Analysis of OpenMP Performance on Sparse Matrix - Vector Multiplcation

Himanshu Mundhra | 16CS10057

4th Year Undergraduate
Department of Computer Science and Engineering
Indian Institute of Technology - Kharagpur, India

INTRODUCTION

Matrix Multiplication is a common operation in multiple fields of applied Mathematics as well as Computer Science. From Linear Algebra to Machine Learning, matrices are an easy way of representing coefficients in equations and vectors and matrix operations provide a cleaner and efficient way to perform mathematical operations on multiple vectors or a set of equations together.

An **(k x m) matrix A** and an **(m x n) matrix B** are said to be compatible with each other with respect to matrix multiplication, and produce an **(k x n) matrix C**. Each element of the resultant matrix **Cij** is the dot-product of the row **Ai** and the column **Bj**. This tells us that each element of the (k x n) matrix C can be computed in O(m) time. Therefore the **Matrix Multiplication algorithm takes O(m * (k x n)) time**.

Another thing to notice here is that the value of each element Cij depends solely on the row Ai and the column Bj. So if we had **(k x n) processing elements**, we could calculate the value at each position in parallel and hence our algorithm would take **effectively only O(m) time.** It is this scope of parallelism that we employ in this assignment.

A special form of matrix multiplication is a matrix - vector multiplication where B, as defined above is a vector, more precisely a column vector with the second dimension being 1. So we can redefine B to be an **(m x 1) vector X** such that Matrix Multiplication algorithm serially takes **O(m * k) time** and in the parallel model, takes **k processing elements**.

In this assignment, we deal with special kinds of matrices called **sparse matrices**. Sparse matrices are those matrices where many of the elements have a value of zero. The representation of these matrices can be changed in a way where we can exploit the scarceness of the **Non-Zero values**. This enables us to optimise memory usage and also optimise the time complexity of the operations which can be performed on a matrix.

We aim to use this modified representation of sparse matrices and perform a **parallel matrix-vector multiplication** using OpenMP Wrapper Library and analyse the variation of SpeedUp and Execution time with the *number of threads*, for different *scheduling algorithms* and different *chunk sizes*.

We also try to qualitatively analyse the type of matrices given to us as the input data by observing the patterns obtained and predict what kind of a scheduling algorithm might be the best for that particular type of matrix.

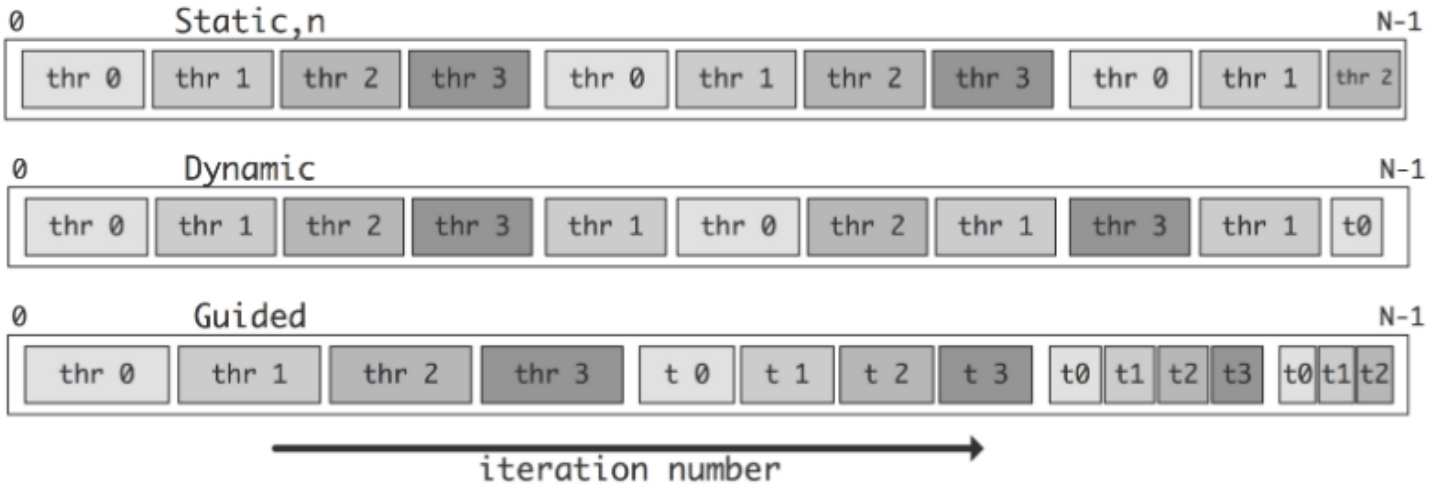
OPENMP SCHEDULING

OpenMP provides us with a framework to manually determine what kind of scheduling is followed, i.e. in what way are the threads assigned with work. We can do this with the help of the schedule clause. When applied to a loop directive, the iterations of the loop are distributed amongst the team of threads according to the parameters specified in the **schedule**(***schedule_type***, *chunk_size*) clause.

schedule(***static***, *chunk_size*) - When static scheduling is specified, the iteration-space is divided into chunks of size *chunk_size* and **assigned to the team of threads** (created by a parallel directive) in a **round robin fashion**. In a way we can say that even before the threads start executing, the chunk(s) of iterations which they need to perform is known.

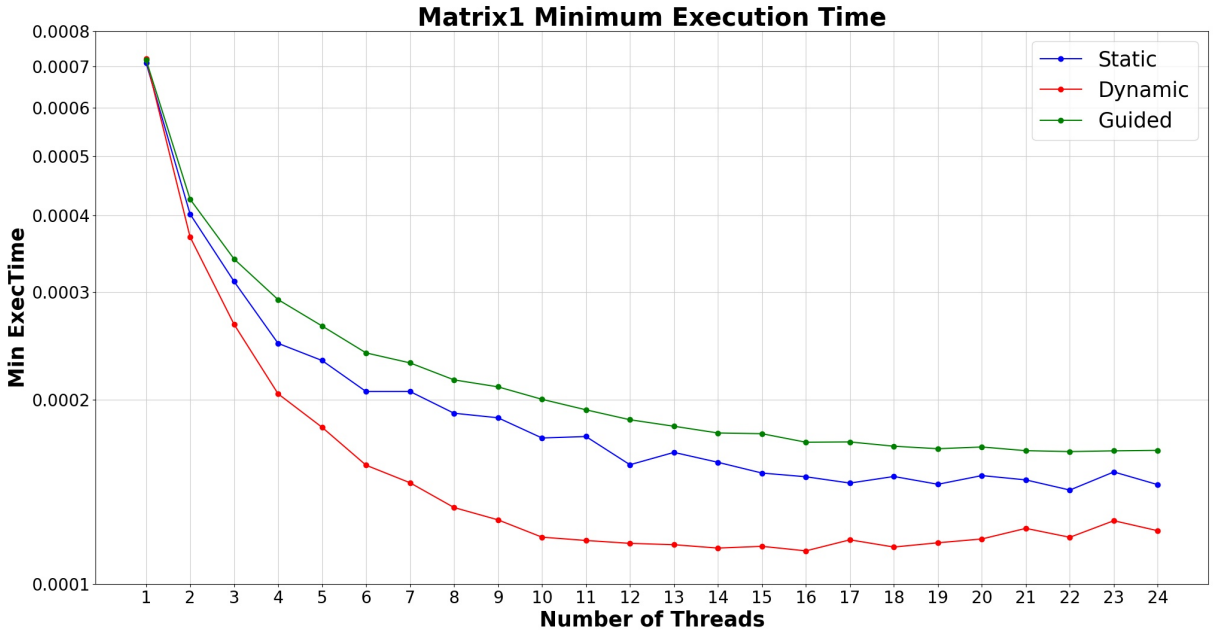
schedule(***dynamic***, *chunk_size*) - When dynamic scheduling is specified, the iteration-space is divided into chunks of size *chunk_size* and initially **distributed to each thread** of the team. As and when a thread finishes processing, it **requests for another chunk** and the chunk is then **assigned to it from a queue** which the system maintains and updates as and when required.

