Homework 2

Walt Williams

March 7, 2023

1 Problem 1

1.1 a

# of threads					
1	2	4	8	16	32
1.43s	1.30s	.87s	.52s	.32s	.25s

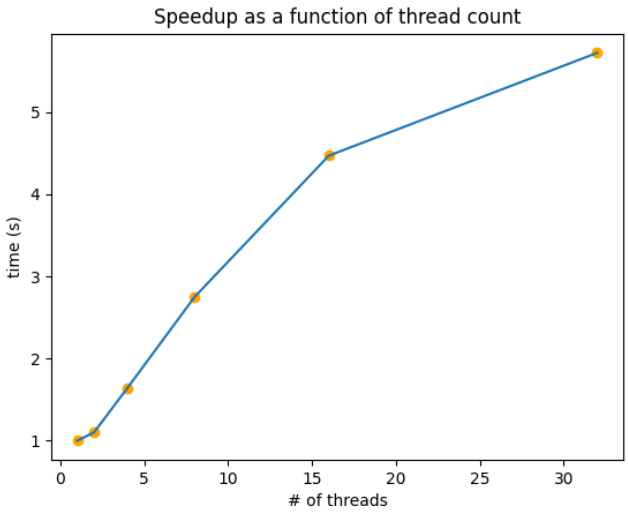

Figure 1: A visualization of performance vs. thread count on our compute node
In figure 1 we're visualizing the performance gain we get by increasing the thread count. If this code were perfectly parallelizable then we would expect the speedup curve to be a linear piecewise function where the slope of each individual piece is just the number of threads.

1.2 b

# of threads					
1	2	4	8	16	32
1.43s	.77s	.45s	.28s	.17s	.13s

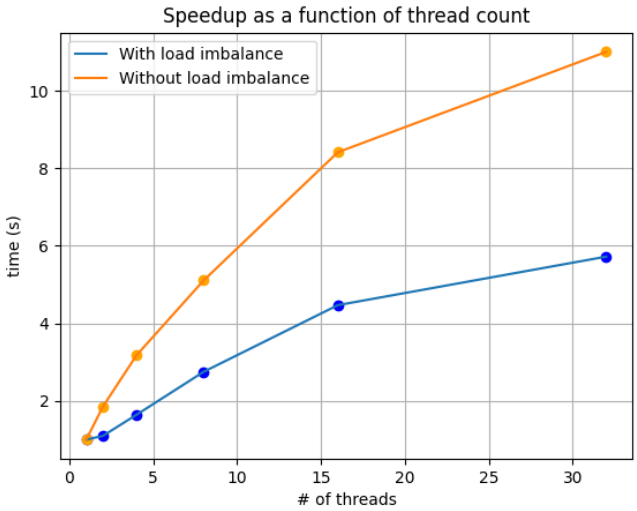

Figure 2: A comparison of parallelism with/without load balance on a compute node with 32 cores

In order to get this second figure we simply added a static schedule with 32 cores to our code to make sure that all 32 cores that all of the cores share their work evenly. By looking at the plot there is indeed a noticeable difference in performance when we handled the load sharing issue.

1.3 C

Since we're using a node with 32 cores we can achieve this by using a schedule(static, 32) to make sure that a cache block holds an entire chunk of data and that only one thread can access this block.

