CITS3402 Assignment 2

REDACTED, Student ID: REDACTED & Sebastian REDACTED, Student ID: REDACTED

2024

1 CITS3402 Assignment 2

1.1 Running the code

- Compile the code with `make release`, which adds some optimisation flags to the compile commands.
  - Doing `make` by itself compiles in the debug profile, which we didn't use for the tests.
- To run the program locally, run
  `mpirun -np <number of processes> ./proj <matrix size> <density> <threads per process> [-o]`.
  To output the matrices to disk, add the `-o` option AFTER the other parameters.
- For example, `mpirun -np 3 ./proj 100 0.05 4 -o`.

1.2 Strategy

We successfully wrote our program to use MPI and OpenMP together. To turn off one, we just reduce nodes/processes/threads to one.

We started with 4B integers for our elements, but we realised we only need 2B unsigned integers. As a generous example, lets say we're multiplying two 500k size matrices with 5% density, and elements are generated between 1 and 9. That means each element in the result with have an average value of $500000 \times 0.05^2 \times ((9-1)/2)^2 = 20000$, which isn't close to the max value of 65535.

1.2.1 Distribution of Work

Initial testing showed us that sending and receiving data was slow and should be avoided as much as possible. We also found that memory usage could explode at high matrix sizes, or even at higher process counts, if we weren't smart about how we created and distributed the matrices. For a good distribution strategy, we identified a few requirements:

- Our project 1 multiplication strategy already treated individual rows as single units that were never split apart and/or shared between threads. When a thread was assigned to multiply a row of the left-hand matrix, no other thread would ever read from that row. This worked well to keep threads from interfering with each other, and something similar could work for multiprocessing by assigning rows to processes. These groups of rows will be called "sections" throughout the report.
- Since each process is assigned a rank, they should be able to coordinate with other threads to some extent, without actually sending anything.
- Processes have to agree on what the matrices contain, even when they are scattered.
- Any work that can be done without communication should be done without communication. This means random generation and compression should be done locally. Any communication that sends large amounts of data should try to reduce the amount of API calls to reduce overhead.

- Has to scale well for any number of nodes/processes/threads. Preferably, memory usage should not change significantly with different numbers of nodes/processes/threads.

Each process gets assigned rows with this function. There are 2 global variables, **mat_size** and **size**, which correspond to the matrix row/column size and the number of processes, respectively.

```
void get_process_rows(const int rank, int* nrows, int* first_row_idx)
{
    int base = mat_size / size;
    int remainder = mat_size % size;
    *nrows = base;
    if (rank < remainder)
    {
        (*nrows)++;
    }

    *first_row_idx = rank * base;
    if (rank < remainder)
    {
        *first_row_idx += rank;
    }
    else
    {
        *first_row_idx += remainder;
    }
}
```

This calculates, for a given rank, the rank of rows that a process is responsible for, starting at **first_row_idx** and ending at **first_row_idx + nrows**. If **size** is not a factor of **mat_size**, then some processes are given an extra row to compensate. For example, if there are 15 rows and 4 processes, each process gets rows `[0,3]`, `[4,7]`, `[8,11]`, and `[12,14]` (using set notation). Therefore the rows are spread evenly between all processes.

When each process starts, they generate their left- and right-hand matrix sections with the correct number of rows and compresses them, and then moves onto multiplication. Generation and compression are both multithreaded, and since each process is responsible for it's own section, it also benefits from multiprocessing.

This strategy checks off all our requirements above. All processes own their own sections, with full-size rows. No communication takes place, and all processes can use the above function to determine what rows other ranks have, by changing the **rank** argument. Memory usage stays the same because the matrix is being split into sections that add to one matrix.

1.2.2 Multiplication

Multiplication was a bit tricky, since each process needs the entire right-hand matrix, but only has a section of it. We thought of 3 ways around this problem:

1. Every process gathers the entire right matrix and has their own copy. This is what we mean by "explosive memory usage," since for a 100k matrix with 5% density, a compressed matrix would be about 2GB if we use **unsigned shorts** as elements. If we run 64 processes, that would amount to 128GB wasted on needless copies, which violates our scaling requirement.

2. Make one main process act as the supervisor for the other worker processes. The workers can request rows one-by-one. We didn't test this strategy, but we avoided it entirely because it seemed slow for

individual threads to stop and ask for a single row, wait for the whole transmission to finish, then continue processing. Not to mention, the main thread would be bombarded with requests, which might be slow. This violates our overhead requirement, since we'd need to send potentially hundreds of thousands of requests between processes.

