University of Pisa
Department of Computer Science

MASTER'S DEGREE IN COMPUTER SCIENCE AND NETWORKING

Parallel And Distributed Systems:
Paradigms And Models

Academic Year: 2021/2022
Date: 12/07/2022

COMPUTING SOLUTION OF LINEAR SYSTEMS OF EQUATIONS VIA JACOBI ITERATIVE METHOD

Author
Hassan Shabir
h.shabir@studenti.unipi.it

Abstract

In this project report, we aim to analyze the sequential Jacobi Method to find independent but highly loaded functional areas in sequential code, analyzing which algorithmic skeletons can be used to implement them in parallel. In the end, we will be discussing performance gain and some overheads.

Contents

Introduction

In numerical analysis, the Jacobi method is an iterative method for solving linear systems, that is, a method that calculates the solution of a system of linear equations after a theoretically infinite number of steps. To calculate this result, the method uses a sequence $x^{(k)}$which converges towards the exact solution of the linear system and progressively calculates its values, stopping when the solution obtained is sufficiently close to the exact one. It was invented by the German mathematician Carl Jacobi. (Wiki)

This project aims to implement the Jacobi algorithm using the parallel approach to solve linear systems. The entire workflow of the project is structured in multiple phases:

1. **Sequential Phase:** Sequential Analysis and algorithm implementation is performed.

2. **Parallel phase:** Designed and modelled a possible parallel implementation of the Sequential algorithm.

3. **Evaluation Phase:** Evaluation of the model is performed and derived expected performances.

4. **Implementation Phase:** Implementation of multiple parallel versions of the Sequential algorithm using the model designed in phase 2.

5. **Testing:** Parallel versions from Phase 2 are executed using different input and some other parameters combinations to measure the model's actual performance.

6. **Conclusion** we will conclude our findings

Sequential Phase

In this phase, we performed sequential implementation of the Jacobi Method using; we can find a snippet of pseudo code.

Algorithm 1 Pseudo-code of the Jacobi method

Input:

matrixA matrix
ndim Dimension of Matrix
resultB Solution Vector B
x_old vector Initialized to 0 at the beginning

Output:

x_curr the computer solution vector

1. **Function** *Jacobi (ndim, matrixA, resultB, x_old, MaxIt)*
2. **repeat**
3. **for** *row=1, ndim* **do**
4. *x_curr(row): = resultB(row)*
5. **for** *col=1, ndim* **do**
6. **if** *row! =col* **then**
7. *x_curr(row): = x_curr(row) – matrixA (row, Col) * x_old(col)*
8. *x_curr(row): = x_curr(row)/ matrixA (row, row)*
9. **return** *x_curr any juice*
10. **Check** *Cosine similarity*
11. **Print Results**

The algorithm starts with the x_curr vector initialized with 0; then, in the first for loop, we are saving the value of resultB[row] in the x_curr[row] vector, and in the second for loop, we use the if condition to make sure we are avoiding diagonal values and takes off from x_curr all the others multiplied for the corresponding value matrixA[row][col]. Then, finally, we can divide by the diagonal value matrixA[row][row].

Parallel Design Phase

The design phase analyzes the sequential Jacobi's algorithm to produce a parallel model, but before starting analysis, let's suppose that matrix size = M, Iterations = K, row = I, and columns = j.

Analysis Starting from the sequential code, we can observe that the sources of parallelism are:

Line 3, the iterations of this for loop are independent. Because the element $x\_curr_i^k$ is calculated using only the row i of the iteration matrix and the previous solution $x^{(k-1)}$ .We do not need to parallelize other for loops because they are too fine-grained a compared to the loop in line 3; moreover, the ost of the loops is also negligible compared to the loop in line 3. Suppose a matrix of size M then on line 3 it will have a complexity of M^2 while other loops have a complexity of M.

Hence, we need to parallelize loop in line3, which means we need to assign this computation

to different cores/processors so that they can run in parallel.

Modelling: operations performed by the sequential code are as below

Read

Here we generate a linear system matrixA, known vector resultB and a solution vector x_curr, and store them in Ram.

init

Initializing Solution vector x_curr (line3)

Calculate

Computing Solution vector (line 7, line 8)

Verify

Checking Cosine Smimlarity of x_curr with our randomly created vector X of variables that are used generate resultB.

Let's Define a function f[x] with x set of instructions and T (f[x]) the function that returns the time that is required to execute f with parameter x. Define also init that contains all the initialization operations (in this case only allocate).

We can model the total time required by the sequential algorithm as below

$$T(jacobi[k,m]) = T(Read[m])\}\, T(Init[m]) + k\,\{\, T(Calculate[m]) + T(verfiy[m])\} \qquad \text{(i)}$$

Where k is the number of iterations and m is the size of a matrix as defined above.

Now let's suppose that calculate operation is parallelized using n workers.
Hence three other operations are introduced to run this operation in parallel:

- **Split**: Dividing original data into chunks and assigning a set of rows to each worker.
- **Spawn**: Spawn operation is performed to create corresponding threads.
- **Collect**: the collect operation will gather all the results and reconstruct the solution vector.

Algorithm 2 Pseudo-code of Workers

Input:

Id: Assigning number to each worker

Output:

Update: updating vector x_old with x_curr

1. **Function** *Compute Unknowns (id)*
2. **for** *row=id*, row<=*ndim, row+=nworker's* **do**
3. *x_curr(row): = resultB(row)*
4. **for** *col=1, ndim* **do**
5. **if** *row! =col* **then**
6. *x_curr(row): = x_curr(row) – matrixA (row, Col) * x_old(col)*
7. *x_curr(row): = x_curr(row)/ matrixA (row, row)*
8. **sync**

A single worker is performing the following operations:

Calculate: to compute the solution.

Sync: waiting for all workers to finish their job.