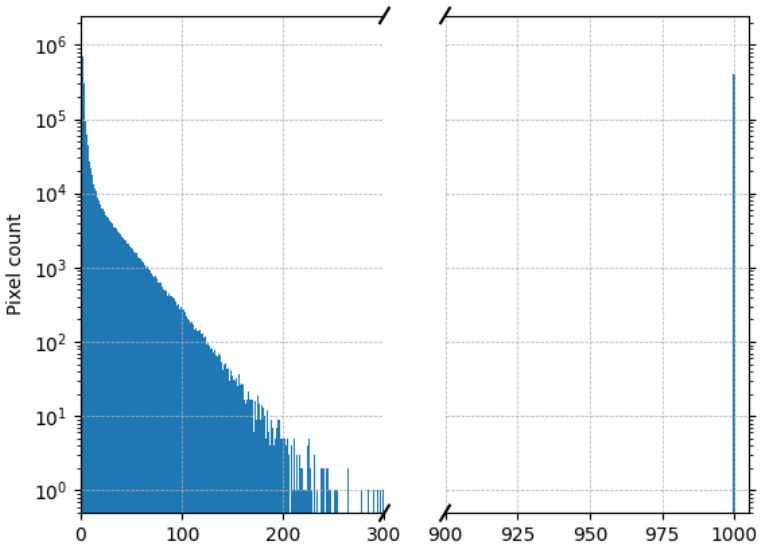

Figure 3: Histogram of paint colors

1.4 D

2 2

2.1 a

2.1.1

DAXPY:
$y = \alpha x + y$
$yi = \alpha x_i + y_i$
$f_i = 1_{mul} + 1_{add} = 2$
$m_i = 2_{read} + 1_{write} = 3$
$F = 2n$
$M = 3n$, and since this is double precision p=8
Therefore: $I(n) = \frac{F}{Mp} = \frac{nf_i}{nm_ip} = \frac{2}{24} = \frac{1flop}{12bytes}$ so it is $O(1)$

2.1.2

SGEMV: Given that n is a large number and Matrix A and vector x would not fit in the cache, since A is a square matrix of size nxn we can treat it as a vector and for each iteration i:
$y = Ax + y$

$yi = \alpha_{ij} x_j + y_i$
$f_i = n_{mul} + (n_{add} - 1_{add}) + 1_{add} = 2n$
$m_{ij} = 2n_{read} + 1_{read} + 1_{write} = 2n + 2$
$F = 2n^2$
$M = 2n^2 + 2n$
Since this is single precision p=4, therefore: $I(n) = \frac{F}{Mp} = \frac{nf_i}{nm_ip} = \frac{f_i}{m_ip} = \frac{2}{8} = \frac{1flop}{4bytes}$
so once again this is constant time complexity or O(1)

2.1.3

DGEMM: If n is large, A,B, and C would not fit in the cache so,
C = AB + C $c_{ij} = a_{ik} b_{kj} + c_{ij}$
$f_i = n_{mul} + (n_{add} - 1_{add}) + 1_{add} = 2n$
$m_{ij} = 2n_{read} + 1_{read} + 1_{write} = 2n + 2$
$F = n^2 f_{ij} = 2n^3$
$M = n^2 m_{ij} = 2n^3 + 2n^2$
Once again, double precision so p=8
Therefore: $I(n) = \frac{F}{Mp} = \frac{n^2 f_i}{n^2 m_i p} = \frac{2}{16} = \frac{1flop}{8bytes}$ so it is $O(1)$
If n is a small enough size so that all matrices fit in the cache then $m_{ij} = 4$ and the bound would be:
$I(n) = \frac{F}{Mp} = \frac{n^2 f_i}{n^2 m_i p} = \frac{2n}{32} = \frac{nflops}{16bytes}$
so it would be linear time complexity or O(n)

2.2 b

If we assume that the cache is sufficiently large and has no capacity to 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.
$f = N(2_{mul}) + N(2_{add}) - 1_{add} + 2_{mul} = 4N + 1$
$m = N3_{read} + 2_{write} = 3N + 2$
Operating, reading, and writing complex numbers is 2 therefore: Operation intensity=$\frac{N(4N+1)}{8*N(3N+2)} = \frac{1flop}{6bytes}$

3

3.1 a

The roofline model is a performance model that helps to analyze a hardware system's computational efficiency. In other words, it shows how efficiently a program is using our memory bandwidth and raw computing power. The model conveys this information by plotting a line at the best possible performance for a given operational intensity. If we have a program that meets this performance then we know that our code is making the best use of our memory bandwidth (which is the constraining factor until reaching balanced operational intensity).

The model also plateaus at a I_b which is our balanced operational intensity meaning that we have achieved both the highest operational intensity and the maximum capacity of our memory bandwidth.

3.2 b

3.2.1

$\pi = f * n_c * I_s * \phi$
$I_s = \frac{w_s}{p}$
$\pi_a = 2.5 * 8 * 128/32 * 1 = 80 Gflop/s$
$\pi_b = 2 * 32 * 512/32 * 2 = 2048 Glop/s$
since processor B is superscalar we multiply π_b by 2 to get 4096 Gflop/s

3.2.2

$I_b = \frac{\pi}{\beta}$
Processor A: $I_b = \frac{80}{30} \approx 2.66$
Processor B: $I_b = \frac{4096}{80} = 51.2$

3.2.3 (attached as figure 4)

4

4.1 a

The bug in this code is that there is a deadlock when MPI_Bcast is being called since all processes need to call MPI_Bcast. MPI_Bcast is a collective communicatine routine that broadcasts a message from the root process to all the other processes in the context. It's important that all processes call MPI_Bcast with the same parameters, including the communicator, data type, count, root process rank etc so that all processes receive data at the same time. To do this, we can just remove the MPI_Recv call.