3. Do something in between the previous 2 ideas, where we broadcast large numbers of rows to each process. Conveniently, we already have the right-hand matrix split into sections, so all we need to do is broadcast them one-by-one using MPI.

Obviously we went with the third option. Each process takes turns in sending their sections to all other nodes. When the section copy is finished being sent/received, multiplication is performed with that section and then it's discarded. Then it's the next process's turn, and so on, until all sections have been sent to all other processes and multiplied. Each process will have a section with the final results.

Since the nodes only have 230GB of memory on Setonix, and about 384GB on Seb's homelab, we might not have had enough space to store the resulting matrix, so we left the result in this fragmented state. If it was required to join the sections into a whole matrix, we could; we have a function for it. However we decided against it because the project never says anything about it, and it would mean we can only utilise the memory on one node. We DO join when we output however, since we're only doing small matrices and it's simpler.

There are some important things to notice about this strategy:

- It continues to let every process be responsible for it's own section.
- Things like section row counts and who's turn it is to send is dictated by rank number, so communication for those things is not needed.
- Each process also only sends it's section out to each process once, in it's entirety and all in one go.
- If we have `P` processes and a matrix size of `M`, then each section is `M / P` rows big, assuming each section is the exact same size. If every process holds a copy of a section, then there are `P` copies of that section. The number of rows across all these copies is `M / P * P = M`, which is just 1 matrix. This holds true for any matrix size and any configuration, which makes it very scalable in terms of memory usage.

While testing on the homelab, we noticed that there's a pause where every process waits for the next section to be sent/received. We tried to avoid this by making one thread request the next section while the current section is being processed. This thread then continues multiplying after it finishes. Depending on the number of nodes, processes and threads, this could lead to there being no pause at all. In the worst case, it could lead to no change because the process only has one thread, so sending/receiving must be done sequentially.
The actual multiplication for each process is called **`CompressedMatrix_multiply`** and goes like this:

Given two input sections `X` (left) and `Y` (right):

- Create a matrix section for intermediate results, with the same size as the input sections. Set all elements to 0.
- Make a pointer to store the current section called `C`, and the next section called `N`.
- Request the first section, and store it in `N`.
- Loop over every process rank, i.e. from 0 to **`size`**. The loop variable determines which process should send, and all others receive.
  - Set `C` to `N`, which moves the next section to be the current section.
  - A single thread will do:
    - If it's this process's turn to send, send `Y`. Set `N` to `Y`.

    - Else, allocate space for `N` and receive another process's `Y` section.
    - Continue multiplying as below.
  - Loop over each row in `X`, using `#pragma parallel for`. Call the index of the row `i`.
    - Loop over each element in the row. Call the index of the element `j`.
      - Loop over each element in the corresponding row `j` of `C`. Call the index of the element `k`.
        - Multiply `X[i, j]` and `C[j, k]`, and add the product to element `[i, j]` of the intermediate results.
    - Once the loops are finished, free `C` if it's not equal to `Y`. In other words, we don't delete `Y` ever, we only delete the copies of `Y`.
- Once finished, compress the "intermediate" results matrix, which is now actually just the results section.

There are some parts left out for simplicity's sake, mainly to do with correcting for index offsets because everything's fragmented, but they're not important for understanding the overarching method we're using. See our code if you want to know more.

Notice we iterate over rows in the right matrix, and not columns. This means we don't splice columns, which leads to better cache efficiency. We also don't need to transpose the matrix.

1.3 MPI Shenanigans

MPI only lets you use integers for most, if not all, of their API calls. This is fine for smaller matrix sizes, but for larger matrices we need to send more than `INT_MAX` elements. As a result, we've had to write our own workarounds that allow us to use `size_t` counts and offsets instead. Apparently MPI makes optimisations under the hood to make composite functions like `MPI_Gather` faster. Since we're forced to use `MPI_Send` and `MPI_Recv` we don't get those optimisations.

There are a few different wrapper functions we've made for different MPI functions, but the general idea of the workaround is this:

Given that we want to send `N` elements of some datatype `D`:

- Make a new datatype that is `INT_MAX` elements long, with each element being of type `D`, using `MPI_Type_contiguous`.
- Send `floor(N / INT_MAX)` elements of this new datatype.
- Send the remaining `N % INT_MAX` elements of type `D`.

