CS205 HW2

Isaac Lee

February 2023

1 Question 1

1.1 a)

Number of Threads	Computing Time (seconds)
1	1.42
2	1.30
4	0.91
8	0.55
16	0.29
32	0.24

Table 1: Computing Time vs. Number of Threads

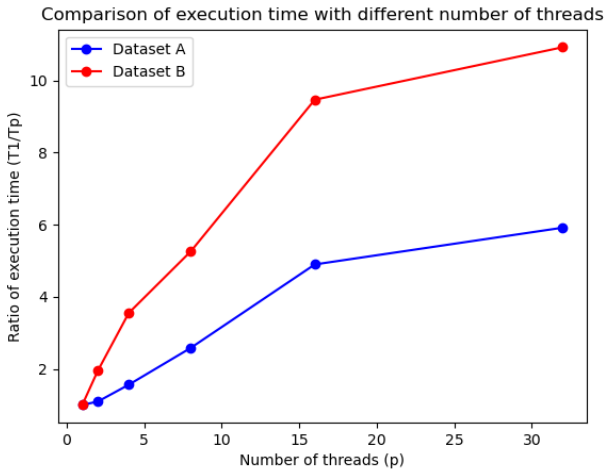

Figure 1: number of threads on the x-axis and the speedup (defined as T1/Tp) on the y-axis.

The plot shows how the speedup increases as the number of threads increases.

If the code were perfectly parallelizable, we would expect to see a linear speedup curve with a slope equal to the number of threads. In other words, if we double the number of threads, we would also expect the speedup to double. However, in practice, there are diminishing returns as the number of threads increases, so the speedup curve may start to flatten out at higher thread counts.

1.2 b)

By specifying schedule(static, 32), the iterations are divided into equally-sized chunks of 32 iterations each, and each chunk is assigned to a thread. This ensures that the workload is balanced across threads, and that each thread gets a similar amount of work to do. There is a performance boost, as can be seen in figure 1.

Number of Threads	**Computing Time (seconds)**
1	1.42
2	0.73
4	0.40
8	0.27
16	0.15
32	0.13

Table 2: Computing Time vs. Number of Threads

1.3 c)

Given that we are using 32 cores, this can be achieved by using schedule(static, 32). As described above, this makes sure that a cache block holds the whole chunk and that this block is only accessed by one thread.

1.4 d)

Histogram

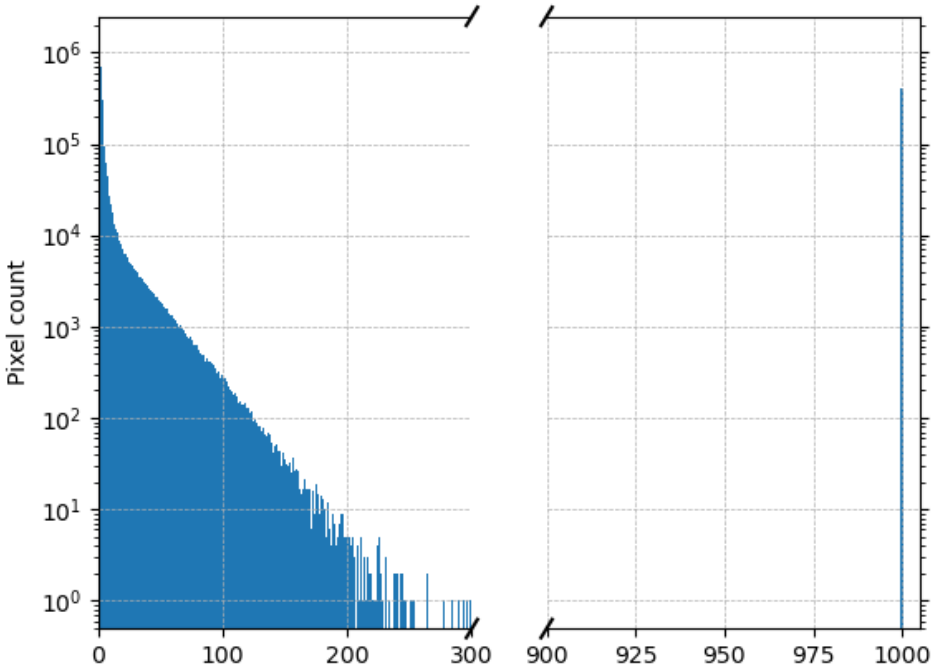

Figure 2: Histogram of how much paint of each color is needed

2 Question 2

2.1 a)

