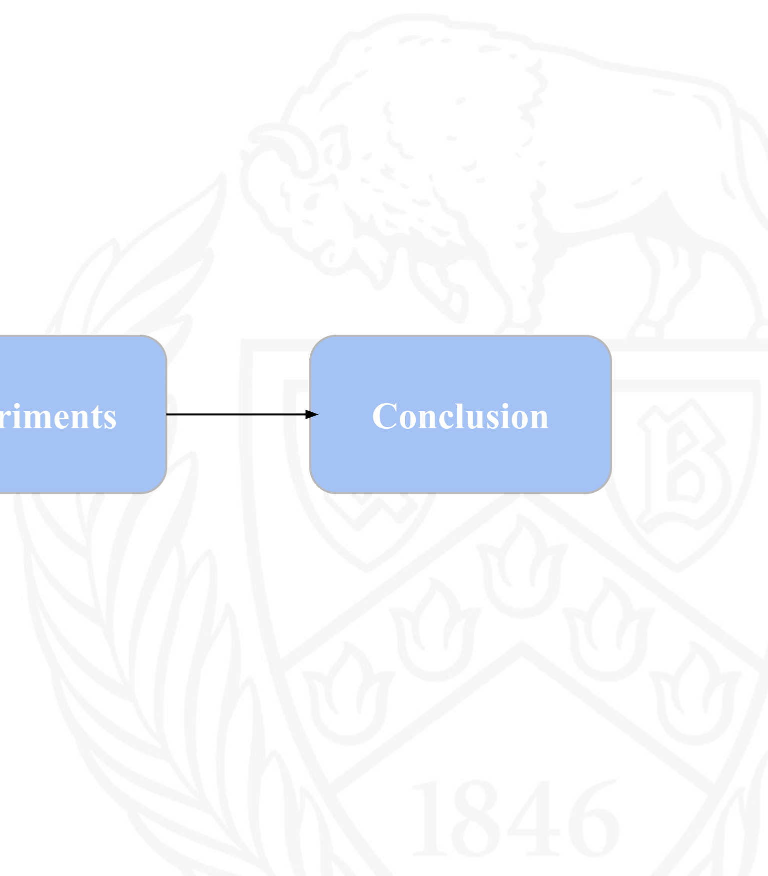
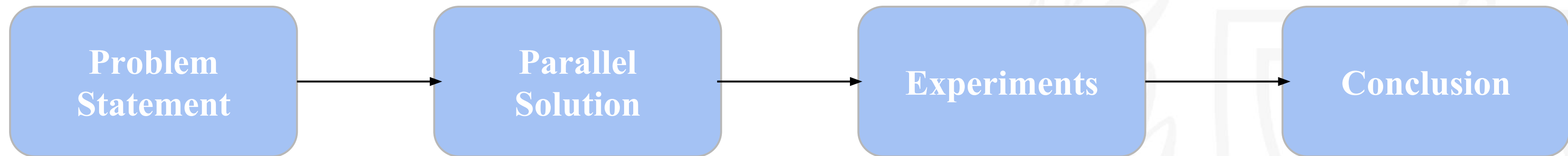
CSE 633: Parallel Matrix Multiplication

