TDT4200 PS 5 - PThread and OpenMP

A. C. Elster, B. Gotschall and Z. Svela

Due Monday, Oct 18th, 2019

Introduction

This Problem Set is mandatory and evaluated as Pass/Fail. I.e. a passing grade is required for taking the final exam.

This assignment has two parts. Part 1 involves task parallelization using **pthreads** whereas Part 2 covers parallelizing matrix-matrix using **OpenMP** and a BLAS library call.

NB!: Carefully read the section "Deliverables" on how to submit your results!

Part I: Task parallelization with pthreads

In this task, you will use a consumer/producer scheme in which a dynamic job queue is processed by multiple worker threads. Therefore, you are provided with a Mandelbrot set renderer which uses a subdivision algorithm.

Mariani-Silver and Escape Time

The renderer provided supports two different rendering algorithms, the Mariani-Silver and the Escape-Time algorithm – see Figure 1.

You can switch between the algorithm by using the parameter -t or the key e in mandelNav. Since you already know the Escape-Time algorithm which just computes the dwell value for each pixel, we focus now on the Mariani-Silver algorithm. This subdivision algorithm is a recursive algorithm which takes the input dimensions and checks if they are under a threshold, which we call the block dimension. If so, it will compute the current block using the Escape-Time algorithm, else it will divide the block by a subdivision factor into x smaller blocks and repeats for each block this process.

The Mariani-Silver algorithm itself is a huge optimization in rendering the Mandelbrot set. It only works through the fact that the Mandelbrot set is a connected set, means complex numbers of the set are connected. Derived from this property we can define a rule: Any shape in the rendering area having the same dwell value on all border pixels, can be filled using this dwell value. This means before a block is checked for its threshold it is evaluated for the common border and only computed or subdivided if none is found.

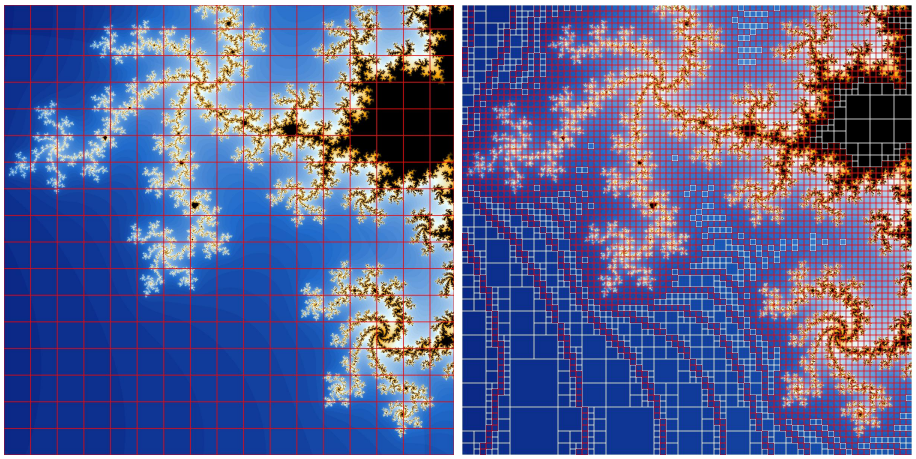


Figure 1: Escape Time Algorithm rendered in blocks (left) and the Mariani-Silver subdivision algorithm (right)

About the code

The code handed out already implements the Mariani-Silver algorithm and the Escape-Time algorithm. It is not required to change any code related to the computation or division of the rendering area. The following files are provided:

- **mandelCompute.h** includes he computational functions, while
- **mandelColours.h** contains the colour functions
- The **main** file holds only one function for the Escape-Time algorithm and one for the Mariani-Silver algorithm.

The rest is already prepared code for you to implement a consumer/producer threaded renderer.

You are given a job struct implementation which holds all the information of one task (which function and parameters). The job queue is a linked list which can be populated with jobs using **putJob**. The function **popJob** can be used to take a job from that list (only if the list is not empty).

Task 1: Baseline measurement

a) **Report:** Measure the runtime of your application by using `time` to execute it or simply running `make time`. Find a set of reasonable parameters (iterations, resolution, x, y, colours) to take the baseline. Always compare in future tasks to this baseline using the same parameters.

b) **Report:** Render different areas of the Mandelbrot set using the Mariani-Silver algorithm and measure the runtime. Alter the block dimension and subdivision parameters as well as the resolution. Think about what has an impact on the rendering time (use the mark border feature!). Render 2 different areas of the Mandelbrot set using the Marini-Silver algorithm with marked borders and use the same values for all other parameters (resolution, iterations, subdivision and block dimension). Include the pictures in your report together with their rendering times. Give a **short** reasoning why the times may or may not differ.

Task 2: PThread Consumer/Producer

Pthread data structures and functions that might help you:

- pthread_t (and associated functions)
- pthread_create
- pthread_join
- pthread_mutex_t (and associated functions)
- pthread_cond_t (and assiciated functions)

a) Implement a consumer/producer scheme using pthreads so that every block in the Mariani-Silver Algorithm and Escape-Time algorithm is handled by one thread. Every thread is supposed to pick up an available job, process it and wait for new jobs to arrive. Remember that a job itself can create new jobs e.g. in the Mariani-Silver algorithm when subdividing the current block. After a new job is created waiting threads must be informed that new jobs are available. Make sure that inserting and picking up jobs is done only by one thread at a time.

b) **Report:** Measure the runtime of the Mariani-Silver Algorithm and the Escape-Time Algorithm using multiple threads.

