Dense Linear Algebra in Parallel

This paper sets out to describe the algorithms used to compute dense linear algebra in parallel using MPI, then benchmark the scaling performance. We warm up by studying a matrix-vector multiplication, $y \leftarrow Ax$, then move onto matrix-matrix multiplication $C \leftarrow AB + C$, using the Cannon and SUMMA algorithms. Matrices (size NxN) and corresponding vectors of size N were used, with N specified by the program user. The parallel implementation was handled by MPI, using several processes, p. We conducted two experiments on each of the algorithms which were a weak and strong scaling investigation (the benchmark data), which enlightens us on the performance of the algorithms under larger workloads. We found a few issues with the benchmark data and investigated possible unforeseen pitfalls such as overheads and core architecture.

I. INTRODUCTION

Two important terms to explain which will be used throughout this paper are strong and weak scaling. Both are performance tests used to evaluate the implementation of parallel code and are very similar but with a subtle difference. Strong scaling looks at the code tackling a fixed total work while varying the number of processes used. The effect of this is that the work per process will decrease the more processes used. Weak scaling differs in that instead of the total work being fixed, the work per process is kept constant. To effectively analyse our implementations, we need a theoretical basis so we know what data good code should produce. For strong scaling we have Amdahl's Law [1] which assumes that you can perfectly parallelise the parallel fraction of the code, F_p and defines the serial fraction, $F_s = 1 - F_p$. Then the time spent executing the code on p processes:

$$T_p = T_1 \left(F_s + \frac{F_p}{p} \right), \tag{1}$$

where T_1 is the time spent on a single process. We can define a new term speedup, which quantifies the performance of the code of p processes and is the ratio:

$$S_p = \frac{T_1}{T_p}. \tag{2}$$

The limit for the speed up of the code can be found from the serial fraction of the code:

$$S_\infty = \lim_{p \to \infty} \frac{T_1}{T_1 \left(F_s + \frac{F_p}{p} \right)} = \frac{1}{F_s}, \tag{3}$$

this result is exactly Amdahl's Law. Similarly, for weak scaling there is a comparable analysis which comes in the form of Gustafson's Law [2]. Here we start with a problem running in parallel, with again known serial and parallel fractions, then if we know T_p on p processes we can derive:

$$T_1 = T_p F_s + p F_p T_p. \tag{4}$$

Then we can find the speedup:

$$S_p = \frac{T_1}{T_p} = F_s + p F_p = p + (1 - p) F_s. \tag{5}$$

There are various assumptions made with this model such as that the serial fraction of the code is independent of the problem size. As if this is not the case then as you increase total work N then F_s and F_p will change so the speedup equation 5 will not apply. But these theoretical models give us a good basis to start analysing the performance of our algorithms.

II. DESCRIPTION OF ALGORITHMS

Vector-Matrix Multiplication

Primarily the data was decomposed as shown in figure 1, each process is responsible for computing a portion of the product vector y. For this algorithm to work we require the size of matrix and vector N to be divisible by both the number of processes p, and $\sqrt{p}$. A small pseudocode is show in figure 2, in order to implement a parallel version of this, functions from the MPI library will be required. The first step is to broadcast each x_k to every process so each rank has the vector x. Then multiply the matrix available on the current rank $A(i, j)$ by the appropriate vector segment. We can select the correct portion of the vector x as it will be from $rank * N/\sqrt{p}$ to $(rank + 1) * N/\sqrt{p}$. Then each rank will have its own partial answer of y, so we reduce all the answers across to ranks using MPI_SUM operation, so each rank has the whole answer. Then each rank will pick out its specific portion of the complete solution y_p.

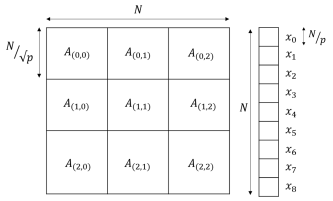

FIG. 1: Data decomposition for matrix-vector multiplication over 9 processes.

- Gather all partial x vectors from each process.
- Multiply local matrix A(i,j) by the corresponding part of vector x(i) and store it as a partial answer yx[i].
- Sum up all the partial answers yx[i] into yy[i].
- Set corresponding yy[i] answer to solution vector y.

