Kennesaw State University

HPC & Parallel Programming

Project - MPI

Instructor: Kun Suo
Points Possible: 100
Difficulty: ★★★★☆

Part 1: (30 points)

General matrix multiplication (GEMM) is typically defined as:

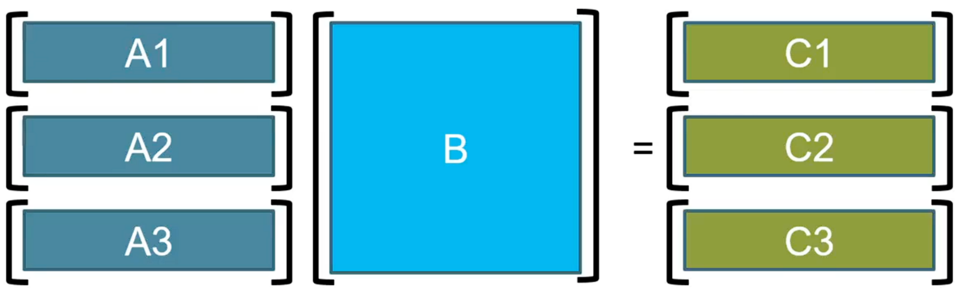
$$C = AB$$

$$C_{m,n} = \sum_{n=1}^{N} A_{m,n} B_{n,k}$$

where A is an m*n matrix and B is an n*k matrix.

Problem: Implement a parallel solution using MPI for general matrix multiplication

- Input: Three integers M, N, and K (512 ~ 2048)
- Problem Description: Randomly generate two matrices A (M*N) and B (N*K), and perform matrix multiplication to obtain matrix C.
- Output: The time taken for the matrix calculation.



In your implementation, you must include a verification function that checks whether the output matrix C produced by the MPI optimization is identical to the original matrix C. (Note that if your matrix elements are floating-point numbers, elements within a certain tolerance can be considered the same. You can set a very small threshold (Epsilon), such as 10^{-9}. The equality check formula is: $|a - b| < \varepsilon$. If the difference between a and b is less than this small value, we consider them "equal".) The time of verification function should not be included in the parallel solution.

Hint: The main thread divides matrix A into blocks based on the number of threads, distributing one block to each thread. It also sends the entire matrix B to each thread. After each thread finishes computing its assigned block, it sends the result back to the main thread. The main thread computes the last block and then aggregates all the results.

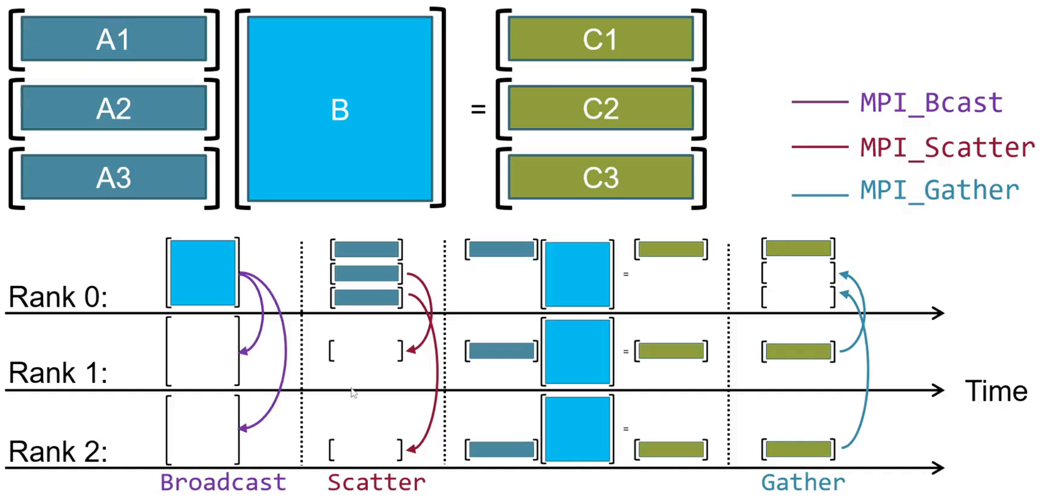
Part 2: (30 points)

Implement inter-process communication in matrix multiplication using

(a) MPI point-to-point communication (e.g., MPI_Send/MPI_Receive)
(b) MPI collective communication (e.g., MPI_Reduce/MPI_Allreduce/MPI_Broadcast, etc.)

and compared the performance of the two implementations.