schedule(***guided***, *chunk_size*) - When guided scheduling is specified, the iteration-spaces assigned to the team of threads is in chunk sizes **proportional to the number of unprocessed iterations**. This value never reduces below *chunk_size*.



EXPERIMENT AND ANALYSIS

The Sparse Matrix - Vector Multiplication was performed on matrices mat1, mat2, mat3 and mat4 with varying parameters like *num_threads*, *scheduling_type* and *chunk_size* with a **Sampling of 10000** and on a Server with **$(nproc) = 48**. Interesting patterns were observed.

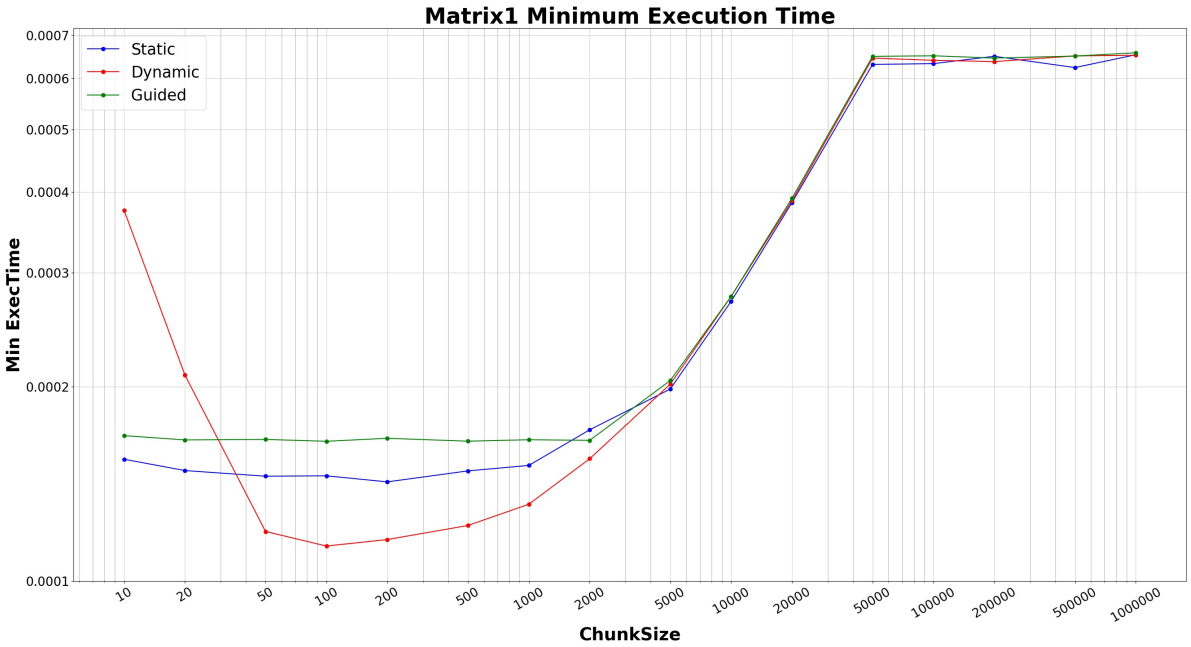


For each team of threads with *num_threads* runners, we vary *chunk_size* and apply each of the three scheduling methods to record the time taken to execute the algorithm.

We observe that an increase in *num_threads* leads to a steep and steady exponential decay in the minimum execution time. This can be attributed to the fact that more number of iterations are being carried out in parallel and hence the time taken for the algorithm to reach completion is lesser.

At times there is an increase or a spike in execution time with an increase in *num_threads*. This maybe due to the fact that sometimes a certain distribution of the iteration space amongst the threads is such that the thread with comparatively more time expensive iterations is not scheduled properly on the processing element which the system assigns to it. Remember the OpenMP Schedule clause simply assigns iterations to threads, how these threads are executed is still the jurisdiction of the operating system.

Moreover after a certain number of threads, the time taken for communication while accessing the shared variables starts dominating, as the inter-core communication time is greater by multiple orders of 10 as opposed to the per unit computation time.



For each *chunk_size*, we vary the *num_threads* and apply each of the three *scheduling_type* to record the time taken to execute the algorithm.
For **Dynamic Scheduling**, the execution time is pretty high in the beginning, decreases steeply and then increases steeply, after which it saturates. For **Static and Guided Scheduling**, the execution time is pretty much constant in the beginning and then increases steeply, after which it saturates.

Dynamic Scheduling is the scheduling which provides the best efficiency in executing these parallel programs. How does it go about doing so? The iteration scheduler maintains a queue where the unprocessed chunks are present. Whenever a thread demands for another chunk, one of the chunks from the queue is assigned to that thread. This way the threads which become available earlier are not kept idle and are given another chunk to process. Since this waiting time is minimised, this scheduling method gives the most efficient and optimal results. But with every optimisation there comes an overhead. The overhead of maintaining a queue with the chunks and granting the threads' requests progressively during run time creep in in this scheduling. For smaller chunk sizes this happens fairly often and hence this overhead over compensates for the speedup which the dynamism may have been causing.

