operating with the simple lock functionality etc. They can be available by including only one file: **#include <omp.h>**. The most useful functions for the practicum are:

omp_set_num_threads() - set the number of thread created in the following parallel regions (if the number of threads is not specified by the corresponding clause there);

omp_get_num_threads() - obtain number of threads running on the current parallel region;

omp_get_thread_num() - returns id of the thread.

More of the functions together with their description can be found here:
https://computing.llnl.gov/tutorials/openMP/#RunTimeLibrary

NUMA architecture

Most of modern many and multi-core servers are heterogeneous systems. An access speed to different regions of memory in such systems is different for the different processing elements (for CPU cores or for CPUs themselves). In order to achieve a maximum speed of computation, it is necessary to control the distribution of memory between tasks. For the control the topology of a server should be known.

The topology of NUMA systems is shown on an example of the lxir075.gsi.de server (GSI, Darmstadt, Germany) on Fig. 2. The server is equipped with four Intel Xeon E7-4860 CPUs (10 physical cores at 2.27 GHz). Each physical core of this CPU can run simultaneously two threads with the hyperthreading technology. Therefore two logical cores correspond to each physical core. Each physical core has 32 kB of the level one (L1) cache memory and 256 kB of the level two (L2) cache memory shared between two logical cores. Each CPU has 24 MB of the level three (L3) cache memory, which is shared between all cores of the CPU. The total RAM of 64 GB is equally distributed between CPUs (16 GB for each of them).

During runtime the operating system can move threads from one logical core to another depending on the load of cores. It can happen, that the thread can jump to another CPU. For the NUMA architectures it is preferable to use only local RAM for the maximum performance. And if the thread would be moved to another CPU all data, which was used by this thread, would be located in the remote RAM. In order to prevent such jumps, which ruin the performance, each thread can be pined to a certain core. OpenMP itself doesn't contain tools for pinning. In order to do so for example the Pthreads library can be used. To use the library the corresponding header file should be included:

#include “pthread.h"

Then the pinning can be done using pthread_setaffinity_np() function:

```
int cpuId = … // index of the logical core
pthread_t thread  = pthread_self(); // get the current thread
cpu_set_t cpuset; // declare a cpu_set_t variable
CPU_ZERO(&cpuset); //macros initialises this variable
CPU_SET(cpuId, &cpuset); //set the cpuset variable according to the cpuId
int s = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset); //set the affinity
```

5_OMP/0_HelloWorldParallel: description

The first exercise on OpenMP is aimed to demonstrate how to create the parallel regions and synchronise them. It is proposed to parallelise a simple “Hello word” program:

Part of the source code of HelloWorld.cpp

```
#include <iostream>
using namespace std;

int main() {

  int id = 0;
  cout << " Hello world " << id << endl;

  return 0;
}
```

The tasks for the exercise are:

- Parallelise the program with OpenMP, create 10 threads.
- Synchronise threads using **omp critical**. Compare the results with and without synchronisation.
- Get the id of a current thread and print it out. Use variable **id** for this.

5_OMP/0_HelloWorldParallel: solution

The first task is solved by including **omp.h** and adding parallel region:

Part of the source code of HelloWorld_solution1.cpp

```
#include <omp.h>
#include <iostream>
using namespace std;

int main() {

  #pragma omp parallel num_threads(10)
  {
    int id = 0;

    cout << " Hello world " << id <<  endl;
  }

  return 0;
}
```

Such program will print symbols on the screen chaotically, because all 10 threads try to do this at the same time and only one of them can print at the current moment. To prevent such situation the threads should be synchronised. For example, omp critical can be added to the parallel region just before the printing:

Part of the source code of HelloWorld_solution2.cpp

```
int main() {

  #pragma omp parallel num_threads(10)
  {
    int id = 0;

    #pragma omp critical
    cout << " Hello world " << id <<  endl;
  }

  return 0;
}
```

To obtain the id of the thread the function **omp_get_thread_num()** should be used. Each thread has it's own id, therefore the function should be called in the parallel region by each thread individually. Then id is stored to the local variable of each thread and printed on the screen:

Part of the source code of HelloWorld_solution3.cpp

```
int main() {

  #pragma omp parallel num_threads(10)
  {
```

```
Part of the source code of HelloWorld_solution3.cpp
    int id = omp_get_thread_num();

    #pragma omp critical
    cout << " Hello world " << id <<  endl;
  }

  return 0;
}
```

5_OMP/1_Bugs: description

To illustrate different features of OpenMP and get familiar with it's functionality a set of simple programs containing bugs was created:

bug1.cpp - a program creates two arrays: **input** is filled with random numbers, **output** is filled by copying **input** element-wise in a parallel loop. Because of the bug entries in the **output** differ from the **input** and the program reports, that the output array is not correct.

bug2.cpp - a program fills two arrays: one in a scalar loop (outputScalar), another - in a parallel (outputParallel). Each element of arrays is filled with a sum of indices of all previous elements normalised on the current index. Because of the bug the scalar and parallel results are not the same.

bug3.cpp - a program creates an input array and fills it with random numbers. After this it calculates in a scalar loop the sum over all elements of the array and stores into the outputScalar array corresponding element of the input normalised by the sum. The same calculations are also done in the parallel loop but with a bug and the result is stored in a outputParallel array.   Because of the bug the scalar and parallel results are not the same.

bug4.cpp - a program do the same, as the previous, but the **omp for** loop is replaced here with a parallel construct only. Again, due to the bug in a parallel region results calculated in the scalar and parallel loops are not the same.