This means that we only need a maximum of two send/receive API calls to send/receive a long array. This is not ideal compared to just using builtin MPI functions, but there seems to be no other way with our current version of MPICH.

1.4 Data Analysis

1.4.1 The Setup

Initially we ran our code on Setonix, and while we did get some data from experimentation conducted on Setonix, we ran the majority of our tests on another system after we were told we were unable to use Pawsey anymore. For help with understanding our data (and choices we made), this is a quick description of the setup that was used.

The setup consists of two main machines, `machine_A`, and `machine_B`, which were separated into two "nodes" each; this was done to replicate 4 nodes, by using a separate gigabit connect for each "node" on each machine. From here on, `node_A_1` and `node_A_2` refer to the two nodes on `machine_A`, and `node_B_1` and

node_B_2 refer to the nodes on **machine_B**. While creating nodes like this had some fallbacks, for example, **node_A_1** and **node_A_2** shared the same cache (which wouldn't be the case if each node was a physical machine), it did help us experiment with how a differing number of nodes would be impacted by network performance (although, this also had it's own fallbacks, as two nodes located on the same physical machine were able to communicate much faster than two nodes on separate machines were able to). Assigning each node its own gigabit connection was beneficial however, as it meant that the data transfer rate of a node would not be impacted if the second node on the same machine was also receiving (or sending) data (providing that the node on the separate machine was not sending or receiving data from both of these nodes).

While both machines had 384GB of RAM, running at 1066 MT/s, **machine_A** had 24 CPU cores, whereas **machine_B** only had 16 CPU cores, both had hyperthreading enabled (which we found did not decrease performance), we effectively had 80 threads available for our testing - however, when we wanted homogeneous nodes, we were limited to only using 16 threads per node (64 threads total). Both machines were running a recent version of MPICH (version 4.2.3).

1.4.2 Data Analysis

The first thing that we wanted to test was the impact of multiple nodes on the speed of the calculations. We were expecting to find that the limited network speed would cause a significant decrease in performance as the number of nodes increase.

