# PHYS52015 Coursework: Dense Parallel Linear Algebra

[redacted]

## 1 Introduction

In this report, implementations of a parallel algorithm for the matrix-vector multiplication $y \leftarrow Ax$ and two parallel algorithms for the matrix-matrix multiplication $C \leftarrow AB + C$ are described, where $A, B, C \in \mathbb{R}^{n\times n}$ and $x, y \in \mathbb{R}^n$. The strong and weak scaling performance of these implementations is also studied and discussed.

## 2 Parallel Matrix Multiplication Algorithms

Parallelism is achieved using MPI, decomposing the data evenly across a $s \times s$ grid of $p = s^2$ processes. Therefore, for matrix-matrix multiplication we require $p \in \left\{s^2 : s|n\right\}$ and for matrix-vector multiplication we require $p \in \left\{s^2 : s^2|n\right\}$ so that each process stores a square block matrix of size $n/s$ and a block vector of length $n/p$. The process at position $(i, j)$ of the process grid is denoted by $P(i, j)$ and its local blocks of matrix $M$ and vector $v$ denoted by $M^{(i,j)}$ and $v^{(is+j)}$ where $0 \leq i, j < s$.

### 2.1 Matrix-Vector Multiplication

Pseudocode of the algorithm written to perform the matrix-vector multiplication $y \leftarrow Ax$ is shown in Alg. 1. This algorithm utilises the fact that if $A$ is partitioned into $p \times q$ blocks $A_{ij}$, $x$ into $q$ blocks $x_j$ and $y$ into $p$ blocks $y_i$, then $y_i$, the $i^{\text{th}}$ block of $y \leftarrow Ax$, is given by

$$y_i = \sum_{j=0}^{q-1} y_i^{(j)} \quad \text{where} \quad y_i^{(j)} = A_{ij}x_j \tag{1}$$

Given that the result depends only on the $i^{\text{th}}$ row of blocks in $A$, the calculation is mapped onto the process grid so that $P(i, j)$ calculates $y_i^{(j)}$, the results being reduced among the $P(i, \cdot)$ to give the correct $y^{(is+j)}$ in each process.

In the implementation, the processes exchange their $x^{(is+j)}$ such that $P(i, j)$ receives from the $P(j, \cdot)$ the elements of $x_j$ and sends to the $P(\cdot, i)$ its elements of $x_i$; this is achieved using non-blocking `MPI_Isend` and `MPI_Irecv` with `MPI_Waitall` to avoid deadlock between processes. Then $y_i^{(j)}$ in $P(i, j)$ contains $s$ blocks of $n/p$ elements, the $k^{\text{th}}$ block of which is reduced by summation among the $P(i, \cdot)$ to give $y^{(is+k)}$ at $P(i, k)$. To simplify the reduction code, each row of processes is grouped into a communicator using `MPI_Comm_split` which allows reduction among a row of processes with `MPI_Reduce` .

**Algorithm 1** Matrix-vector multiplication $y \leftarrow Ax$ in process $P(i, j)$

```
1: $x_j \leftarrow$ Empty block vector of length $n/s$
2: for $n \leftarrow 0$ to $n \leftarrow s - 1$ do
3:     Send $x^{(is+j)}$ to $P(n, i)$
4:     Receive $x^{(js+n)}$ from $P(j, n)$ and append to $x_j$
5: end for
6: $y_i^{(j)} \leftarrow A^{(i,j)}x_j$
7: for $k \leftarrow 0$ to $k \leftarrow s - 1$ do
8:     Reduce by summation at $P(i, k)$ the $k^{\text{th}}$ block of $n/p$ elements of $y_i^{(j)}$ to give $y^{(is+k)}$
9: end for
```

### 2.2 Cannon's Algorithm

Cannon's algorithm is given in pseudocode in Alg. 2. To set up the calculation, each $P(i,j)$ shifts its $A^{(i,j)}$ upwards by $j$ rows and its $B^{(i,j)}$ leftwards by $i$ columns in the process grid. Then the local $C^{(i,j)} \leftarrow A^{(i,j)}B^{(i,j)}+C^{(i,j)}$ is evaluated, the $A$'s shifted leftward, and the $B$'s shifted upward in the process grid (wrapping around the edges), this step being repeated $s$ times. Each row & column of processes is grouped into a new communicator using `MPI_Comm_split` which simplifies the implementation and allows `MPI_Sendrecv_replace` to be used to rotate data within each communicator; this function has the added benefit that no additional data buffer is needed as the same buffer can be used for sending & receiving.