2.1.1 1

DAXPY

$$y = \alpha x + y$$

$$y_i = \alpha x_i + y_i$$

$$f_i = 1_{mul} + 1_{add} = 2$$

$$m_i = 2_{read} + 1_{write} = 3$$

$$F = 2n$$

$$M = 3n$$

Since it is double precision, $p = 8$

$$I(n) = \frac{F}{Mp} = \frac{nf_i}{nm_ip} = \frac{f_i}{m_ip} = \frac{2}{3 \times 8} = \frac{1}{12}\frac{Flop}{Byte}$$

Constant $\mathcal{O}(1)$

2.1.2 2

SGEMV Given n is big and that A and x would not fit in the cache,

Since A is $\mathbf{R}^{n \times n}$, we can treat it as a vector for each loop iteration i.

$$y = Ax + y$$

$$y_i = a_{ij}x_j + y_i$$

$$f_i = (n_{mul}) + ((n_{add}) - 1_{add}) + 1_{add} = 2n$$

$$m_i = 2(n_{read}) + 1_{read} + 1_{write} = 2n + 2$$

$$F = 2n^2$$

$$M = 2n^2 + 2n$$

Since it is single precision, $p = 4$

Keeping only the leading order of operations:

$$I(n) = \frac{F}{Mp} = \frac{nf_i}{nm_ip} = \frac{f_i}{m_ip} = \frac{2}{2 \times 4} = \frac{1}{4}\frac{Flop}{Byte}$$

Constant $\mathcal{O}(1)$

2.1.3 3

DGEMM Again, given n is big and that A, B and C would not fit in the cache,

$$C = AB + C$$

$$c_{ij} = a_{ik}b_{kj} + c_{ij}$$

$$f_{ij} = (n_{mul}) + ((n_{add}) - 1_{add}) + 1_{add} = 2n$$

$$m_{ij} = 2(n_{read}) + 1_{read} + 1_{write} = 2n + 2$$

$$F = n^2 f_{ij} = 2n^3$$

$$M = n^2 m_{ij} = 2n^3 + 2n^2$$

Since it is double precision, $p = 8$

Keeping only the leading order of operations:

$$I(n) = \frac{F}{Mp} = \frac{n^2 f_i}{n^2 m_i p} = \frac{f_i}{m_i p} = \frac{2}{2 \times 8} = \frac{1}{8}\frac{Flop}{Byte}$$

Constant $\mathcal{O}(1)$

If n is small enough such that A, B and C fits in the cache, then $m_{ij} = 3 + 1 = 4$, and the asymptotic bound would be

$$I(n) = \frac{F}{Mp} = \frac{n^2 f_i}{n^2 m_i p} = \frac{f_i}{m_i p} = \frac{2n}{4 \times 8} \leq \frac{n}{16}\frac{Flop}{Byte}$$

$\mathcal{O}(n)$

2.2 b)

Assume that cache is big and has no capacity miss.

considering the pre-computed constant factors, we can simplify the equation to $X = \frac{1}{N}Wx$, where x is a real vector of size N and W is a complex matrix.

$$f = N(2_{mul}) + (N(2_{add}) - 1_{add}) + 2_{mul} = 4N + 1$$

$$m = N(3_{read}) + 2_{write} = 3N + 2$$

Operating, reading/writing complex number is 2.

Operation intensity =

$$\frac{N(4N+1)}{N(3N+2)8} = \frac{1}{6}\frac{Flop}{Byte}$$

3 Question 3

3.1 a)

The roofline model is a performance model that helps to analyze the computational efficiency of a hardware system by comparing its operational performance and memory bandwidth. The model displays a log-log plot with operational performance on the x-axis and operational intensity on the y-axis. The graph is bounded by two lines, which represent the memory bandwidth and the peak operational performance of the hardware, with ceiling optimizations (ex. single core, TLP, ILP, DLP). The graph provides information about the theoretical maximum performance of a system for a given computational problem and the bottlenecks that limit the system's performance, making it a useful tool for optimizing code on modern hardware. The roofline model primarily relates to the hardware's floating-point performance and memory bandwidth.

