TDT4200: Parallel Computing

Parallel Computing - Assignment 2

September 26, 2018

Bart van Blokland
Björn Gottschall
Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)

- **Delivery deadline: October 5th, 2018 by 23:00.**
- **This assignment counts towards 5% of your final grade.**
- You can work on your own or in groups of two people.
- Deliver your solution on *Blackboard* before the deadline.
- Upload your report as a single PDF file.
- Upload your code and results as a single ZIP file solely containing the source files (src directory). Do not include binaries or additional resources provided with this assignment, such as mesh object files. Not following this format may result in a score deduction.
- All tasks must be completed using C++.
- Use only functions present up to and including C++11.
- Do not include any additional libraries apart from standard libraries or those provided.
- The delivered code is taken into account with the evaluation. Ensure your code is documented and as readable as possible.

Questions which should be answered in the report have been marked with a [**report**] tag.

Objective: Use the Message Passing Interface (MPI) to distribute work over multiple CPU nodes in different ways to speed up executions.

Parallel Computing - MPI

The Message Passing Interface is a library which simplifies communication between processes. The main reason why this is useful because these processes do not necessarily need to run on one machine only (although they definitely can). On large computing clusters, which typically consist of large numbers of separate computers connected by a networking interface, MPI is the primary means by which communication between machines is facilitated.

MPI is an example of SPMD: Single Program Multiple Data. The main idea is that data can be processed in parallel by executing *exact copies* of the same program on many different machines, each processing different data.

The ease at which work can be divided over a large number of machines or processors make it a powerful tool for distributing work to speed up computations. Better yet, you don't need a large computing cluster to run MPI programs; you can even create a number of instances of your program on your own machine, and have them divide their work over your CPU's cores.

At the very basic level, MPI consists of a few basic functions to facilitate sending a message from one instance to another (primarily `MPI_Send()` and `MPI_Recv()`). However, based upon these functions, the library offers many convenience functions intended for more complex communication and cooperation behaviour (e.g. broadcast, gathering, and reductions), as well as synchronisation features like barriers. These features tend to be implemented using only the basic send/receive operations.

For this assignment you are provided with the already known rasterisation algorithm from the first assignment, which has now been optimised to run fast. It also has been extended to support rendering of objects with basic materials (yay colour!).

The rasterisation process alone is very fast and wouldn't demonstrate a good use for parallelization using MPI. To make sure your machine's processor actually has a decent amount of work to do, the algorithm now renders the input mesh in a 3D Sierpinski carpet scheme[1]. We made sure the recursion depth is variable, so you can taylor it to the abilities of your machine.

Unfortunately, this algorithm by default still runs on only a single CPU core. In this assignment, you will distribute the work evenly across all CPU cores using MPI.

Although all operating systems can be used for solving this assignment, Linux is highly recommended and best supported. Keep in mind that you are not allowed to use any additional libraries apart from the C++ standard library or those provided. You are allowed to create additional source files, but your main focus should be to optimise the

[1] `https://en.wikipedia.org/wiki/Sierpinski_carpet`

existing ones. Please leave the original command line parameters intact (-i, -o, -w, -h, and -d).

This assignment will contribute with 5% to your final grade.

Remember that at the very basic level, we're looking for whether you've understood the concepts and methods this assignment touches upon. Make sure you show this when answering a question.

If you have any further questions please ask them first on the blackboard discussion board, as it is likely others have them too.

Task 0: Preparation [0 points]

a) Ensure the following tools are installed:

- CMake or make
- Git
- Git LFS (optional)(`https://git-lfs.github.com/`)
- A C++ compiler, such as G++ or MSVC++.
- mpirun, mpic++ [2]

These have already been installed for you on the lab machines.

b) Clone the assignment repository using the command:

```
git clone https://github.com/bgottschall/TDT4200-Assignment-2.git
```

If you have not cloned the repository with git-lfs (large file support), you have to unzip the *inputs.zip* inside the *inputs* folder.

c) Test your mpi setup for correct functionality by compiling and executing the included mpi test application:

```
mpic++ mpitest/main.cpp -o mpitest/mpitest
mpirun -np 4 mpitest/mpitest
```

Note that `mpic++` is the same as g++ (which we have used to compile C++ code before), with the addition of setting up everything you need for compiling a program wihch makes use of the MPI library.

d) Compile the project. You are provided with three ways for getting started. The included Makefile, which takes care of the compilation, run arguments, CMake for generating automatically a *new* Makefile solely compiling the project or the manual way doing everything on your own. You are free to choose, adapt and extend any of these.

- Make

  ```
  make all
  ```

- CMake

  Choosing this method will override the shipped Makefile. You will not be able to use make features mentioned in the following parts.