The init phase is not containing alloc operation only, but it also has split and spawn functions. In this case, total time can be modelled as

$$T(jacobi[k,m,n]) = T(Read[m]) + T(init[m,n]) + k\left[\frac{T(Clculate[m])}{n} + T(collect[m])\right] \quad (ii)$$

From the convention we are following, k is the number of iterations, and n is the number of workers. T is a constant time operation. Let's assume that we are in shared memory architecture, so T(collect) becomes negligible so let's not consider this in the further calculation. Now Substituting T(init) and T(calculate) with the worker code, the formula becomes

$$T(jacobi[k,m,n]) = T(Read[m]) + T(aloc[m,n]) + T(split[m,n]) + T(spawn[m,n]) + +k\left[\frac{T(Calculate[m])}{n} + T(sync[m])\right] \quad (iii)$$

Evaluation Phase

In order to evaluate this phase, we have taken two assumptions.

1. We are not including reading operations in our evaluation because they are common to all of Jacobi's implementations.
2. The input data is of size M x M always.

Sequential

Referring the formula 1 let's analyze the complexity of each part of the algorithm.

- Calc: performs a matrix-vector multiplication hence the complexity is $O(M^2)$.
- Other operations are performed with the complexity of $O(M)$

Parallel

To evaluate the performance of our Parallel Model, we used the following metrics:

Speedup s(n)

it is the ratio between the best sequential time and the parallel execution time.

$$\boldsymbol{S(n)} = \frac{T(Seq)}{T_{Par}(n)} \quad (iv)$$

Scalability Scalab(n)

The ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to n.

$$\boldsymbol{Scalab(n)} = \frac{T_{Par}(1)}{T_{Par}(n)} \quad (v)$$

Efficiency (ϵ)

The ratio between the ideal execution time and actual execution time.

$$\in (\boldsymbol{n}) = \frac{T_{Seq}}{n\,.T_{par}(n)} \quad (vi)$$

We start our analysis with formula (iii), in which we modeled our parallel version of the algorithm. We can neglect split operation because it is a constant time operation which will be executed only one time. After, this we are left with calc, and sync time. Let's try to analyze them.

- **Calc**: Here, while performing calculate operation, we can see that we are dividing our matrix of size M by the number of workers n, so the smallest amount of work that a worker can perform is just row by column multiplication.
- **Sync**: The evaluation of the synchronization time depends on the underlying architecture and the number of workers.

The main overhead that will be introduced when we are parallelizing algorithms on multi-core shared memory systems is due to synchronization overhead. There are two possibilities

1. It can grow up linear to the number of workers
2. It can grow up faster than the number of workers.

It is not very useful to consider only synchronization time. Still, it is more interesting when we also consider the calculation time that workers spend while performing a given task. Because if synchronization time is very large but computation time of workers is also very long, then synchronization is not an overhead. If computation time is not too long, then synchronization time is a bottleneck we need to consider.

To have a deeper understating of this problem, let's assume some projections,
That our sequential code is taking 20000 µsec in computing solution of matrix size M = 10000 with max iteration K =1, and each thread in my architecture takes about 50µsec from start to join, and we have a total number of works NW= 100.With these assumptions, we can calculate the amount of overhead that we can have

$$\emptyset(overhead) = \left(T_{start} + T_{join}\right) \times NW = 50\mu\,\text{sec} \times 100 = 5000\mu sec$$

If we don't consider this overhead, then ideal Service time will be

$$T_{Ideal} = \frac{T_{Seq}}{NW} = \frac{20000}{100} = 200$$

Now we will add our projected overhead into our computations to see the maximum speed up that we will get with 100 works.

$$\Omega\big(S(n)\big) = \frac{T_{Seq}}{T_{Ideal\,+\,\emptyset}} = \frac{20000}{200 + 5000} \approx 3.84$$

Why is this the maximum speed up that we can get?

Because we are only considering thread management here, other parameters such as cache miss, waiting for some synchronizations, scheduling, and other overheads are not considered.

Implementation Phase

For the parallel implementation, we have two approaches

1. Stream parallel Approach
2. Data parallel Approach

Approach 1 is mainly used when we have data for computation coming from input steam; with the help of this approach, we improve speed up or decrease service time. Some stream parallel approaches such as (Pipelines and Farms) can be used for this purpose.
Approach 2 is used when data is readily available for computation. With the help of a data-parallel approach, we mainly optimize completion time. Some data-parallel approaches such as (Map, Reduce, and Stencils) can use for this purpose.

According to our problem there can be many ways for implementation such as Emitter Worker and Collector.

