CSE455

High-Performance Computing

Project Report

Marwan Atef Hamed Ali Mohamed 18P8678
TEAM 139

Openmp:

```
#include <iostream>
#include <omp.h>
#include <cmath>
#include <vector>
#include <cstdlib>

using namespace std;

int main() {
    int n, p;
    cout << "Enter the size of the matrices (n): ";
    cin >> n;
    cout << "Enter the number of processes (p): ";
    cin >> p;

    int sqrt_p = sqrt(p);
    int block_size = n / sqrt_p;

    vector<vector<int>> A(n, vector<int>(n));
    vector<vector<int>> B(n, vector<int>(n));
    vector<vector<int>> C(n, vector<int>(n));

    srand(time(NULL));

    // Initializing A,B with random numbers
#pragma omp parallel for collapse(2) num_threads(p)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = rand() % 5;
            B[i][j] = rand() % 5;
            C[i][j] = 0;
        }
    }

    // Printing matrices A and B
#pragma omp master
    {
        cout << "matrix A: " << endl;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cout << A[i][j] << " ";
            }
            cout << endl;
        }

        cout << "matrix B: " << endl;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cout << B[i][j] << " ";
            }
            cout << endl;
        }
    }

    double start_time, end_time;
```

```
#pragma omp parallel num_threads(p)
    {
        // Record the start time
#pragma omp master
        {
            start_time = omp_get_wtime();
        }

        int thread_id = omp_get_thread_num();
        int row = thread_id / sqrt_p;
        int col = thread_id % sqrt_p;

        for (int p = 0; p < sqrt_p; p++) {
            // Iterate over block size of matrix B instead of n
            for (int i = 0; i < block_size; i++) {
                for (int j = 0; j < block_size; j++) {
                    int temp = 0;
                    for (int k = 0; k < block_size; k++) {
                        temp += A[row * block_size + i][k + p * block_size] * B[k +
p * block_size][(col * block_size + j)];
                    }
                    // No need for atomic operation as each thread is working on a
unique block of matrix C
                    C[row * block_size + i][col * block_size + j] += temp;
                }
            }

            // Rotate blocks of matrices A and B
#pragma omp barrier
#pragma omp master
            {
                // Rotate matrix B
                for (int i = 0; i < block_size; i++) {
                    int temp = B[p * block_size + i][(col + 1) * block_size - 1];
                    for (int j = (col + 1) * block_size - 1; j > col * block_size;
j--) {
                        B[p * block_size + i][j] = B[p * block_size + i][j - 1];
                    }
                    B[p * block_size + i][col * block_size] = temp;
                }
                // Rotate matrix A
                for (int i = 0; i < block_size; i++) {
                    int temp = A[(row + 1) * block_size - 1][p * block_size + i];
                    for (int j = (row + 1) * block_size - 1; j > row * block_size;
j--) {
                        A[j][p * block_size + i] = A[j - 1][p * block_size + i];
                    }
                    A[row * block_size][p * block_size + i] = temp;
                }
            }
#pragma omp barrier

        }
#pragma omp master
        {
            // Record the end time
            end_time = omp_get_wtime();
```

```
            cout << "resultant matrix C: " << endl;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    cout << C[i][j] << " ";
                }
                cout << endl;
            }
        }
    }

    // Calculate and print the time taken
    double time_taken = end_time - start_time;
    cout << "Time taken for parallel Cannon's multiplication algorithm: " <<
time_taken << " seconds" << endl;

    return 0;
}
```

Description and explanation:

This code is an implementation of the parallel Cannon's matrix multiplication algorithm using OpenMP in C++.
The parallel Cannon's matrix multiplication algorithm works by dividing the matrices into smaller blocks and performing matrix multiplication on these blocks in parallel across multiple threads. After each multiplication step, the blocks of matrices A and B are rotated along their respective rows and columns. This process is repeated for a total of `sqrt(p)` times where `p` is the number of OpenMP threads.

1. Include statements and namespace: The code includes necessary header files and uses the `std` namespace.

```
#include <iostream>
#include <omp.h>
#include <cmath>
#include <vector>
#include <cstdlib>

using namespace std;
```

2. Matrix size and number of processes: The program reads matrix size `n` and the number of OpenMP threads `p` from the user.

```
int n, p;
cout << "Enter the size of the matrices (n): ";
cin >> n;
cout << "Enter the number of processes (p): ";
cin >> p;
```

3. Initialize matrices: The code calculates the square root of `p` and the block size of the matrices. It then initializes matrices A, B, and C using `vector<vector<int>>`.

```
int sqrt_p = sqrt(p);
int block_size = n / sqrt_p;

vector<vector<int>> A(n, vector<int>(n));
vector<vector<int>> B(n, vector<int>(n));
vector<vector<int>> C(n, vector<int>(n));

srand(time(NULL));

// Initializing A,B with random numbers
#pragma omp parallel for collapse(2) num_threads(p)
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      A[i][j] = rand() % 5;
      B[i][j] = rand() % 5;
      C[i][j] = 0;
   }
}
```

