CMOR 421/521, Homework #3: LaTeX Submission

amc50

April 12, 2024

1 Compilation

1.1 Accessing NOTS Cluster

All of the following processes and results were run on the NOTS Cluster. The compilation process is as follows.

For this assignment testing the two algorithms with large matrices is necessary, and as such the process cannot be run on a login node. In order to avoid being kicked off and the job killed using an interactive node is the best practice.

Command used to run interactive node with multiple tasks:

```
[amc50@bc8u27n1 homework-3]$ srun --pty --partition=interactive --reservation=cmor421
--ntasks=16 --mem-per-cpu=1G --time=00:30:00 $SHELL
```

Once logged into NOTS, the modules `GCC` and `OpenMPI` must be loaded. It is important to note that the loading of these modules is order dependent. If `GCC` is not loaded first, `OpenMPI` is not available.

Commands used to load modules needed for compilation:

```
[amc50@nlogin3 homework-3]$ module load GCC/13.2.0

[amc50@nlogin3 homework-3]$ module load OpenMPI/4.1.6
```

For this assignment, there is significant overlap between the functions used in the SUMMA and Cannon's algorithm implementations. As such, for convenience, I created a header file to include in both. Additionally, for this project, there are now two driver files—one for each implementation.

Command used to compile SUMMA algorithm implementation:

```
[amc50@nlogin3 homework-3]$ mpic++ -I include -o summa summa.cpp
src/matrix_operations.cpp
```

To execute the program, the number of processors and the matrix size must be specified. For this demonstration, 4 processors were used, and the matrix size was set to 2048 to allow testing with sufficiently large matrices.

Command used to run SUMMA algorithm mplementation:

```
[amc50@nlogin3 homework-3]$ mpirun -np 4 ./summa 2048
```

The same requirements as previously mentioned are necessary for the implementation of Cannon's algorithm. However, for Cannon's algorithm the number of processors was increased to 16.

Command used to compile Cannon's algorithm implementation:

```
[amc50@bc8u27n1 homework-3]$ mpic++ -I include -o cannon cannon.cpp
src/matrix_operations.cpp
```

Command used to run Cannon's algorithm implementation:

```
[amc50@nlogin3 homework-3]$ mpirun -np 16 ./cannon 2048
```

2 Introduction

As quickly referenced in the compilation section there is quite a bit of overlap between the implementation and testing of SUMMA and Cannon's algorithm. The shared components are described below.

2.1 Assumptions

Several assumptions are defined to simplify the implementation of the matrix-matrix multiplication algorithms:

1. The number of processors s is set to $p \times p$, where p is an integer. This shaping allows the creation of a square grid of processors.
2. The matrices involved in the multiplication are square matrices of size $n \times n$, and the block size for distribution across processors is defined as $b = \frac{n}{p}$. This ensures that each processor receives a sub-matrix of size $b \times b$, facilitating even storage and efficient parallel computation.
3. Each rank is responsible for storing and computing a single block of the matrices. Specifically, rank 0 stores the blocks A_{00}, B_{00}, and C_{00}. More generally, the processor at the i^{th} row and j^{th} column of the processor grid stores the blocks A_{ij}, B_{ij}, and C_{ij}. This assignment aids in how the information is stored and distributed for efficient memory accessing.

2.2 Matrix Generation

In order to test each of the implementations the result is compared to a serial matrix-matrix multiplication routine. For the purpose of this assignment the serial routine was selected to be blocked matrix-matrix multiplication, as its innermost loop is what each processor does for the local matrix-matrix multiplication.

As such, for this comparison matrices A and B must be randomly generated. This was accomplished using the libraries `cmath` and `ctime`. To prevent the regeneration of identical matrices in successive runs due to the pseudo-random nature of number generation, the random number generator was seeded. The `ctime` library provided the current time as a seed, allowing for the generation of different matrices in each run. Additionally, the elements of the matrices were defined as doubles in the range $[-\text{RAND\_MAX}, \text{RAND\_MAX}]$, where RAND_MAX is defined as the maximum value producible by `rand()`.

2.3 Scatter and Gather

For both matrix-matrix multiplication algorithms after randomly generating matrices A and B on the root rank, the relevant blocks need to be scattered to their specific processors. This process requires reshaping the matrix into an array where the blocks are stored as contiguous chunks of memory of size $b \times b$. Additionally, after the parallel matrix-matrix multiplication is completed, the results need to be gathered from the processors back to the root rank and unscrambled.