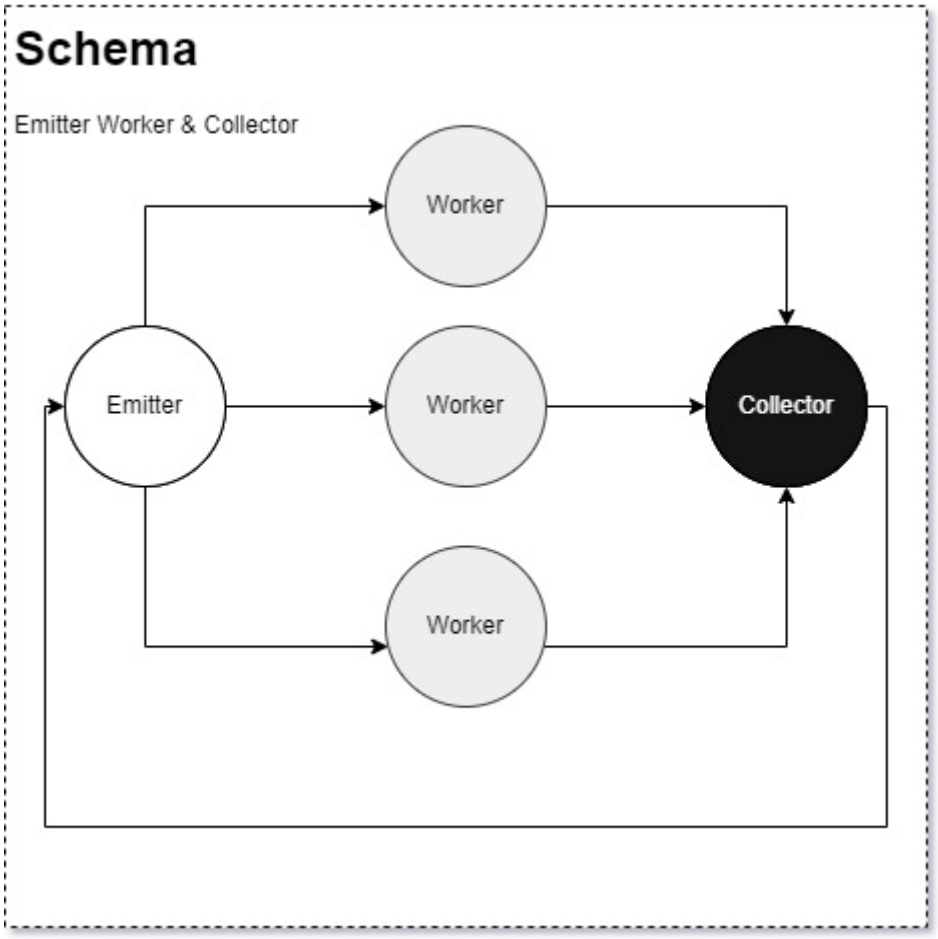

Figure 1 Emitter Worker and Collector Schema

Let's assume that every worker is getting chunk of a Matrix to compute and send results to collector which is responsible for gathering results from all workers and send results back to emitter then emitter update temporal vector and send results back to worker for next iteration until convergence.

This scheme will not be an efficient implementation because emitter need to wait for synchronization if we are able to do such computation where synchronization is not required such that works can work on partially updated data and converge to a solution then we can consider this methodology. In our case we need to wait for whole works to complete their computation.

It is recommended to use both approaches (Stream and Data Parallel) together to achieve the best performance

Now coming back to our problem of implementing linear system of Equations by Jacobi Method. We already have discussed that we have matrix of size M x M (means data for computation is available from beginning) and the computation of element $x\_curr_i^k$ is calculated using only the row i of the iteration matrix and the previous solution $x^{(k-1)}$ which in turns to be a stencil operation. Hence, we decided to perform stencil of composition and dividing data on all works in this way we can divide highly loaded area of computation i.e., line 7 of algorithm 1 among the number of workers to compute in parallel but as we know Jacobi is an iterative method so before moving to next iteration master need to perform synchronizations (i.e. wait for all thread for compete their operation then master will update temporal vector) as this phase is very important to get correct results from Jacobi so any parallel approach that we will follow have to go through this synchronization. To minimize this synchronization overhead we opted strategy i.e., start threads together so they can finish task together.

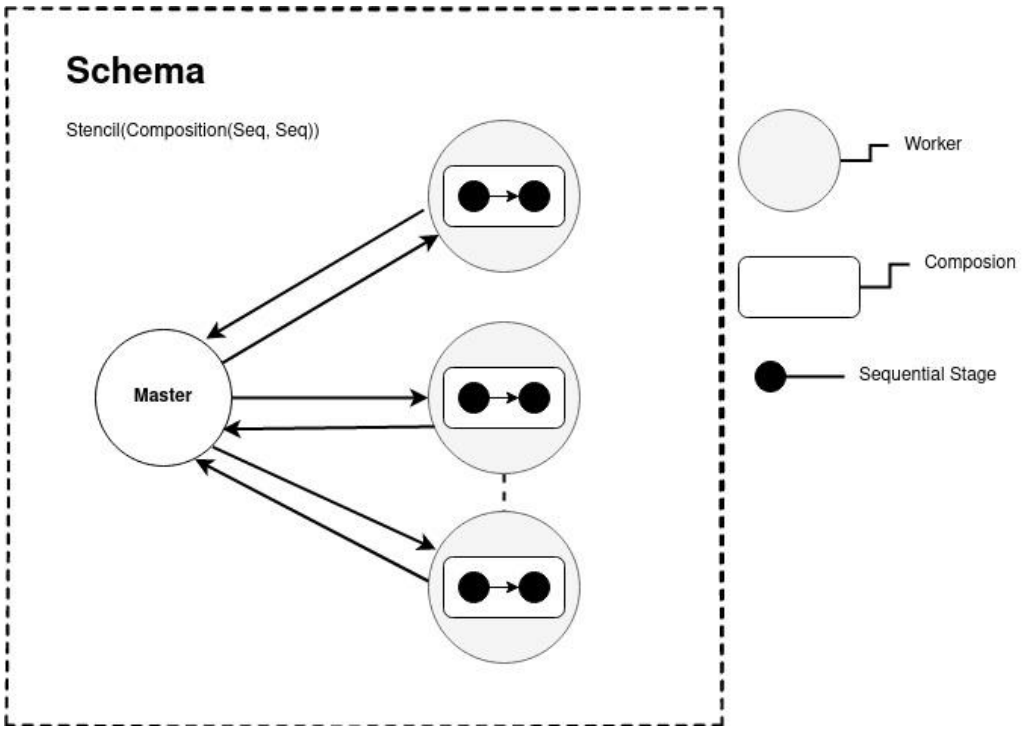

Figure 2 The final schema adopted, Stencil (Composition (Seq, Seq))

Std: Thread based Implementation

During the implementation of this project with Std: library, we took advantage of std: barrier class by providing an update () function as a waiting condition with number of workers and let barrier class take care of synchronization. Each thread that will be used for computing Jacobi function in parallel will start waiting after reaching std: barrier. arrive_and_wait () call. we also noticed that it is taking less time to perform synchronization as compared to very large vector computation. Hence, we can say this synchronization overhead is almost negligible. We performed static distribution of work among workers specifically we used a cyclic approach, but chunk-based distribution policy can also be used.

Fastflow based implementation

We let Fastflow library to handle thread synchronization and parallel implementation is performed by means a parallel for skeleton which parallelizes the matrix-vector multiplication present in line 7 of algorithm 1.

