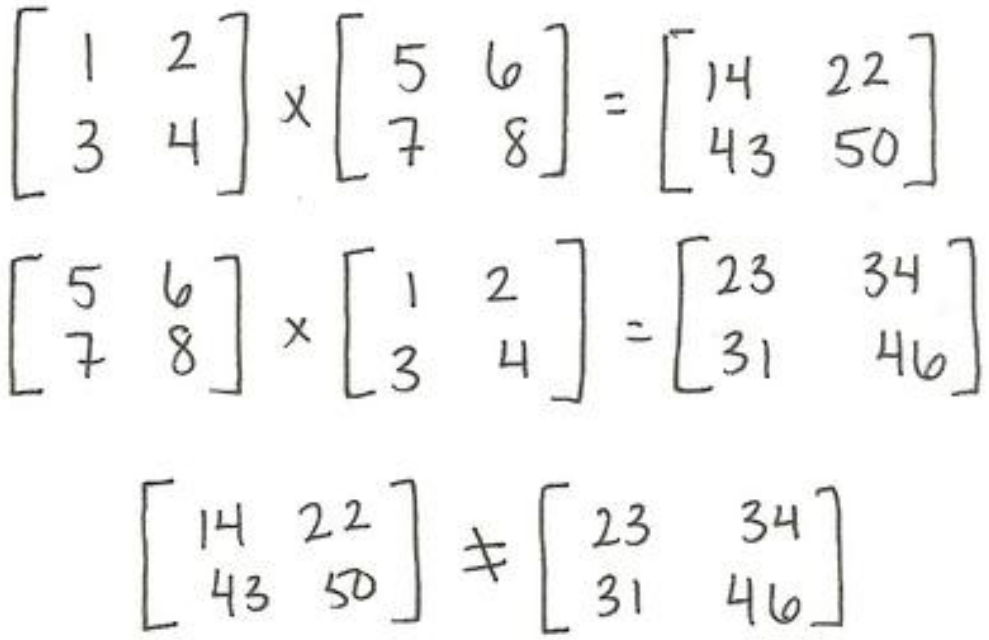
PARALLEL MATRIX MULTIPLICATION

Using Collective Communications and Open MPI



Tim Whitaker
CSCI 551 - Spring 2015

PARALLEL MATRIX MULTIPLICATION

Using the Collective Communications and Open MPI

Tim Whitaker
CSCI 551 - Spring 2015

Introduction

In this assignment, we will use Open MPI to multiply 2 large matrices together in parallel. There are a few different "ijk" forms of matrix multiplication that we will explore as well. These different forms can have different performance qualities due not to any method having fewer steps, but in how the cache is utilized for each.

My Approach

The formula for multiplying 2 square matrices, named A and B respectively, looks like this.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj} \qquad 1 \leq i \leq m, \quad 1 \leq j \leq \ell.$$

There are several ways to structure this operation in code, and they all have to deal with looping over the matrices in a different order. There are 3 models, the inner product, middle product and outer product model. There are 6 different forms in total which correspond to these models: ijk, jik, ikj, jki, kij and kji. We will look at examples of each of these models by timing and analyzing one form of each model: ijk, ikj, and kij. The trick to parallelizing this operation is going to be with partitioning the matrix data so that each process can work on a different part of the result without affecting the other processes answers. In order to do that, I elected to go with a process called row wise decomposition. This involves broadcasting the whole B matrix to every process, and scattering the A matrix by different rows to the different processes. Let's look at how each form, ijk, ikj, and kij performed.