FIG. 2: Pseudocode written for the vector-matrix algorithm before writing formally.

Cannon

The Cannon algorithm [3] handles the multiplication of square matrices of size N, and in this case for 2D matrices it is assumed that the number of processes p is square and N is divisible by $\sqrt{p}$. The two matrices are divided up in a checkerboard fashion into smaller square matrices of size $N/\sqrt{p}$ (as seen in fig.2), so the total number of square matrices is equal to p. Again, a simple pseudocode is shown in figure 4. First we completed the initial "skewing", this entailed each submatrix $A(i, j)$ being shifted left by i and each

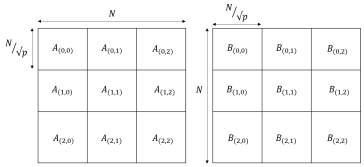

FIG. 3: Checkerboard data decomposition for 2D matrix-matrix multiplication using $p = 9$.

submatrix $B(i, j)$ shifted up by j. Obviously the shifts are wrapped so that for instance B(0,0) will shift up to B(2,0) position for the example in figure 3. Then each submatrix $A(i, j)$ is multiplied by its corresponding $B(i, j)$ and added to the result $C(i, j)$. Then matrices A and B are shifted left for A and up for B respectively by one, a similar process to the initial skew but each block only moves by 1 instead of by i or j. Then each $A(i, j)$ and $B(i, j)$ is again multiplied and added to $C(i, j)$. This shift and multiplication process is then repeated for a total of $\sqrt{p}$ times, and $C(i, j)$ is found from the addition of all the submatrix multiplications.

- Perform initial skew:
  - Left-circular-shift row i of A by i.
  - Up-circular-shift B column j by j.
- Main loop, for k = 1 to $\sqrt{p}$:
  - Perform local multiplication of A and B.
  - Left-circular-shift each row of A by 1.
  - Up-circular-shift each row of B by 1.

FIG. 4: Pseudocode written for the Cannon algorithm before writing formally.

SUMMA

The SUMMA method [4] is a more general recipe for matrix multiplication and can handle non square products, but we will only concern ourselves with the case of perfectly square matrices. Again, here the two matrices are decomposed as shown in figure 2, instead of shifting the submatrices like in the Cannon algorithm here we broadcast the data. For $k < \sqrt{p}$ we broadcast $A(i, k)$ to the other processes on the row i, similarly we broadcast $B(k, j)$ to the processes in their column j. Then the submatrix product $C(i, j)$ is calculated by summing the products of $A(i, k)$ and $B(k, j)$. The quick pseudocode is shown in figure 5 and this seems much simpler than the Cannon algorithm as well as more flexible due to the fact it can be used for any matrices. However, theoretically the communication cost on this algorithm is worse but we will investigate that further later.

- Main loop, for k = 1 to $\sqrt{p}$:
  - Broadcast A(k, 0:(n-1)) to the rest of the row.
  - Broadcast B(0:(n-1), k) to the rest of the column.
  - C(i,j) += A(k, 0:(n-1)) · B(0:(n-1), k)

FIG. 5: Pseudocode written for the SUMMA method before writing formally.

III. RESULTS

To conduct my experiments on the algorithms described above I produced a SLURM script for each so that I could run the algorithm with a variety of processes and sized matrices (or vectors). First, I chose to study the algorithms under the strong scaling regime, so the total work done was kept constant despite the number of processes used. Therefore, the work done per process decreases the more processes used. To compare we would need a fixed total sized matrix that was divisible by all the test number of processes, thus we used a total size $N \propto \Pi_i^k \sqrt{p_i}$ where k is the total number of tests conducted. For the vector-matrix algorithm there is an additional condition that N is also divisible by p as well as $\sqrt{p}$ we use $N \propto \Pi_i^k p_i$. Due to the constraints of the skeleton code only certain sized matrices could be created and tested, therefore the number of data points we could collect was limited as the larger k was, N increased. With some trial and error, I found that if we wanted to span the full range of processes (up to 256), I could use 5 different tests for the matrix-matrix algorithms and 3 for the vector-matrix algorithm. To compensate for this, I decided to study two different ranges for the vector-matrix procedure, one short range and one over the full range. This enabled me to run 5 tests on the shorter range as the total sized matrix N did not need to be so large. After doing this I decided to also do the same for the matrix-matrix processes and found 7 tests could be run, so we could maximise the data to study their performance.