**Algorithm 2** Matrix-matrix multiplication $C \leftarrow AB + C$ in process $P(i,j)$ by Cannon's algorithm

```
1: Send $A^{(i,j)}$ to $P(i, j-i \mod s)$, replaced with block from $P(i, j+i \mod s)$
2: Send $B^{(i,j)}$ to $P(i-j \mod s, j)$, replaced with block from $P(i+j \mod s, j)$
3: for $n \leftarrow 0$ to $n \leftarrow s-1$ do
4:     $C^{(i,j)} \leftarrow A^{(i,j)}B^{(i,j)} + C^{(i,j)}$
5:     Send $A^{(i,j)}$ to $P(i, j-1 \mod s)$, replaced with block from $P(i, j+1 \mod s)$
6:     Send $B^{(i,j)}$ to $P(i-1 \mod s, j)$, replaced with block from $P(i+1 \mod s, j)$
7: end for
```

### 2.3 SUMMA

The Scalable Universal Matrix Multiplication Algorithm (SUMMA) calculates $C \leftarrow AB + C$ using the fact if $A$ and $B$ are partitioned into $p \times q$ blocks and $q \times r$ blocks respectively, then block $(AB)_{ij}$ of $AB$ is the sum of products of blocks of $A$ and $B$:

$$(AB)_{ij} = \sum_{k=0}^{q} A_{ik}B_{kj} \tag{2}$$

In SUMMA the processes calculate a term of this sum for their local blocks concurrently; i.e. each $P(m,n)$ calculates $A_{m0}B_{0n}$, then $A_{m1}B_{1n}$ up to $A_{mq}B_{qn}$. At the $n^{\text{th}}$ step of the summation the processes broadcast $A^{(i,n)}$ and $B^{(n,j)}$ within their within their row and column so that each process has the necessary blocks of $A$ and $B$. The implementation uses `MPI_Comm_split` to logically group processes by row & column and `MPI_Bcast` to broadcast data within these communicators. This calculation is shown in Alg. 3 for the $s \times s$ decomposition described above.

**Algorithm 3** Matrix-matrix multiplication $C \leftarrow AB + C$ in process $P(i,j)$ by SUMMA

```
1: for $n \leftarrow 0$ to $n \leftarrow s-1$ do
2:     Broadcast $A^{(i,n)}$ to the $P(i,\cdot)$
3:     Broadcast $B^{(n,j)}$ to the $P(\cdot,j)$
4:     $C^{(i,j)} \leftarrow A^{(i,n)}B^{(n,j)} + C^{(i,j)}$
5: end for
```

## 3 Scaling Performance

### 3.1 Methodology

The two main models for assessing performance of a parallel program are the **strong** and **weak** scaling models. In the strong scaling model, parallel performance is assessed in terms of the **speedup** - how much faster a problem of fixed size runs with additional parallelism, defined as $t(1)/t(p)$ where $t(p)$ is the execution time for $p$ processes. If a program spends a fraction $f_s$ of its execution time on the serial part of the task, the strong scaling model assumes that $t(p)$ is simply $t(1)[f_s + (1-f_s)/p]$. A program with perfect strong scaling ($f_s \to 0$) would then have speedup $p$ when run on $p$ processes.

The weak scaling model instead considers how much extra work can be done with additional parallelism, assuming that the size of the task per process is fixed. In this case the speedup is $f_s + (1-f_s)p$

and a perfectly weak-scaling program would do $p$ times more work with $p$ processes compared to that done in the same time using 1 process.

Both models assume there is no overhead associated with additional parallelism and that data transfer between processes is instantaneous.

To assess the strong scaling performance, the run time of the 3 algorithms was measured on the Hamilton par.7 partition for global matrix sizes $n$ of 6480 and 20160, for between 1 to 256 processes. The weak scaling performance was assessed by measuring the run time over a number of processes, keeping the size of the block matrix *within each process*, $n_{local}$, constant and scaling $n$ to match ($n = n_{local}\sqrt{p}$). Block matrix sizes of 1260 and 5000 were tested.

## 3.2 Results

The run times recorded for the strong and weak scaling tests are shown in Tables 1 and 2 respectively. Fig. 1 shows the speedup in the strong case (left) and the normalised runtime in the weak case (right).

