



F. Wermelinger  
Office: Pierce Hall 211

## Homework 2

OpenMP, Operational Intensity, Roofline model, Introduction to MPI

<b>Issued:</b> February 14, 2023 <b>Due:</b> March 7, 2023 11:59pm
---

Note this is a 3 week exercise. *Do not procrastinate the work.* It is suggested you start with problem 1. Material for problems 2, 3 and 4 will be covered in the following week.

Please keep in mind that when the cluster is under high load with many queued jobs, your submitted job(s) will not execute right away. Depending on the cluster load, your job may execute only in a couple of hours (resources are limited). *Make sure you plan ahead.*

**Submission instructions:** please see the homework submission section in the syllabus.<sup>a</sup>

<sup>a</sup><https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#homework-submission>

### Problem 1: Julia Set with OpenMP (40 points)

Romeo recently heard about the Julia set. Fascinated by its beauty, he decided to impress his beloved Juliet by painting the set on the wall of a large house across the street of her balcony. In order to do so he has to know what color goes where and how much paint of each color he will need. The job is not trivial, as he wants a very high resolution painting.

Romeo had some training in the arts of programming, but he managed only to write a slow serial code for computing the colors. Help him by

1. parallelizing the code
2. computing the required amount of each color

For the purpose of this exercise, we define the Julia set as the set of complex numbers  $w$  for which the numbers  $z_n$ , defined as

$$z_0 = w, \quad z_{n+1} = z_n^2 + c, \tag{1}$$

do not diverge for  $n \rightarrow \infty$ . Here  $z_0$  is an initial condition and  $c$  is some constant complex number. It can be shown that the divergence occurs if  $|z_n|$  becomes larger than 2 for some  $n$  (if  $|c| < 2$ , which is true in our case). In our visualization in figure 1 each pixel represents

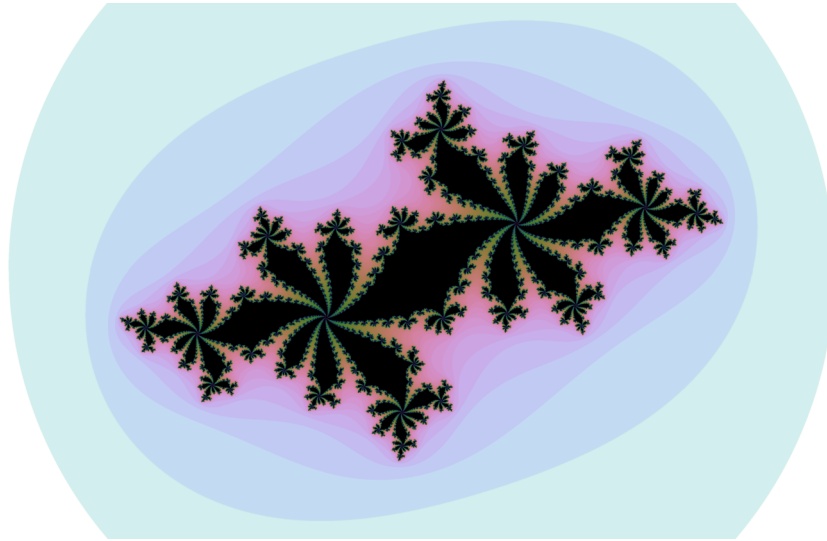


FIGURE 1: Visualization of the Julia set for  $c = -0.624 + 0.435i$ .

one value of  $w$ , where the  $x$  coordinate determines the real component and  $y$  the imaginary component of  $w$ . The pixel color is determined by the minimum  $n$  for which  $|z_n| > 2$ . For some pixels such  $n$  either does not exist or is too large, so we stop iterating after  $n = 1000$ . You are given a skeleton C++ code that computes the iteration count for each pixel (you must complete it, see tasks below) and a plotting script for the visualization.

**Work in directory:** `./code/p1`

**a) 15 points**

Implement the iteration rule in equation 1 and the termination criterion and parallelize the function `julia_set` in the file `julia_set.cpp` using OpenMP. You are given a Makefile for convenience. Use the following sequence of commands

---

```
$ make && make run && make plot
```

---

to compile the code, run it with a single thread and plot the set.