IJK Form

The IJK form is an example of the inner product model. Turns out the ijk form was the worst performing out of all of our tests. The code for our ijk multiplication looks like this.

```
for (int i =0; i < size; i++)
{
     for (int j = 0; j < size; j++)
     {
          for (int k = 0; k < size; k++)
          {
               c[i][j] += a[i][k] * b[k][j];
          }
     }
}
```

The inner product model of 2 vectors is equivalent to a column vector multiplied on the left by a row vector.

$$
\begin{aligned}
\mathbf{a} \cdot \mathbf{b} &= \mathbf{a}^{\mathrm{T}}\mathbf{b} \\
&= \begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\
&= a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \\
&= \sum_{i=1}^{n} a_i b_i,
\end{aligned}
$$

IJK Data

Timing Runs (Seconds)

1	4	8	12	16	20
1.21767E+03	3.09669E+02	1.61574E+02	1.09406E+02	8.27277E+01	6.68550E+01
1.21784E+03	3.06759E+02	1.58592E+02	1.08318E+02	8.23096E+01	6.65851E+01
1.23358E+03	3.18433E+02	1.59622E+02	1.08911E+02	8.23526E+01	6.66088E+01
1.22284E+03	3.08879E+02	1.59903E+02	1.07338E+02	8.22116E+01	6.65585E+01
1.22214E+03	3.10569E+02	1.58297E+02	1.08300E+02	8.23307E+11	6.65300E+01

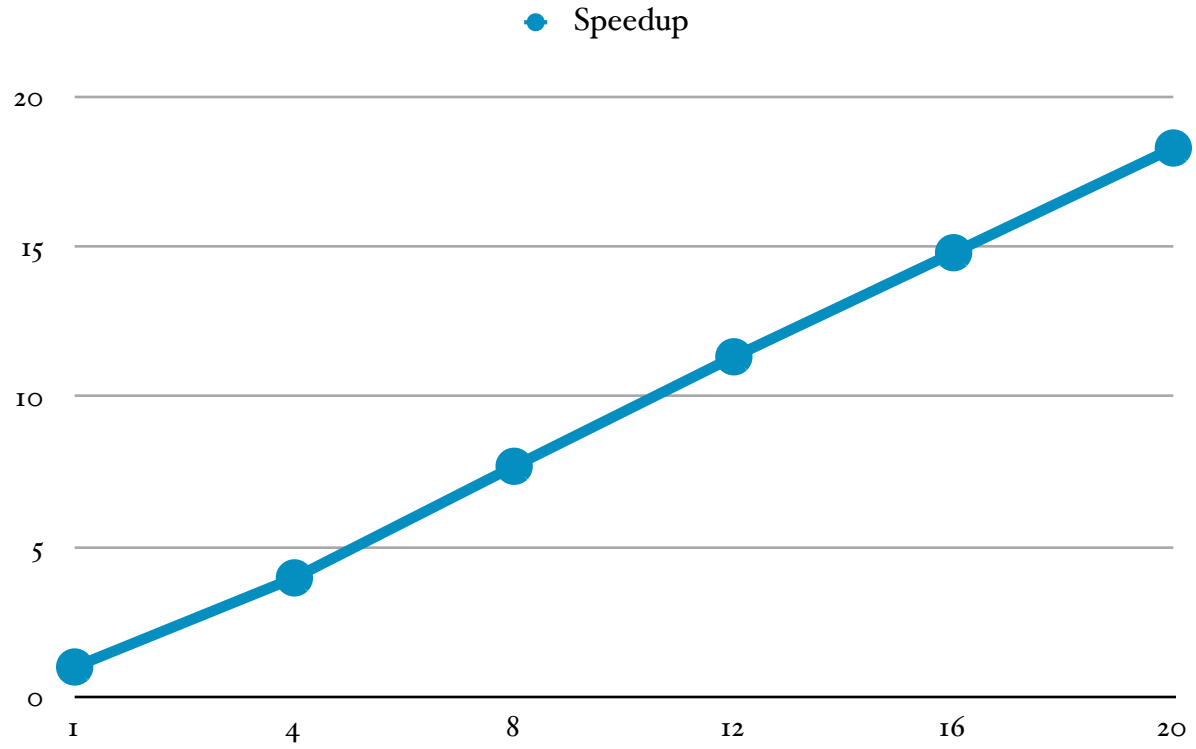
Speedup (T1/Tp)