Cole Severance

University at Buffalo The State University of New York

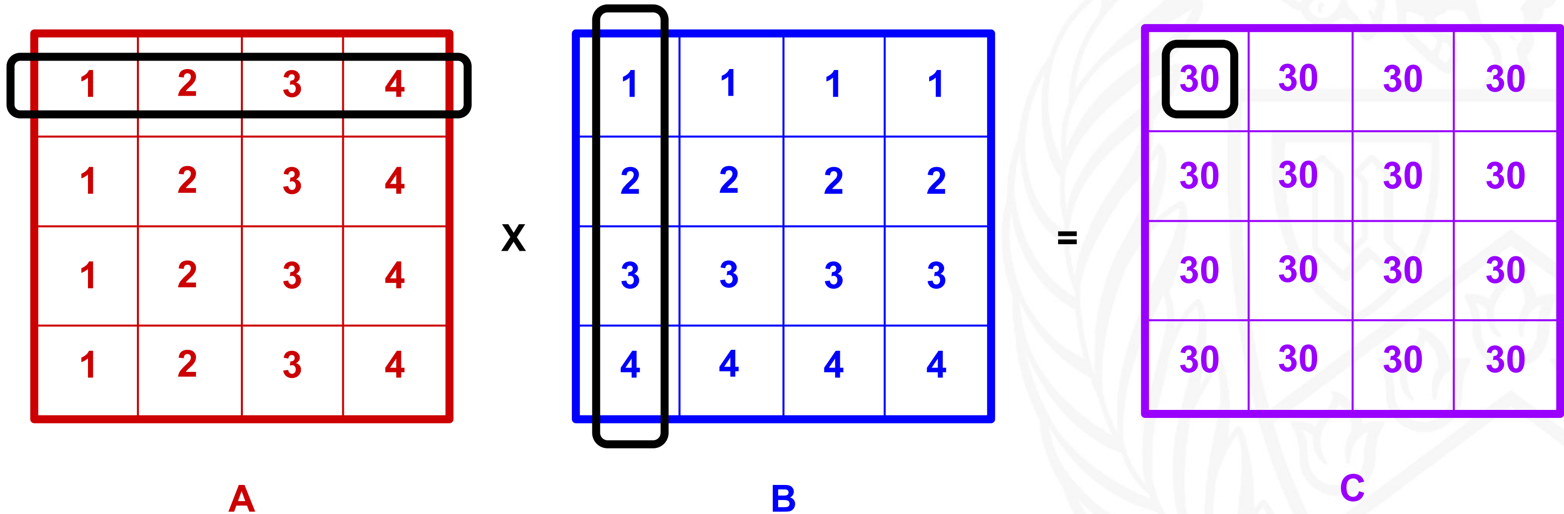
Roadmap





Problem Statement: Matrix Multiplication

- Matrix multiplication is an operation that produces a single matrix from two input matrices, A and B
- It can only be performed in cases where the number of columns in A matches the number of rows in B
- The resulting matrix C has the same number of rows as A and the same number of columns as B
  - Each element is the sum of the product of each corresponding pair of elements from A and B



Sequential Solution

- The sequential algorithm is simple: it consists of a triple loop over the rows of A, the columns of B, and the shared dimension (representing the columns of A and rows of B, which should have the same size)
- Runtime Complexity:
  - O(i*j*k), where:
    - i = rows of A
    - j = columns of B
    - k = shared dimension
- Extra Space Requirements:
  - O(1)
- These calculations are independent and need not be done sequentially, so this problem can be parallelized

Pseudocode:

```
C = [rows of A][columns of B]
For i in rows of matrix A:
    For j in columns of matrix B:
        For k in the shared dimension:
            C[i][j] += A[i][k] * B[k][j]
```

Parallel Solution

- Many parallel algorithms have been designed for this problem, but the algorithm I am using is called the SUMMA algorithm
  - It uses a mesh of processors and assigns each one to be responsible for calculating the results for a subregion of the output matrix
  - Each processor also gets assigned a subregion of each input matrix
  - It is simplest to visualize (and implement) this with square matrices, so I will be using square matrices for the duration of the project

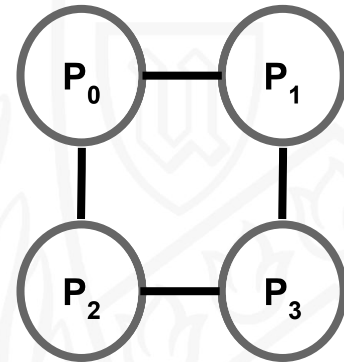
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

A

X

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

B

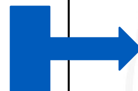


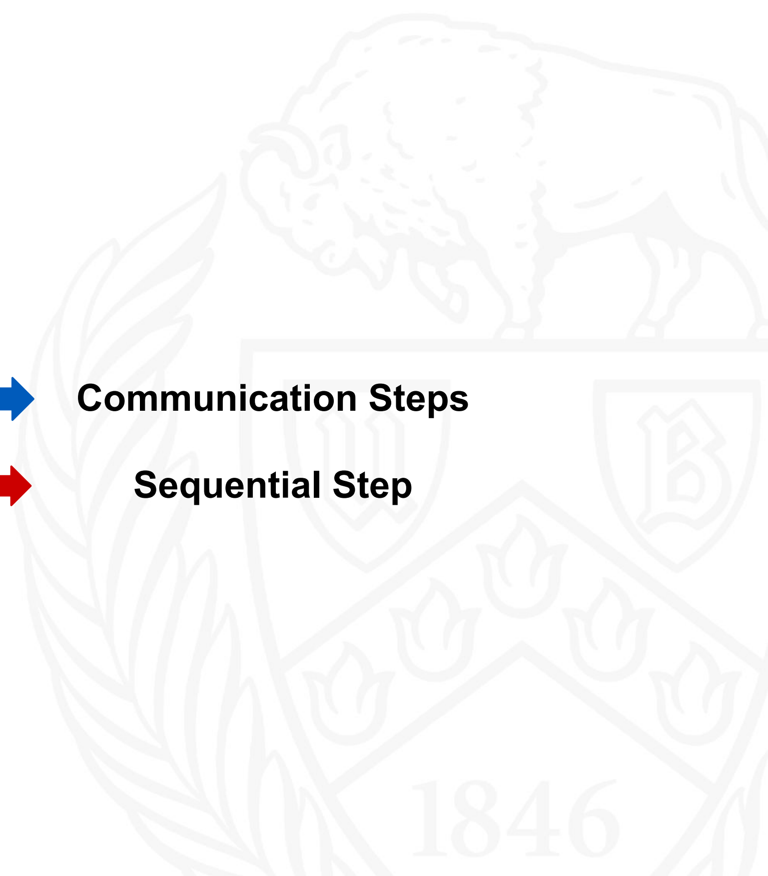
Parallel Solution

Pseudocode:

For each block k in shared dimension:

    Broadcast column k from A across rows
    Broadcast row k from B across columns
    Perform sequential matrix multiplication



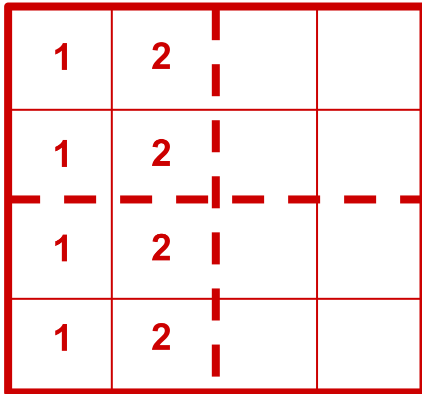


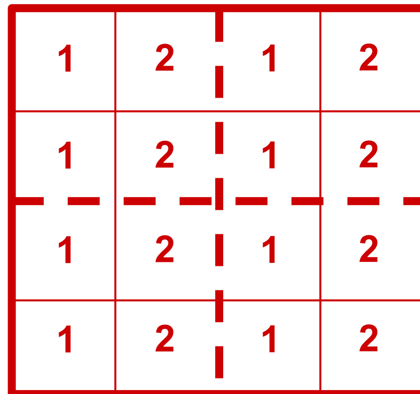
Communication Steps

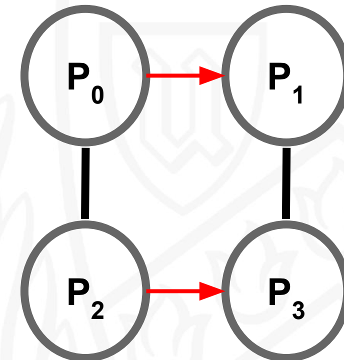
Sequential Step

Parallel Solution

- The algorithm begins by broadcasting a column of blocks from A;
  - The processors with blocks of A being broadcasted send their blocks to the other processors in their row
  - When this step is finished, all of the processors will have the appropriate data from the column of blocks being broadcasted

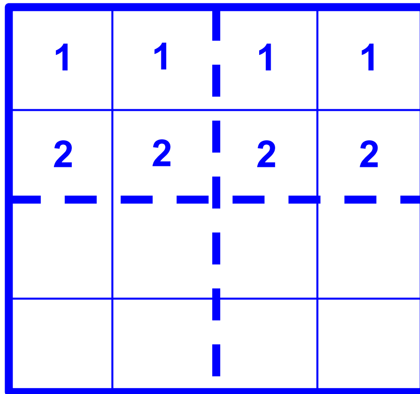


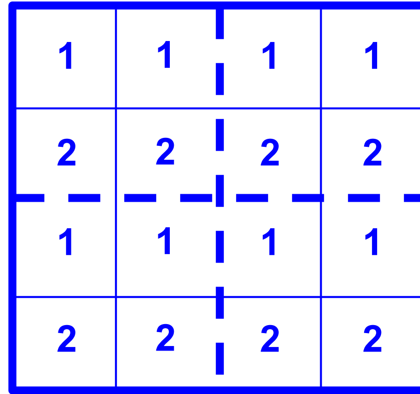


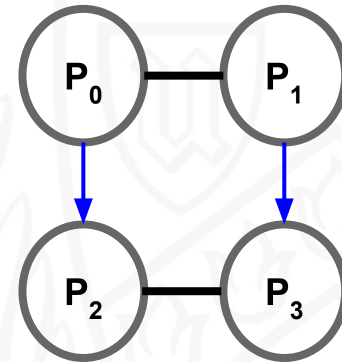


Parallel Solution

- The next step involves broadcasting a row of blocks from B
  - The processors with portions of the row being broadcasted send their blocks of matrix B to the other processors in their column
  - When this step is finished, all of the processors have the right portion of matrix B for their calculations







Parallel Solution

- Now that each processor has the appropriate data, they can update their block of matrix C
  - Simultaneously and in parallel, the matrices multiply their received block from A and their received block from B, just as in the sequential algorithm
  - Then their block of matrix C gets updated with the product