Report the execution time  $T_p$  for  $p = 1, 2, 4, 8, 16, 32$  threads in tabular form (see the run target in the Makefile to see how this is done for  $p = 1$  thread). You must collect this data on a *single* compute node on the academic cluster using a proper SLURM job script (see task 3 in lab 1 for an example). Name this script `p1_job.sh` and add it to the working directory for this problem (you must create this file from scratch, make sure you include it in your submission). Set the SLURM parameter for your job according to what you need for this task and set your OpenMP runtime environment in the script if needed. You can run all the different experiments for  $p$  in the same job and append the results to a file that you can then use for the data post-processing in your write up. You can reuse this job script for the tasks that follow below when you are asked to repeat the experiment (you may adjust it if needed). It is important that you collect your data for  $T_p$  on a full node that is allocated explicitly for you to avoid noise in your measurements due to other users active on the same node.

You can develop your code either on your laptop or checkout an interactive node (or VSCode instance) on the cluster using a *fair* amount of cores if you request the resource for a longer time. See the handouts for lab 1 for reference.

Finally, use your collected data  $T_p$  to create a plot with  $p$  on the abscissa and the ratio  $S_p = T_1/T_p$  on the ordinate. Be sure to label the axes and include this plot in your report. Briefly explain what you are plotting. What would you expect to see in this plot if your code was *perfectly parallelizable*?

**Hint:** Complete the TODO A: labels in the code.

**Hint:** In the Makefile make sure you compile with OpenMP support!

**Hint:** Note that due to the symmetry, the code computes only the bottom half of the image. Furthermore, the plotting script will likely run longer than the C++ code.

b) **8 points**

By definition, the number of iterations may differ from pixel to pixel, which causes load imbalance. Improve your parallelization to fix the load imbalance issue using only OpenMP. Describe your approach.

Repeat the benchmark from the previous task and collect new measurement data for  $T_p$  with your balanced implementation. Tabulate this data in another table and plot the speedup of these measurements in the same plot of the previous task such that you can compare them. Do you observe an improvement?

c) **2 points**

Shakespeare told you it is good practice to avoid accessing the same cache lines by different threads. Propose a simple way to incorporate that into your code.

d) **15 points**

Implement the function `compute_histogram` that will help the prince determine how much paint of each color he needs to buy. Minimize the usage of critical regions, locks and atomics. See the comments in the skeleton code for details. Plot the computed histogram in your report (you can use the provided `plot.py` script).

**Hint:** Complete the TODO D: labels in the code.

## Problem 2: Operational Intensity (20 points)

The purpose of a compute architecture is to perform a certain operation on data. This requires a mechanism to send data along the memory hierarchy<sup>1</sup> to the execution units such that they can perform the operation(s).

The operational intensity is a measure that relates the amount of work  $W$  (operations) to the number of bytes  $Q$  (data) that need to be transferred and is defined as

$$I = \frac{W}{Q} \quad [\text{ops/byte}]. \quad (2)$$

The operational intensity  $I$  is used to characterize a code and reveals information about the expected utilization of the architecture's execution units and memory bandwidth. In practice, operations are defined by floating point operations (FLOP) and memory transfer is defined by access from DRAM to the closest processor cache.

*Hint: Remember that double precision numbers have a size of 8 bytes.*

### a) 12 points

Find an analytic expression for the operational intensity and determine the asymptotic bounds on the operational intensity  $I(n)$  for the following matrix/vector operations, where  $n$  is the dimension of the vector. Start with a cold cache and state any further assumptions.

*Hint: The leading letter in these operations indicate the precision of the operation. "D" stands for double and "S" for single.*

1. DAXPY:  $y = \alpha x + y \quad \alpha \in \mathbb{R}; x, y \in \mathbb{R}^n$
2. SGEMV:  $y = Ax + y \quad x, y \in \mathbb{R}^n; A \in \mathbb{R}^{n \times n}$
3. DGEMM:  $C = AB + C \quad A, B, C \in \mathbb{R}^{n \times n}$

*Hint: Assume that  $A$ ,  $B$  and  $C$  all fit into the cache together*

### b) 8 points

The discrete Fourier transform (DFT) of the real signal  $x \in \mathbb{R}^N$  is given by

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j \omega_N^{-jk}, \quad (3)$$

where  $\omega_N = e^{\frac{i2\pi}{N}}$  is the  $N$ -th root of unity. Assume that  $k = 0, 1, \dots, N-1$  and that you have *pre-computed* the constant factors  $\omega_N^{-jk}$ . Note that the factors  $\omega_N^{-jk} \in \mathbb{C}$  and the transform  $X_k \in \mathbb{C}$  is **complex**. Complex numbers are stored in memory with a real part and imaginary part, both are *individual* floating point numbers.