4.2 b

The bug in this code is a deadlock that's a result of the MPI_Ssend. Since it's a non-local procedure it requires that the receiving process has a matching receive before it gives up access to the data buffer. However, the code uses MPI_Recv to receive its message from the left neighbor which is non-blocking procedure that doesn't guarantee that its message has arrived before the program continues, this is the source of the deadlock. To fix it we can just use MPI_Bsend since it's local.

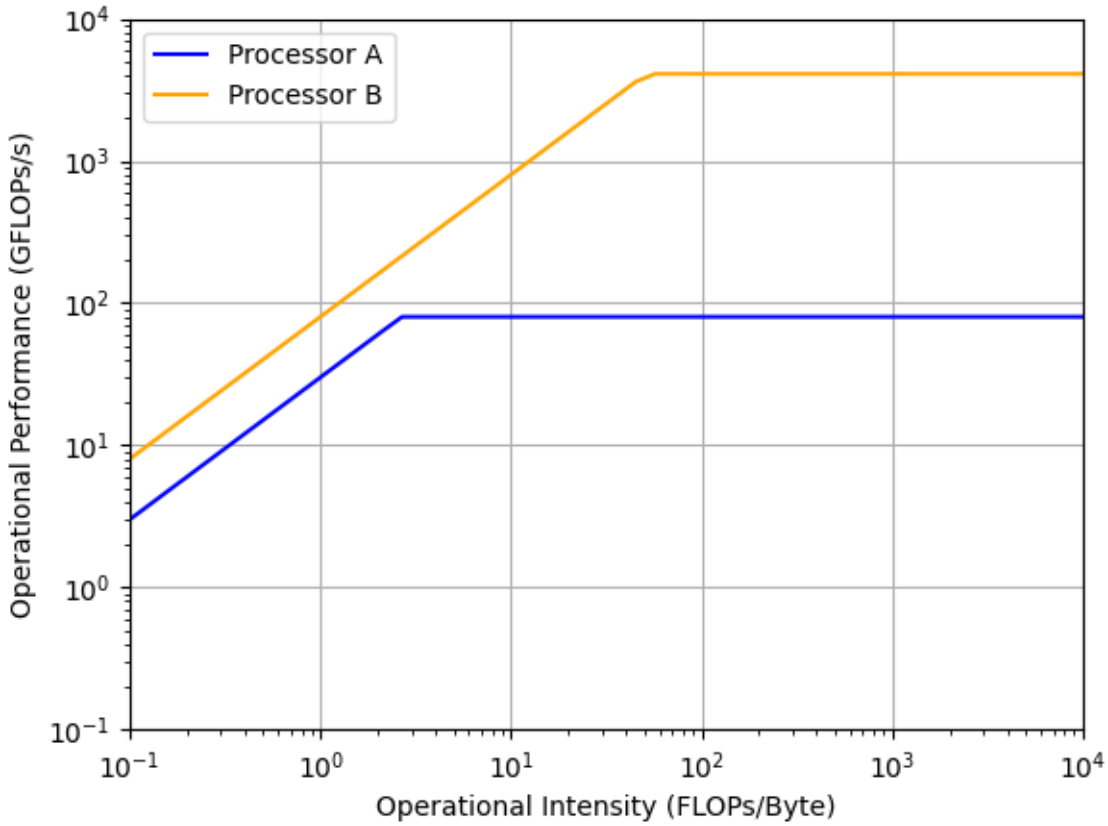

Figure 4: Roofline plot comparing processors A and B

4.3 c

This code will result in a deadlock. The problem is because the root process has 3 threads while the others have 4. All processes call MPI_Barrier but with the current setup process 0 calls it 3 times and the others call it 4 times. When the program gets to MPI_Finalize the non-rank-0 processes will be waiting for the 4th MPI_Barrier call. One way to fix the issue is to add #pragma omp single to MPI_Barrier so that only one thread per rank calls it. This way all the ranks would call it the same number of times and avoid the deadlock.