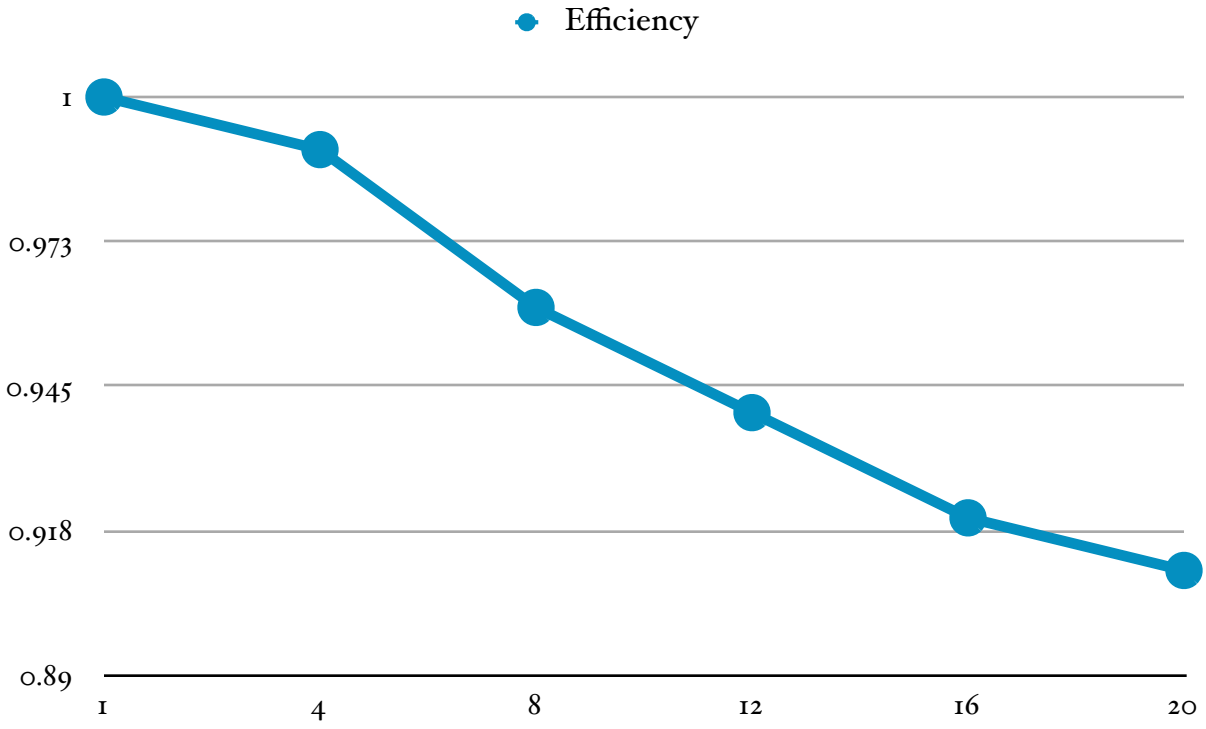
1	4	8	12	16	20
1	3.97	7.69	11.34	14.81	18.30

Efficiency (SU/P)

1	4	8	12	16	20
1	0.99	0.96	0.94	0.92	0.91

IJK Graphs


Speedup
20
15
10
5
0
1
4
8
12
16
20


Efficiency
1
0.973
0.945
0.918
0.89
1
4
8
12
16
20

IKJ Form

The IKJ form is an example of the middle product model. Turns out the ikj form was the best performing out of all of our tests. The code for our ijk multiplication looks like this.

```
for (int i = 0; i < size; i++)
{
     for (int k = 0; k < size; k++)
     {
          for (int j = 0; j < size; j++)
          {
               c[i][j] += a[i][k] * b[k][j];
          }
     }
}
```

IKJ was the best performing of the bunch. After doing some research, I found it was due to a way the compiler can optimize this loop in a loop in a loop. The compiler changes the code from the above to something that looks like this.

```
for (int k = 0; k < n; k++)
{
     int temp = A[i][k];
     for (int j = 0; j < n; j++)
     {
         C[i][j] += temp * B[k][j];
     }
 }
```

This optimization allows the program to completely skip over one of the loop, saving us a lot of computational time.