Testing Phase

All test experiments are performed on Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz 32 cores - 4 hyper threading machine therefore 32 x 4 = 128 threads.

Results

Results that we obtained from parallel implementation is very interesting we observed that matrix size (Dimension) plays very important role. we kept number of iterations same for different matrix sizes and observed behavior of our parallel implementation. Of course, if we increase or decrease this parameter, we will observe different speedup results as number of iterations are increasing means there will be more synchronization overhead among workers and vice versa.
We selected three different matrix sizes 1024x1024 | 4096x4096 | 16384x16384 and Max Iterations 120 and observed following results.

		Minimum Completion Time Of	
Matrix Size	Average Sequential Time in	Thread with Workers	Fastflow With Workers
1024	197144 µSec	34263 µSec | 19	35291 µSec | 34
4096	3243467 µSec	281324 µSec | 16	244952 µSec| 45
16384	51568417 µSec	2591842 µSec | 26	2454755 µSec |89

Table 1 Speed up gained Against Sequential Time

1st observation Completion time

- Case 1: matrix Size 1024

  We achieved Minimum completion time of 34263 µSec with 19 workers with Thread-based implementation and 35291 µSec with 34 workers with Fastflow-based implementation against Average the Sequential time of 201284 µSec with 1 worker.

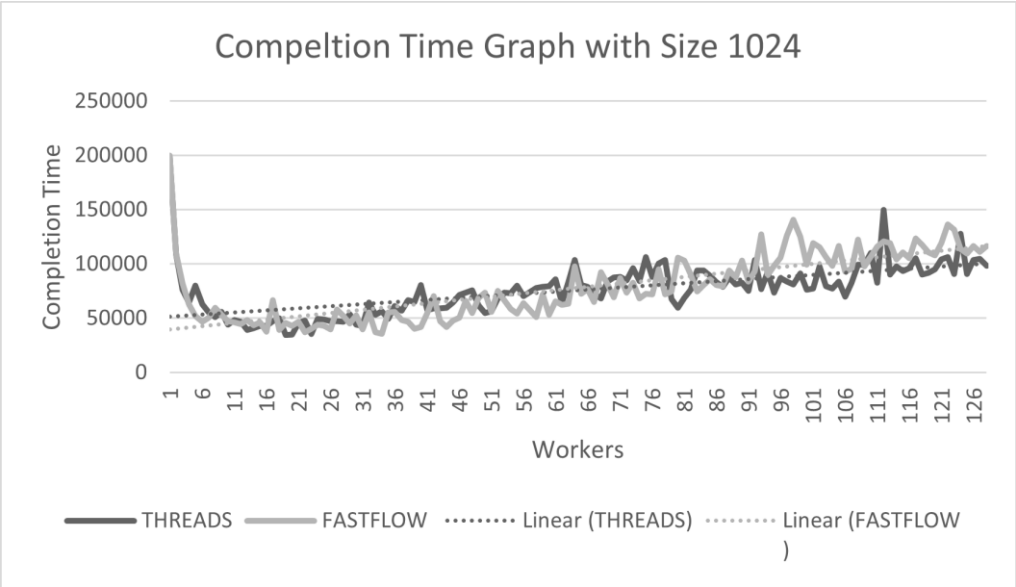

Figure 3 Completion Time with Matrix Size 1024

From the above figure can observe that the problem size is very low, so as we keep increasing number of workers computation time will keep decreasing but thread synchronization overhead overcomes computation time of workers that's why our trend line in graph goes up after reaching minimum completion time.

- Case2: matrix Size 4096

We achieved Minimum completion time of 281324 µSec with 16 workers with Thread-based implementation and 244952 µSec with 45 workers with Fastflow-based implementation against the Average Sequential time of 3243467 µSec with 1 worker

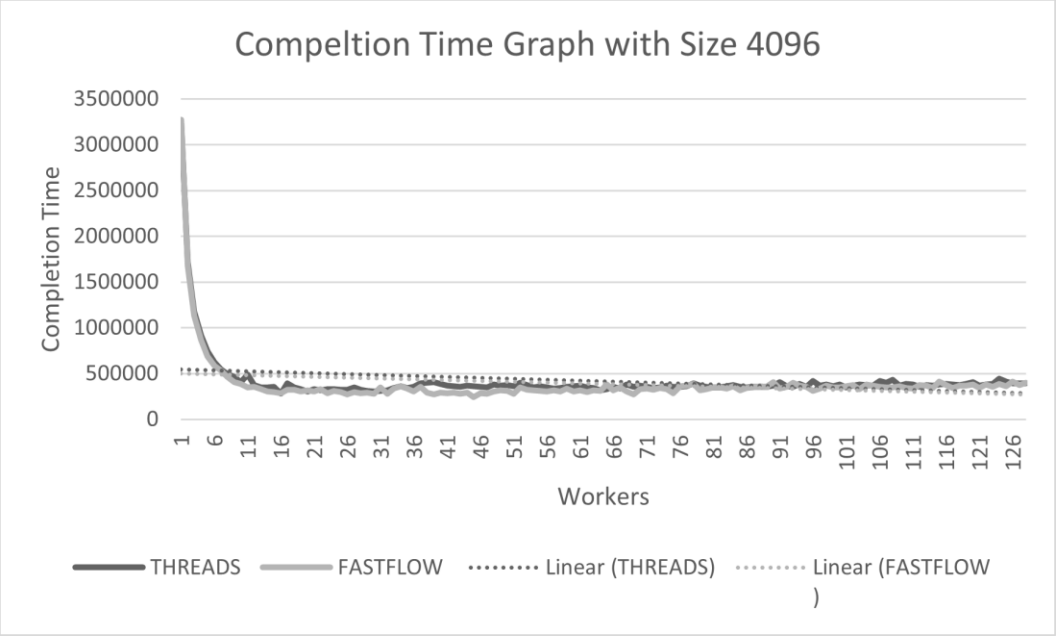

Figure 4 Completion Time with Matrix Size 4096

Now this result is very interesting to observe as we increased problem size with 1024 to 4096, now every worker will have more task to computer in parallel synchronization overhead is not quick as in first case but after reaching to a point where we are getting minimum computation time (16 workers with threads and 45 workers with Fastflow) again synchronization overhead will overcome