[2] On Ubuntu these are contained in the "openmpi-bin" and "openmpi-common" packages. You can install them using the "sudo apt install openmpi-bin openmpi-common" command.

```
cd build
cmake ..
make
```

- Manually

```
mpic++ -std=c++11 -Isrc -c src/rasteriser.cpp -o src/rasteriser.o
mpic++ -std=c++11 -Isrc -c src/main.cpp -o src/main.o
mpic++ -std=c++11 -Isrc -c src/utilities/geometry.cpp \
    -o src/utilities/geometry.o
mpic++ -std=c++11 -Isrc -c src/utilities/lodepng.cpp \
    -o src/utilities/lodepng.o
mpic++ -std=c++11 -Isrc -c src/utilities/OBJLoader.cpp \
    -o src/utilities/OBJLoader.o
mpic++ src/rasteriser.o src/main.o src/utilities/geometry.o \
    src/utilities/lodepng.o src/utilities/OBJLoader.o
    -o cpurender/cpurender
```

e) Run the project. The provided Makefile should give you a comfortable way to execute the compiled application. However, you should also be able to execute it manually for some tasks!

- Make

  The following lines should give you enough ideas on how to use the provided Makefile. For all available options have a look in the make help page (type in 'make help') or review the Makefile itself.

```
make call
(example) make call OUTPUT=output/cubes.png CPUS=1
(example) make time OUTPUT=output/cubes.png CPUS=1
```

- Manually

  The generated binary supports 5 different parameters for defining input, output, width and height in the rendering process. The depth parameter scales the sierpinski expansion to increase the rendering time.

```
cpurender/cpurender -i input/cubes.obj -o output/cubes.png \
    -w 1920 -h 1080 -d 3
cpurender/cpurender -i input/spheres.obj -o output/spheres.png \
    -w 480 -h 270 -d 1
```

f) Familiarize yourself with this project.

You are given new mesh objects for rendering: the sphere and cubes are colorful simple objects to start experimenting with. The rendering time of those can extremely scaled up by the depth parameter (-d) using the Sierpinski expansion. The left three objects are a Audi-R8, a Camaro and a pot plant, all consisting of much more mesh objects and vertices. Try to render some of those objects and use the Sierpinski expansion with the depth option. Get a feeling on how much the runtime scales with the options.

Task 1: Getting started with MPI [2 points]

In this task we will use MPI to give every processor an isolated rendering job, speeding up parallel rendering of independent pictures.

a) **[0.7 points][report]** Parallel image rasterisation

Modify the rasteriser program such that it uses MPI to render multiple images in parallel. Each MPI instance should render a single image by itself. Each image should be different from the others, for instance by modifying the rotation angles[3] and/or lighting atmospheres.

Each instance of the program should solely use its MPI rank to determine how it modifies the image it is going to render. Also make sure that your code can be executed on different numbers of CPU cores.

Next, run your program with different numbers of processes (using the `mpirun -np <process count>` command). Show how the number of processes affects the number of images produced within the same time frame. What happens when you run your program with (far) more processes than you have cores in your CPU?

How good does the MPI execution scale?

Which speedup do you achieve compared to rendering the same number of images consecutively in a single process?

Show the changes you made in the source code for this task in your report (for instance using screenshots).

HINT: Note that calling `MPI_Init()` and `MPI_Finalize()` is mandatory at the start and end of the program, respectively, when using MPI.

HINT: If it's difficult to tell how long your program needs to execute, you can pass in a different depth parameter into the program from the command line. This will recursively render the object in the 3D Sierpinski carpet, thereby making execution time take longer.

b) **[0.6 points][report]**

Modify your program such that all ranks/processes ask a single rank how they should draw their images (for instance at which angle to draw the object). Use `MPI_Send()` and `MPI_Recv()` to communicate between processes.

Just like the previous task, you can decide yourself how each image varies, as long as they are unique to each instance. The "master" node handing out tasks should also draw an image itself.

Show in your report how you implemented the MPI communication "protocol" (you can for example show screenshots of your code).

[3] There's a parameter in the runVertexShader() which allows you to specify the rotation of the object.

c) **[0.7 points] [report]**

If you were to execute your MPI program on a computing cluster, you might be using a double or even triple digit number of machines at once. This, however, can lead to problems from unexpected directions.

For instance, let's say you are starting an MPI program on 100 machines. Also, at the start of your program you're reading a 10MB file. This causes 100 machines to request that file, and requiring a poor single server to transfer 1GB worth of data.

In this task, we'll implement a much more effective method. A single instance of the program loads the file, then broadcasts the loaded 3D Mesh object using the `MPI_Bcast()` function.

Mostly.

If you look in the utilities/geometry.hpp file, you might notice that due to the addition of materials (for making the 3D objects look pretty), there are a few additional fields in the Mesh struct. To simplify this task a bit, we'll therefore *only* be transferring the vertices, normals, and texture coordinates.

We'll cheat a little bit here and have each rank load the mesh file such that the other values are still the same (you're welcome to have a try at transferring them too, though!).

To transfer the vertices, normals, and texture arrays, do the following:

i) Each rank loads the mesh file

ii) All ranks throw out the contents of the vertices, normals, and texture coordinate vectors (apart from the "source" rank, of course).

iii) Use `MPI_Type_create_struct()` to create an MPI data structure for the `float3` and `float4` data types.

iv) Use `MPI_Bcast()` to broadcast the contents of the three arrays to all other ranks.