How big is the speedup compared to your baseline?

How does the speedup scale with the number of threads?

Part II: Matrix-matrix multiplication

In the following tasks, you will time three different versions of a matrix-matrix multiplication:

1. Provided serial version,
2. A version parallelized using OpenMP
3. A version implemented by the highly optimized library BLAS (Basic Linear Algebra Subroutines).

For those of you who have not done matrix-matrix multiplication in a while, it is a operation between a mxk matrix A and a kxn matrix b that produces a mxn matrix C, where each cell $C[i, j]$ is the dot product/inner product between the i'th row of A and the j'th column of B. In all of the provided code, this is the meaning of the variables A, B, C, m, n and k. See the Figure 2 for a graphical representation of matrix-matrix multiplication. Matrix vector multiplication is covered in the Pacheco text. You can also check out "Multiplying matrices" at: https://www.khanacademy.org/

The provided code contains a Makefile that will compile the program with the necessary arguments to use both BLAS and OpenMP. The program will run fine on the lab computers, but rely on that BLAS and OpenMP are installed and linked correctly in your programming environment, so the program will not necessarily run on your own computer, if they are not installed the same way.

You can run any of the three versions of the program using the following:

- `make serial`,
- `make omp` or
- `make blas`.

About BLAS

BLAS is a library of functions for linear algebra operations. Or rather, BLAS is a specification, and there are several implementations, including OpenBLAS and a version that is baked into Intel's Math Kernel Library (a.k.a. MKL). OpenBLAS is installed on all the lab computers and is the one we will be using. We will also be using the part of BLAS called CBLAS, which contains functions that allow for matrices to be row-major, rather column-major as the rest of BLAS assumes.

BLAS functions look quite cryptic at first glance, so be sure to check our the documentation when using them. Documentation can be found on the netlib website (link), and a presentation from TDT4200 in 2017 can be found on blackboard.

Task 3: Baseline measurement

Using code instrumentation, time the serial matrix-matrix multiplication function called `serial_mxm()` in the provided code.

Report: Where did you insert timing functions in the code and why? Provide a short justification. Hint: What do we/don't we want to measure?

Task 4: OpenMP parallelization

Implement the function called `parallel_mxm()` using OpenMP. You can copy code from the serial version if you want. In order to test your new function for correctness, run the program with `make p_test` or just run it with more than one argument (e.g. `./main p t`.

Time the function and measure speedup. Use `omp_get_num_threads()` to check how many threads are running.

Report: What kind of speedup did you measure? How does this compare to the number of threads running? Give a short explanation of your results.

Task 5: BLAS

Implement the function called `blas_mxm()`. The function you will need to use is called `cblas_dgemm()` (CBLAS Double precision General Matrix Matrix multiplication). The documentation for `dgemm()` can be found on Netlib's web sites (link). Calling the cblas-version: `cblas_dgemm(CblasRowMajor, /*arguments for normal dgemm()*/)`, that is, the only difference is the leading "cblas_" and an indication of whether the matrices are row- or column-major. An example of use can be found at Intel's web sites (link).

Report: What kind of speedup do you find compared to the serial version? What kind of speedup or slowdown do you find compared to your parallel version? Give a short explanation of these results. Hint: Compare the number of operations done in a matrix-matrix multiplication with the amount of data used.

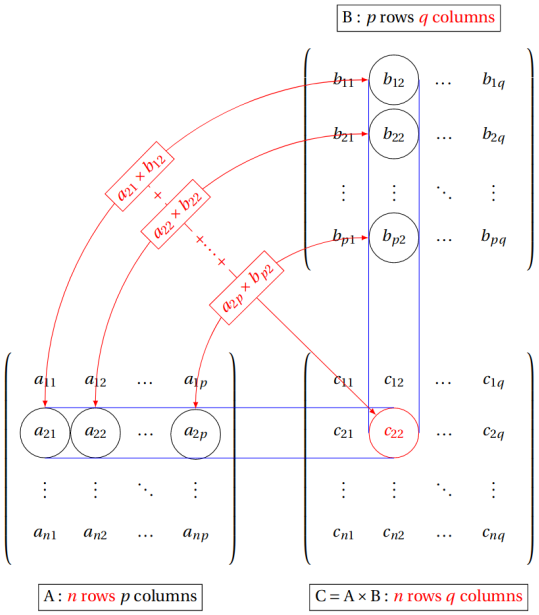
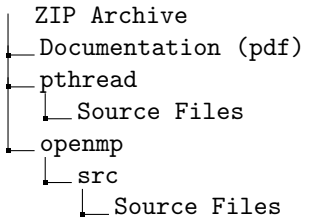


Figure 2: Graphical representation of matrix-matrix multiplication. Keep in mind that this figure uses different letters to indicate the dimensions than in the provided code.

Deliverables

The documentation must be delivered as a PDF file together with the source code packed in a ZIP file and submitted in time on blackboard. The code must be documented and in a runnable state. Do not use any libraries apart from standard libraries or those provided. Do not change the calling conventions (parameters) of the application.

NB!: Deliver your code in the folder structure that is provided to you. Include at least every file that you have changed. Your application must stay compatible with the provided Makefiles or CMake files. **Do not include any object, input or output files**. The folder structure for this assignment is the following:



The following tasks require documentation (if solved):

- 1a, 1b, 2b, 3, 4, 5