IKJ Data

Timing Runs (Seconds)

1	4	8	12	16	20
9.26241E+02	2.34521E+02	1.18406E+02	8.00417E+01	6.07568E+01	4.93065E+01
9.26176E+02	2.34293E+02	1.18250E+02	8.01649E+01	6.07593E+01	4.92400E+01
9.25950E+02	2.33881E+02	1.19644E+02	8.02538E+01	6.09332E+01	4.95411E+01
9.26329E+02	2.34285E+02	1.19120E+02	8.01224E+01	6.08892E+01	4.92870E+01
9.25931E+02	2.36002E+02	1.18438E+02	8.03888E+01	6.07422E+01	4.92614E+01

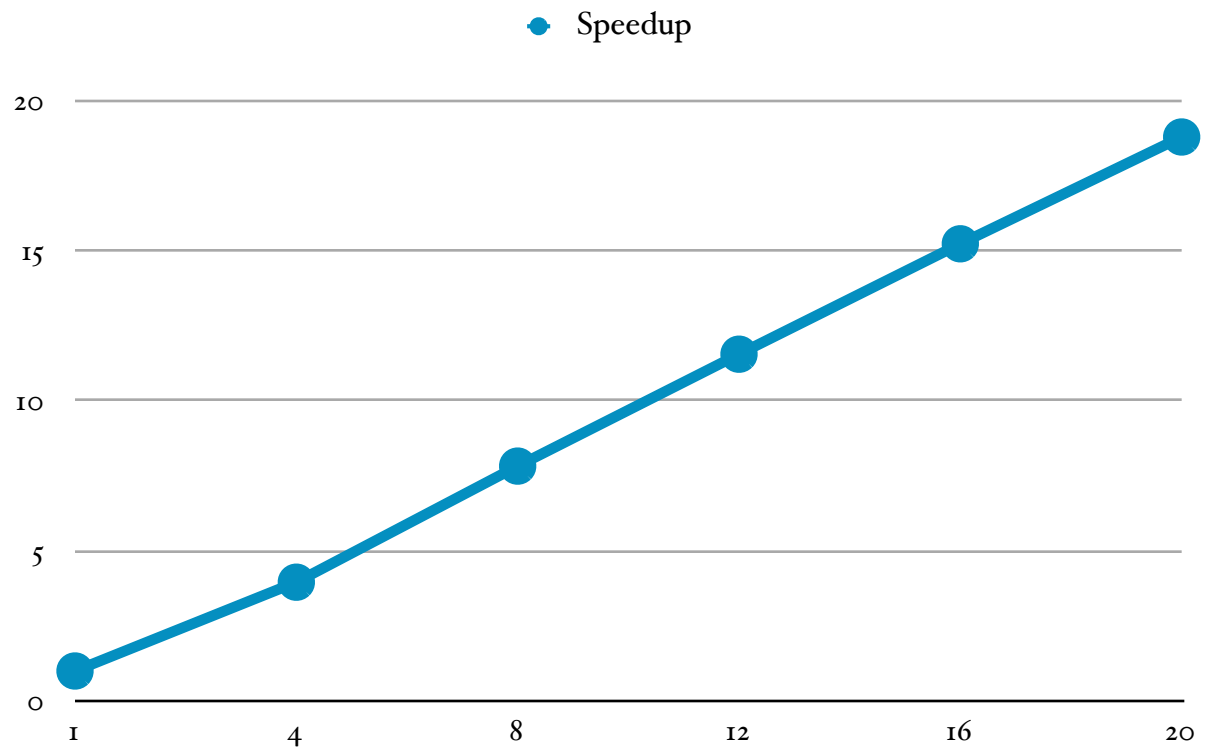
Speedup (T1/Tp)