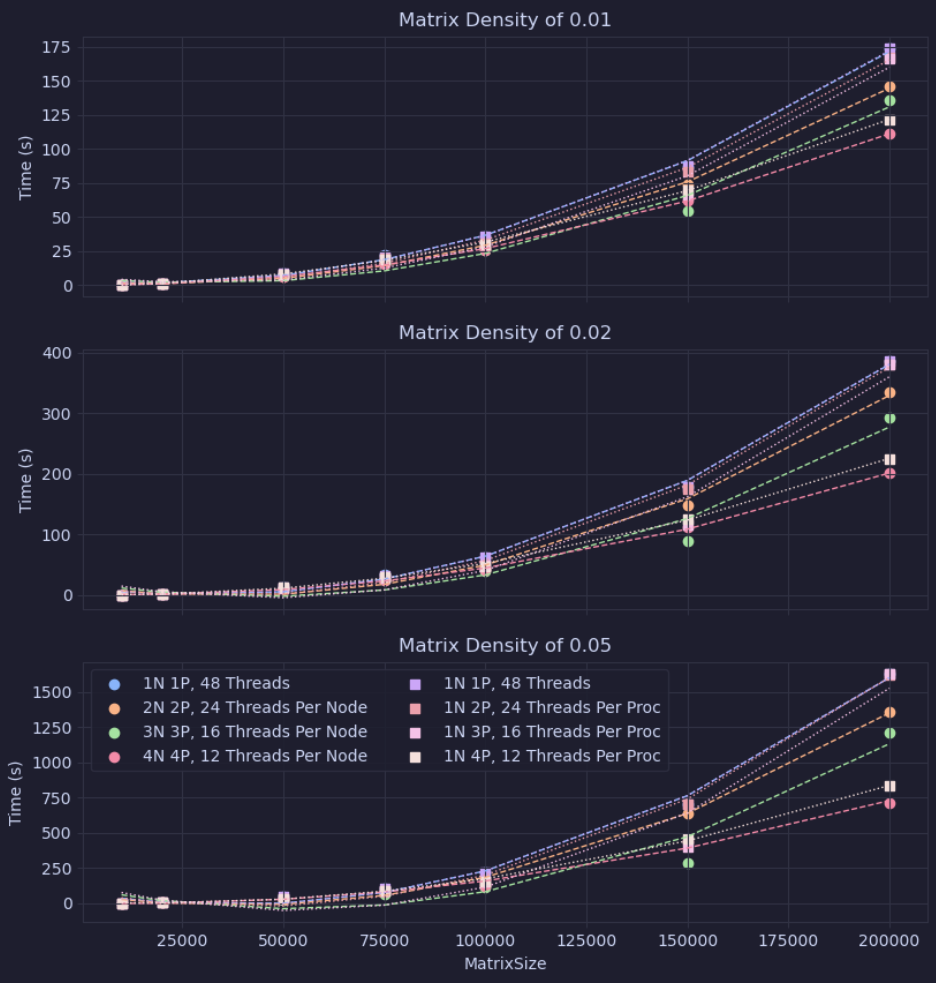

Figure 1: Figure 1 shows the difference in time to calculate values between running a MPI processes on each node compared to running all the processes on one node. The total number of threads was always 48, and was evenly distributed betwen each process.

As shown above in Figure 1, the network limitations did not impact the data in the way that was expected, and we found that multiple nodes increased the calculations speeds. This could be from an increase in the CPU cache, as using multiple nodes doubled the total amount of CPU cache that could be used. That said, if that was the only reason for an increased speed, we would expect that 2 nodes would be fastest (as explained above, two nodes on separate machines have the same amount of cache as 4 nodes on the two machines). However, four nodes did have a higher amount of network bandwidth available to be used, which meant that each process had access to its own gigabit connection with four nodes, whereas with two nodes processes would have to share a gigabit connection (if there were four processes). However, if this was the case, you would expect that two nodes and two processes should be the same speed as 4 nodes and 4 processes, as both runs had 48 threads available to them in total, each process would have had a gigabit connection, and the same amount of CPU cache. That clearly isn't the case here, and so clearly the configuration of MPI and OpenMP has an impact on the performance (as in, an equal number of threads being used can have different results depending on how many MPI processes there are for example).

It seems that more processes increased the speed - this seemed to possibly be related to data transfer speed however, as once a matrix was larger than a certain size $\sim 50k\times$ `numNodes`, the time taken suddenly skyrocketed. This is potentially due to the gigabit connection between nodes, possibly causing a network bottleneck, which would slow the performance down by a considerable amount (as each process would have to wait for a longer amount of time to transfer or receive data). More processes would also decrease the amount of data each process would have to send, which would also explain why increasing the number of nodes with a single process on a node was impacted less by an increase in matrix size. This would explain why an increase in number of processes increased speed (even when running on a single node).

