Lab 2 - synchronisation and reduction in OpenMP

Files to be used in this lab:

- lab2.zip

1) Synchronisation

Last week, in the hello_world.cpp example we've seen that there can occur synchronisation issues between threads. OpenMP offers several mechanisms to combat such problems.

a) `#pragma omp single/master`

The `single` directive marks a block of code to be executed by a single thread from a team. It's not known beforehand which thread will be the one to handle the section nor is it guaranteed that it will be the first thread to reach it - the behaviour here is implementation dependent. An important aspect of this directive is the fact that there is an implicit barrier at the end of its section - all threads that don't execute the `single` section wait for the one thread that does.

Example:

```
#pragma omp parallel
{
  Job1();
  #pragma omp single
  {
    Job2();
  }
  Job3();
}
```

In the above example, all threads from the team execute Job1, then one of them is chosen and executes Job2 while all remaining threads wait for its conclusion. When that happens, all threads start executing Job3.

The `master` directive is similar to `single` - it also forces single-thread execution. However, in this case the thread in question is always the master thread. Another distinction from `single` is the lack of a barrier - other threads don't wait for the completion of the `master` section.

b) `#pragma omp critical`

This directive denotes a section that's executed by a single thread at a time. It guarantees that threads can't interfere with each other.

Example:

```
#pragma omp parallel for
for (int i = 0; i < 10; i++) {
  #pragma omp critical
  {
    std::cout << "Hello world" << std::endl;
  }
}
```

This example illustrates the characterics of the `critical` section - the output won't become garbled.

c) `#pragma omp barrier`

The `barrier` directive is stand-alone - it can't contain any code sections. Its role is just to synchronise. All threads in a team must stop at the barrier and wait for each other.

Example:

```
#pragma omp parallel
{
  Job1();

  #pragma omp barrier

  Job2();
}
```

Job2 can't be started until all threads finish Job1 and reach the barrier.

d) The `nowait` clause

Before we talk about the `nowait` clause, it's important to make note of implicit barriers in OpenMP. Aside from the previously mentioned `single` directive, such a barrier is also present at the end of each `parallel` block and also at the end of each `sections` and `for` directive.

This can be changed with the `nowait` clause. Applying it to `sections`, `for` or `single` removes the implicit barrier from the corresponding block.

Example:

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int i = 0; i < 10; i++) {
    SomeJob();
  }

  AdditionalJob();
}
```

In this example, the AdditionalJob may be reached before some threads finish executing the for loop - the `nowait` clause removes the implicit barrier.

e) `#pragma omp atomic`

The `atomic` directive marks single operations that are to be executed atomically, i.e. without the risk of conflict between threads. The exact behaviour can be controlled via `read`, `write`, `update` and `capture` clauses.

Examples:

```
#pragma omp atomic read
var1 = var2;

#pragma omp atomic write
var3 = var4

#pragma omp atomic update
var5 += var6;

#pragma omp atomic capture
var7 = var8++;
```

Let's go through these examples one by one. Read access to var2 is atomic, but nothing is guaranteed for var1. Write access to var3 is atomic, but nothing is guaranteed for var4. Next, var5 is updated atomically, but the state of var6 is uncertain. Finally, the `capture` clause combines `read` and `update` - var8 is first read and then updated atomically.

2) Reduction

Reduction is an operation that merges (reduces) a collection of values into a smaller number of values (usually one). One common example is summing - we take an array of numbers and sum them together into one value. It's very easy to come up with a parallel algorithm for such a reduction - each thread can sum a certain part of the array and then the master thread (or any other) can sum these partial sums. OpenMP offers a built-in mechanism to handle this operation - the `reduction` clause.

Example:

```
int sumNumbers(std::array inputArray) {
  int totalValue = 0;

  #pragma omp parallel for reduction(+:totalValue)
  for (int i = 0; i < inputArray.size(); i++) {
    totalValue += inputArray[i];
  }

  return totalValue;
}
```

The sumNumbers function above performs the sum reduction on an array of integers. The `reduction` clause takes two arguments - an operator and a list of variables to which that operator will be applied. Take note that summing is completely atomic. In fact, the sumNumbers function is equivalent to the following:

```
int sumNumbers(std::array inputArray) {
  int totalValue = 0;

  #pragma omp parallel
  {
    int tmpValue = 0;
    #pragma omp for nowait
    for (int i = 0; i < inputArray.size(); i++) {
      tmpValue += inputArray[i];
    }
    #pragma omp atomic
    totalValue += tmpValue;
  }

  return totalValue;
}
```

As you can see, for each thread there is a private variable tmpValue which stores a partial reduction result. The actual `reduction` clause works in much the same way, but the initial value of that private variable depends on the operator being used. The built-in operators are: +, -, |, ^, ||, *, &&, &, min, max. It's easy to figure out the initial values for each one (e.g. for sum its 0 and for multiplication it's 1). In the end, the partial reduction results get combined with the values of variables passed (on a `firstprivate` basis) to the `reduction` clause.

Reduction may also work with 2D arrays with the following syntax: `reduction(+:array[:arrayLength]).`

Since OpenMP 4.0, it's possible to define custom operators. For more information on that (and for the initial values of each operator), check the official documentation.

3) Exercise

Discrete Fourier Transform image compression (1 point)

`dft.cpp` file contains implementation of a simple image compression algorithm using Discrete Fourier Transform. We use the fact that usually lower frequencies contain more general informations. If we keep data for only some of those frequencies, we save a lot of space while still keeping the image essence.

Your task is to parallelize both compress and decompress functions, but mind the following:

1. Parallelization should be done inside the `dft.cpp` file only. Do not parallelize the utilities.
2. Prepare two versions of `compress` parallelization. Do your best in the first one. In the second one just use the most naive atomic operations.
3. During decompression we require that only the master thread (`threadId == 0`) is allowed to work on image data. Think of it as the image data is a secret and no other thread than master can see it. You may still use other threads to compute sines / cosines values.
4. You may test your solution on the provided `example.bmp` bitmap or any other, but try to use similar dimensions.
5. **Please note** that there is a problem with opening compressed images in some editors (like VSCode). If you see a blank image, please open the image using system browser.
6. Compile your solution using at least `-O2` optimisation.

Additionally, create a plot that shows the relation of speedup (calculated as sequential time / parallel time) to the `accuracy` parameter. Include three plots in a single image - both versions of the parallel `compress` and the `decompress` implementation. Test the following `accuracy` values: 8, 16, 32. Use different numbers of threads: 2, 4, 8, 16, 32, 64.

Analyse your findings. Is there a significant difference between parallel implementations? If yes - why? How does the number of threads impact the results?

Hints:

- It's quite challenging to parallelize master and other threads during decompression phase. We suggest the following - in a single loop iteration, while master thread is working on some image data, other threads may pre-calculate sin, cos and theta values for the next iteration.

4) Nested Parallelism

`parallel` regions can be nested - if thread executing a parallel region encounters `parallel` block, a new team of parallel worker threads is created. Depending on used environment it may be required to enable nested parallelism, e.g. by setting environment `OMP_NESTED`:

```
export OMP_NESTED=TRUE
```

5) Sources

Your main source for learning OpenMP should be the official documentation

Ostatnia modyfikacja: piątek, 7 marca 2025, 12:22