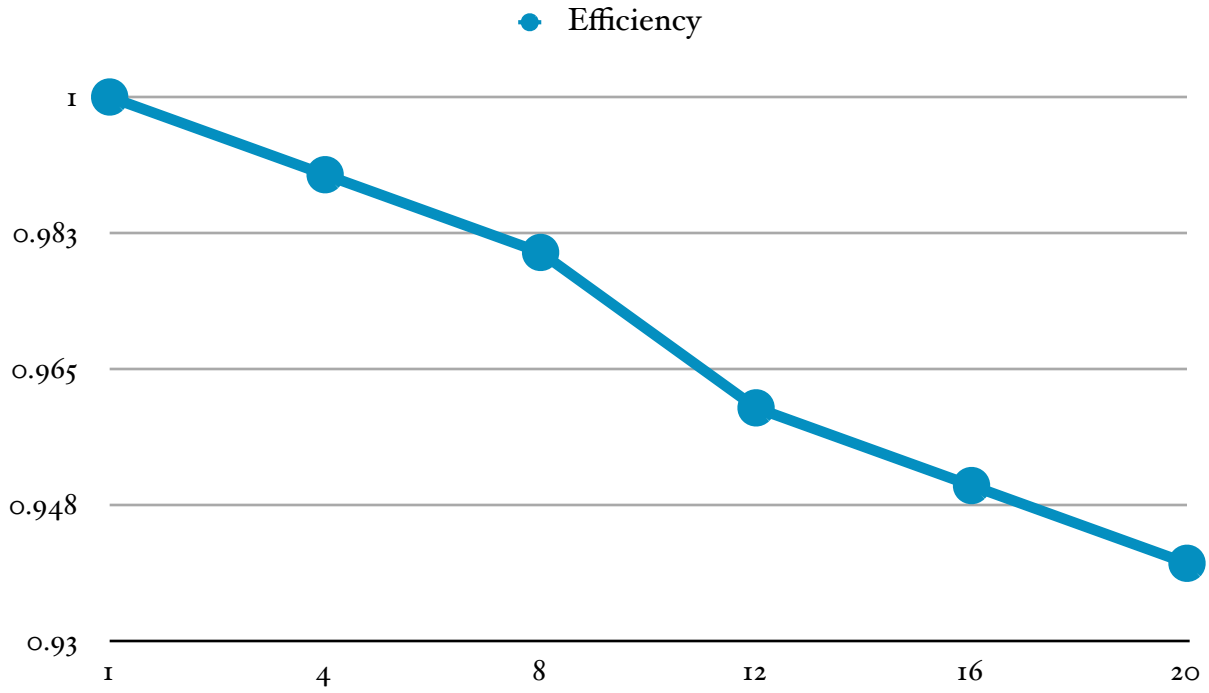
1	4	8	12	16	20
1	3.96	7.83	11.56	15.24	18.80

Efficiency (SU/P)

1	4	8	12	16	20
1	0.99	0.98	0.96	0.95	0.94

IKJ Graphs


Speedup
20
15
10
5
0
1
4
8
12
16
20


Efficiency
1
0.983
0.965
0.948
0.93
1
4
8
12
16
20

KIJ Form

The KIJ form is an example of the outer product model. Turns out the kij form was the second best performing out of all of our tests. The code for our kij multiplication looks like this.

```
for (int k = 0; k < size; k++)
{
     for (int i = 0; i < size; i++)
     {
          for (int j = 0; j < size; j++)
          {
               c[i][j] += a[i][k] * b[k][j];
          }
     }
}
```

The outer product model is also known as the dyadic product, or the tensor product. It is the equivalent of a row vector on the left multiplied by a column vector.

$$\begin{aligned}
\mathbf{a} \otimes \mathbf{b} &= \mathbf{a}\mathbf{b}^{\mathrm{T}} \\
&= \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix} \\
&= \begin{pmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n b_1 & a_n b_2 & \cdots & a_n b_n \end{pmatrix}.
\end{aligned}$$

KIJ Data

Timing Runs (Seconds)

