Exercise 4

hand out: 2020-12-09, hand in: 2021-01-06 14:00

Introduction: OpenMp

OpenMP execution model

OpenMP uses the fork-join model of parallel execution.

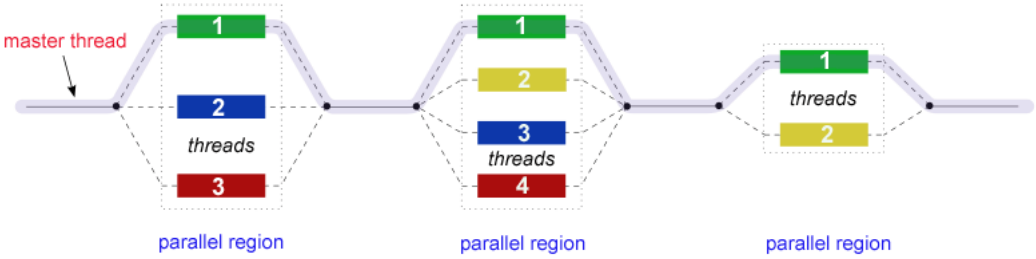


Figure 1: OpenMP `FORK-JOIN` execution model

All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first parallel region construct is encountered. The master thread then creates a team of parallel threads (fork operation). The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads. When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread (join operation). The number of parallel regions and the threads that comprise them are arbitrary.
More info can be found at `http://openmp.org/`.

OpenMP Environment Variables

OpenMP Environment Variables influence runtime behavior.

`OMP_NUM_THREADS`

Sets the maximum number of threads in the parallel region, unless overridden by `omp_set_num_threads` or `num_threads`.

```
OMP_SCHEDULE
```

Allows to specify schedule type and chunk size. The value of the variable shall have the form: `type[,chunk]` where type is one of `static` , `dynamic`, `guided` or `auto`. The optional chunk size shall be a positive integer. If undefined, dynamic scheduling and a chunk size of 1 is used.

OpenMP library API

Returns the number of processors that are available when the function is called.

```
int omp_get_num_procs();
```

Returns the number of threads in the parallel region.

```
int omp_get_num_threads();
```

Returns the thread number of the thread executing within its thread team.

```
int omp_get_thread_num();
```

Returns nonzero if called from within a parallel region.

```
int omp_in_parallel();
```

OpenMP `pragmas`

OpenMP consists of a set of compiler #pragmas that control how the program works. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.
All OpenMP directives in C and C++ are indicated with a `#pragma omp` followed by parameters, ending in a newline. The pragma usually applies only into the statement immediately following it, except for the barrier and flush commands, which do not have associated statements.

`parallel` pragma The parallel pragma starts a parallel block. It creates a team of N threads, all of which execute the next statement or the next block. After the statement, the threads join back into one.

`section` pragma The sections pragma indicates that statements can run in parallel. The following code states that tasks Work1, Work2+Work3 and Work4 can run in parallel, but that Work2 and Work3 must run in sequence.

```
int main() {
  #pragma omp sections
  {
    { Work1(); }
    #pragma omp section
    { Work2(); Work3(); }
    #pragma omp section
    { Work4(); }
  }
}
```

`for` pragma directive The `for` directive splits the for-loop so that each thread in the current team handles a different portion of the loop.
The code below will print numbers from 0...9. As they are printed from different threads, it might happen that they do not appear in order.

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

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);
  }
}
```

There is also the shorthand notation `#pragma omp parallel for` to perform both operations with a single pragma:

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

scheduling pragmas The scheduling algorithm of the for-loop can be explicitly controlled. `static` and `dynamic` are the most important scheduling policies.
`static` scheduling is used by default. Upon entering the loop, each thread independently decides which chunk of the loop it will process.
With `dynamic` scheduling, there is no predictable order in which the loop items are assigned to different threads. Each thread asks the OpenMP runtime for an iteration number, then processes it. Upon completion it asks for next, and so on. The chunk size can also be specified to lessen the number of calls to the runtime.

```
// set the dynamic schedule with chunk size 4
#pragma omp parallel for schedule(dynamic, 4)
for(int n=0; n<10; ++n)
{
  printf(" %d", n);
}
```

ordering pragmas The order in which the loop iterations are executed is unspecified, and depends on runtime conditions.
However, it is possible to force that certain events within the loop to happen in a predetermined order, using the `ordered` clause.