What is the operational intensity of the DFT for computations in double precision?

*Hint: State any further assumptions you make (if you are stuck).*

*Hint: Assume you evaluate the DFT naively without decimation in time or frequency and without exploiting symmetry and periodicity in  $\omega_N$ .*

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Memory\\_hierarchy](https://en.wikipedia.org/wiki/Memory_hierarchy)

### Problem 3: The Roofline Model (20 points)

This problem is about the roofline model discussed in lecture 9. Please write up your answers.

a) **6 points**

Describe in at most five sentences the basic concept of the roofline model. What information does it convey? How does it convey it? Which parts of hardware does it relate? More questions like this can be formalized.

*Hint: We are looking for keywords here that you must formalize in your own words.*

b) **14 points**

We are given two processors *A* and *B*:

Processor	<i>A</i>	<i>B</i>
Frequency	2.5 GHz	2.0 GHz
Cores	8	32
Max. Memory Bandwidth	30 GB/s	80 GB/s
SIMD Register Width	128 bit	512 bit
FMA Instructions	No	Yes

Processor *B* is *superscalar* and supports the simultaneous execution of *two* FMA instructions (fused-multiply-add) in the same clock cycle. Processor *A* is *scalar* and does not support FMA instructions. For processors *A* and *B*, report the following:

i) **6 points**

The peak floating point performance  $\pi$  for single precision data (32-bit float numbers).

ii) **2 points**

The ridge point  $I_\beta$  given the processor peak floating point performance  $\pi$  and memory bandwidth  $\beta$ .

iii) **6 points**

Draw the roofline ceilings for both processors *A* and *B* in a log-log plot. Do not forget to label the axes. Include the plot in your report.

## Problem 4: MPI Bug Hunting (20 points)

The following MPI/OpenMP codes are buggy. Identify the cause of the bug and propose a solution.

### a) 5 points

Identify and explain the bug in the following MPI code. Propose a solution to fix the bug. Assume the code is launched with  $N > 1$  processes.

---

```
#include <iostream>
#include <mpi.h>

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int msg;
    if (0 == rank) {
        msg = 33;
        MPI_Bcast(&msg, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } else {
        MPI_Status status;
        MPI_Recv(
            &msg, 1, MPI_INT, MPI_ANY_SOURCE, 123, MPI_COMM_WORLD,
            &status);
        std::cout << "Rank " << rank << " received " << msg
            << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

---

### b) 5 points

Consider the following MPI code that implements a ring communication pattern. Assume the code is launched with  $N > 1$  processes. Identify and explain the error in the this code. How could you fix this problem?

---

```
#include <iostream>
#include <mpi.h>

int main(int argc, char *argv[])
{
    int rank, size; // assume size > 1
```

```

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int left = rank - 1;
int right = rank + 1;
if (0 == rank) {
    left = size - 1;
}
if (size - 1 == rank) {
    right = 0;
}

MPI_Status status;
int buf;

MPI_Ssend(&rank, 1, MPI_INT, right, 0, MPI_COMM_WORLD);
MPI_Recv(&buf, 1, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
std::cout << "Process " << rank << " received " << buf << std::endl;

MPI_Finalize();
return 0;
}

```

---

c) **10 points**

Suppose you have a heterogeneous cluster where not all nodes have the same number of cores. The following program tries to synchronize the execution of all OpenMP threads spawned by the MPI processes. Explain why the code is not correct and what will happen when this code is run. Propose a fix for the bug that implements the correct behavior. All OpenMP constructs have to remain in place.

**Hint:** Collective MPI calls require all processes to participate in the communication.

---

```

#include <iostream>
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    int rank, provided;

    // MPI_THREAD_MULTIPLE: If the process is multithreaded, multiple
    //                        threads may call MPI at once with no
    //                        restrictions.
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

```

```

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) { // root process
    omp_set_num_threads(3);
} else {
    omp_set_num_threads(4);
}

std::cout << "Rank " << rank << " before OpenMP parallel region\n"
    << std::flush;
#pragma omp parallel
{
    // synchronize the execution of all threads
    MPI_Barrier(MPI_COMM_WORLD);
}
std::cout << "Rank " << rank << " after OpenMP parallel region\n";

MPI_Finalize();
return 0;
}

```

---