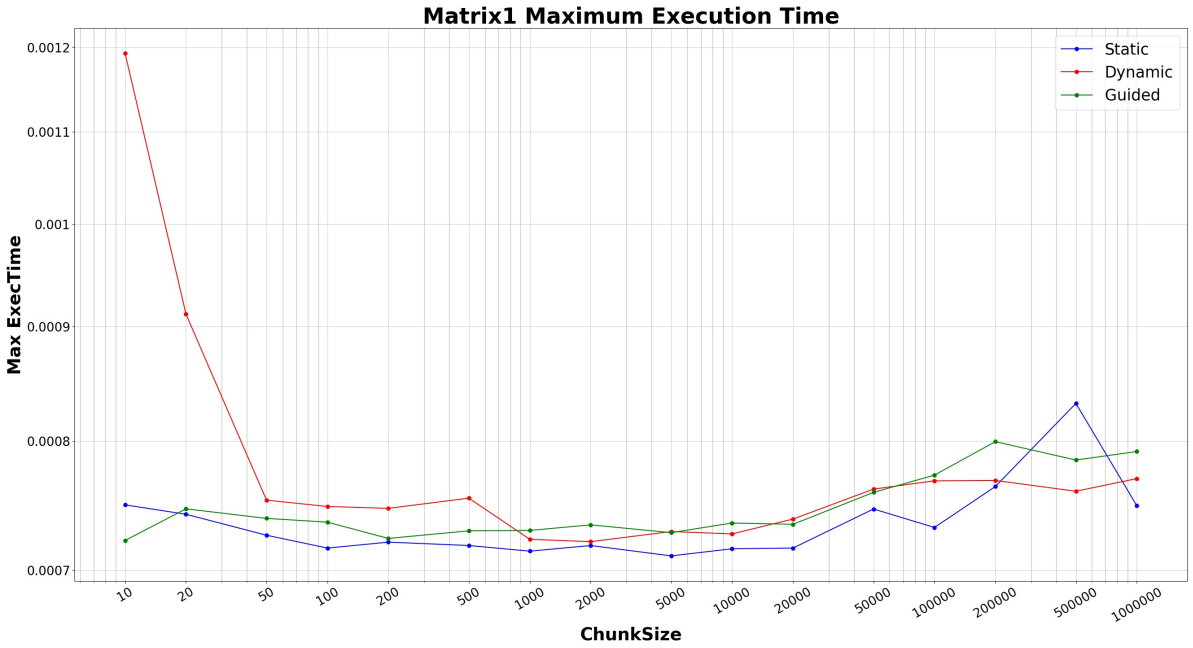
Static and Guided Scheduling are seemingly unaffected initially with the increase in *chunk_size*, but with a further increase there starts a steep ascent in the execution times.

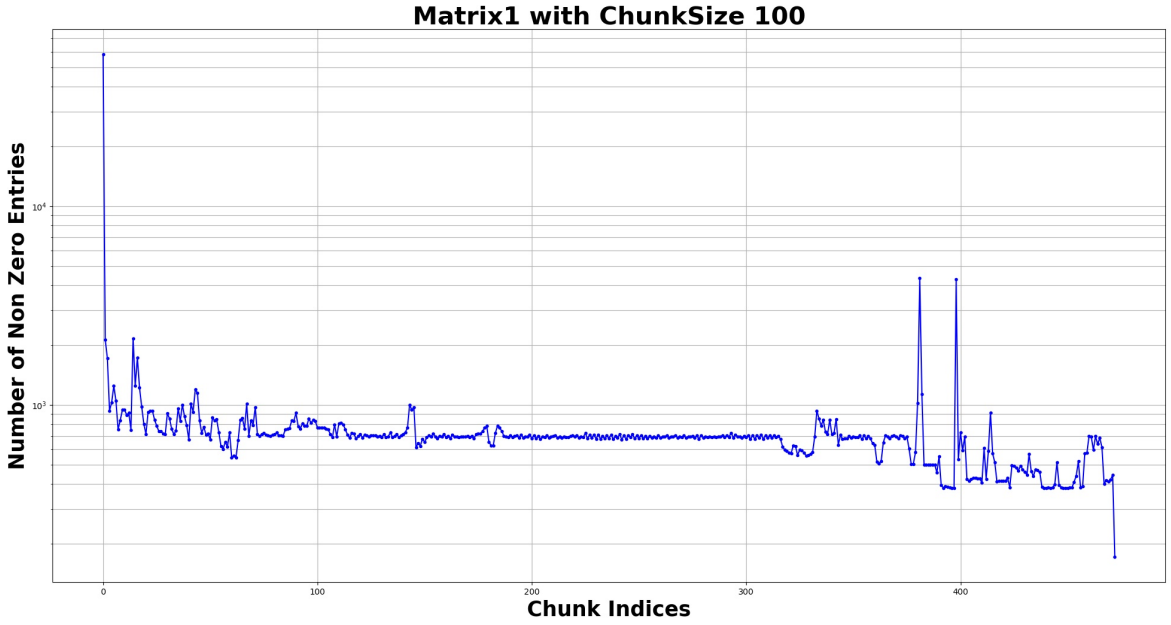
The justifications for this steep rise can be drawn from the fact that bigger *chunk_size* implies that all the threads may not even get a subset of the iteration space. The big chunks are assigned to the initial threads and get finished by the time the remaining threads are reached. This is why after a certain threshold where the complete iteration-space is within one chunk, the execution time is similar to when there is only one thread executing.

We can deduce from the trend here that the **Maximum Execution Time** is more or less similar (except for a few outlying spikes) for all the threads and for all the *scheduling_type*.

The maximum time any team of threads takes to execute an algorithm is when only one thread is doing all the computing and all the other threads are idle. This as we have discussed earlier happens when the chunk size is so large that effectively only one thread out of the team is assigned the whole iteration-space. Hence the maximum execution time for each thread is when, due to a huge value of *chunk_size* provided as a scheduling parameter, the computation is effectively reduced to a serial implementation.

Similarly for the variation of **Maximum Execution Time** with *chunk_size*, we can more or less infer that the computation of the algorithm when the team of threads contains only one runner, i.e. the Serial Implementation, takes the maximum amount of time for each *chunk_size*.



