CSE 6220 / CX 4220 – Introduction to High-Performance Computing

PA3 Report
Due Monday, April 14, 2025

Authors:
Tamas Mester & Dylan Keskinyan

1 Distribute Function

The purpose of the distribute function is to create a 2D block partition of the nonzero elements across the processor grid. From Ed we know that we can assume a normal distribution for the input grids and from the assignment we are told that the processors will always be organized in a $\sqrt{p} \times \sqrt{p}$ grid. We also know that we can use MPICartget to retrieve processor coordinates and grid dimensions. The root processor created a vector of vectors to organize the elements into a list for each destination. Then it went through each element and used the following to determine the destination:

$$\text{proc\_row} = \left\lfloor \frac{i \cdot \text{dims}[0]}{m} \right\rfloor, \quad \text{proc\_col} = \left\lfloor \frac{j \cdot \text{dims}[1]}{n} \right\rfloor$$

Next, the root process packs entries into buffers and sends them with MPISend. The non-root processes receive data with MPIRecv. We know this accomplishes our goals because we are told that we can assume a uniform distribution so we can assume that each processor will get similar amounts of data.

2 Implementation of the 2D SUMMA SpGeMM Algorithm

We designed our SpGeMM algorithm to follow the sparse SUMMA algorithm based on a 2D processor grid. Here are some of the important details and steps of the algorithm:

2.1 Setup

Processors are split into row and column groups that are in a $\sqrt{p} \times \sqrt{p}$ grid like explained above. Matrix A (dimensions $m \times p$) is partitioned into dims[0] row blocks. Matrix B (dimensions $p \times n$) is partitioned into dims[1] column blocks. The elements are distributed in accordance to the function explained above.

2.2 SUMMA Communication Pattern

We followed the communication pattern and strategy in the pseudocode given in the PA3 instructions. Basically, for each iteration from 0 to p we want to broadcast the appropriate column to the rest of the row and the appropriate row to the rest of the column. This is shown below with the graphics from the instructions given.

```
On each process P(i, j):
let block_size = n / proc_rows (assuming proc_rows = proc_cols)

for(int k = 0; k < proc_rows; ++k) {
  Bcast block column k of A (A[i*block_size:(i+1)*block_size - 1,
              k*block_size:(k+1)*block_size- 1]) within proc row i
  Bcast block row k of B (B[k*block_size:(k+1)*block_size - 1,
              j*block_size:(j+1)*block_size - 1]) within proc column j
  Do C += block of A * block of B
```

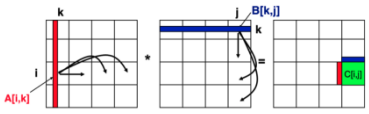

Figure 1: Pseudocode for blocked SUMMA on a square processor grid and associated illustration.

2.3 Communication Buffer

In order to send the data with MPIIbcast we needed to convert the `pair<pair<int,int>,int>` type into three contiguous ints in a vector and concatenate these with the rest of the data in both the A and B blocks that we wanted to send. This is because MPI is unable to send the std::pair data type.

2.4 Local Sparse Multiplication

An MPIWaitAll is used to ensure that all the processors have received the appropriate A and B blocks before they continue with local sparse multiplication. Additionally, before the local sparse multiplication the processors need to unpack the buffers that were used to send the data by doing the opposite of the operation defined in step 3.

Now each processor builds a map and preprocesses B into the map in order to group them by their row index. This allows for quicker lookups when iterating over the elements in A. Now we iterate over the elements in A and find the corresponding elements in B that have the same row as the column of the element in A. Then we uses the semiring operations to multiply and store the results into the temporary C data.

After all the multiplication, we sort the temporary C data by the row and column indices and use the plus operation to add elements with the same coordinates. If the coordinates don't exit yet in the local C matrix, then we insert them directly.

2.5 Convert to COO format

The last step is to convert the local C data from a map to COO format as required by the instructions.

