# Matrix Addition with MPI: An Exercise in Poor Distributed Programming Application

Ryan Jones

November 7 2018

## 1 Introduction

For this assignment, the task was to use the Message Passing Interface (MPI) to perform matrix multiplication on two same-sized matrices. This project is to introduce the concept of applying the PCAM (Partitioning, Communication, Aggregation, Mapping) model using MPI in the C programming language. As an introduction to MPI, this project offers a simpler separation of the data and simpler computation, however.

I believe that this project should be offered in the future as either an introduction project to using MPI, or as an extra credit assignment to demonstrate how some problems that may do well with a parallel implementation do not gain the same performance increase on a distributed processor program.

## 2 Partitioning

My program generates two matrices, $A$ and $B$, which have equivalent rows and columns to ensure no issues would occur with the matrix addition. For the benefits of cacheing I partitioned the $A$ and $B$ matrices by rows using the formula $L = \frac{P}{Rows}$, where $P$ is the number of processors specified at runtime. If the number of rows is not evenly divisible by the number of processors, I assigned any rows left over by the partitioning formula to $Pth$ processor.

For each processor used at runtime, processor zero distributes $L$ rows of $A$ and B sequentially, which each processor then recieves in a separate function. Because of the nature of my partitioning, this method of partitioning will work on any size matrix and any number of processors.

## 3 Data

Table 1: Data for 1024x1024 matrices on multiple cores

| Matrix | $p$ | $t_p$ | $s$ | $e$ |
|---|---|---|---|---|
| 1024x1024 | 1 | 0.989 | 1.0 | 1.0 |
| | 2 | 1.214 | 0.82 | 0.41 |
| | 4 | 1.133 | 0.87 | 0.218 |
| | 8 | 1.510 | 0.66 | 0.08 |