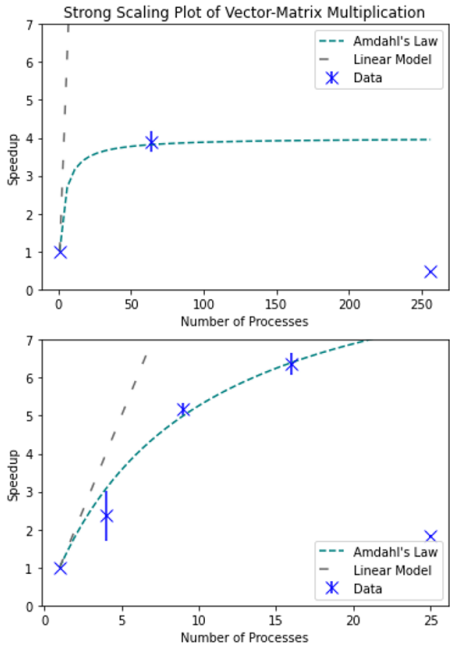

FIG. 6: A Strong scaling plot of the time taken to run the matrix-vector algorithm as a function of the number of processes used. Both experiments conducted are shown on different plots, the same y-axis is used to enable an effective comparison. A Linear model and Amdahl's Law are plotted with $F_p = 0.90$ for the short regime and $F_p = 0.75$ for the experiment over 256 processes.

Looking at the strong scaling performance primarily all the algorithms start to scale linearly but then should deviate according to Amdahl's Law. Starting with the vector-matrix algorithm it was clear that the algorithm did not scale as expected. The speedup over 256 processes displayed very odd behaviour as Amdahl's Law predicts a fast initial rise and a slow level off to a maximum value S_∞. But both plots display a decay in speedup after a certain number of processes. Initially, the first four tests under 20 processes scale very well with Amdahl's Law with $F_p = 0.9$, which seems very logical. However, the final value at 25 processes is

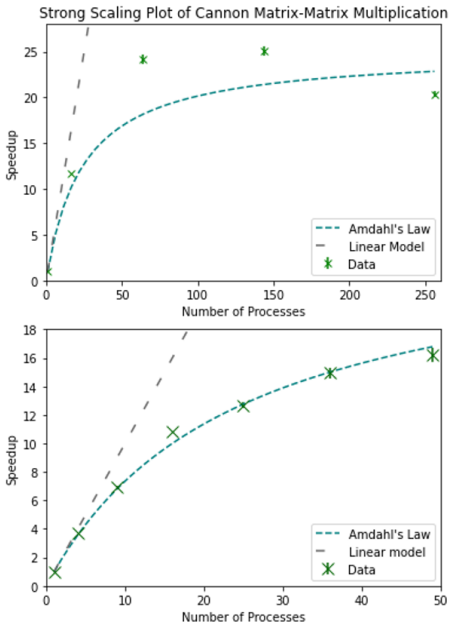

FIG. 7: A Strong scaling plot of the time taken to run the Cannon algorithm as a function of the number of processes used. Each regime is placed on its own plots, with a linear model and Amdahl's Law plotted with $F_p = 0.96$.

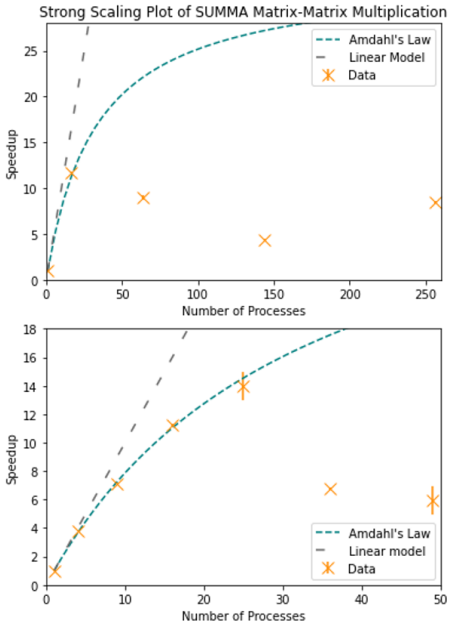

FIG. 8: A Strong scaling plot of the time taken to run the SUMMA algorithm as a function of the number of processes used. Each regime is placed on its own plots, with a linear model and Amdahl's Law plotted with $F_p = 0.97$.

much smaller and shows no resemblance to the trend. This is then reflected on the larger plot over 256 processes as the value for 64 processes would follow Amdahl's Law if $F_p = 0.75$ and the value for 256 would only be correct with a negligible value for F_p. We can conclude then that for more processes than roughly 20 that code starts to deviate from Amdahl's Law for some reason.