Happening in Each Processor

State of Matrix C After this Step

Parallel Solution

Before

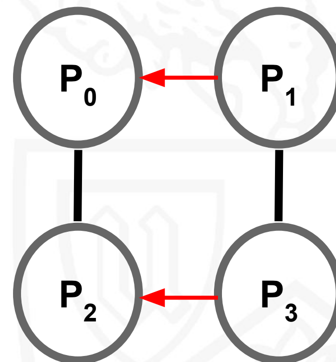
		3	4
		3	4
		3	4
		3	4

A

After

3	4	3	4
3	4	3	4
3	4	3	4
3	4	3	4

A



Parallel Solution

Before

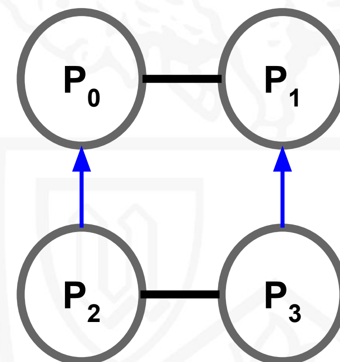
3	3	3	3
4	4	4	4

B

After

3	3	3	3
4	4	4	4
3	3	3	3
4	4	4	4

B



Parallel Solution

Happening in Each Processor

3	4
3	4

A

X

3	3
4	4

B

=

25	25
25	25

C

5	5
5	5

C

+

25	25
25	25

C

=

30	30
30	30

C

State of Matrix C After this Step

30	30	30	30
30	30	30	30
30	30	30	30
30	30	30	30

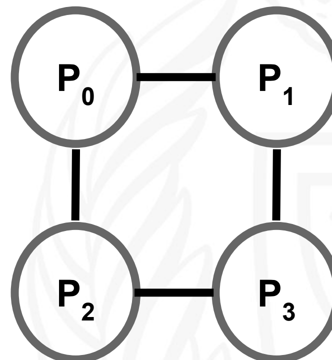
C

Parallel Solution

After k Updates

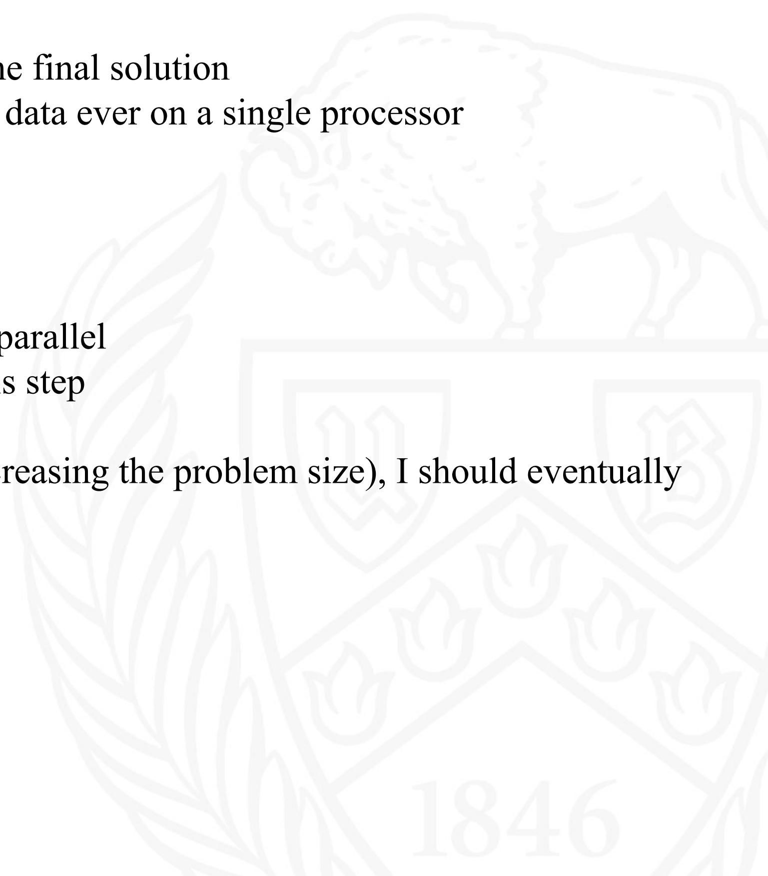
30	30	30	30
30	30	30	30
30	30	30	30
30	30	30	30

C



Parallel Solution

- When the algorithm finishes, each processor contains one block of the final solution
  - At no point during the duration of the algorithm was all of the data ever on a single processor
- At a higher level, the algorithm involves 3 steps:
  1. Row broadcast
  2. Column broadcast
  3. Sequential computation step
- The 3rd step happens on all of the processors simultaneously and in parallel
  - The performance gains from parallelism should come from this step
- Steps 1 and 2 are communication steps
  - If I continue to increase the number of processors (without increasing the problem size), I should eventually see the runtime of the algorithm limited by these steps