| **Num. Processes, $p$** | **Test Case Run Time (s.)** | | | | | |
|---|---|---|---|---|---|---|
| Algorithm: | CANNON | | SUMMA | | $y \leftarrow Ax$ | |
| Matrix size, $n$: | 6480 | 20160 | 6480 | 20160 | 6480 | 20160 |
| 1 | 1.05 | 26.08 | 1.05 | 26.10 | 0.0070 | 0.0488 |
| 4 | 3.25 | 54.14 | 2.03 | 46.19 | 0.0219 | 0.0677 |
| 9 | 1.50 | 36.13 | 1.46 | 35.71 | 0.0143 | 0.0430 |
| 16 | 1.53 | 39.59 | 1.52 | 39.38 | 0.0052 | 0.0347 |
| 25 | 1.11 | 36.87 | 1.05 | 32.12 | - | - |
| 36 | 0.78 | 18.33 | 0.78 | 18.32 | 0.0035 | 0.0201 |
| 64 | 0.49 | 10.71 | 0.50 | 10.72 | - | - |
| 81 | 0.41 | 8.63 | 0.42 | 8.72 | 0.0033 | - |
| 100 | 0.35 | 7.07 | 0.36 | 7.25 | - | - |
| 144 | 0.27 | 5.21 | 0.29 | 5.42 | 0.0019 | 0.0124 |
| 225 | 0.18 | 3.45 | 0.20 | 3.64 | - | - |
| 256 | 0.16 | 3.09 | 0.19 | 3.30 | - | - |

**Table 1: Strong scaling**: Test case run time in seconds, for 1 to 256 processes, at fixed matrix size $n$.

| **Num. Processes, $p$** | **Test Case Run Time (s.)** | | | | | |
|---|---|---|---|---|---|---|
| Algorithm: | CANNON | | SUMMA | | $y \leftarrow Ax$ | |
| Block matrix size, $n_{local}$: | 1260 | 5000 | 1260 | 5000 | 1260 | 5000 |
| 1 | 0.02 | 0.49 | 0.02 | 0.49 | 0.0012 | 0.0054 |
| 4 | 0.14 | 7.51 | 0.11 | 6.20 | 0.0180 | 0.0240 |
| 9 | 0.65 | 43.18 | 0.36 | 15.15 | 0.0068 | - |
| 16 | 0.78 | 38.78 | 0.77 | 38.49 | 0.0038 | 0.0334 |
| 25 | 1.02 | 48.38 | 0.94 | 46.14 | 0.0051 | 0.0343 |
| 36 | 1.15 | 58.39 | 1.16 | 58.39 | 0.0142 | - |
| 64 | 1.55 | 78.72 | 1.58 | 78.63 | - | 0.0475 |
| 81 | 1.74 | 88.43 | 1.78 | 88.64 | 0.0118 | - |
| 100 | 1.93 | 98.15 | 2.00 | 99.06 | 0.0051 | 0.0481 |
| 144 | 2.37 | 119.07 | 2.49 | 120.22 | 0.0056 | - |
| 225 | 2.89 | 148.08 | 3.05 | 150.13 | 0.0091 | - |
| 256 | 3.11 | 158.37 | 3.30 | 160.68 | - | - |

**Table 2: Weak scaling**: Test case run time in seconds, for 1 to 256 processes and fixed **block** matrix size $n_{local}$.

In the strong scaling plot all algorithms are far from achieving perfect strong scaling, with speedup per process of c. 2% for $n$ of 6480 and c. 3% for n of 20160. The poor scaling is also obvious in the timing data (Table 1), where the run time decreases much slower than $1/p$. It is noted that the scaling is slightly better for $n$ of 20160, likely because increasing $n$ at fixed $p$ increases the size of the parallel calculation per process i.e. $f_s$ decreases when $n$ is increased at fixed $p$. This results in a process with better strong scaling. The effect of increasing $n$ is modest, however; athough $n$ increased by a factor of 3, the speedup only increased by c. 20% whereas the work of each local matrix multiplication increased by a factor of $3^3$ (assuming the local multiplication is $O(n^3)$) which suggests increasing $n$ had only a minor effect on $f_s$. This would suggest that the total run time is not dominated by the local calculation, but instead by other factors such as the speed of the collective communications used in the implementation, or by the available bandwidth between processes that may be on the same or different nodes. It would be interesting to study this further by decomposing the run time into calculation & communication time