Order of Matrix	1024	2048	4096	8192
MPI point-to-point communication				
MPI collective communication				



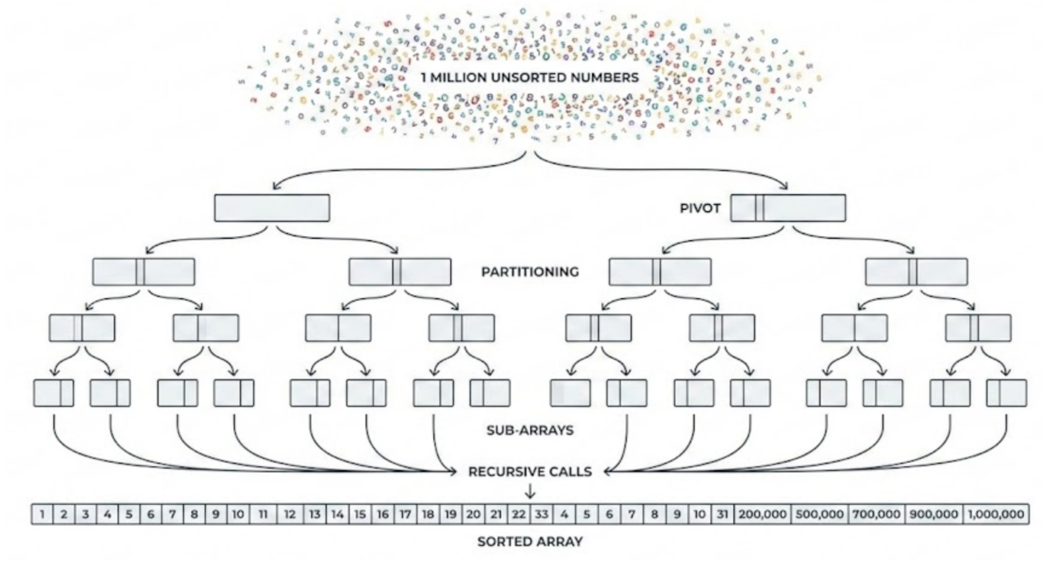
In your implementation, you must include a verification function that checks whether the output matrix C produced by both MPI optimization (a) and (b) are identical to the original matrix C. (Note that if your matrix elements are floating-point numbers, elements within a certain tolerance can be considered the same. You can set a very small threshold (Epsilon), such as 10^{-9}. The equality check formula is: $|a - b| < \varepsilon$. If the difference between a and b is less than this small value, we consider them "equal".) The time of verification function should not be included in the parallel solution.

Hint: Idea in (a) is the same as part 1. Idea in (b) is to divide Matrix A into blocks and sent to each process, while matrix B is broadcast to all processes. After the calculations are completed, the results are aggregated.

Part 3: (40 points)

https://github.com/kevinsuo/CS4522/blob/main/data.txt

The above link is a file which contains 1 million unsorted numbers.



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int data[1000000];

void swap(int* a, int* b)
{
          int t = *a; *a = *b; *b = t;
```

```c
}

int partition (int arr[], int low, int high)
{
          int pivot = arr[high];       // pivot
          int i = (low - 1);                        // Index of smaller element

          for (int j = low; j <= high- 1; j++)
          {
                    if (arr[j] < pivot)
                    {
                             i++;    // increment index of smaller element
                             swap(&arr[i], &arr[j]);
                    }
          }
          swap(&arr[i + 1], &arr[high]);
          return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
          if (low < high)
          {
                    int pi = partition(arr, low, high);
                    quickSort(arr, low, pi - 1);
                    quickSort(arr, pi + 1, high);
          }
}

void printArray(int arr[], int size)
{
          int i;
          for (i=0; i < size; i++)
                    printf("%d\n", arr[i]);
}

int main()
{
          //read the unsorted array
          char str[100];
          int count = 0;
          struct timespec start, end;
          FILE* fp = fopen("data.txt", "r");
          while (fscanf(fp, "%s", str) != EOF) {
                    data[count] = atoi(str);
                    count++;
          }

          //quick sort the array
          clock_gettime(CLOCK_MONOTONIC, &start);
          quickSort(data, 0, count - 1);
          clock_gettime(CLOCK_MONOTONIC, &end);

          u_int64_t diff = 1000000000L * (end.tv_sec - start.tv_sec) + end.tv_nsec -
start.tv_nsec;
          printf("elapsed time = %llu nanoseconds\n", (long long unsigned int) diff);

//        printArray(data, count);

          fclose(fp);

          return 0;
}
```

---

https://github.com/kevinsuo/CS4522/blob/main/quicksort.c

The above source code file quicksort.c is a Divide and Conquer algorithm which sorts the above 1 million unsorted numbers. However, the program executes in the sequential implementation. Please write a parallel program quick sort using MPI based on this sequential solution. Compare the parallel program execution time with the sequential version and write a report with data and figures introducing your implementation. (Hint: MPI_Send and MPI_Recv are required.)

In your implementation, you must include a verification function that checks whether the output array produced by MPI is identical to the array solved by sequential method. The time of verification function should not be included in the parallel solution.

Submission

Submit your assignment file through D2L using the appropriate link.
The submission must include the ***source code***, and ***a report describe your code logic. Output screenshot of your code, Figures & Tables*** should be included in the report.