Experiment 1: Communication Bottleneck

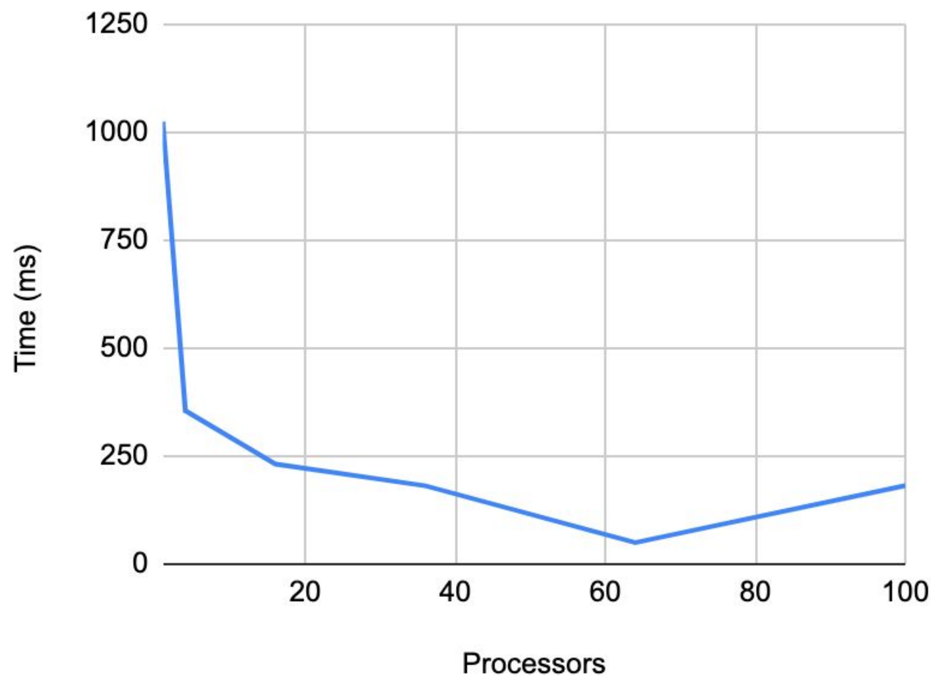
- The goal for this first experiment was to show that increasing the number of processors, while keeping the problem size the same, results in the following:
    - A decrease in runtime, up to a certain point
    - A point beyond which increasing the number of processors increases the runtime (since communication costs take over)
- I kept the problem size the same (A, B and C are 1320 x 1320 matrices) and varied the number of processors
- I submitted the script to slurm with 1 process per node, and the exclusive flag set to ensure that they entire node is just running my job
- I ran the experiment 10 times and averaged the runtime over the 10 trials to mitigate the effects of outliers

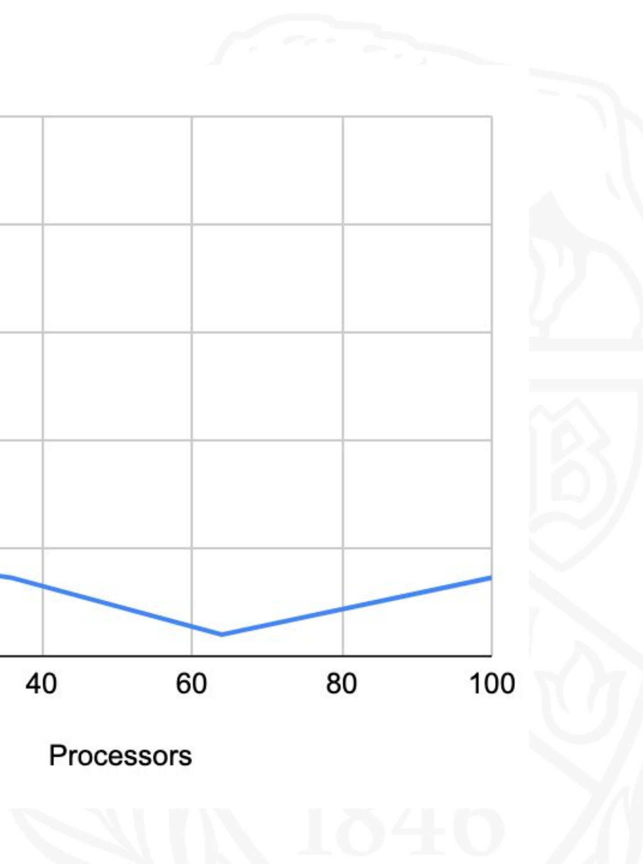
```
#!/bin/bash
#SBATCH --nodes=100

#SBATCH --ntasks=100
#SBATCH --exclusive
#SBATCH --ntasks-per-node=1
#SBATCH --time=00:05:00
```

