Details

In this assignment, you will practice with shared-address-space coding seen in class and, to this end, you need to learn something about OpenMP and its directives. The topic of the first two tasks is how to handle data in order to reduce the data access overheads while the last one focus the OpenMP tasks.

OpenMP is a very wide API and the entire course wouldn't be enough to appropriately cover all its aspects. We will focus on some of the mostly known constructs, clauses, and commands. At the end of the this document is attached a short appendix which explains the directives and functions offered by OpenMP for C++ required in this assignment: read and test all of them (as well as the material presented during the lecture) before trying to solve the tasks. For those who are interested, by the way, the OpenMP website (Links to an external site.) contains a more exhaustive documentation.

You are allowed to use only the directives described in the appendix of this document for implementing your solutions (even if some shortcuts are available).

Note for Windows Users

Visual Studio may do not support the most recent versions of OpenMP and you need OpenMP 3.0+ to get rid of Task #3.

My suggestion is that you use a Linux-based machine or, if you have Win10, you install WSL (version 1 should be enough, here is a guide) but you can find something better). If you have installed WSL, you can use a windows editor (choose your favorite) for coding and the WSL Terminal to compile and run your programs.

In both the cases (native Linux or WSL), you need to install the GCC compiler: open the terminal and just write

```
sudo apt install g++
```

General Notes

First of all, download lab2.zip Download lab2.zipcontaining the required files.

It's necessary that you use the given 'makefile' for compiling and testing your programs. It also generates a couple of macro (i.e., *nproc* and *cachelinesize*) that you need in the second task and you cannot edit. To compile and test the program, simple write:

```
make filename_without_extension [n=positive_integer]
```

possibly you can append n=... to change the input size of some tasks.

It's mandatory that your solutions work regardless of the values of the aforementioned macros (i.e., macros have to be considered as parameters of your solution).

Task #1 : approximating π by means of numerical integration

Given that ππ is the area of a circle with unit radius, we approximate ππ by computing the area of a quarter circle using *Riemann sums* as described in the following method. Let $f(x)=1-x^2-----\sqrt{}$f(x)=1−x2 be the formula describing the quarter circle for $x\in[0,1]\subseteq R$x∈[0,1]⊆R. We approximate the quarter circle area by means of the formula:
$\pi/4\approx\sum_{N-1i=0}\Delta xf(x_i)\pi/4\approx\sum i=0N-1\Delta xf(xi)$
where $x_i=i\Delta xxi=i\Delta x$ and $\Delta x=1/N\Delta x=1/N$. Such an approximation becomes more accurate as NN approaches to ∞∞. Your task consists in to analyze the sequential version in the file 'riemannsumpi.cpp' and write the parallel version of the method. Once you are done, compare the elapsed times obtained for the sequential and parallel versions by varying *n* from few thousands to hundreds of millions.
As *n* increases, the speedup should be reach a value similar to the number of physical cores available on your CPU while the error approximation error should decreases.

Requirements:

- Minimize the part of code that require the atomic execution
- Minimize the usage of shared data to avoid delays due to the cache coherence mechanism and related problems.

Task #2 : counting sort

Counting Sort (described here (Links to an external site.)) is an algorithm for sorting values belonging to a specific range [0, *k*). The file 'countingsort.cpp' contains the implementation of the sequential version of the algorithm and a naïve parallel solution. Your task is to implement a second parallel version that minimizes the effect of the *false sharing* problem. To this end, you need to use the proper memory

alignment (read about the **alignas** specifier in this link (Links to an external site.)) and calculate the minimal **padding** value required to avoid that different processors access the same cache-lines for writing.

NOTE: Insert the general formula to calculate the padding in your solution (e.g., by means of a macro). The formula should get *k* as parameter. Don't write just the result of your calculation since the function should work for different values of *k*.

Test you program changing the value of *k* from 90 to 2000 then compare the performance of the two parallel versions: try to find out the reason behind the performance gap.

Task #3 : height-balance check for BSTs

As you probably remember, OpenMP parallel loops don't fit well in case of accessing linked data structures (e.g., linked lists and trees) or programming recursive functions and the OpenMP tasks are more effective. The 'balancetree.cpp' file contains a program that builds a height-balanced Binary Search Tree then verifies such a property according to the definition reported here (Links to an external site.) (in case you forgot). Your task is to provide an equivalent and efficient version of the functions **height** and **isbalanced** that calculate the height of the tree and check if a tree is height-balanced, respectively. Do not expect any kind of improvement from the parallelization, just check that the result of your implementation is correct.