As with the previous tasks, show how you implemented this task in your report (again, screenshots tend to work best, but make sure they are readable!).

Task 2: Collective MPI computation [3 point]

Problems such as the one we saw in Task 1 (rendering a number of separate images) are a class that are known as *embarrassingly parallel*; parallelising them requires little to no effort [4]. In our case, the problem consisted of rendering completely separate images, which can be divided over any number of processors running in parallel.

We'll now look at a more complicated problem: using multiple MPI ranks/instances to cooperatively render a *single* image.

For this task, you may apply any optimisations desirable, *as long as the output image is unchanged*, and you're not obviously using that freedom to work around requirements of the individual tasks.

Also, you should start with a fresh copy of the *rasteriser.cpp* file for this task. The optimised version you end up with at the end should also be the one you submit in your source code archive.

a) **[1.5 point]** **[report]** Collective Construction

The `renderMeshFractal()` function is the primary piece of code taking up execution time.

Spread the execution of this function out over your MPI ranks.

It's important here to make sure that the workload is balanced well between nodes. Therefore, for a given number of ranks, the number of rendered objects each rank draws should be approximately equal (which is proportional to the number of times the `renderMeshFractal()` function is invoked, *also counting its recursive calls*).

This work division may not be trivial. My recommendation is to convert the recursive `renderMeshFractal()` function to an iterative one first, although you are free to choose your preferred approach.

As a result of this workload division, all ranks should produce a partial image of the scene. Force each rank to write to a separate image file, run the program with 4 ranks, and show the produced images in your report.

If you looked through the code in some level of detail, you may have noticed that the rendering process uses a "framebuffer" and "depth buffer".

The frame buffer is essentially the output image. It contains the colour of individual pixels, stored as an array of consecutive colours, with one byte representing each colour channel

[4] Little to no effort in the sense that you don't need to do extensive restructing of your algorithm to allow (parts of) it to be executed in parallel. Your efforts here are of course anything *but* little to no effort ;)

(red, green, blue, alpha (transparency)). The depth buffer is used to figure out which objects appear before other ones during rendering. If a pixel being rendered has a lower depth value than another, it appears in front of another from the viewpoint of the camera.

Since you have now split up the rendering into multiple separate processes, each rank has its own version of the depth and frame buffer. We'll therefore as a final step have to merge all images back together. That is, when we have used n ranks, we have n frame buffers and n depth buffers.

The basic process here is that for each pixel in these images, we need to figure out which of the ranks has the pixel that's closest to the camera. Next, we set the pixel in the final output image to the colour of the closest rank's frame buffer.

If you look through the handout source code, you can already find how to perform one of these "depth tests" on lines which I have copied below:

```
float pixelDepth = face.getDepth(u,v,w);
if( pixelDepth >= -1 &&
    pixelDepth <= 1 &&
    pixelDepth < depthBuffer.at(y * width + x)) {
```

Now we could have all ranks send their depth and frame buffer back to the "master" rank, which subsequently runs through all images and computes the final output image. However, such a solution causes one rank to do a large chunk of computations, while all the others are mostly waiting around.

Fortunately, we can do much better by (ab)using reductions.

b) **[1.0 point]** Compute the correct output image by performing the following reductions using the `MPI_Reduce()` and `MPI_Allreduce()` functions in the listed order:

1. Compute the depth buffer of the final image by applying the MIN operation on each value in the depth buffer. That is, if your image has a size of w x h, your depth buffer is $w \cdot h$ long, and you should apply the MIN operation $w \cdot h$ times to end up with an output buffer of size $w \cdot h$.
2. Each rank now creates an empty frame buffer (with the same dimensions and thus number of pixels as the output image). Make sure to initialise all values in this new buffer to 0. Hint: you can copy and paste the frame buffer initialisation code already present for this purpose.
3. Each rank loops through the previously computed final depth buffer, and compares the depth value in each pixel to the depth value in its own depth buffer.

If they match, the rank sets the colour in the newly created frame buffer to the one in its own frame buffer. Note that we assume no two ranks have the same depth here, so you may end up with some pixels with weird bright colours in some cases.

4. Use a binary OR reduction to compute the final output image. Since we assumed that for each pixel there was only one rank with the closest depth value, all ranks combined write to each pixel once. A binary OR will therefore combine all frame buffers together.
5. The "master" rank writes the combined image to the output file.

Make sure to synchronise the ranks to avoid race conditions. Using an `MPI_Barrier()` can help.

c) **[0.5 point]** **[report]** Measure the execution time of your current implementation. How does it scale with varying numbers of MPI ranks? What happens to the execution time when you launch more ranks than you have cores in your CPU?

Compare your measured runtime to the average execution time needed for computing a single image (which you measured in task 1a)). In both cases, use a number of MPI processors equal to the number of cores in your CPU.

What is the speedup of the cooperative construction of images over a single-threaded approach and briefly speculate on what could cause that approach to be faster than the other.