Experiment 1 Results

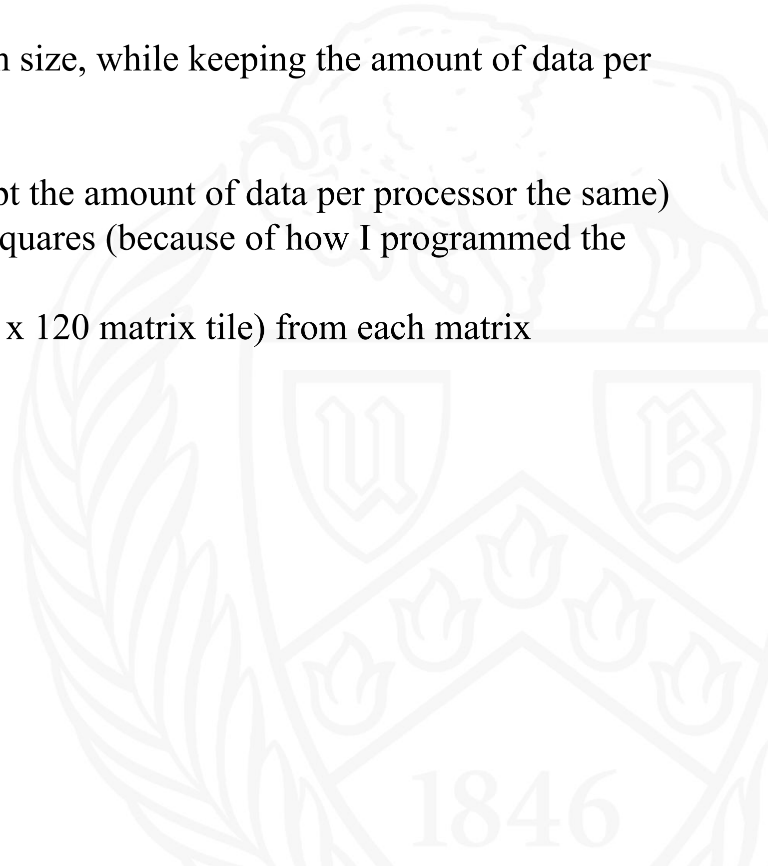
# Processors	Average Runtime of 10 Trials (ms)
1	1025.44
4	355.66
16	231.98
36	181.48
64	50.38
100	182.10





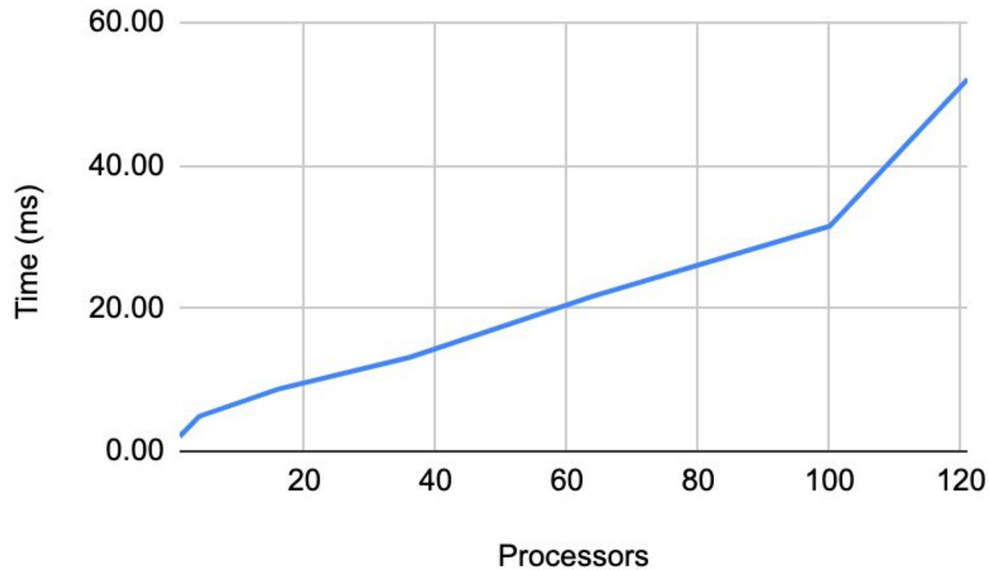
Experiment 2: Scaled Speed-Up

- The goal for this experiment was to show that increasing the problem size, while keeping the amount of data per processor the same, results in the following:
    - A steady increase in the runtime as more processors get added
- I varied both the number of processors and the problem size (and kept the amount of data per processor the same)
    - The number of processors and the input data must be perfect squares (because of how I programmed the algorithm)
    - Each processor had 14,400 pieces of data (equivalent of a 120 x 120 matrix tile) from each matrix
- I ran 10 trials and averaged the runtime over the trials