Graph 1: $1024^2$ Matrices

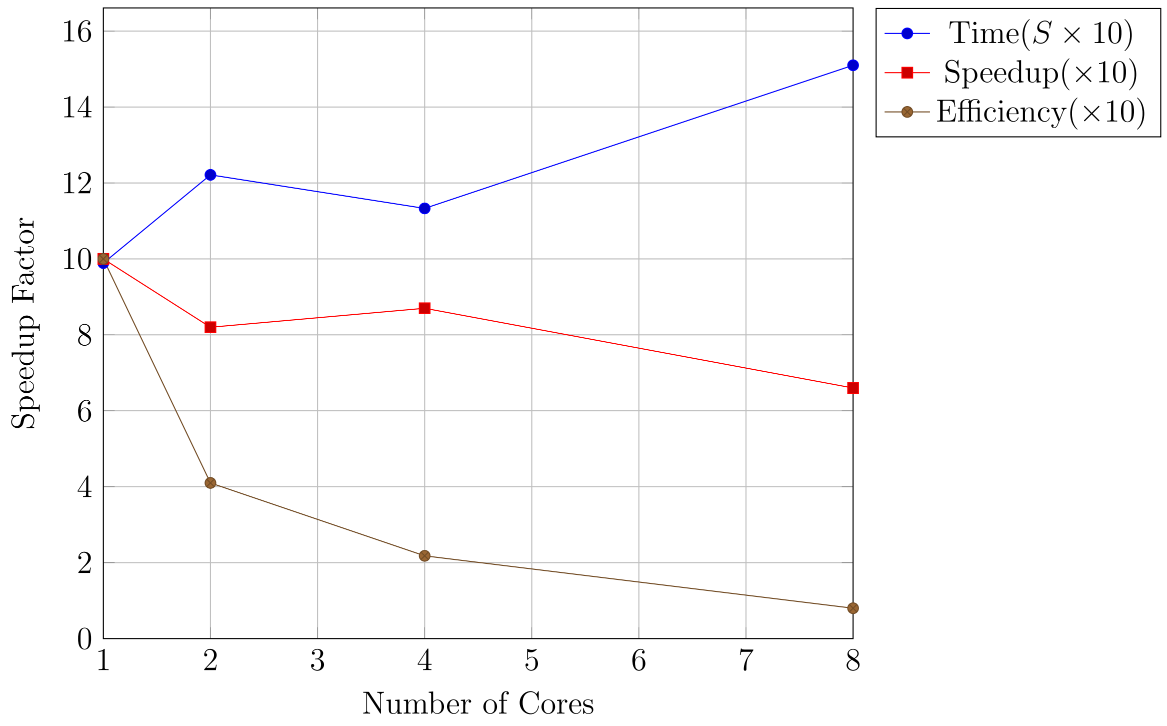


Table 2: Data for 4096x4096 matrices on multiple cores

| Matrix | $p$ | $t_p$ | $s$ | $e$ |
|---|---|---|---|---|
| 4096x4096 | 1 | 11.146 | 1.0 | 1.0 |
| | 2 | 11.114 | 1.0 | 0.5 |
| | 4 | 13.884 | 0.80 | 0.2 |
| | 8 | 16.679 | 0.67 | 0.08 |

Graph 1: $1024^2$ Matrices

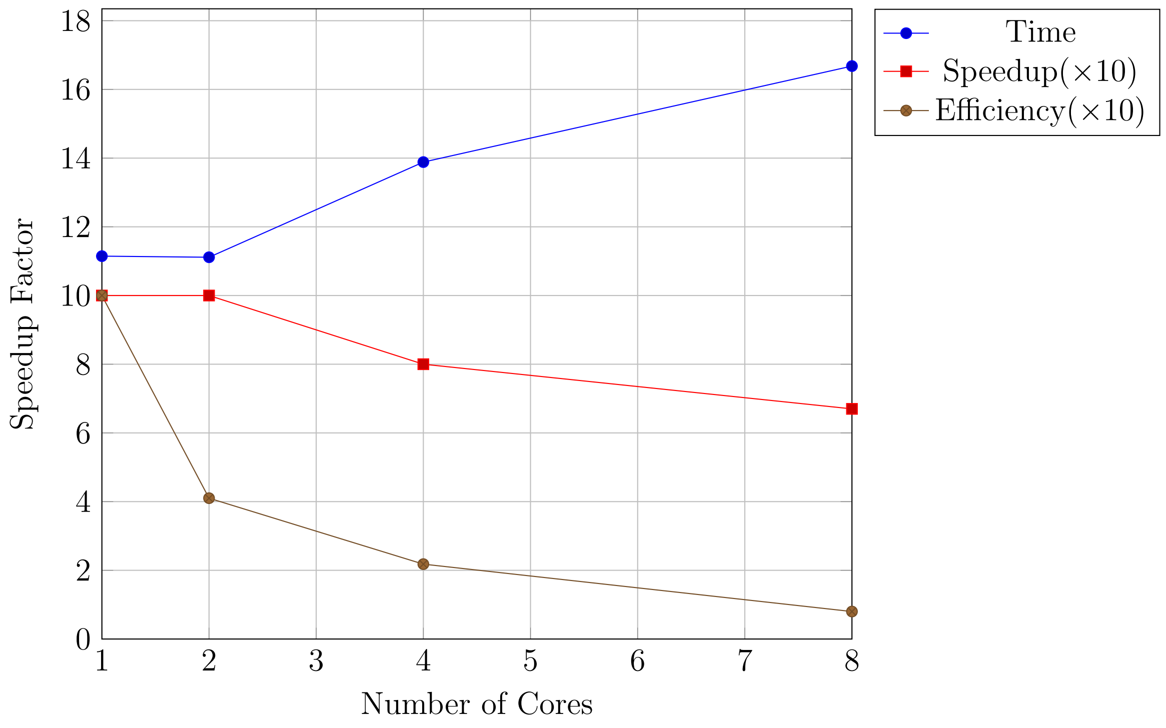
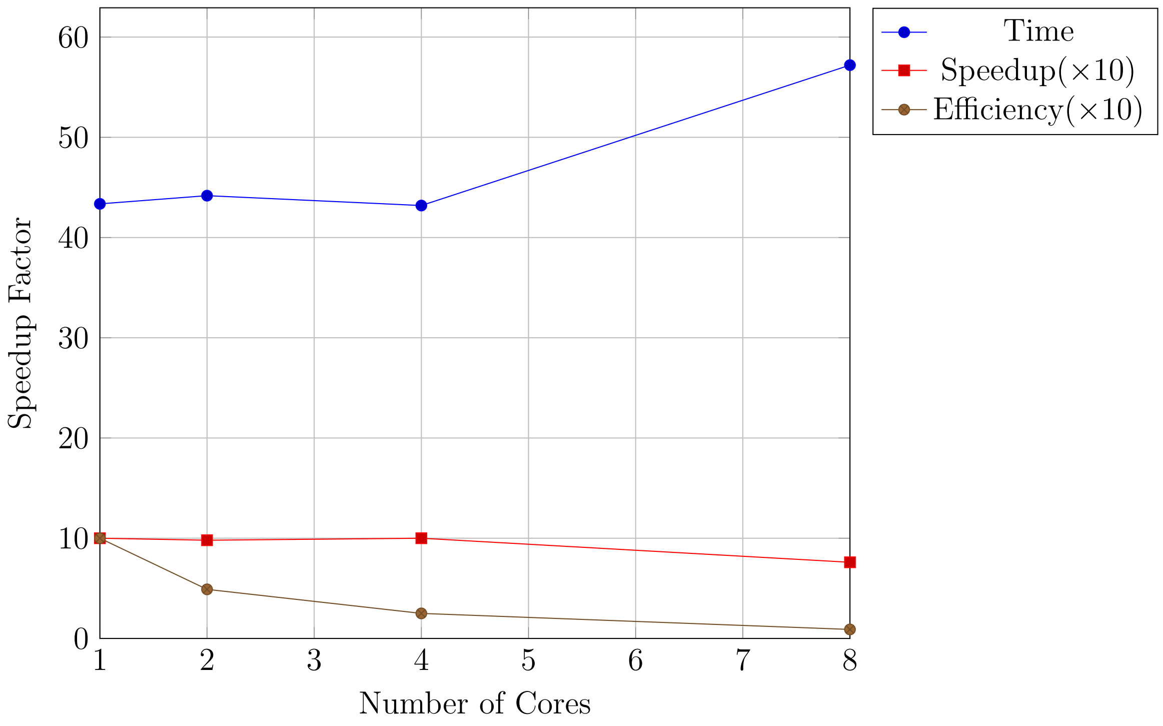