2.3.1 Scattering Process

In order to scatter the matrix, first the reshaping is completed by the function `convert_matrix`. This function reorders the elements of the matrix to be scattered into a temporary array, where each segment of size $b \times b$ corresponds to a different processor. This is effective as the processor grid rank is defined left to right and from top to bottom. As such, creating a row and column index for the matrix block can be used to access the specific elements going to a processor. By iterating through the elements within a block row index i_b and column index j_b, the absolute position in the flattened array can be determined. This system of indexing is defined by $index = (i + i_b * b) \times n + (j + j_b)$. Each element is then copied to the corresponding position in the temporary array, maintaining block continuity in memory. Once the matrix to be scattered is formed, the root rank uses `MPI_Scatter` with the `sendcount` being $b \times b$.

2.3.2 Gathering Process

The gathering operation is the exact opposite process of scattering and done by the function `revert_matrix`. When using `MPI_Gather` to collect the local C matrices at the root rank, they are ordered according to the ranks from which they originate. As such, the same indexing strategy used for scattering can be reversed and will give the elements original position within the matrix. The reassembly creates a final gathered matrix that has correctly combined the outputs of all the processor's results.

2.4 Implementation Check

In order to check the correctness of the two algorithms the results are compared to the serial method of blocked matrix-matrix multiplication. This process was more thoroughly defined for Homework 1, but a quick overview will be described here. The resulting C matrix from the serial implementation is checked to see if it is identical to the parallel C up to machine precision. This process utilizes an element-wise comparison and the absolute difference between each element is added to a sum. If the sum exceeds $1 \times 10^{-15} \times n$, where n is the size of the matrix, the matrices are not identical. As seen for both implementations, the matrices matched.

3 SUMMA Algorithm

The Scalable Universal Matrix Multiplication Algorithm (SUMMA) strategically reduces memory demands by partitioning the matrix multiplication workload among processors that compute outer products. The efficiency of the algorithm is enhanced through the use of custom `MPI` communicators, which help minimize communication overhead and focus much of the operational complexity on problem setup.

3.1 Setup

The implementation involves constructing custom communicator groups specific to the needs of the matrix operations. This is achieved by converting rows and columns in the processor grid into row and column communicator groups, respectively, using `MPI_Comm_split`. By establishing these groups, processors can broadcast data more efficiently by limiting communication to relevant row or column groups instead of the entire processor grid.

3.2 Implementation

After the communicator groups are constructed, on the k-th iteration the k element of each communicator group follows this procedure:

1. For row communicator, broadcast k element along the row using `MPI_Bcast`.
2. For column communicator, broadcast k element along the column using `MPI_Bcast`.

Each processor then computes its local matrix-matrix value of C. After completing p iterations the local C matrices are gathered at the root rank.

This strategic use of communicators and focused broadcasting operations in SUMMA significantly enhances performance by reducing the frequency and volume of data transfers, making it particularly effective for large-scale matrix multiplication tasks.

3.3 Method Generalization

As the Scalable Universal Matrix Multiplication Algorithm (SUMMA) name implies, extending the implementation to a $p \times p$ block matrix is very straightforward. The primary requirement is to adjust the number of processors to a perfect square of size $p \times p$ and ensure that the block size $b = \frac{n}{p}$, where n is size of the matrix, is an integer, to match the original implementation assumptions. The reason that it is so convenient to scale is that on iteration k, the algorithm broadcasts the k-th element within a communicator group to the rest of that group. Consequently, this group is defined to be of length p, matching the grid's dimensions without further modifications. During testing, increasing the number of processors to 16 resulted in a successful execution on a 4×4 grid, with $p = 4$.

4 Cannon's Algorithm

After the scattering of the blocks to each process, Cannon's algorithm has two phases: an initial skewing phase and a computation phase involving shifting and matrix-matrix multiplication. Unlike SUMMA, Cannon's algorithm uses point-to-point communication.

4.1 Initial Skewing Phase

In the initial skewing phase, the blocks of matrix A are shifted leftwards, with the i-th row of blocks being shifted i times. Simultaneously, the blocks of matrix B are shifted upwards, with the j-th column of blocks being shifted j times.

4.2 Shifting and Multiplication Phase

After skewing, the algorithm utilizes point-to-point communication between processors in a grid for p iterations. Each processor, designated by rank i, follows this procedure:

- Computes local matrix-matrix multiplication and adds to local C.
- Sends its local block of A to the processor on its left.
- Sends its local block of B to the processor above it.
- Receives the corresponding block of A from the processor on its right.
- Receives the corresponding block of B from the processor below it.

This circular shifting requires that if the processor is in column 0, it should send data to the processor in the last column, p, for row shifting. Similarly, if it is in row 0, it should send data to the processor in the last row, also p, for column shifting.

To ensure that the local blocks A and B are not overwritten during point-to-point communication, the MPI function `MPI_Sendrecv_replace` is used. This method is advantageous because it combines send and receive operations into a single non-blocking call, thus avoiding potential deadlocks and ensuring safe data transfer. After all iterations, the results are gather into the final matrix in the root rank.