Experiment 2 Results

p	n	n/p	Runtime (ms)
1	57600	14400	2.11
4	230400	14400	4.92
16	518400	14400	8.72
36	921600	14400	13.15
64	1440000	14400	21.73
100	2073600	14400	31.51
121	3686400	14400	52.06



Takeaways and Improvements

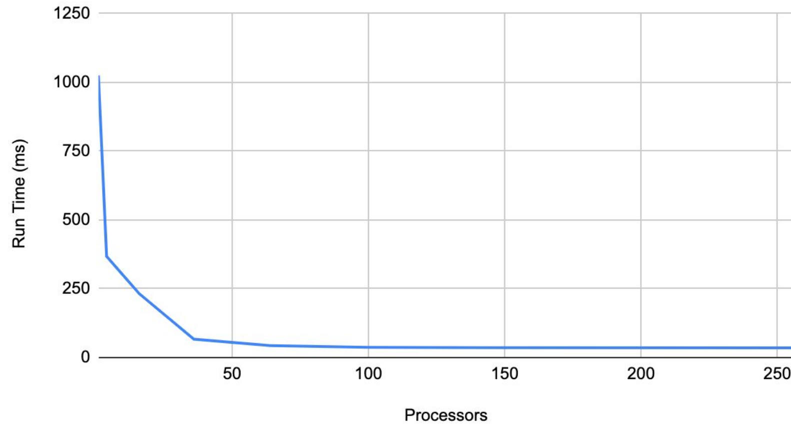
- The two big things I would have done differently if I could go back are the following:
  1. Spend more time understanding slurm at the project's outset
  2. Use larger input matrices so that runtime measurements are more meaningful
- The results of my first round of experiments ended up being invalid for what I wanted to test because I was running the experiments with the wrong slurm specifications
  - My slurm script was initially set up properly, with 1 process per compute node and the exclusive flag set
  - However, when I was unable to submit jobs with 144 and 256 nodes, I convinced myself that 1 process per CPU was the same as 1 process per node
  - So I ran the experiments with 1 process per CPU, and even though the results showed the effects I wanted to see they were invalid

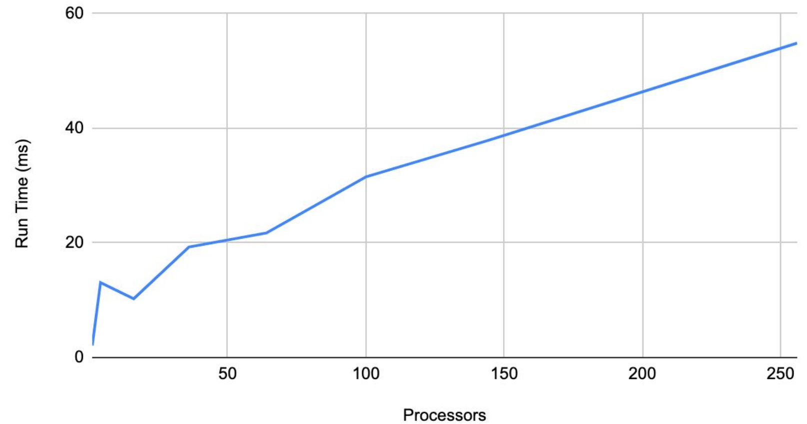
```
#!/bin/bash
#SBATCH --nodes=72

#SBATCH --ntasks=144
#SBATCH --exclusive
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=1
```

Takeaways and Improvements

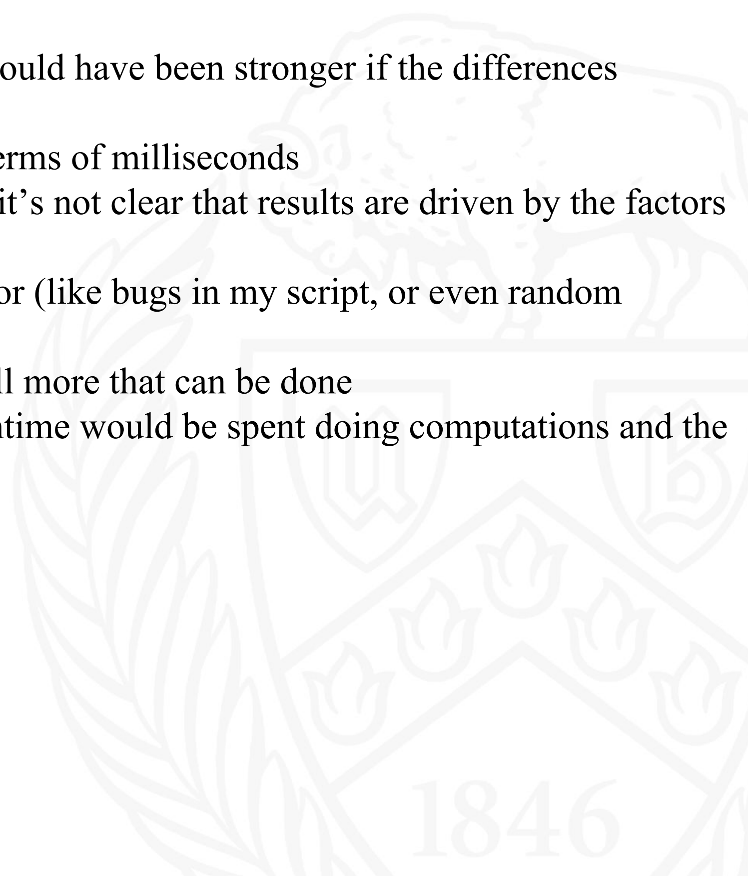
- The following graphs show the invalid results from my first set of experiments:





Takeaways and Improvements

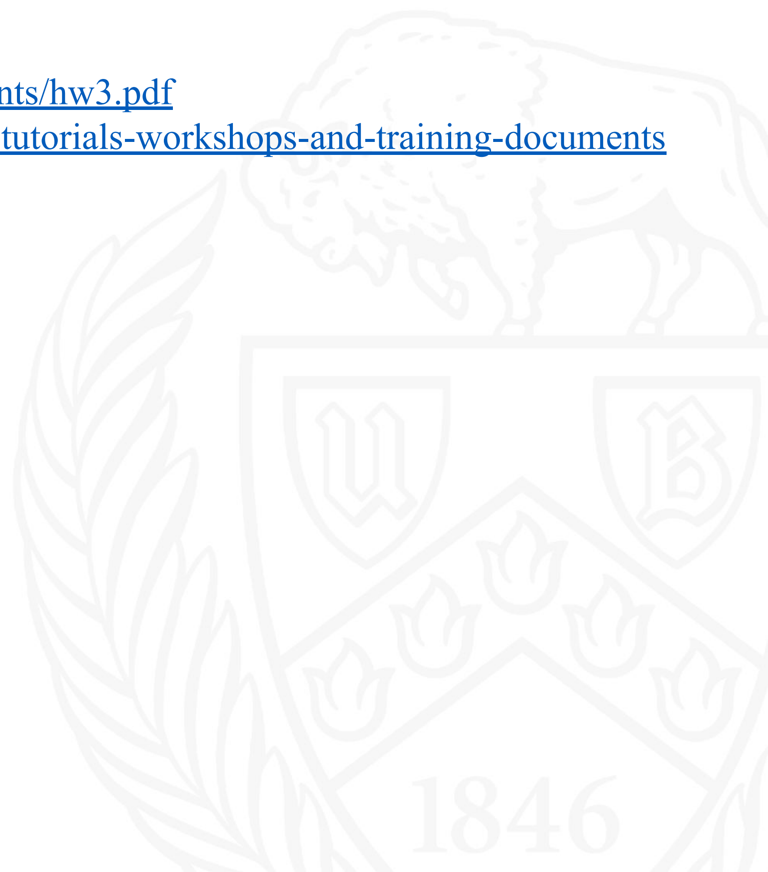
- The final results showed the patterns that I wanted to see, but they would have been stronger if the differences between runtimes were more significant
    - For example, my results for the second experiment are all in terms of milliseconds
    - Even though the graph shows the trend that I expected to see, it's not clear that results are driven by the factors that I intended to test for
    - There could be factors related to things that I did not control for (like bugs in my script, or even random chance) that impact the results
    - Running 10 trials mitigated the impacts of this, but there is still more that can be done
- Had I used larger input matrices for the experiments, more of the runtime would be spent doing computations and the potential impact of other factors would be greatly reduced



Future Work

- Run experiments 1 and 2 with different sizes of input matrices
  - The results currently don't show how the runtimes vary with the size of the input matrices and testing for this could add to my understanding of the problem
  - For experiment 2, this would also shed light on whether the pattern in the results (a steady increase in runtime with number of processors and problem size) holds for scales larger than a few milliseconds
- Program the same algorithm with a combination of OpenMP and MPI to see if I can get even faster speedup
  - And also learn about how to program with OpenMP
- Implement a parallel solution for a more complicated problem
  - The parallel portion of my script ended up being the easiest part to write since the SUMMA algorithm is so simple; I would have liked to spend more time learning what is possible with MPI

Sources



- https://sites.cs.ucsb.edu/~gilbert/cs140/old/cs140Win2009/assignments/hw3.pdf
- https://ubccr.freshdesk.com/support/solutions/articles/13000026245-tutorials-workshops-and-training-documents
- https://courses.cs.washington.edu/courses/csep524/02au/summa.pdf
- https://en.wikipedia.org/wiki/Matrix_multiplication