The data from the Cannon Algorithm is more encouraging but equally raises some issues. Looking at the smaller plot from 1-50 processes we see that the code scales very well with the value of $F_p = 0.96$. Expanding over 256 processes the data follows Amdahl's law less precisely but does seem to reach the speedup limit predicted $S_\infty = 25$, but a lot sooner than theoretically predicted. This method clearly scales much better than the vector-matrix algorithm but hits the speedup limit much faster than predicted, before levelling off and if anything drops towards 256 processes. This could be down to a myriad of factors that we explore later but is more encouraging as both plots do not seem to contradict themselves unlike the vector-matrix algorithm.

Finally, the SUMMA implementation shows similar properties to the vector-matrix algorithm, for less than 30 processes it scales well with a $F_p = 0.97$ but beyond that it starts to decay and roughly levels off as you go to 256 processes. This is not ideal and clearly shows this method is inefficient on larger processes as completely deviates from Amdahl's Law and does not go near the speedup limit.

Next, we wanted to look at the algorithms by computing their performance on weak scaling. Therefore, as the total work will be proportional to the size of the matrix, we had to keep the size of the local matrix on each process constant. The maximum number of processes used was 256 and to ensure that the tests did not complete too fast each local process was given a square matrix of size 20 so the total size N was equal to $20 * \sqrt{p}$.

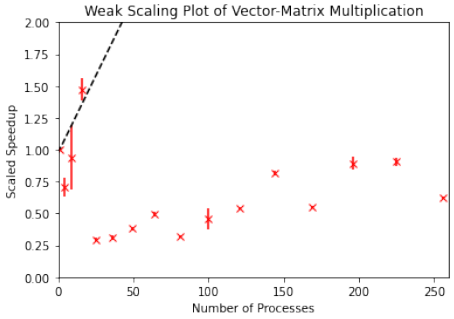

FIG. 9: A weak scaling plot of the time taken to run the vector-matrix multiplication algorithm as a function of the number of processes used. The dashed line added shows the fitted curve based on Gustafson's Law for the first 4 values.

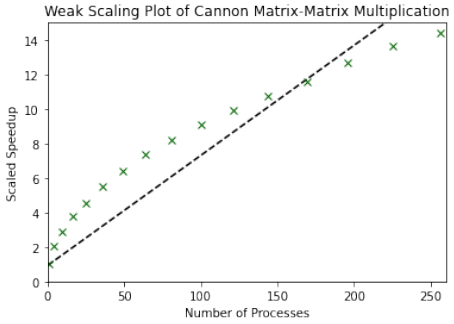

FIG. 10: A weak scaling plot of the time taken to run the cannon algorithm as a function of the number of processes used. The dashed line added shows the fitted curve based on Gustafson's Law iwth $F_p = 0.94$.

It could be argued that the vector-matrix algorithm started to scale correctly fo less than 20 processes before breaking down. Gustafson's Law is fitted for the first four values

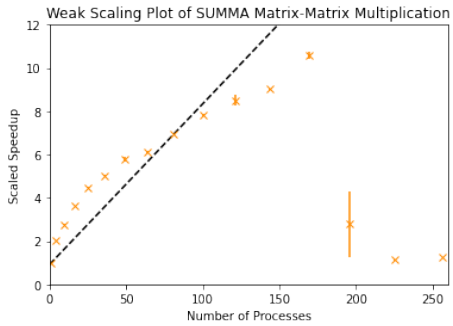

FIG. 11: A weak scaling plot of the time taken to run the SUMMA algorithm as a function of the number of processes used. The dashed line added shows the fitted curve based on Gustafson's Law excluding the last 3 values.