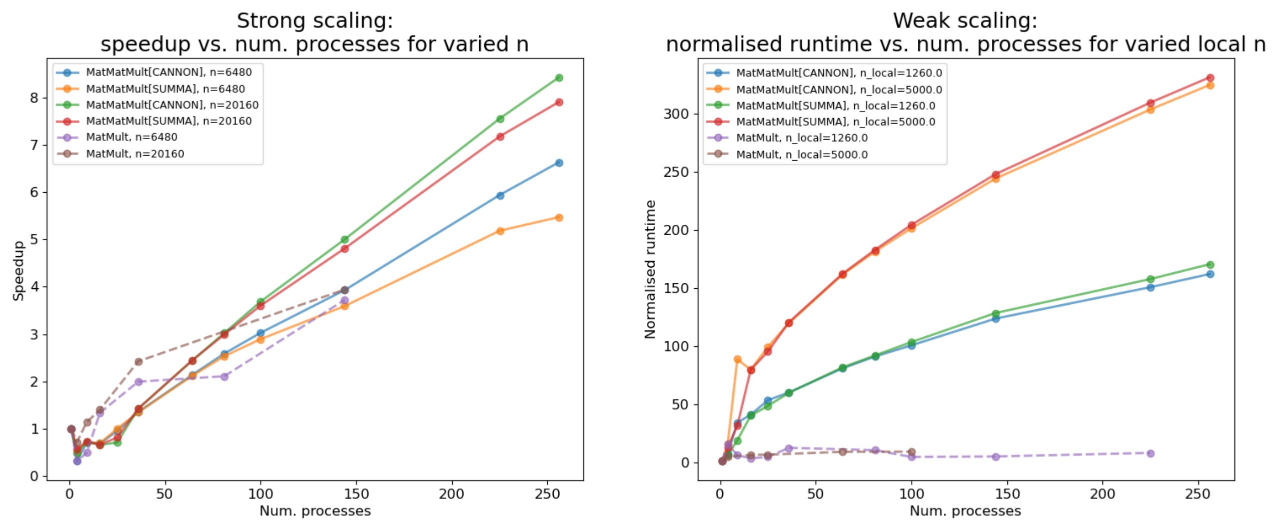

**Figure 1:** Speedup in the strong case (left) and normalised runtime in the weak case (right), as a function of the number of processes, $p$. **Left:** sub-optimal strong scaling was seen and increasing $n$ 3-fold gave only a modest improvement, likely due to the runtime being dominated by communication speed rather than calculation speed. **Right:** normalised runtime ($t(p)/t(1)$) scales as $\sqrt{p}$ rather than being constant as predicted by the weak scaling model. This demonstrates a limitation of the weak scaling model - although problem size per process may be constant, the actual work done by each process grows as $\sqrt{p}$ because each process performs $\sqrt{p}$ multiplications.

to observe how each scales with $p$. This also suggests that better strong scaling may have been observed had significantly larger $n$ been tested.

The weak scaling results are shown in terms of the total runtime in Fig. 1 (right) and clearly illustrate that the implementations do not show weak scaling in the sense described in 3.1. The runtime for both matrix-matrix multiplication algorithms scales as $\sqrt{p}$ despite the size of the problem per process $n_{local} = n/\sqrt{p}$ being fixed. It is not surprising that the total runtime scales as $\sqrt{p}$ because, as can be seen in Alg. 2 and Alg. 3 the number of local matrix multiplications (i.e. the number of loop iterations) performed by each process is $\sqrt{p}$. This highlights a limitation of the weak scaling model; while the size of the problem per process is fixed, the actual work done per process is not fixed since the processes must communicate with each other. The weak scaling model assumes the processes are independent of each other and that there is no overhead associated with added parallelism i.e. the work done by each process does not depend on the number of processes. This is clearly not the case here, since each process must do more total work when there are more processors.

It is interesting to observe from Table 2 that the Cannon and SUMMA runtimes increased by a factor of c. 50 when $n_{local}$ was increased by a factor of c. 4, which implies that the OpenBLAS matrix multiplication calculation scales as $O(n^{2.8})$ rather than $O(n^3)$. This is not surprising, as one might expect the library to be using an optimised algorithm such as Strassen's algorithm, rather than the naive $O(n^3)$ algorithm, for matrix-matrix multiplication.

## 4 Conclusion

Parallel implementations of matrix-vector multiplication $y \leftarrow Ax$ and matrix-matrix multiplication $C \leftarrow AB + C$ by Cannon's Algorithm and SUMMA have been described and their strong & weak scaling performance measured.

Poor strong scaling was observed (Fig. 1, left) with speedup per process c. 2%, only improving to 3% when $n$ was increased 3-fold. This poor strong scaling was attributed to the calculation time being dominated by communication between processes rather than calculation within processes, although this was not measured directly. It is hypothesized a significantly larger $n$ would be required for more efficient strong scaling to be observed.

In the weak scaling case, the implementations did not show the constant runtime predicted by the weak scaling model (Fig. 1, right); instead the Cannon & SUMMA runtime for fixed block matrix size $n_{local}$ grew as $\sqrt{p}$ due to each process having to perform $\sqrt{p}$ local matrix calculations. This highlights a limitation of the weak scaling model; that while the problem size per process is fixed, the total work done per process is not since the processes must exchange data between themselves - the work per process scales as $\sqrt{p}$ despite the fixed size of the block matrices.