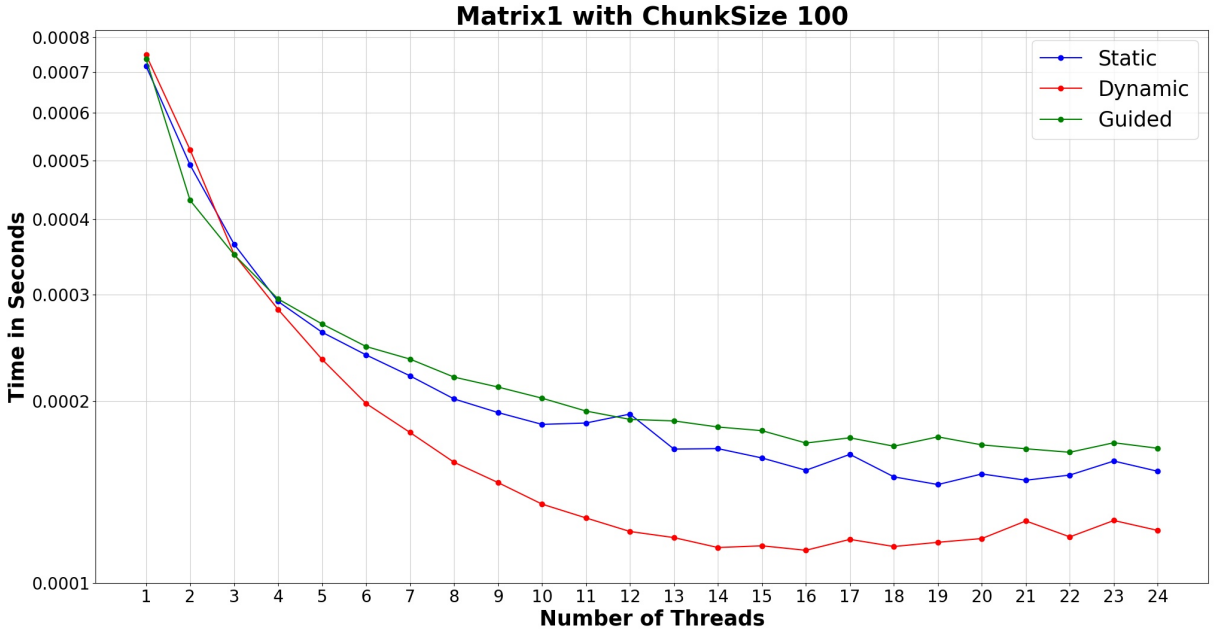


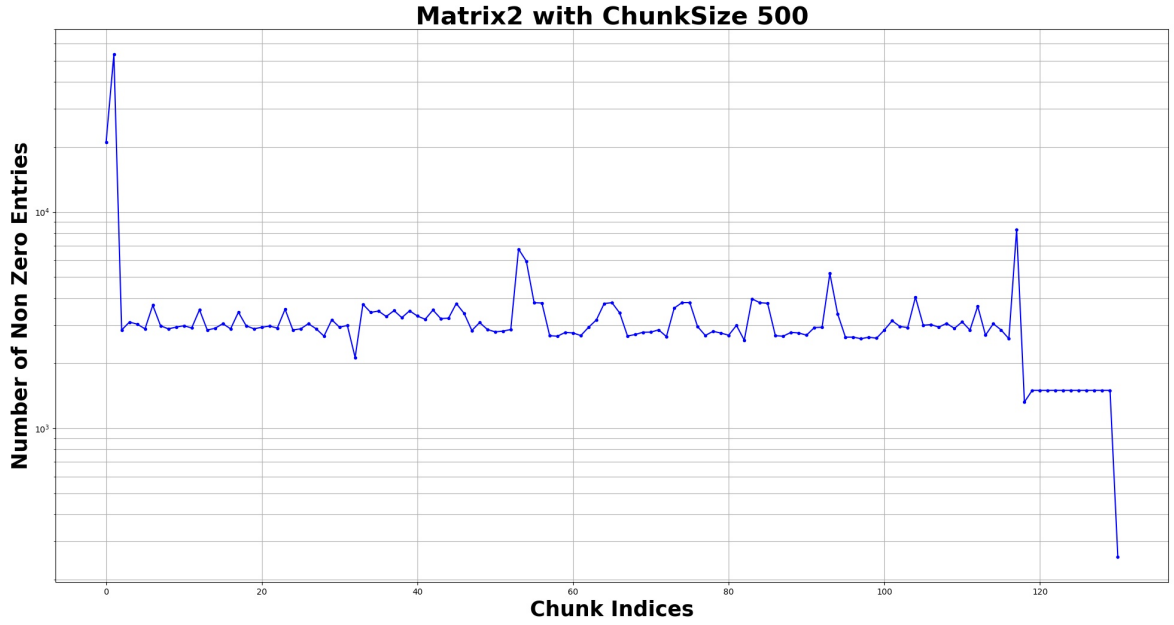
This is a plot of the number of non-zero entries in each chunk of size 100, which is the *chunk_size* where the minimum execution time occurs in case of **Matrix1** .

As we can observe here that the number of non-zero entries in the chunks in the middle is roughly the same and hence the type of scheduling method does not play much of a role here.

However there are minor fluctuations in the starting as well as the end, and it is such an uneven distribution of work amongst the chunks where **dynamic** scheduling trumps out **static and guided**.

Moreover, the first chunk is exponentially larger than all the other chunks and possibly larger than each one of them combined. In the case of **static scheduling**, the iteration chunks grouped together with the *chunk1* would have to wait for a long time before the thread assigned to them reaches them, and this would cause a significant slowdown.
This is the reason we can see a decent speedup in case of **dynamic scheduling**, because here in this case the thread assigned to *chunk1* would keep executing whilst the other threads finish off their chunks and get assigned new ones without much waiting.