The task for this exercise is to find and fix the bugs.

5_OMP/1_Bugs: solution

bug1.cpp - in the initial program the number of elements in an array **N** is declared **private** in a **parallel for** construct:

```
Part of the source code of bug1.cpp
  #pragma omp parallel private(N) num_threads(NThreads)
  {
    #pragma omp for
    for(int i=0; i<N; i++)
      output[i] = input[i];
  }
```

With such declaration local variables **N** of the treads will not be initialised. It should be declared as **firstprivate** in order to initialise each local copy with a value of the master thread.

```
Part of the source code of bug1_solution.cpp
  #pragma omp parallel firstprivate(N) num_threads(NThreads)
  {
    #pragma omp for
    for(int i=0; i<N; i++)
      output[i] = input[i];
  }
```

bug2.cpp - the iterations of the loop in the parallel region are not independent:

```
Part of the source code of bug2.cpp
  #pragma omp parallel num_threads(NThreads)
  {
    #pragma omp for
    for(int i=1; i<N; i++)
    {
      tmp += i;
      outputParallel[i] = float(tmp)/float(i);
    }
  }
```

The variable **tmp** in current implementation depends on all previous iteration. Therefore when the loop is divided into several parts in the parallel region, values of local variables **tmp** are corrupted. Taking into account that value of **tmp** is a sum of the arithmetical progression, the iterations can be rewritten in an independent way:

```
Part of the source code of bug2_solution.cpp
  #pragma omp parallel num_threads(NThreads)
  {
    #pragma omp for
    for(int i=1; i<N; i++)
    {
      tmp = (1+i)*i/2;
      outputParallel[i] = float(tmp)/float(i);
    }
  }
```

bug3.cpp - in the initial program in the **omp for** construct during the sum calculation **nowait** clause destroys the implicit barrier and some threads start to fill the outputParallel array when the sum is not yet correctly calculated:

```
Part of the source code of bug3.cpp
  sum = 0;
  #pragma omp parallel firstprivate(N) num_threads(NThreads)
  {
    #pragma omp for nowait
    for(int i=0; i<N; i++)
    {
      #pragma omp atomic
      sum += input[i];
    }

    #pragma omp for
    for(int i=0; i<N; i++)
      outputParallel[i] = input[i]/sum;
  }
```

The solution would be to get rid of the **nowait**. For example, it can be done like this:

```
Part of the source code of bug3_solution.cpp
  sum = 0;
  {

```

```
Part of the source code of bug3_solution.cpp
49      for(int i=0; i<N; i++)
50      {
51        sum += input[i];
52      }
53
54      #pragma omp parallel firstprivate(N) num_threads(NThreads)
55      #pragma omp for
56      for(int i=0; i<N; i++)
57        outputParallel[i] = input[i]/sum;
58    }
```

bug4.cpp - unlike the previous exercise, here the barrier is missed:

```
Part of the source code of bug4.cpp
57    sum = 0;
58    #pragma omp parallel num_threads(NThreads)
59    {
60      int id, i, Nthrds, istart, iend;
61      id = omp_get_thread_num();
62      Nthrds = omp_get_num_threads();
63      istart= id * N / Nthrds;
64      iend= (id+1) * N / Nthrds;
65      if (id == Nthrds-1)
66        iend= N;
67      float sumLocal = 0;
68
69      CalcuateSum(input,istart, iend, sumLocal);
70
71      #pragma omp atomic
72      sum += sumLocal;
73
74      #pragma omp for
75      for(int i=0; i<N; i++)
76        outputParallel[i] = input[i]/sum;
77    }
```

Again, because of the missed barrier threads will continue without waiting for sum calculation to be completed. To prevent this the barrier should be put after the line 72:

```
Part of the source code of bug4_solution.cpp
57    sum = 0;
58    #pragma omp parallel num_threads(NThreads)
59    {
60      int id, i, Nthrds, istart, iend;
61      id = omp_get_thread_num();
62      Nthrds = omp_get_num_threads();
63      istart= id * N / Nthrds;
64      iend= (id+1) * N / Nthrds;
65      if (id == Nthrds-1)
66        iend= N;
67      float sumLocal = 0;
68
69      CalcuateSum(input,istart, iend, sumLocal);
70
Part of the source code of bug4_solution.cpp
    #pragma omp atomic
    sum += sumLocal;

    #pragma omp barrier

    #pragma omp for
    for(int i=0; i<N; i++)
      outputParallel[i] = input[i]/sum;
  }
```