1	4	8	12	16	20
9.29789E+02	2.35192E+02	1.19200E+02	8.05595E+01	6.05609E+01	4.89121E+01
9.30631E+02	2.35035E+02	1.19228E+02	8.05973E+01	6.06314E+01	4.88789E+01
9.30113E+02	2.35343E+02	1.28565E+02	8.07476E+01	6.06037E+01	4.95531E+01
9.30267E+02	2.37132E+02	1.20263E+02	8.06257E+01	6.05167E+01	4.88877E+01
9.30181E+02	2.35265E+02	1.19140E+02	8.07335E+01	6.07045E+01	4.95350E+01

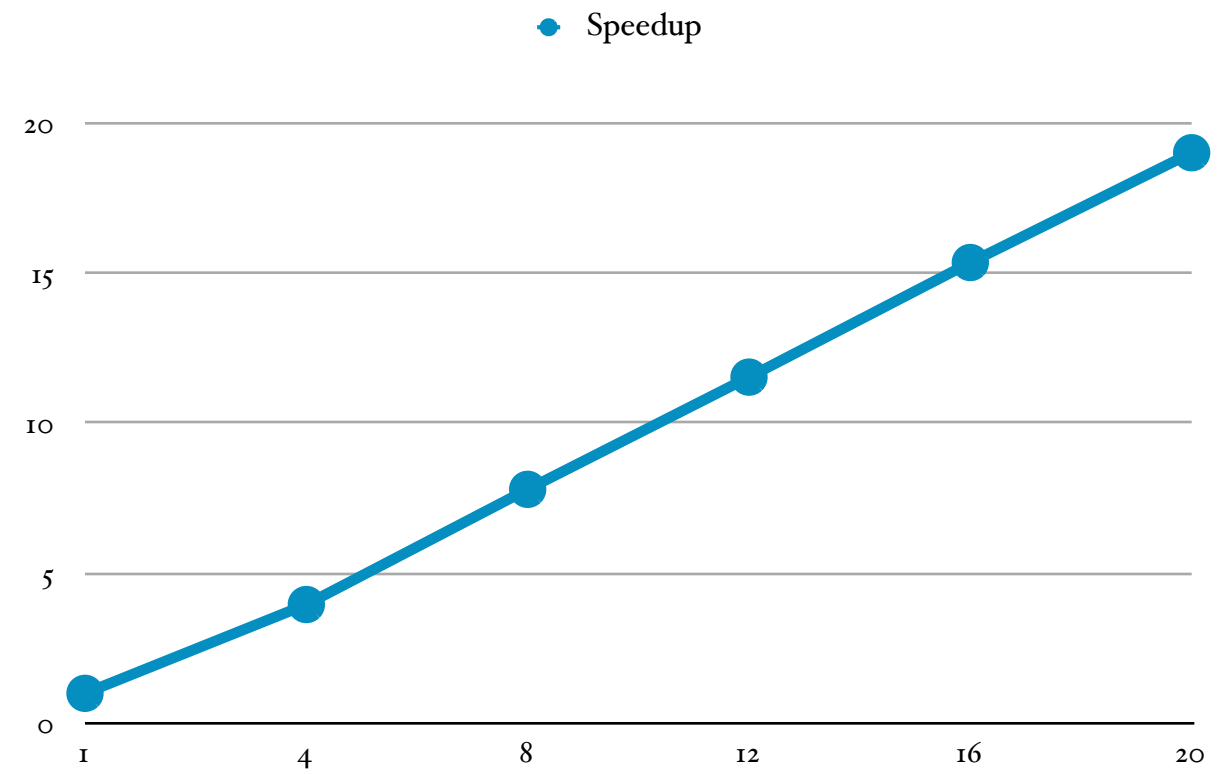
Speedup = (T1/Tp)