- Case3: matrix Size 4096

We achieved Minimum completion time of 2591842 µSec with 26 workers with Threads-based implementation and 2454755 µSec with 89 workers with Fastflow-based implementation against Average Sequential time of 51568417 µSec with 1 worker

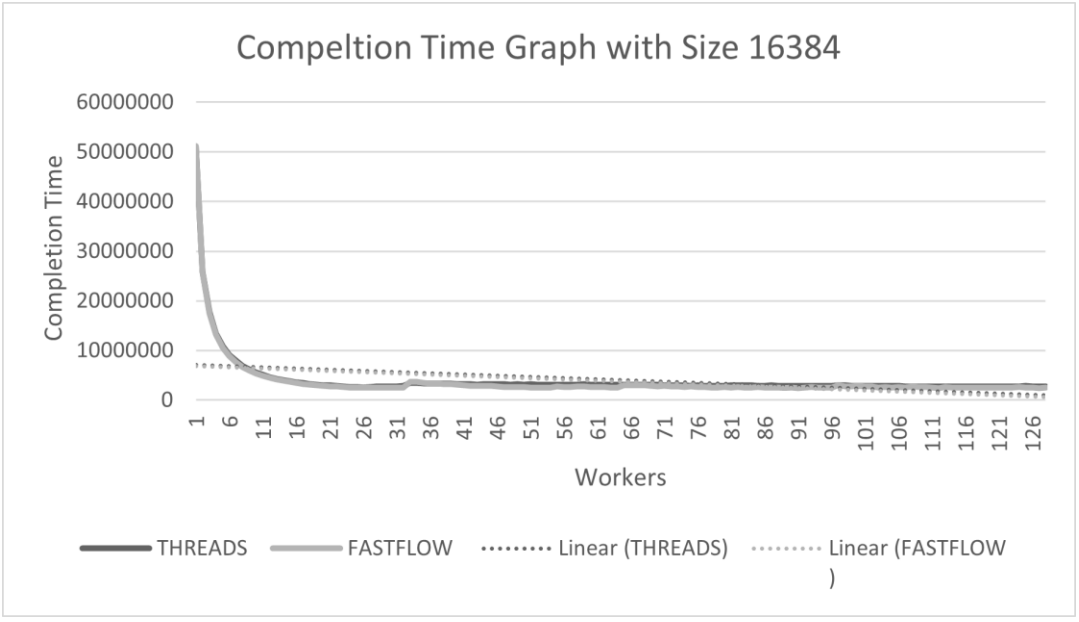

Figure 5 Completion Time with Matrix Size 16384

From results of figure3, we can say that as we increase our problem size and then assign this problem to number of workers, we are getting more speedup but on the other hand efficiency keep on decreasing. Hence selection on right parallelism degree depends on our usage of applications.

After this analysis let's see graphs of other performance measures such as Speed up, Scalability and Efficiency that we obtained