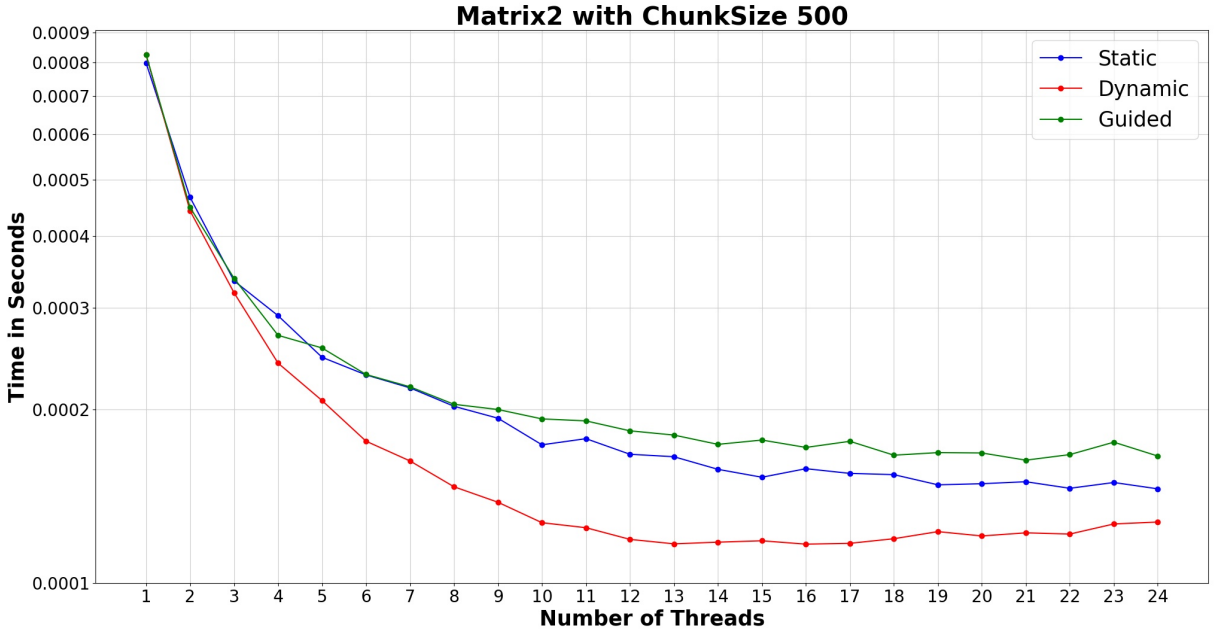


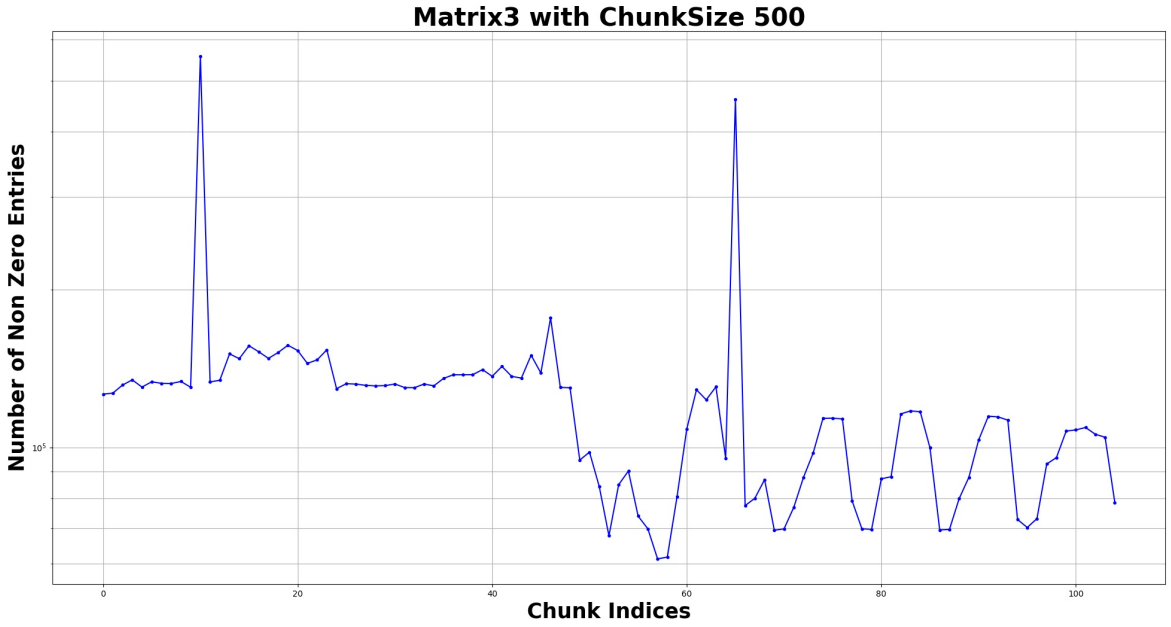


This is a plot of the number of non-zero entries in each chunk of size 500, which is the *chunk_size* where the minimum execution time occurs in case of **Matrix2** .

There are minor fluctuations throughout, and it is such an uneven distribution of work amongst the chunks where **dynamic** scheduling trumps out **static and guided**.

Moreover, again here the first chunk is exponentially larger than all the other chunks. Similarly, in the case of **static scheduling**, the iteration chunks grouped together with the *chunk1* would have to wait for a long time before the thread assigned to them reaches them, and this would cause a significant slowdown.
This is the reason we can see a decent speedup in case of **dynamic scheduling**, because here in this case the thread assigned to *chunk1* would keep executing whilst the other threads finish off their chunks and get assigned new ones without much waiting.