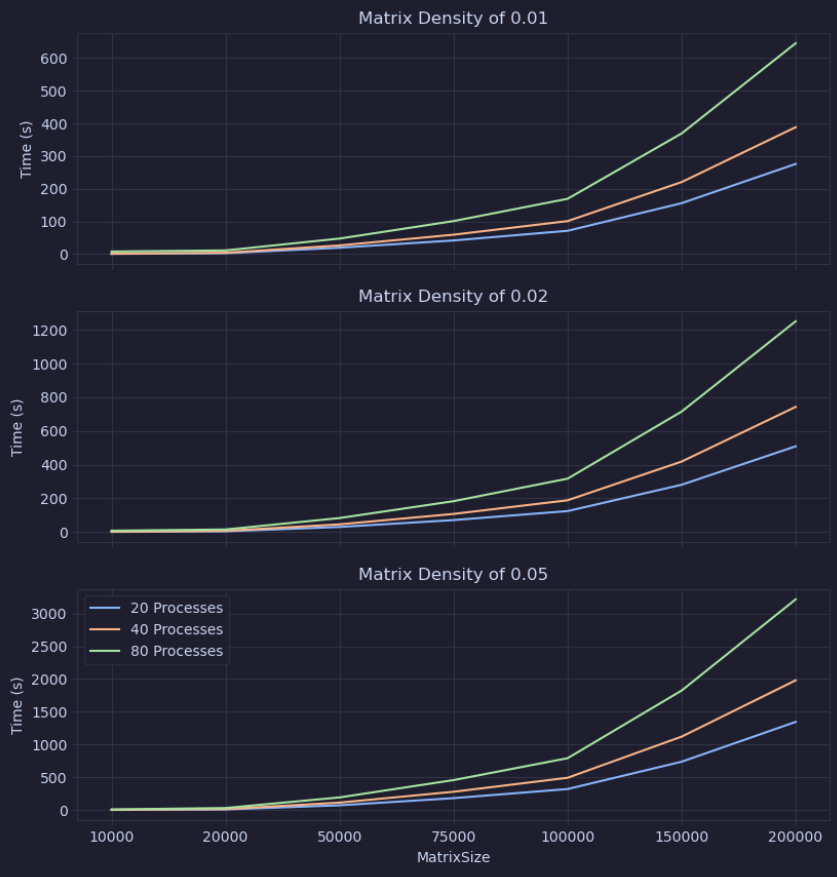

Figure 2: Figure 2 uses data gathered from running 20, 40, and 80 processes across 4 nodes, with each process using a single thread.

The next trial we tried was to see if not using OpenMP multithreading was optimal, and so we tried creating a MPI process for each thread. We found that the initial trend of the speed increasing as we increased the number of processes did not hold true for a larger number of processes when using multiple nodes; We found that a large number of processes would slow the speed of calculating matrices down considerably (see Figure 2). This slowdown was assumed to be caused by each process (and node) requiring a lot more communication with other nodes/processes, which meant that increasing the number of processes did not speed up the calculation, and in fact would slow it down when the number of process was taken to the extreme. These results were seen especially in single threaded processes (while still present in multithreaded processes), however all of these results were only seen in multi-node runs. Single threaded processes would have been hit the hardest as every thread would have to communicate with another thread (and while this was not bad on a single node, as communication speeds are a lot faster), communication across nodes is a lot slower. A larger num-

ber of processes mean that there is more fragmentation of the data, and as discussed above, more processors means that there needs to be more communication between nodes, which means each process would slow down considerably while waiting for data.

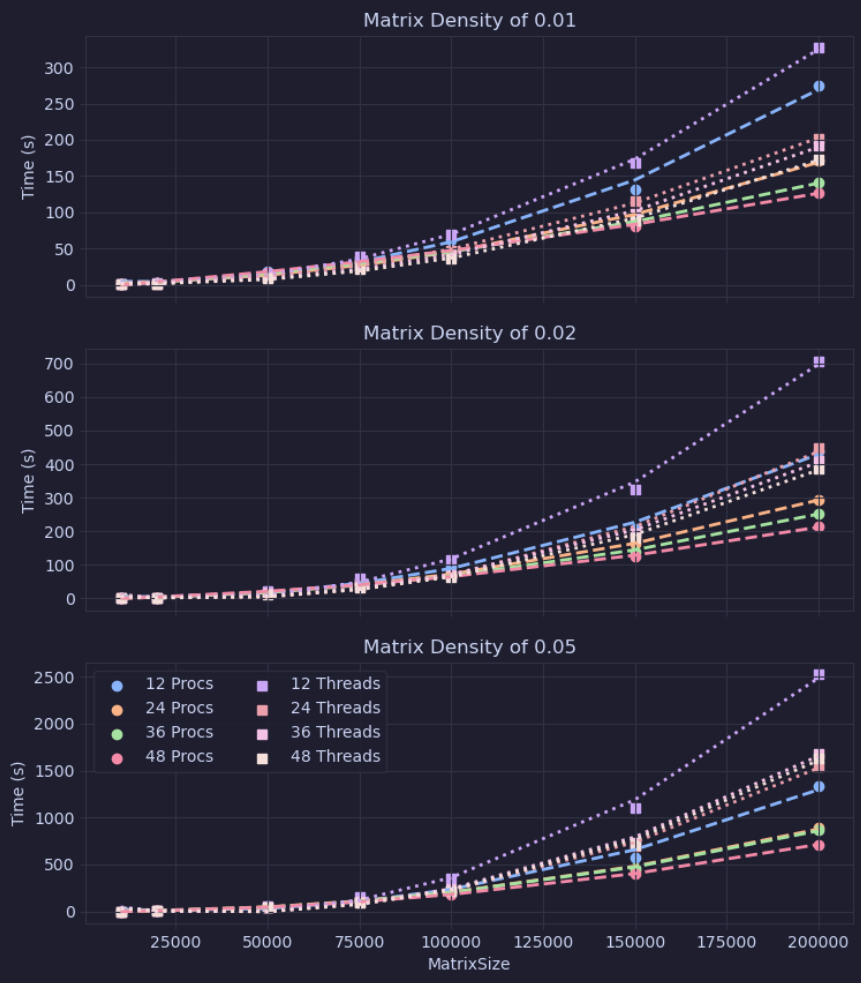

Figure 3: Figure 3: A comparison between the speed taken by a varying level of `x` processes on a single node (each process has a single thread), and a single process with `x` threads (also on a single node).