3.2 b)

3.2.1 i)

$$\pi = f \times n_c \times l_s \times \phi$$

$$l_s = \frac{w_s}{p}$$

$$\pi_A = 2.5 \times 8 \times 4 \times 1 = 80\ Gflop/s$$

π_B is superscalar, so times 2

$$\pi_B = 2 \times 32 \times 16 \times 2 \times 2 = 4096\ Gflop/s$$

3.2.2 ii)

$$i_\beta = \frac{\pi}{\beta}$$

$$I_{\beta,A} = \frac{80}{30} = 2.667\ Flop/Byte$$

$$I_{\beta,B} = \frac{4096}{80} = 51.2\ Flop/Byte$$

3.2.3 iii)

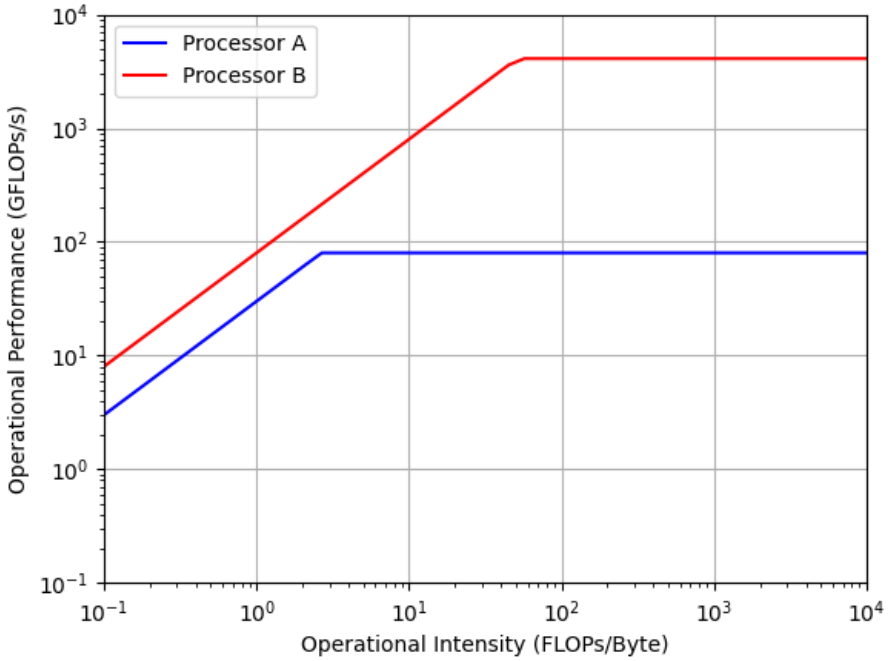

Figure 3: Roofline graph of processor A and B

4 Question 4

4.1 a)

There is a deadlock at MPI_Bcast because all processes need to call MPI_Bcast if they want to receive the broadcasted data. MPI_Bcast is a collective communication routine that broadcasts a message from the root process to all other

processes in a communicator. The root process provides the data to be broadcasted and all other processes receive the data.

It is important that all processes call MPI_Bcast with the same parameters, including the communicator, data type, count, root process rank, and so on. This ensures that all processes receive the same data at the same time and the communication is correctly synchronized.

To fix, I removed the MPI_Recv call.

```
#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);

    std::cout << "Rank " << rank << " received " << msg << std::endl;

    MPI_Finalize();
    return 0;
}
```

4.2 b)

There is a deadlock because MPI_Ssend is a non-local procedure. MPI_Ssend sends the message to the right neighbor, which is a blocking send operation that requires the receiver to start a matching receive operation before the send can complete. However, the code uses MPI_Recv to receive the message from the left neighbor, which is a non-blocking receive operation that does not guarantee that the message has arrived before the program continues. Therefore, there is a potential deadlock in the code, where the sender is blocked waiting for the receiver to receive the message, while the receiver is blocked waiting for the message to arrive.

Using MPI_Bsend fixes the issue since Bsend is local.

```
#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_Request reqs[2];
    MPI_Status stats[2];
    int buf;

    MPI_Isend(&rank, 1, MPI_INT, right, 0, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf, 1, MPI_INT, left, 0, MPI_COMM_WORLD, &reqs[1]);

    MPI_Waitall(2, reqs, stats);

    std::cout << "Process " << rank << " received " << buf << std::endl;

    MPI_Finalize();
    return 0;
}
```

4.3 c)

The code is not correct and the code will deadlock upon running. The problem arises because the root rank has 3 threads while other threads have 4. All ranks have to call MPI_Barrier the same number of times. However, with the current setup, rank root would call it 3 times while others call it 4 times. At the point of MPI_Finalize, the root rank would have only called 3 MPI_Barrier calls, with other ranks still waiting for the 4th one.

One possible way to fix the issue is to add #pragma omp single to MPI_Barrier, such that only one thread per rank call it. This way, all ranks, including the root, would call it the same number of times, thus avoiding the deadlock.

```
#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::f

    #pragma omp parallel
    {
        // synchronize the execution of all threads
        #pragma omp single
        MPI_Barrier(MPI_COMM_WORLD);
    }

    std::cout << "Rank " << rank << " after OpenMP parallel region\n";

    MPI_Finalize();
    return 0;
}
```