Table 3: Data for 8192x8192 matrices on multiple cores

| Matrix | $p$ | $t_p$ | $s$ | $e$ |
|---|---|---|---|---|
| 8192x8192 | 1 | 43.359 | 1.0 | 1.0 |
| | 2 | 44.177 | 0.98 | 0.49 |
| | 4 | 43.192 | 1.0 | 0.25 |
| | 8 | 57.200 | 0.76 | 0.09 |

Graph 1: $1024^2$ Matrices



## 4 Analysis

Compared to other parallel and distributed computing projects we have done, this project had very interesting results. For every test I ran, efficiency went down when doubling from one to two processors. The program occasionally resulted in a speedup that was at least 1, but otherwise was less than one.

My interpretation of why these results occur lies in the operation being performed on the data set and the partitioning of the data. Matrix addition is essentially an $n$-runtime operation. For a sequential algorithm this program runs in approximately $5n$ time. The first and second $n$ comes from the program reading in the $A$ and $B$. The second comes from the actual computation while running through $A$ and $B$. The last $n$ comes from writing the $C$ matrix to disk.

For a distributed algorithm, the runtime is increased to $7n$ Like the sequential algorithm, the distributed algorithm reads in two matrices $(2n)$, computes the values between them $(2n)$, and writes the resulting matrix to disk $(n)$. The difference with a distributed algorithm is that there is an extra read of $A$ and $B$, which adds at least $2n$ runtime overhead to every program run. This reduces distributed program runtime by about 30% for two processors, and that deficit grows as the number of processors increases with more repetitions of the distribution process.

I also tried a parallel implementation of this computation using Posix Threads. I did this because I wanted to avoid the overhead of distributing whole chunks of the data, and rather compute starting points for the processes to compute from. On a shared memory system, the process of assigning starting points for each process added overhead not found in a sequential implementation. Only on data sets of extreme sizes would a distributed or parallel implementation have the potential to solve a problem like this faster than a sequential implementation.

## 5 Conclusion

Distributed matrix addition is a good example of a computation that does not benefit from distributed programming. This is an exercise that benefits from being done as an extra credit assignment, or as an assignment to introduce the use of MPI in a way that shows thinking about a problem and whether or not it will benefit from distribution and/or parallelization.