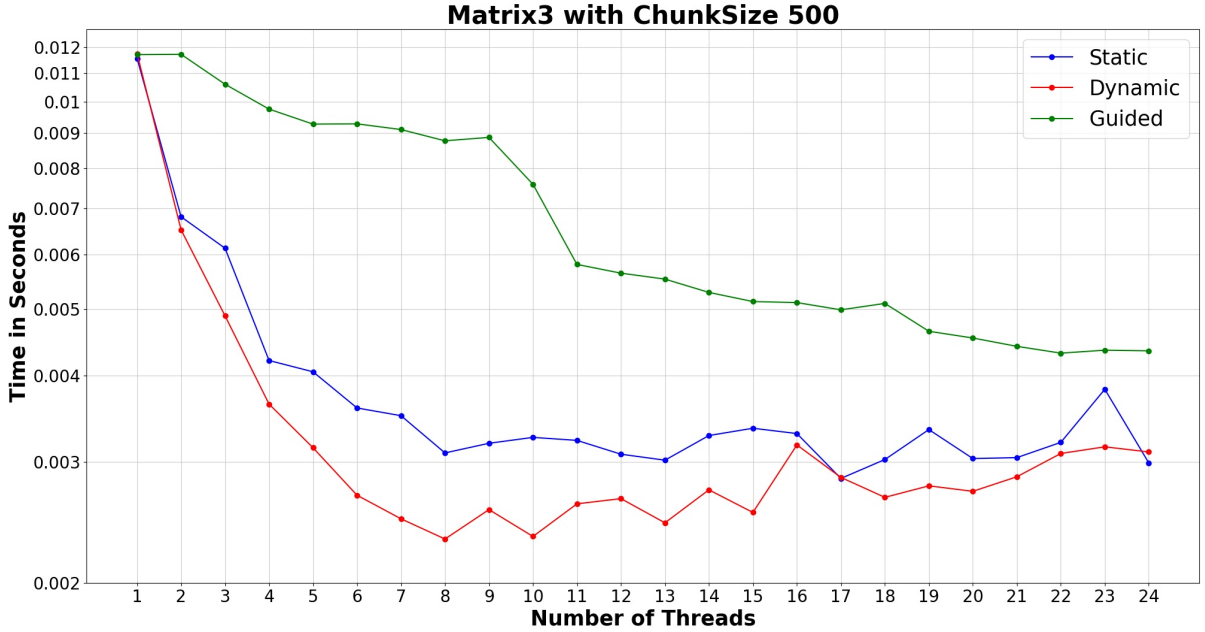


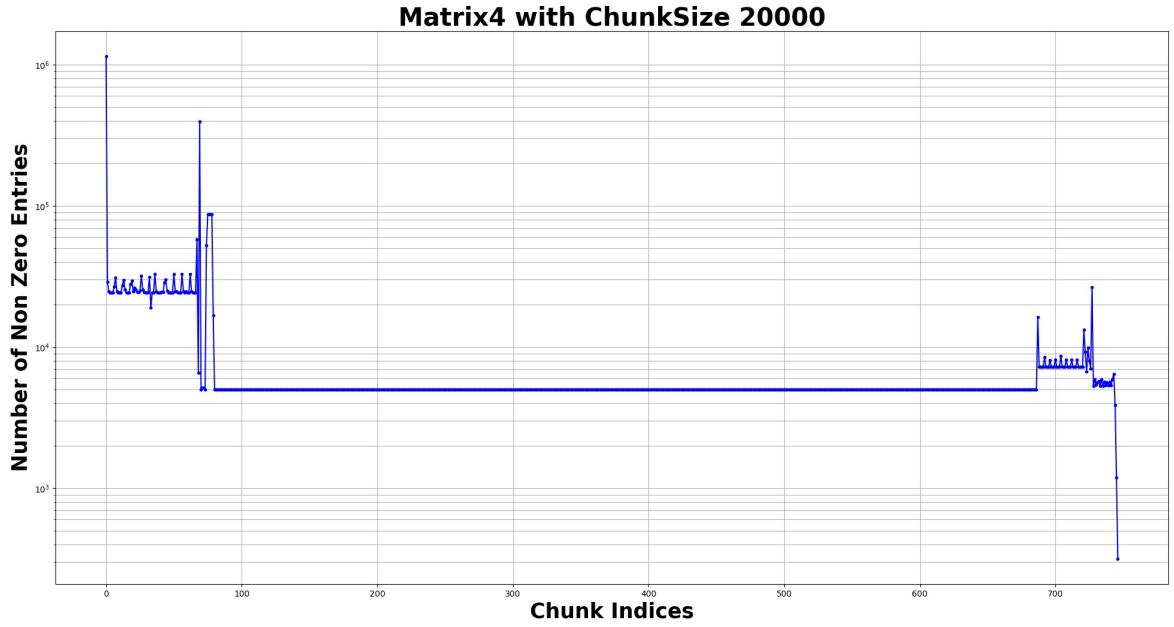


This is a plot of the number of non-zero entries in each chunk of size 500, which is the *chunk_size* where the minimum execution time occurs in case of **Matrix3** .

There are huge fluctuations throughout the distribution, with peaks and valleys at regular intervals, implying a very unequal distribution of values amongst the rows of the matrices. This is reason why **static and guided** scheduling have really bad performances as opposed to **dynamic**.

As opposed to the previous matrices vis-a-vis **Matrix1 and Matrix2**, there are no chunk sequences where the sizes are relatively equal in length due to which there is vivid difference in the trends for the different *schedule_type*.

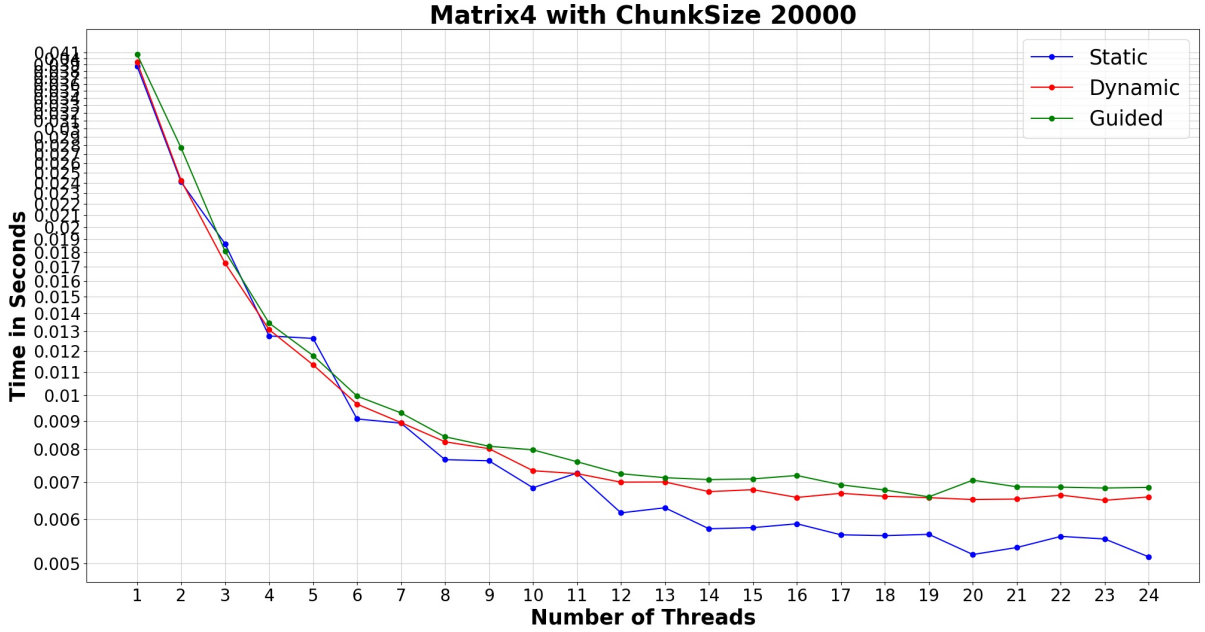




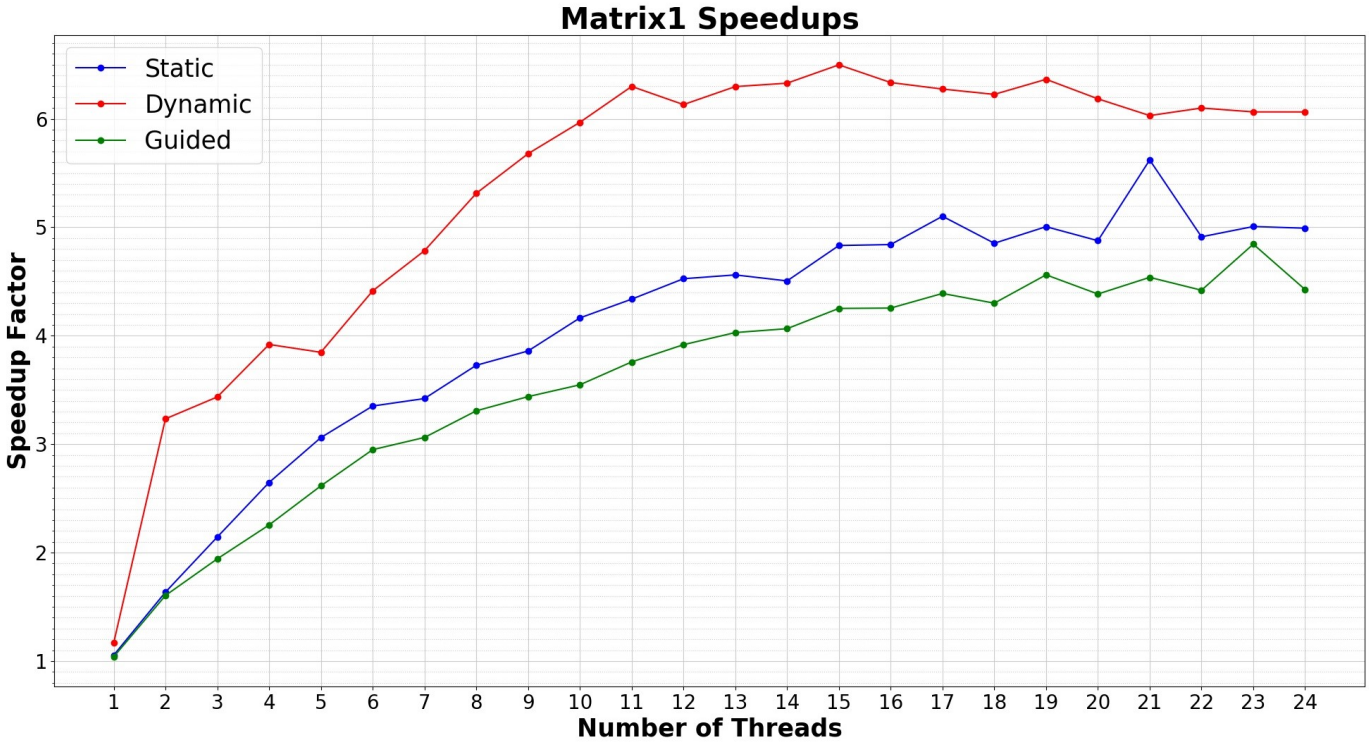
This is a plot of the number of non-zero entries in each chunk of size 20000, which is the *chunk_size* where the minimum execution time occurs in case of Matrix4 .