4. Print matrices A and B: The code prints the initial matrices A and B using a single master thread.

```
#pragma omp master
{
  // code to print matrix A and B
}
```

5. Parallel Cannon's algorithm: The main part of the code that contains the implementation of the parallel Cannon's matrix multiplication algorithm using OpenMP. **(will be explained in details in the next section)**

```
#pragma omp parallel num_threads(p)
{
  // Record the start time
  #pragma omp master
  {
    start_time = omp_get_wtime();
  }

  // code for the Cannon's algorithm, including block size iteration, matrix multiplication, and rotations

  #pragma omp master
  {
    // Record the end time
    end_time = omp_get_wtime();
    // print resultant matrix C
  }
}
```

```

6. Calculate and print the time taken: Finally, the code calculates the time taken for the parallel Cannon's algorithm and prints it.

```
double time_taken = end_time - start_time;
cout << "Time taken for parallel Cannon's multiplication algorithm: " << time_taken << " seconds" << endl;
```

---

The main idea behind Cannon's algorithm is to divide the matrices into smaller blocks and perform block-wise matrix multiplication in parallel. In this implementation, the algorithm uses OpenMP to parallelize the multiplication process.

Thread ID, row, and column calculation: Each thread calculates its own ID, row, and column based on the square root of the number of processes sqrt_p.

```
int thread_id = omp_get_thread_num();
int row = thread_id / sqrt_p;
int col = thread_id % sqrt_p;
```

Block-wise matrix multiplication and rotation: The algorithm iterates over sqrt_p steps. In each step, it performs block-wise matrix multiplication and then rotates the blocks of matrices A and B.

```
for (int p = 0; p < sqrt_p; p++) {
  // Block-wise matrix multiplication
  for (int i = 0; i < block_size; i++) {
    for (int j = 0; j < block_size; j++) {
      int temp = 0;
```
```

```
        for (int k = 0; k < block_size; k++) {
            temp += A[row * block_size + i][k + p * block_size] * B[k + p * block_size][(col * block_size + j)];
        }
        C[row * block_size + i][col * block_size + j] += temp;
    }
}

// Rotate blocks of matrices A and B
#pragma omp barrier
#pragma omp master
{
    // Rotate matrix B
    for (int i = 0; i < block_size; i++) {
        int temp = B[p * block_size + i][(col + 1) * block_size - 1];
        for (int j = (col + 1) * block_size - 1; j > col * block_size; j--) {
            B[p * block_size + i][j] = B[p * block_size + i][j - 1];
        }
        B[p * block_size + i][col * block_size] = temp;
    }
    // Rotate matrix A
    for (int i = 0; i < block_size; i++) {
        int temp = A[(row + 1) * block_size - 1][p * block_size + i];
        for (int j = (row + 1) * block_size - 1; j > row * block_size; j--) {
            A[j][p * block_size + i] = A[j - 1][p * block_size + i];
        }
        A[row * block_size][p * block_size + i] = temp;
    }
}
#pragma omp barrier
```

}

```

In the block-wise matrix multiplication part, each thread calculates the product of its corresponding blocks in matrices A and B, and accumulates the result in matrix C.

After the block-wise multiplication, the program rotates the blocks of matrices A and B. The rotation is done in such a way that each block in a row of matrix A is shifted one step left, and each block in a column of matrix B is shifted one step up. This rotation is done by the master thread, while other threads wait at the barrier.

The algorithm repeats the block-wise multiplication and rotation steps for a total of sqrt_p times. At the end of these iterations, the product of matrices A and B is stored in matrix C.

## Screenshots:

Input:

```
Enter the size of the matrices (n): 4
Enter the number of processes (p): 4
```

Output:

```
matrix A:
2 2 2 0
1 4 4 3
1 4 4 3
1 4 4 3
matrix B:
1 1 4 2
2 0 4 3
2 0 4 3
2 0 4 3
resultant matrix C:
10 2 24 16
23 1 48 35
23 1 48 35
23 1 48 35
Time taken for parallel Cannon's multiplication algorithm: 9.20007e-06 seconds
```

Time taken is 0.00000920007 seconds.
```