When we look at running multiple processes on a single node, we found that this slowdown from a high number of processes was not present, meaning that this only seemed to impact the speed when using multiple nodes. We found that on a single node, increasing the number of processes seemed to always increase the speed of calculating the matrix multiplication. Hence as this difference came from using a single node compared to multiple nodes, it seems that the factor causing a slowdown with increasing the number of processes (when the number of processes is large) when using multiple nodes was processes waiting on data transfers for extended period of time. This could have been from the slow the network communication speeds and data transfer rate. Sadly there wasn't a way to speed up the network transfer rates to test this theory, however we think if we ran these trials on Setonix, we would find that the slowdown from a high number of processes when using multiple nodes would be significantly decreased, or potentially not even present - at the very least the number of processes that could be run before experiencing slowdown would be much higher.

This wasn't a factor when using a single node, as the communication between processes was not limited to a gigabit connection, and so they were able to communicate their data much faster. We can also see that the speed when using the same number of threads in a single process (i.e. an OpenMP implementation only) is decreased significantly, when compared to using the same number of threads, each with their own process.

This is especially the case when the density of the matrix increases, which is presumed to be due to parts of the OpenMP implementation where only a single thread can be used at a time.

We found that an increase in the number of threads used by OpenMP increases the speed, however the effectiveness of this does wane as the number of threads (per process) grows larger, as you can clearly see the difference between 6 and 12 threads is much larger than the difference between 24 threads and 48 threads. This could be attributed to the cache more regularly needing to be swapped out as the number of threads increase.

Figure 4: Figure 4: the difference between two threads per process when compared to twice the number of processes with a single thread each

We can also see that even on a single node, OpenMP is important, and that even when network bottlenecks are not a factor, using a single thread for each process is not ideal. We can see above in Figure 4 that using multiple threads for each process does increase speed. This could be due to the implementation, where if a process has multiple threads, it simultaneously processes the data with one thread, while the second thread requests the next section. It could also be due to interprocess communication, as although communication between processes is a lot faster when there is a single node, OpenMP threads were implemented here in such a way that they do not need to communicate with each other. However, it seems that this speed gain

is more likely to be the latter reason, as the difference in speed that this created did not seem to change by much with an increase in matrix size, which would make sense as the impact from the former reason would be expected to change as matrix size and density (and hence the amount of data that needed to be sent between processes) increased. This is also potentially a reason as to why 80 single threaded processes over four nodes took longer than 20 processes, because each process could not request for the next data section while doing calculations (see Figure 2).

Hence, we found that increasing the number of processes, while being beneficial to the speed when used on a single node, was only effective across multiple nodes when kept to a relatively small number (roughly 5 in our setup). We also found that using multiple nodes does pose a significant speed boost, even with the slowdown from the network communication, which can be attributed to increasing the size of cache available. Finally, we found that increasing the number of threads for each process, while always beneficial, only really increased speed when communication between processes took a longer amount of time (i.e. more than one node).

Hence, we needed to find a number of processes that was relatively small, that allowed us to maximise the number of threads per process, which would allow us to keep network communication between processes to a minimum, while using all of the nodes.