There is a peak in the start followed by some minor fluctuations followed by some really sharp spikes up and down after which an absolute serene constant plot is observed. Some fluctuations are present towards the end as well but those are smaller in magnitude in terms of deviations.

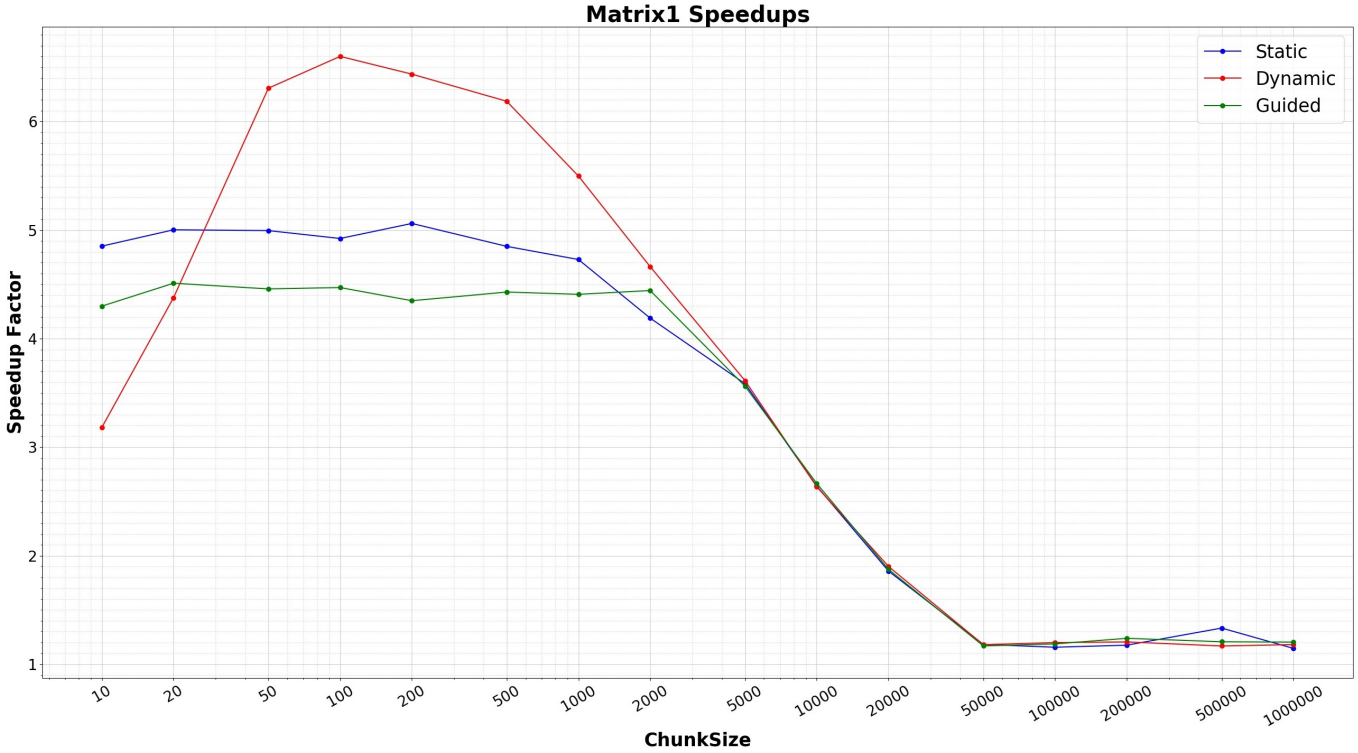
Here after the initial storm of fluctuations is taken care of, the following chunks of equal sizes render all the different *schedule_type* to have the same effect. Moreover, for a majority of the chunks, this equality is observed and hence the overall effect that this has is that all of **static, dynamic and guided** have plots which are really close by and have similar values throughout.



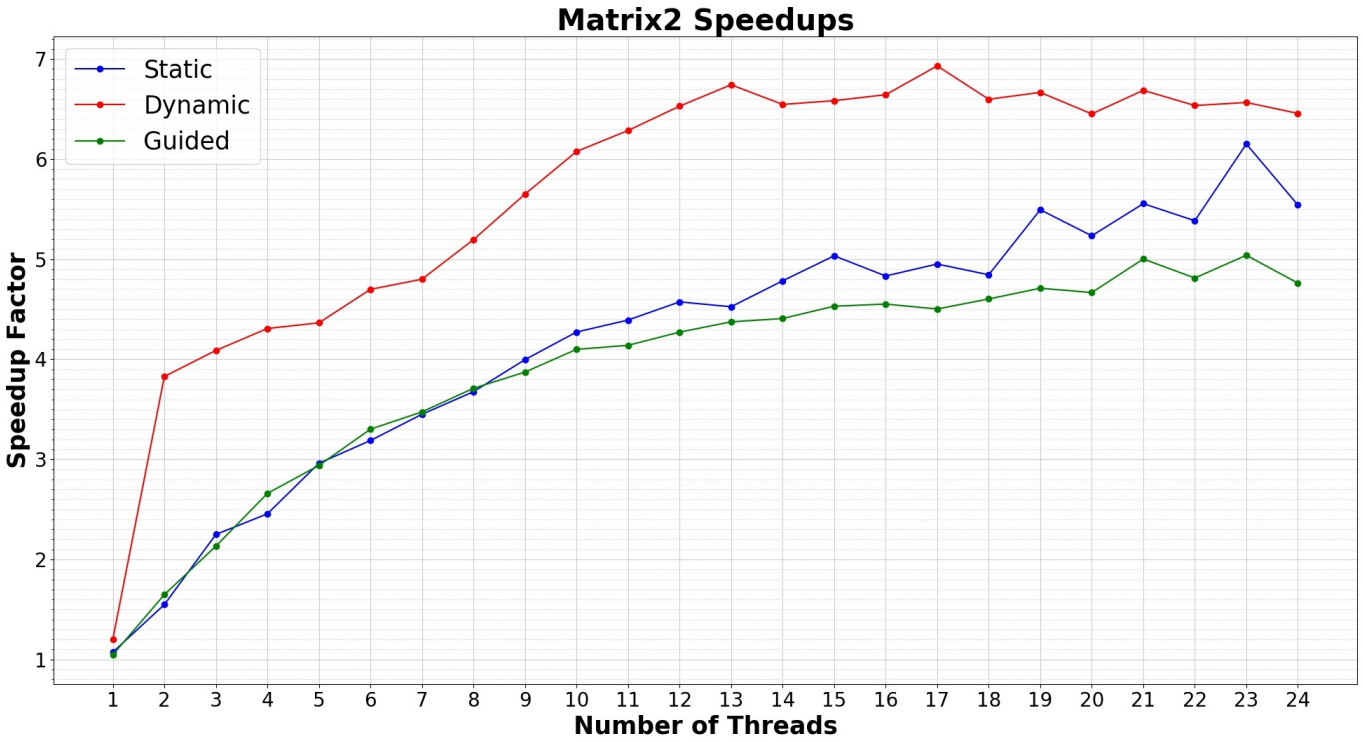
The **Speedup** of a Multi-Threaded Program is determined as
Max ExecTime/ Min ExecTime