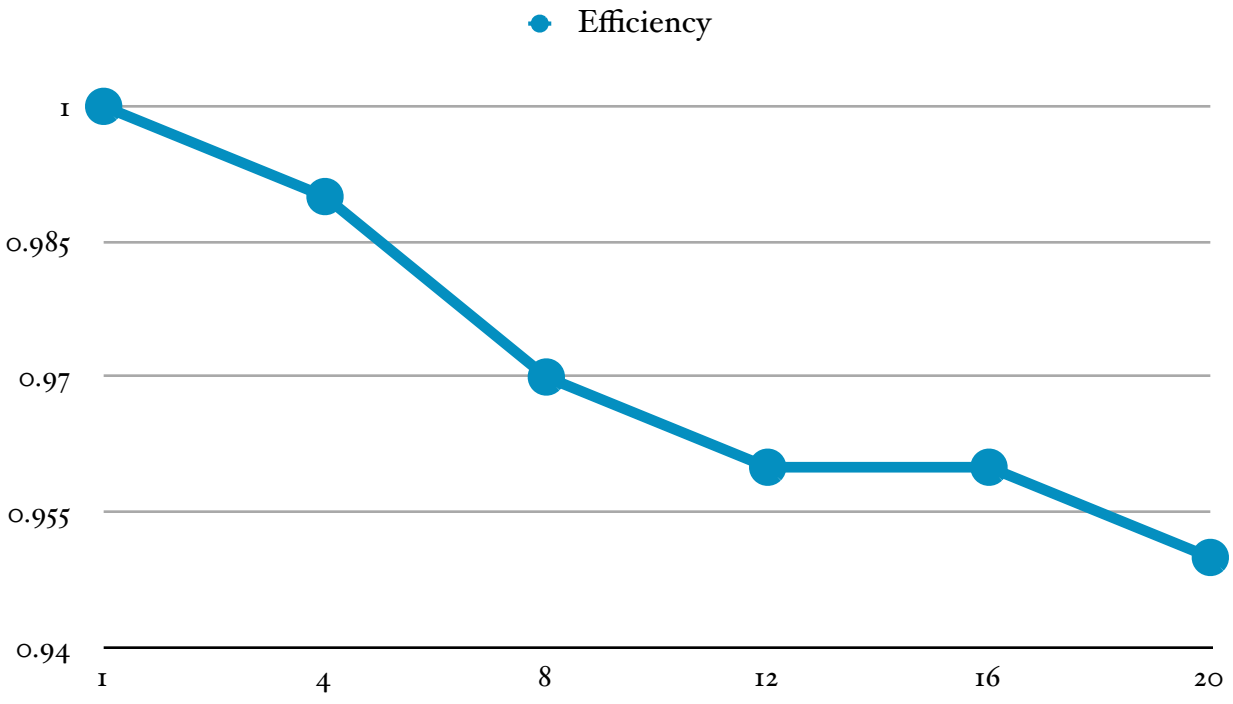
1	4	8	12	16	20
1	3.96	7.80	11.54	15.36	19.02

Efficiency (SU/P)

1	4	8	12	16	20
1	0.99	0.97	0.96	0.96	0.95

KIJ Graphs


Speedup
20
15
10
5
0
1
4
8
12
16
20


Efficiency
1
0.985
0.97
0.955
0.94
1
4
8
12
16
20

Conclusion

I had a lot of fun programming this assignment. I found a lot of motivation in trying to eek out performance and optimizations as much as I could. The first time I wrote the program, I used vectors of vectors. It worked, but for 4800x4800 size matrices, it ran really slow. I rewrote my program by allocating memory manually and initializing memory using memset over looping through it. I did notice a big bottleneck being the initialization of random matrices. By looping over a 4800x4800 matrix, we call the rand() function 23 million times. I kept trying to come up with a quicker way, but memset only works for single values, and doing bitwise operations on garbage would leave me with huge numbers. Parallelizing matrix multiplication was another good chance to practice Foster's Methodology; Partition, Communication, Aggregation, Mapping. We can see through the timing runs that adding more processors definitely sped up our program. The efficiency does drop off at a slow rate as we add more processors. This is because there will always be some bottlenecks that deal with communication and aggregation that can't be improved like calculations and data processing can.