Speedup

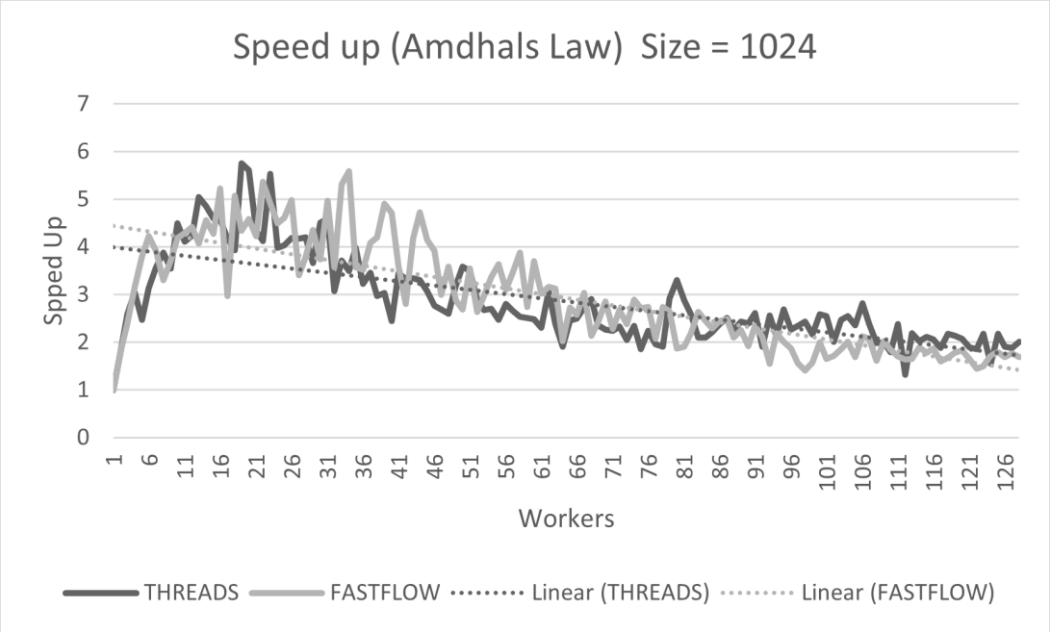

Figure 6 Speedup with Size 1024

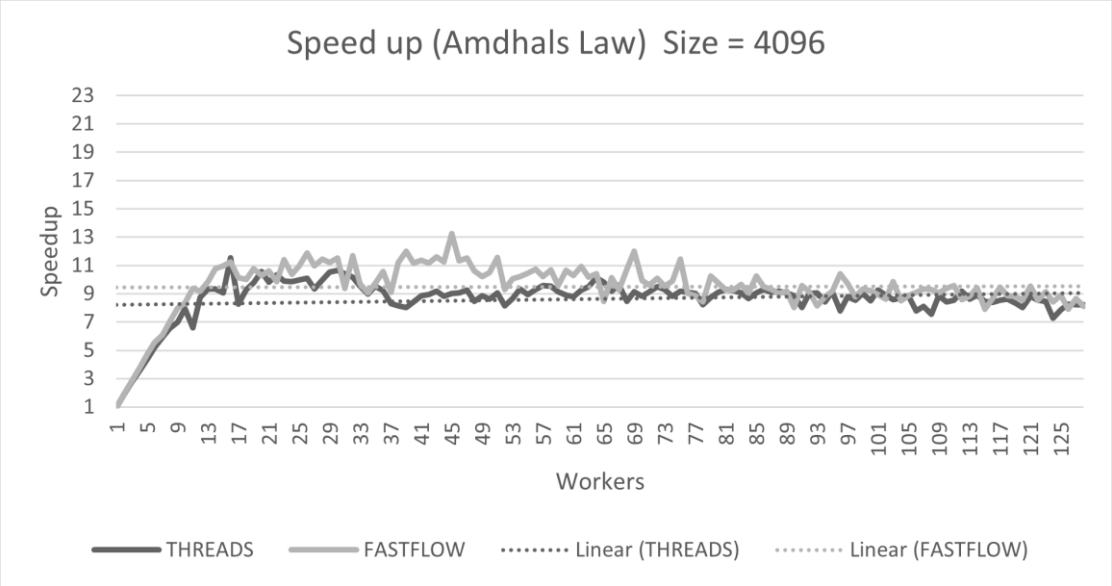

Figure 7 Speedup with Size 4096

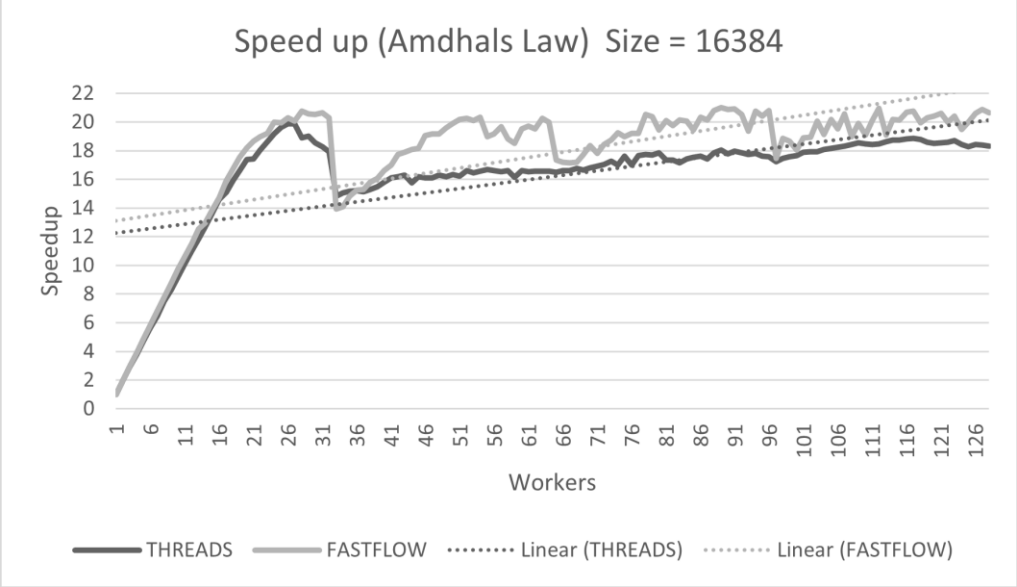

Figure 8 Speedup with Size 16384

Scalability

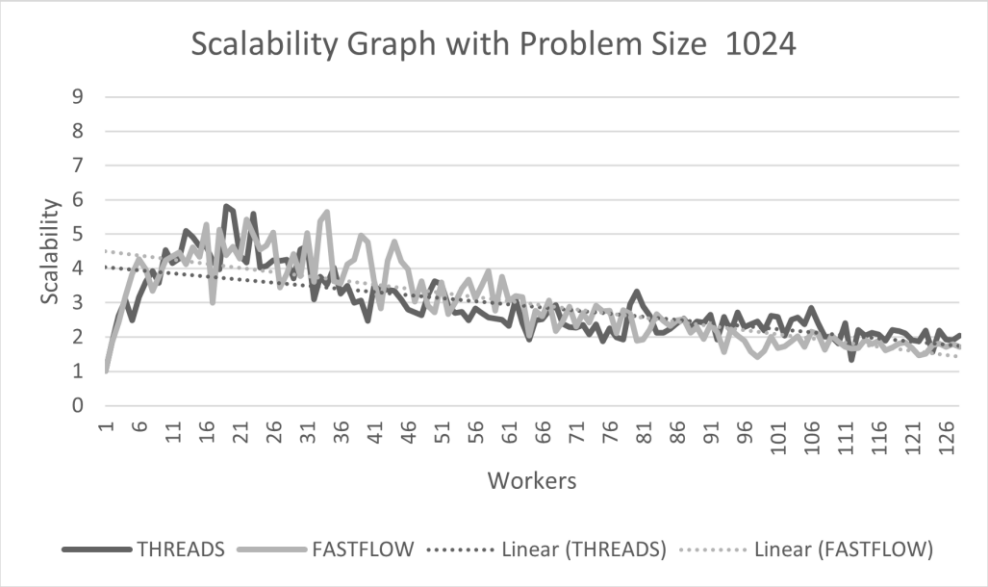

Figure 9 Scalability with Size 1024

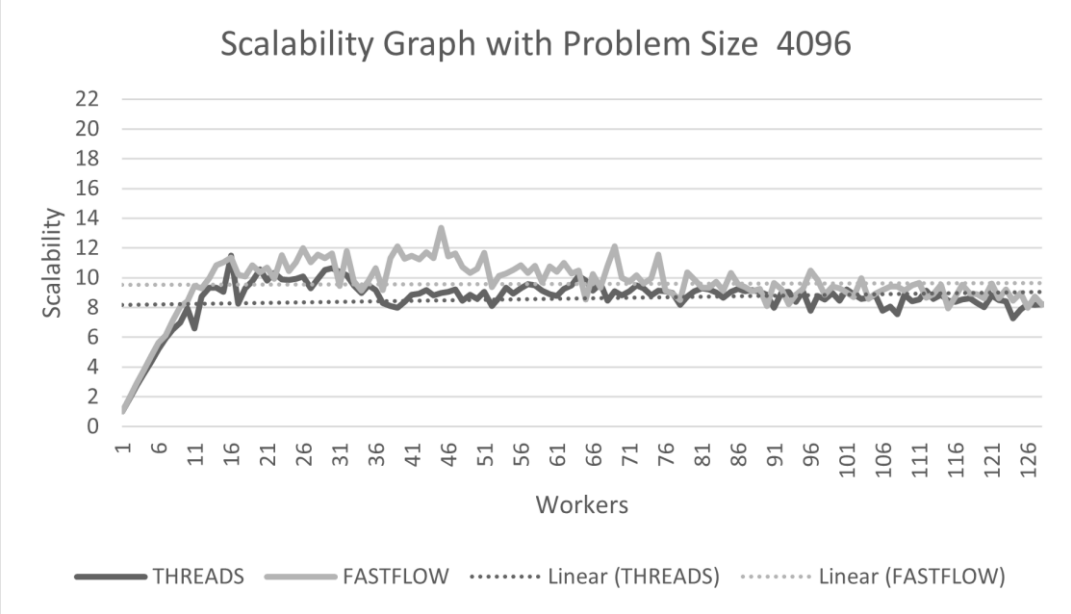

Figure 10 Scalability with Size 4096

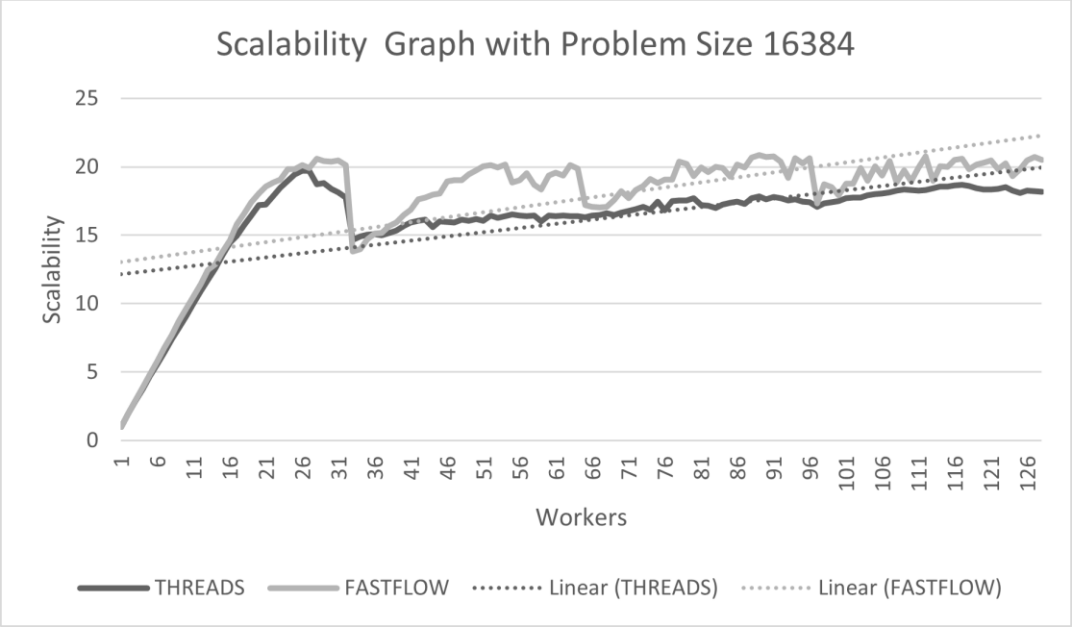

Figure 11 Scalability with Size 16384

Efficiency

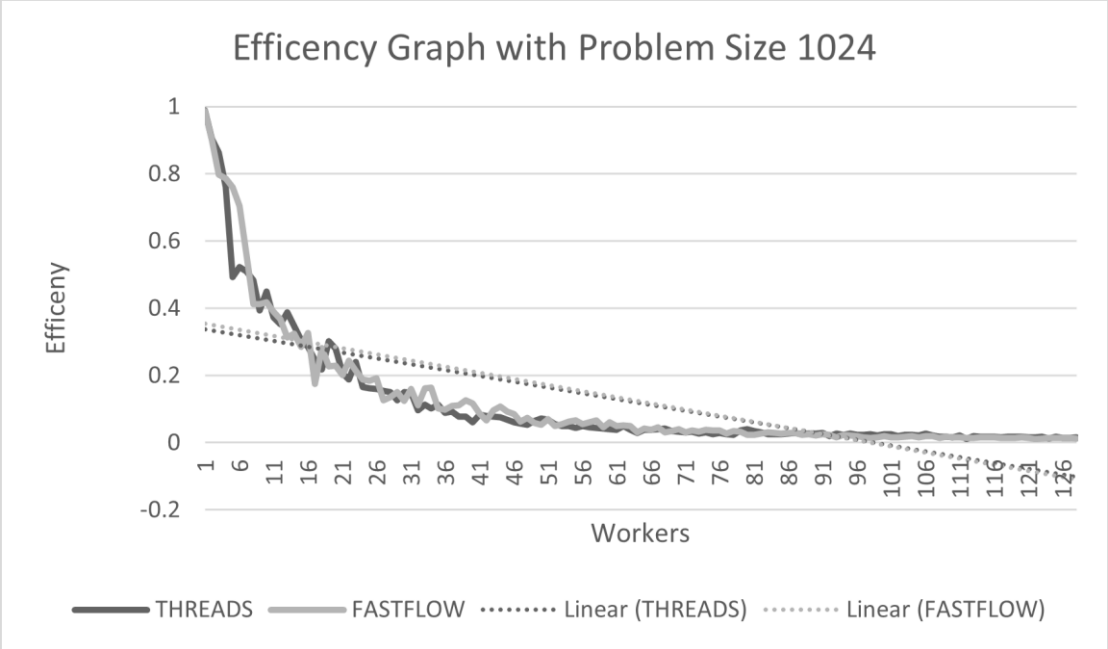

Figure 12 Efficiency with Size 1024