The trends which the speedup follows is akin to the inverse of the Min ExecTime plot, which is what is expected since the Max ExecTime, as we showed previously, is almost the same for a particular team of threads or for a particular chunk size.



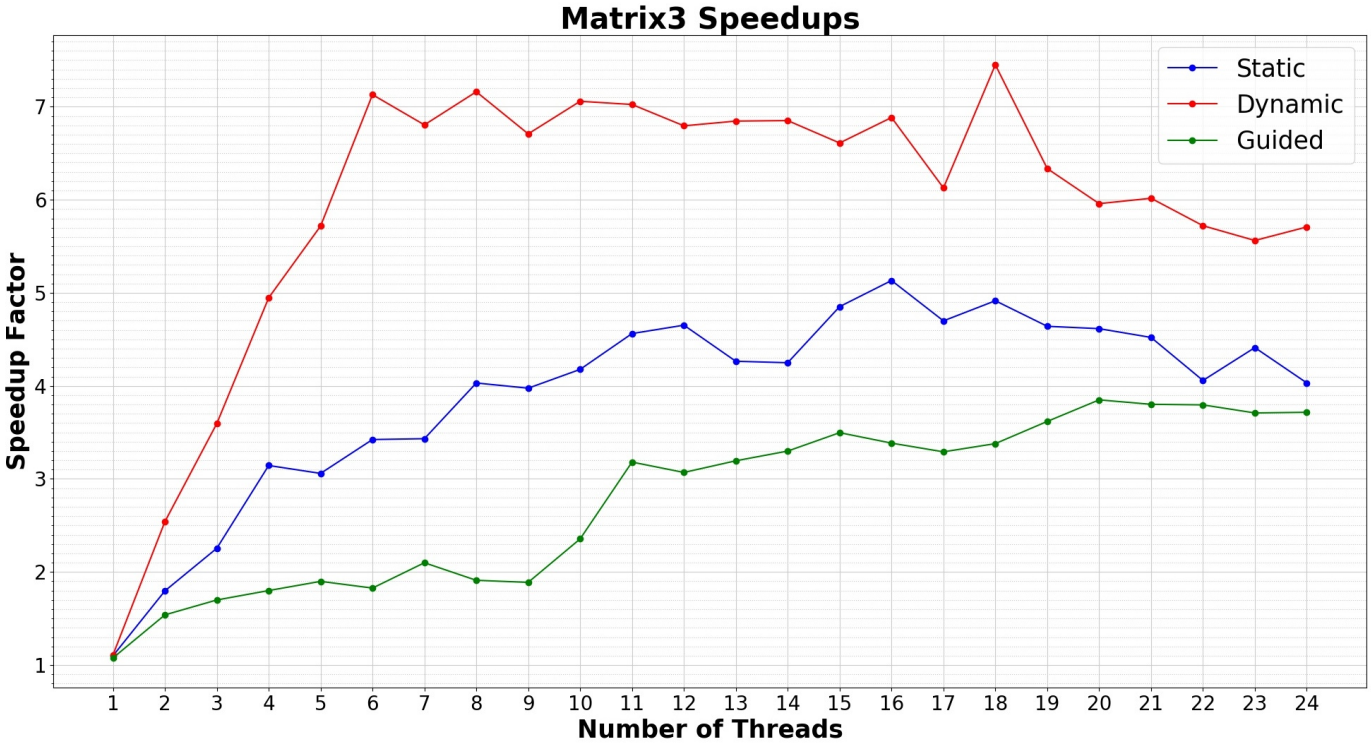
These experiments were performed on other matrices namely Matrix2, Matrix3 and Matrix4 as well. Here we can see similar trends for the variation of SpeedUp with *num_threads* for Matrix2 and Matrix3.



Another interesting think to observe is the value of the maximum possible speedup for each Matrix. There is a steady increase in the maximum speedup as the Matrix Size is increasing.
This can be correlated to the Amdahl's Effect.

As the Matrix indices are increasing, the size of the matrix is increasing by leaps and bounds, with Matrix1 requiring around 1MB of space, Matrix2 with 3MB and Matrix3 with 30MB space requirements. Amdahl's Law of Parallel Processing states that with an increase in total number of operation, the fraction of operations which are inherently serialised and cannot be parallelized decreases.

As the percent of code which needs to executed serially decreases, the minimum time taken by multi-threaded program decreases. The maximum time taken remains constant since it is a serial implementation which executed the slowest, hence the decrease in denominator cause an increase in the SpeedUp values.



INFERENCE

After analysing the trends of the various different plots, we can safely infer the following.

Increasing the *number of threads* on a multi-core processing system boosts the performance by decent factors, except when the threads increase the number of load-free processors available in the system at that instance.

Moreover the **dynamic** *scheduling algorithm* more often than not ensures better performance as opposed to its **static and guided** counterparts. However this is true only when the *chunk_size* is large enough for the queuing overhead to be negligible in comparison to the speedup achieved due to the compact fitting of chunks and thread runner schedules.

Given a particular sparse matrix, we can analyse the distribution of values amongst its rows to figure out which kind of a *scheduling_type* should be adopted for the best results.

For a matrix where the number of non-zero values for a particular *chunk_size* are similar over most of the chunks, there will be no particular advantage of using **dynamic** scheduling over **static or guided**. Moreover, **static** is the best option in that case since no additional construct like a queue needs to be maintained and that reduces a significant amount of overhead.

On the other hand in case the non-zero values are really unevenly distributed amongst the chunks, **dynamic** scheduling is the best option we have to minimise the waiting time for each chunk and hence optimise the overall execution time as much as possible.