HINTS:

- Tasks produced by the two functions can be collected in the same *pool*
- Usage of *final* clause is optional
- Pay attention to the visibility of the variables

SUBMISSION:

Submit your implementations on Canvas.

Appendix on OpenMP

Here is a brief summary of some functionalities offered by the OpenMP API. Read all paragraphs and test the code snippets on a computer.

OpenMP offers many types of constructs for different needs, but we are interested to the subset of them for which we can (somehow) predict the behavior on the system. This material is an adaptation of the content extracted from several pages found on the Web. Read

Including the header file

```
#include <omp.h>
```

You only need it to call OMP functions like, for example, omp_get_wtime(). If all you need are pragmas you don't need to include it.

The parallel construct

The parallel construct starts a parallel block. It creates a *team* of N threads (where N is determined at run-time, usually from the number of CPU cores, but may be affected by a few things), all of which execute the next statement (or the next block, if the statement is a {...} -enclosure). After the statement, the threads join back into one.

```
#pragma omp parallel
{
  // Code inside this region runs in parallel.
  printf("Ciao!\n");
}
```

This code creates a team of threads, and each thread executes the same code: it prints the text "Ciao!" followed by a newline, as many times as the threads created in the team. For a dual-core system, it will output the text twice. Once the program reaches the '}', the threads are joined back into one, as if in non-threaded program.

Variables

A variable in an parallel region can be either shared or private. If a variable is shared, one instance exists and it is shared among all threads. If a variable is private, then each thread in a team of threads has its own local copy of the private variable. The implicit rule is that a variable declared locally within the parallel region is private, otherwise is shared. Look at the following examples.

```
int a = 1;
#pragma omp parallel
{
  int b = 9;
  printf("%d", b);
}
```

In the example above, the variable 'a' is shared while 'b' is private. Sometimes, you may need to specify if a variable is shared or private. This is can be done with the attributes: shared and private. The example above can be rewritten using the explicit attributes.

```
int a = 1, b;
#pragma omp parallel shared (a) private (b)
{

  b = 9;
  printf("%d", b);
}
```

The for-loop construct
The for construct splits the for-loop so that each thread in the current team handles a different portion of the loop.

```
#pragma omp for

for(int n=0; n<10; ++n)

{

  printf(" %d", n);

}

printf(".\n");
```

This loop will output each number from 0...9 once (in arbitrary order). Internally, the above loop becomes into code equivalent to this:

```
int num_threads = omp_get_num_threads(); // number of threads in the team
int this_thread = omp_get_thread_num(); // thread id in [0,num_threads-1]
int my_start = (this_thread  ) * 10 / num_threads;

int my_end   = (this_thread+1) * 10 / num_threads;

for(int n=my_start; n<my_end; ++n)

  printf(" %d", n);
```

So each thread gets a different section of the loop, and they execute their own sections in parallel.

Note: #pragma omp for only delegates portions of the loop for different threads in the *current team*. A *team* is the group of threads executing the program. At program start, the team consists only of a single member: the master thread that runs the program.

To create a new team of threads, you need to specify the parallel keyword. It can be specified in the surrounding context:

```
#pragma omp parallel
{
 #pragma omp for
 for(int n=0; n<10; ++n) printf(" %d", n);
}
printf(".\n");
```

Equivalent shorthand is to specify it in the pragma itself, as #pragma omp parallel for:

```
#pragma omp parallel for
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

You can explicitly specify the number of threads to be created in the team, using the num_threads attribute:

```
#pragma omp parallel num_threads(3)
{
  /* This code will be executed by 3 threads. The loop is equally divided in as
     many chunks of consecutive iterations as the threads of the current team. */
  #pragma omp for
  for(int n=0; n<10; ++n)
     printf("%d\n", n);
}
```

If you prefer, the loop iterations can be assigned to threads on-demand (instead of statically as it happens above):

```
#pragma omp for schedule(dynamic)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