5_OMP/2_Pi: description

The loops can be parallelised using OpenMP in two ways: manually (using only omp parallel construct and OpenMP functions) and using constructs together with clauses. In order to illustrate the difference it is proposed to parallelise a simple program using these two approaches.
The initial program is a scalar code to calculate the value of π by a formula:

$$\int_0^1 \frac{4}{1+x^2}dx = \pi$$

```
Part of the source code of pi.cpp
#include <stdio.h>
#include <omp.h>

static long num_steps = 100000000;
double step;
int main ()
{
  int i;
  double x, pi, sum = 0.0;
  double start_time, run_time;

  step = 1.0/(double) num_steps;

  start_time = omp_get_wtime();

  for (i=1;i<= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }

  pi = step * sum;
  run_time = omp_get_wtime() - start_time;
  printf("\n pi with %d steps is %f in %f seconds \n",num_steps,pi,run_time);
}
```

The tasks for the exercise are:
- parallelise the initial program using only **omp parallel** construct and OpenMP functions;
- parallelise the initial program making as less changes, as possible.

5_OMP/2_Pi: solution

For the solution of the first task it is necessary to define the number of threads (line 30), allocate variables to store the temporary sums calculated by each thread (line 31) and set the number of threads (line 32). In a parallel region it is needed to obtain the id of the thread (line 35, it is defined from zero to (nThreads-1)), calculate manually the maximum index in the loop-range of the current thread (line 37) and then perform the calculations. When calculating x the index i should be modified taking into account the offset for the current thread (line 38). In the end local sums should be summed up together (lines 43 - 44).

Part of the source code of pi_solution1.cpp

```
15  #include <stdio.h>
16  #include <omp.h>
17  
18  static long num_steps = 100000000;
19  double step;
20  int main ()
21  {
22  
23    double pi, sum = 0.0;
24    double start_time, run_time;
25  
26    step = 1.0/(double) num_steps;
27  
28    start_time = omp_get_wtime();
29  
30    const int nThreads = 2;
31    double sums[nThreads] = {0};
32    omp_set_num_threads(nThreads);
33    #pragma omp parallel
34    {
35      int id = omp_get_thread_num();
36  
37      for (int i=1;i<= num_steps/nThreads; i++){
38        double x = (i + id* num_steps/nThreads -0.5)*step;
39        sums[id] = sums[id] + 4.0/(1.0+x*x);
40      }
41    }
42  
43    for (int i = 0; i < nThreads; i++ )
44      sum += sums[i];
45  
46    pi = step * sum;
47    run_time = omp_get_wtime() - start_time;
48    printf("\n pi with %d steps is %f in %f seconds \n",num_steps,pi,run_time);
49  }
```

The solution of the second task would be to add only one line before the loop (line 30):

Part of the source code of pi_solution2.cpp

```
15  #include <stdio.h>
16  #include <omp.h>
17  
18  static long num_steps = 100000000;
19  double step;
20  int main ()
```

```
Part of the source code of pi_solution2.cpp
{
  int i;
  double x, pi, sum = 0.0;
  double start_time, run_time;

  step = 1.0/(double) num_steps;

  start_time = omp_get_wtime();

#pragma omp parallel for reduction(+ : sum) private(i,x)
  for (i=1;i<= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }

  pi = step * sum;
  run_time = omp_get_wtime() - start_time;
  printf("\n pi with %d steps is %f in %f seconds \n",num_steps,pi,run_time);
}
```

In the parallel region x and i should be declared as private and sum should be reduced. It is also worth to notice, that such code will compile and work even when OpenMP is disabled.

5_OMP/3_Matrix: description

This exercise is aimed to demonstrate the full power of CPU: the already vectorised exercise with matrix calculation (similar to 2_Vc/1_Matrix) should be parallelised between cores with OpenMP. In addition to the vectorised code the program contains now the part prepared for parallelisation:

```
   Part of the source code of Matrix.cpp
75     ///OpenMP
76    TStopwatch timerOMP;
77    for( int ii = 0; ii < NIter; ii++ ) // repeat several times to improve time measurement precision
78    {
79        //TODO modify the code below using OpenMP
80      for( int i = 0; i < N; i++ ) {
81        for( int j = 0; j < N; j+=float_v::Size ) {
82            float_v &aVec = (reinterpret_cast<float_v&>(a[i][j]));
83            float_v &cVec = (reinterpret_cast<float_v&>(c_omp[i][j]));
84            cVec = sqrt(aVec);
85        }
86      }
87    }
88    timerOMP.Stop();
```

5_OMP/3_Matrix: solution

Since the calculations are absolutely independent it is enough to add only one line to the initial code (line 79):

```
   Part of the source code of Matrix_solution.cpp
75     ///OpenMP
```

Part of the source code of Matrix_solution.cpp

```
  TStopwatch timerOMP;
  for( int ii = 0; ii < NIter; ii++ ) // repeat several times to improve time measurement precision
  {
#pragma omp parallel for num_threads(omp_get_num_procs())
    for( int i = 0; i < N; i++ ) {
      for( int j = 0; j < N; j+=float_v::Size ) {
          float_v &aVec = (reinterpret_cast<float_v&>(a[i][j]));
          float_v &cVec = (reinterpret_cast<float_v&>(c_omp[i][j]));
          cVec = sqrt(aVec);
      }
    }
  }
  timerOMP.Stop();
```