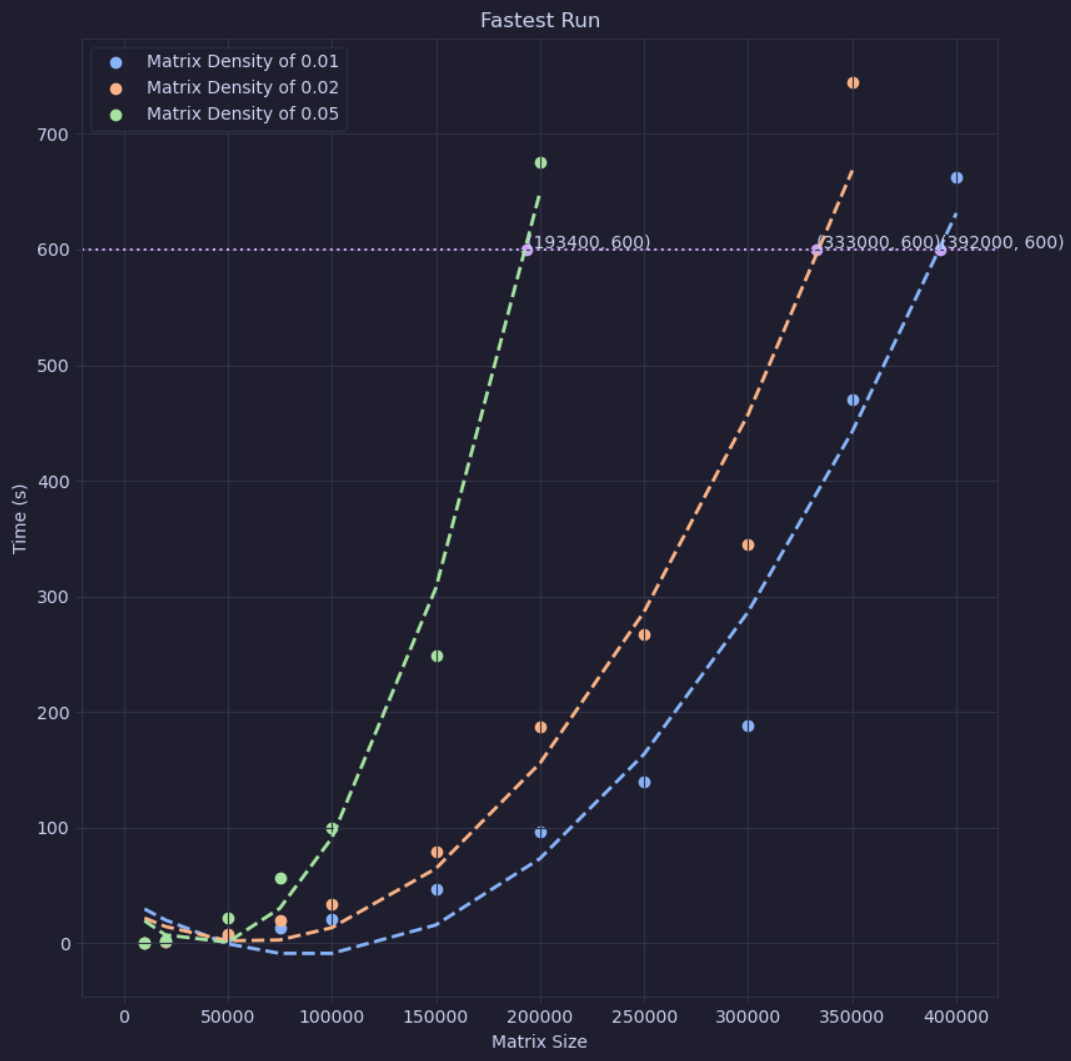

Figure 5: Figure 5: shows the fastest run we had, and shows the rough theoretical estimate of the maximum sized matrix we could calculate in ten minutes using our line of best fit (392000, 333000, and 193400 for densities 0.01, 0.02, and 0.05 respectively)

We found that the ideal number of process was roughly around 5 processes for our 4 node setup. This meant that we could have 16 threads per process, which, while 24 threads would be more performant, we found that increasing the number of processes to 5 performed better than more threads per process (with fewer processes). We assumed, tested, and found that 10 processors with 8 threads each had too much communication between processes and nodes, and seemed to be bottlenecked by our slow data transfer speeds between nodes, and that 5 processes with 16 threads each ran faster. These numbers were also chosen as we were limited with 80 threads in total, and with one machine having more threads than the other (32 compared to 48) we had to find values that would allow one machine to have three processes and one machine to have two. An improvement to speed could be to add another gigabit connection and node to **`machine_A`** to increase network speeds to the fifth process. However, even without this, we found that 5 processes with 16 threads each gave us our fastest result, which tracks with the predictions and reasoning we found above.

Figure 5 shows the fastest runs we managed to achieve for each matrix density. it also shows the limit of 10 minutes (600 seconds), and so we can see the largest matrix size that we could multiply in under 10 minutes (to the nearest 50k) - for `0.01` it was `350000`, `0.02` it was `300000`, and for `0.05` it was `150000`. We also show a rough estimate for the maximum sized matrix we could multiply in ten minutes using our lines of best fit.