In the dynamic schedule, there is no predictable order in which the loop iterations are assigned to different threads. The OpenMP run-time support keeps track of the progress state of the loop while each thread asks for the next iteration to be computed. As soon as the computation of the current iteration ends, the thread asks for another one, and so on. Even if it is more flexible because it allows to better distribute loop-iterations with significantly different execution times among the

processor, the overhead for the dynamic scheduling is higher than the static one. In both approaches, however, it is possible to tune the chunk size assigned to a thread:

```
#pragma omp for schedule(dynamic, 3)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

In this example, each thread obtains 3 iterations of the loop at a time. Straightforwardly, the last chunk could be smaller than the specified chunk size.

Synchronization: barrier, atomic, critical, single

The barrier directive causes threads encountering the barrier to wait until all the other threads in the same team have encountered the barrier.

```
#pragma omp parallel
{
  SomeCode();
  #pragma omp barrier
  SomeMoreCode();
}
```

In the example, all threads execute SomeMoreCode(), but not before all threads have finished executing SomeCode(). Note: There is an implicit barrier at the end of each parallel block, and at the end of each for and single statement, unless the *nowait* clause is used.

Atomicity means that something is inseparable. In cases like the one below, the processor loads, recalculates, and writes the variable x. If another processor accesses x in the meanwhile, the result of the computation may be incorrect. By using the atomic directive, the three operations become inseparable so that the other threads cannot intervene during their execution.

```
#pragma omp atomic
x += value;
```

The atomic keyword in OpenMP specifies that the denoted action happens atomically. It is commonly used to update counters and other simple variables that are accessed by multiple threads simultaneously. Among the expressions you can use with atomic, there are:

```
x++; x--; ++x; --x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

where 'binop' can be one of these: +, *, -, /, &, ^, |, <<, or >>. If the statement to execute atomically doesn't fall in the previous cases because it's more "complex", you can use the critical construct which has higher overheads. The critical construct restricts the execution of the associated block to a single thread at time.

```
#pragma omp critical
{
  foo();
}
```

The single construct specifies that the related block is executed by only one thread in the team (it is unspecified which one) while the other threads skip the block and wait at an implicit barrier at the end of the construct.

```
#pragma omp parallel
{
  Work1();
  #pragma omp single
  {
    Work2();
  }
  Work3();
}
```

In a 2-cpu system, the example above will run Work1() twice, Work2() once and Work3() twice. There is an implied barrier at the end of the single construct, but not at the beginning of it. Even in this case the implicit barrier can be disabled with the clause *nowait*.

Tasks

By means of the task mechanism, you can implement algorithms that, otherwise, would be difficult to implement with the for-loop construct. For instance, if a certain operation needs to be applied to all elements of a linked list, you can have one thread go down the list, generating a task for each element of the list:

```
#include <stdio.h>

struct listnode {
   listnode(int data, listnode *next) : data(data), next(next) {}
  ~listnode() { delete next; }
   int data;
   listnode *next;
};

void print_unordered(listnode* head) {
   if(head) {
      #pragma omp task
      printf("%d\n", head->data);
      print_unordered(head->next);
   }
}

int main() {
   int n = 10;
   listnode *head = nullptr;
   for(int i=0; i<n; i++) head = new listnode(n-i, head); //1->2->3->...->10
   #pragma omp parallel //create the team: producer and consumers
   #pragma omp single //one producer (without this each thread call the function)
   print_unordered(head);
   delete head;
   return 0;
}
```

In general, new tasks are inserted into a pool from which all threads can take and execute them. Typically, one thread executes the code segment that creates a task which will be executed by another thread in the team later. This leads to the following idiom:

```
#pragma omp parallel

#pragma omp single

{

   ...

   #pragma omp task

   {
      ...
   }

   ...

}
```

Description of the code above:

1. A parallel region creates a team of threads;
2. One thread in the team then produces the tasks;
3. All the threads in that team (possibly including the one that generated the tasks) consume the tasks and when the pool is empty the threads join and parallel region ends

The *taskwait* construct specifies a barrier on the completion of child tasks of the current task. In the previous example, to print the list in a sorted way, each task should wait for the completion of the task previously created as done below here:

```
void print_ordered(listnode* head) {
   if(head) {
      #pragma omp task
      printf("%d\n", head->data);
      #pragma omp taskwait
      print_ordered(head->next);
   }
}
```