```
#pragma omp for ordered schedule(dynamic)
for(int n=0; n<100; ++n)
{
  data[n].process();

  #pragma omp ordered
  file.write(data[n]);
}
```

Data is processed in parallel, and possibly out of order, but the `ordered` directive ensures that the data is written in order to the file.
The must be exactly one ordered block per ordered loop. In addition, the enclosing for directive must contain the `ordered` keyword.

`atomic` pragma Atomicity means that something is inseparable; an event either happens completely or it does not happen at all, and another thread cannot intervene during the execution of the event.
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.
Unfortunately, the atomicity setting can only be specified to simple expressions that usually can be compiled into a single processor opcode, such as increments, decrements or xors.

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

`critical` pragma The `critical` directive restricts the execution of the associated statement or block to a single thread at a time.
The critical directive may optionally contain a global name that identifies the type of the critical directive. No two threads can execute a critical directive of the same name at the same time.

Data sharing

In the parallel section, it is possible to specify which variables are shared between threads and which are not. By default, all variables are shared except those declared within the parallel block.

```
int a, b=0;
#pragma omp parallel for private(a) shared(b)
for(a=0; a<50; ++a)
{
#pragma omp atomic
  b += a;
}
```

This example explicitly specifies that `a` is private (each thread has their own copy of it) and that `b` is shared (each thread accesses the same variable).

Execution synchronization

The `barrier` directive causes threads encountering a barrier to wait until all the other threads in the same team have also reached that barrier.

```
#pragma omp parallel
{
  /* All threads execute this. */
  SomeCode();

#pragma omp barrier

  /* All threads execute this, but not before
  * all threads have finished executing SomeCode().
  */
  SomeMoreCode();
}
```

Note: There is an implicit barrier at the end of each parallel block, and at the end of each section, for and single statement, unless the `nowait` directive is used.

```
#pragma omp parallel
{
#pragma omp for nowait
  for(int n=0; n<10; ++n) Work();

  /* This line may be reached while some threads are still
  * executing the for-loop. */
  SomeMoreCode();
}
```

The `single` directive specifies that the given statement/block is executed by only one thread. It is unspecified which thread. Other threads skip the statement/block and wait at an implicit barrier at the end of the construct.
The `master` directive is similar, except that the statement/block is run by the master thread, and there is no implied barrier; other threads skip the construct without waiting.

```
#pragma omp parallel
{
  Work1();

  // This...
#pragma omp master // use single to execute only once on any thread
  {
    Work2();
  }

  // ...is practically identical to this:
  if(omp_get_thread_num() == 0)
  {
    Work2();
  }

  Work3();
}
```

Task 1: Parallel Mandelbrot

30 Punkte

The Mandelbrot set calculation lends itself to parallel execution as all pixels can be calculated independent from each other. We can use OpenMP to calculate the Mandelbrot set with multiple concurrent threads.

Tasks:

a) (6 P.) Use OpenMP to parallelize the Mandelbrot set calculation provided.[1] Use the necessary pragmas and API functions to achieve a parallel execution of the code. Make sure that the number of threads can be specified by using the environment variable `OMP_NUM_THREADS`. Hint: You can specify the environment for just a single program run as follows:

```
OMP_NUM_THREADS=4 ./mandelbrot
```

b) (6 P.) Assign a different color to each thread, and represent all pixels calculated by the same thread in the same color. This can be done by altering one or more color channels depending on the OpenMP thread number. Create a Mandelbrot image with `static` scheduling and explain the resulting color variations.

c) (6 P.) Change the thread scheduling from `static` to `dynamic` and experiment with different blocksizes. Choose at least two interesting examples and explain the variations in the resulting images.

d) (6 P.) Running multiple threads allows us to occupy multiple CPU cores in parallel. In addition to realtime, the time using a processor (cpu time) becomes interesting. The template provides a stopwatch class to measure real and cpu time which you may use.

Measure and report the times spent by using a different number of threads, blocksizes and schedulings.[2] Discuss how the numbers behave if the number of OpenMP threads exceed the available real and logical cores in your system.[3]

e) (6 P.) Evaluate the difference in the image and execution time when you parallelize the first, second and both for-loops. Give a short explanation of the results.

[1] You will need to use the `-fopenmp` flag to tell the compiler to recognize the OpenMP pragmas and link the needed libraries. If you use OpenMP functions you need to include the omp.h header. The template already takes care of both.

[2] To get meaningful measurements the runtime should be at least a few seconds. If your system is too fast or slow you can adjust the pixel size of the image. To be able to compare your measurements against each other all of them must have the same pixel size.

[3] If you are running in a VM make sure you have assigned more than one core to the VM.

Task 2: Parallel Calculations

10 Punkte

Assume the runtime of an application for a problem is 100 s for problem size 1. It has an initialization phase which lasts for 10 s. This initialization phase can not be parallelized and stays constant for any problem size. Secondly, it has a problem solving phase which can be perfectly parallelized. However, this phase has a quadratic complexity with respect to the problem size.

a) (5 P.) What is the function of the speedup for the given application in dependence of the number of processors p and the problem size n?

b) (3 P.) What is the execution time and speedup of the application with problem size 1, if it is parallelized and run on 4 processors?

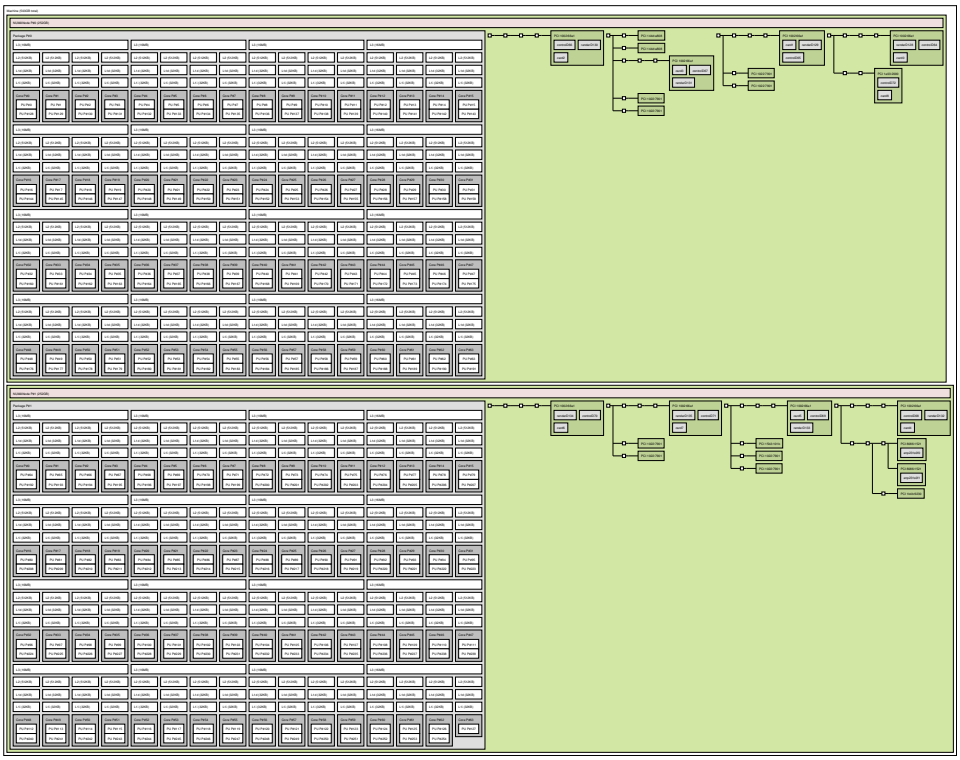
c) (2 P.) What are the execution times of the application if the problem size is increased to 4 and it is run on 4 processors and on 16 processors? What is the speedup of both configurations compared to a non-parallel version?

Task 3: Hardware Topology

5 Punkte

Understanding the hardware platform's topology can be important when optimizing applications for a particular system. This is not limited to details of the CPU architecture like core count and cache topology. Features of the base system, i.e., the topology of the attached periphery, can be similarly important. Knowing the locality of the network interface, for example, can help to improve communication latency by assigning the communication thread to the respective CPU.

There are multiple Linux tools and OS features to discover the local hardware architecture. One notable example is the `lstopo` command that comes with the Portable Hardware Locality (hwloc) software package. It allows displaying the hierarchical topology of modern architectures, including NUMA memory nodes, sockets, shared caches, cores and simultaneous multithreading. It also gathers various system attributes such as cache and memory information as well as the locality of I/O devices such as network interfaces, InfiniBand HCAs or GPUs.[4] Beside text-mode output `lstopo` can create graphical representations of the system. An example for a dual-socket server with two AMD EPYC 7742 64-core processor is depicted in the figure below.



a) (3 P.) Use `lstopo` to create a visual representation of the topology of your system and include it in your solution.[5]

b) (2 P.) Briefly explain the cache topology of your system. Include the associativity of each cache level. Tip: the text-mode output can report additional details in verbose mode.

[4] `https://www.open-mpi.org/projects/hwloc/`

[5] On most Linux systems, you get the `lstopo` command by installing the hwloc package. For Ubuntu, you can do this by running `sudo apt install hwloc`