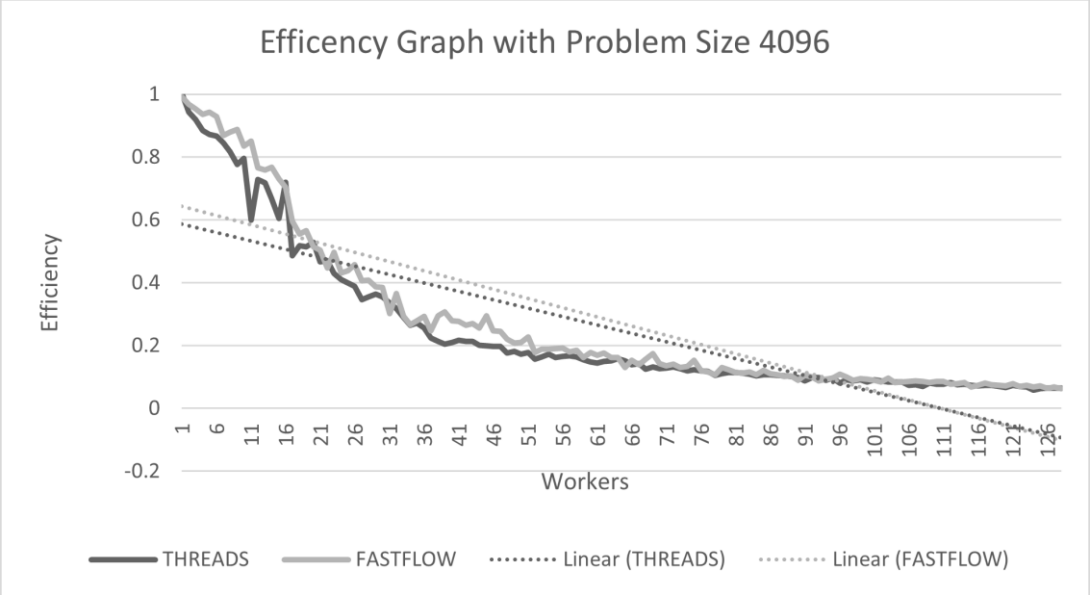

Figure 13 Efficiency with Size 4096

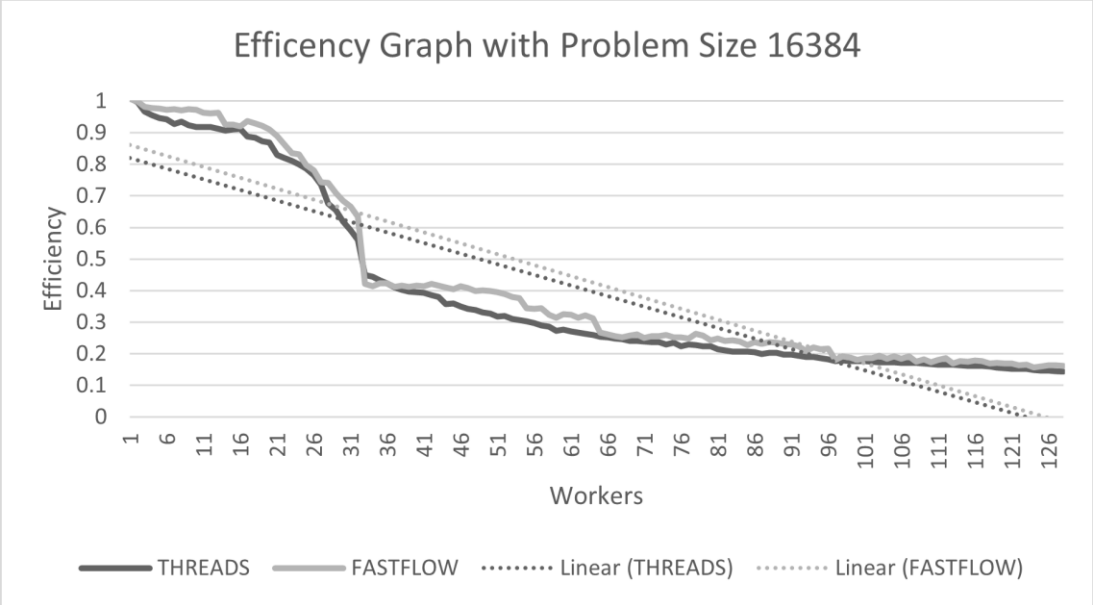

Figure 14 Efficiency with Size 16384

Comparison

Size	Implementation	Speedup	Scalability
1024	Thread	**5.753773**	**5.8177798**
	Fastflow	5.58617	5.655738
4096	Thread	11.5292	11.50638
	Fastflow	**13.2412**	**13.3581**
16384	Thread	19.8964	19.7086
	Fastflow	**21.0075**	**20.853**

Table 2 Show best achieved results from Both Implementation

Conclusion

To conclude our finding, we say that the performances we measured during the evaluation phase are very close to those we obtained from our implementations (thread-based and fast flow-based) for a large enough matrix size. On the other hand, we also observed that we are not getting good performance by using a small matrix because of synchronization overhead. Both implementations show promising results, but the performance of Fastflow is also very good in thread utilization. With a matrix size of 16384 with threads-based implementation, initialization time dominates after 26 workers impacting performance and scalability. In contrast, Fastflow-based implementation shows very good behavior with 89 workers.