3 APSP Algorithm using SpGeMM

We were able to solve the All-Pairs Shortest Path (APSP) problem with SpGEMM by doing repeated squaring. Essentially, the path length was doubled in each iteration with SpGEMM until

all of the paths are explored. We were able to change the plus semiring operation to a min operation that would help keep track of the shortest possible path. The times operation was changed to plus to add edge weights in the path. The initial matrix L is the adjacency matrix, and the final result holds shortest-path weights.

4 Performance & Scalability of SpGEMM & APSP

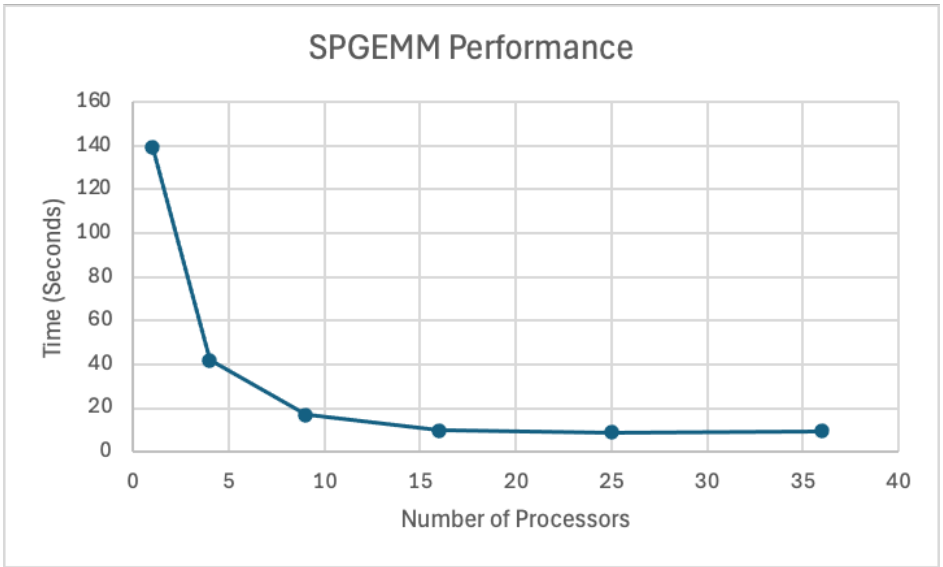

Figure 2: Performance of the 2D SUMMA SpGeMM Algorithm

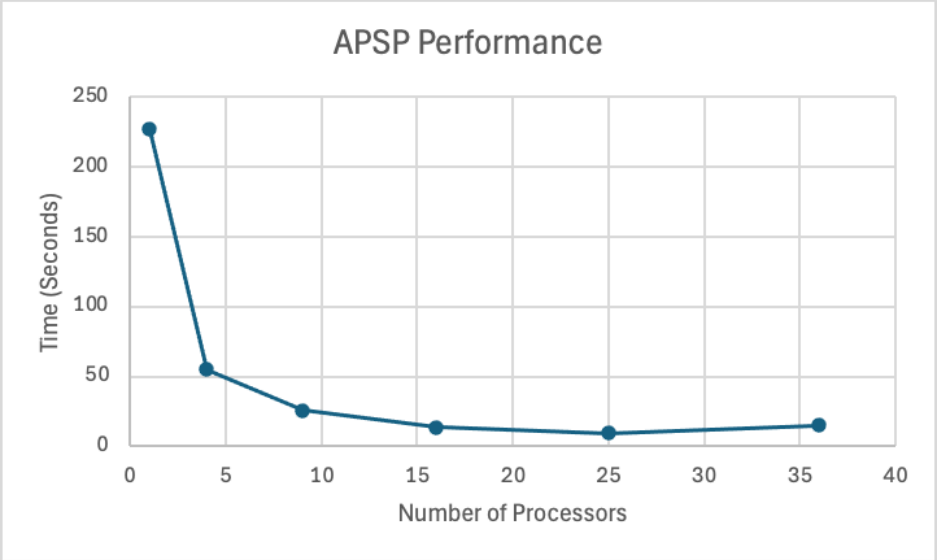

Figure 3: Performance of the APSP algorithm using SpGeMM