and gives a reasonable value for $F_p = 0.96$ which seems promising and makes some logical sense. However, similar to the strong scaling investigation, the algorithm seemed to fail beyond 20 processes again. Therefore, this seems to be a problem with the algorithm that beyond this limit it does not scale properly. This weak scaling attempt was also hampered by the fact that the size N needed to be divisible by both p and $\sqrt{p}$ which meant that the size of the local matrices was much smaller than the matrix-matrix investigations. I found that the smaller matrices gave much poorer scaling results, and we were limited to using local sizes of 500 much smaller than what we used for the other experiments (around 3000 for matrix-matrix).

The Cannon algorithm scales well under the weak scaling regime, one of the benefits of the cannon matrix is that we could set the matrices on each process to have size 3000. This solved the issue of the code finishing too fast in serial and seemed to give far more accurate data. There was a slight discrepancy in the value of F_p reported in the weak and strong scaling (0.94 and 0.96 respectively), but this is attributed to the different approximations in each law.

The SUMMA method was slightly less successful than Cannon but produced good data up until 196 processes. Again, as seen with the vector-matrix algorithm, the larger processes speedup drops off dramatically. Ignoring the last three values and fitting Gustafson's Law we find a value of $F_p = 0.93$ which is plausible and again the difference between the strong and weak estimates can be explained by the differing assumptions.

IV. DISCUSSION OF BENCHMARKING DATA

Amdahl's and Gustafson's Laws are very useful theories but do have their short comings as they were conceived in 1967 and 1988 respectively; in subsequent time computing power and techniques have improved exponentially. The laws do not consider overhead which is associated with initializing parallelism and the time spent in communication or synchronization between parallel tasks. Using MPI creates this overhead and so as the number of processes increase, and the number of messages sent increase, the more significant the overhead will be. The amount the overhead affects performance is due to how the program is written and can explain why the SUMMA and vector-matrix method run into problems above a threshold number of processes. So, above this threshold the overhead time becomes significant in slowing down the program producing a notable speedup drop in both the strong and weak scaling regimes. The Cannon algorithm does not seem to be affected and so must be written in a much more effective manner so that the overhead time is negligible even with large numbers of processes. If given more resources and time, we could investigate higher numbers of processes to see if indeed the algorithm would succumb to the overhead problem and start to break down.

Also, it is assumed that the type of processor used to compute the speedup of p processes and a single process are the same, but this may not be the case. Due to the size of the experiment, multiple cores were used (up to 11 for 256 processes), and many would have had different setups. This heterogeneity means that some processes would have been running on "fat" processors with a higher clock rate and higher instruction level parallelisation than the "thin" processors. So, if some of these cores contained one or more "fatter", quicker processors then the assumption that all processors are identical is wrong and could explain some of the deviation from laws observed. To accurately start to dissect how big a factor this is we would need to know the architecture of the chips used.

V. CONCLUSIONS

It can be argued that Amdahl's or Gustafson's Law are no longer relevant in today's computing climate and speedup is no longer solely dependent on the number of processes used. This is not surprising as the assumptions were made back in the 20th century and can explain why the benchmark data at points do not seem to behave under scaling. However, the extent to which these issues, such as overheads and core architecture can be to blame for the scaling discrepancies will remain unknown unless more time and resources are spent. A more efficient use of time would be to look back at the code written for these algorithms and redesign them in a fashion to negate issues (especially overheads). This is clearly possible as seen from the Cannon algorithm (scaled especially well in the weak regime) and should be consulted to see why it did not suffer under larger processor loads like its counterparts.

References

[1] Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, AFIPS '67 (Spring), Association for Computing Machinery, 1967, Pages 483-485.

[2] Gustafson JL, *Reevaluating Amdahl's law*, 1988, Commun ACM 31(5):532–533

[3] Y. Li and H. Li, *Optimization of Parallel I/O for Cannon's Algorithm Based on Lustre*, 2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science, Guilin, 2012, pp. 31-35, doi: 10.1109/DCABES.2012.61.

[4] R. A. Van De Geijn J. Watts, *SUMMA: scalable universal matrix multiplication algorithm*, Concurrency: Practice and Experience, Vol. 9(4), 255–274, April 1997