MPI:

```
#include <iostream>
#include <mpi.h>
#include <math.h>
#include <cstdlib>
#include <ctime>
using namespace std;


void print_complete_matrix(int* matrix, int block_size, int n) {
	for (int row_block = 0; row_block < n; ++row_block) {
		for (int i = 0; i < block_size; ++i) {
			for (int col_block = 0; col_block < n; ++col_block) {
				int base_idx = (row_block * n + col_block) * block_size *
block_size + i * block_size;
				for (int j = 0; j < block_size; ++j) {
					cout << matrix[base_idx + j] << " ";
				}
				cout << " ";
			}
			cout << endl;
		}
		cout << endl;
	}
}

int main(int argc, char* argv[])
{
	int rank, size;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);


		// Check if the number of processes is a perfect square
		int sqrt_size = sqrt(size);
	if (sqrt_size * sqrt_size != size) {
		cerr << "Error: The number of processes must be a perfect square." <<
endl;
		MPI_Finalize();
		return 1;
	}

	srand(time(NULL));

	int n = sqrt(size);
	int block_size = atoi(argv[1]) / n;
	int* A = new int[block_size * block_size];
	int* B = new int[block_size * block_size];
	int* C = new int[block_size * block_size];
	for (int i = 0; i < block_size * block_size; i++) {
		A[i] = rand() % 5;
		B[i] = rand() % 5;
		C[i] = 0;
	}
```

```
	// Create Cartesian topology
	int dims[2] = { n, n };
	int periods[2] = { 1, 1 };
	int reorder = 0;
	MPI_Comm comm_cart;
	MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &comm_cart);

	// Get Cartesian coordinates for this process
	int coords[2];
	MPI_Cart_coords(comm_cart, rank, 2, coords);

	double start_time, end_time;

	// Perform initial alignment of blocks
	int rank_source, rank_dest;

	MPI_Cart_shift(comm_cart, 1, coords[0], &rank_source, &rank_dest);
	MPI_Sendrecv_replace(A, block_size * block_size, MPI_INT, rank_dest, 0,
rank_source, 0, comm_cart, MPI_STATUS_IGNORE);

	MPI_Cart_shift(comm_cart, 0, coords[1], &rank_source, &rank_dest);
	MPI_Sendrecv_replace(B, block_size * block_size, MPI_INT, rank_dest, 0,
rank_source, 0, comm_cart, MPI_STATUS_IGNORE);

	start_time = MPI_Wtime();

	// Perform local block multiplication
	for (int i = 0; i < block_size; i++)
		for (int j = 0; j < block_size; j++)
			for (int k = 0; k < block_size; k++)
				C[i * block_size + j] += A[i * block_size + k] * B[k *
block_size + j];

	// Perform next block multiplication and add to partial result
	for (int l = 1; l < n; l++)
	{
		// Shift blocks of A and B
		MPI_Cart_shift(comm_cart, 1, 1, &rank_source, &rank_dest);
		MPI_Sendrecv_replace(A, block_size * block_size, MPI_INT, rank_dest, 0,
rank_source, 0, comm_cart, MPI_STATUS_IGNORE);

		MPI_Cart_shift(comm_cart, 0, 1, &rank_source, &rank_dest);
		MPI_Sendrecv_replace(B, block_size * block_size, MPI_INT, rank_dest, 0,
rank_source, 0, comm_cart, MPI_STATUS_IGNORE);

		for (int i = 0; i < block_size; i++)
			for (int j = 0; j < block_size; j++)
				for (int k = 0; k < block_size; k++)
					C[i * block_size + j] += A[i * block_size + k] *
B[k * block_size + j];
	}

	end_time = MPI_Wtime();
	double elapsed_time = end_time - start_time;
	double max_elapsed_time;
	MPI_Reduce(&elapsed_time, &max_elapsed_time, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);
```

```
	// Synchronize the processes before printing the output
	MPI_Barrier(MPI_COMM_WORLD);

	// Gather all blocks of matrix C in the root process (rank 0)
	int* gathered_A = nullptr;
	int* gathered_B = nullptr;
	int* gathered_C = nullptr;
	if (rank == 0) {
		gathered_A = new int[n * n * block_size * block_size];
		gathered_B = new int[n * n * block_size * block_size];
		gathered_C = new int[n * n * block_size * block_size];
	}
	MPI_Gather(A, block_size * block_size, MPI_INT, gathered_A, block_size *
block_size, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Gather(B, block_size * block_size, MPI_INT, gathered_B, block_size *
block_size, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Gather(C, block_size * block_size, MPI_INT, gathered_C, block_size *
block_size, MPI_INT, 0, MPI_COMM_WORLD);

	// Print the entire matrix C in the root process (rank 0)
	if (rank == 0) {
		cout << "Matrix A:" << endl;
		print_complete_matrix(gathered_A, block_size, n);
		cout << "Matrix B:" << endl;
		print_complete_matrix(gathered_B, block_size, n);
		cout << "Matrix C:" << endl;
		print_complete_matrix(gathered_C, block_size, n);

		cout << "Time taken: " << max_elapsed_time << " seconds" << endl;
	}

	delete[] A;
	delete[] B;
	delete[] C;
	if (rank == 0) {
		delete[] gathered_A;
		delete[] gathered_B;
		delete[] gathered_C;
	}
	MPI_Finalize();

}
```

Description and explanation:

This code is an implementation of the Cannon's algorithm for matrix multiplication using the Message Passing Interface (MPI). Cannon's algorithm is an efficient method for performing large matrix multiplications in parallel on a distributed system.

Here's a detailed explanation of the code:

1. Including necessary header files:

```
#include <iostream>
#include <mpi.h>
#include <math.h>
#include <cstdlib>
#include <ctime>
```

2. print_complete_matrix function:

This function is used to print the entire matrix in a formatted manner. It takes a pointer to the matrix, the block size, and the number of blocks in one dimension (n). It uses nested loops to print each element of the matrix.

3. main function:

The main function contains the primary logic of the program.

a. Initializing MPI and obtaining the process rank and size:

```
int rank, size;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

b. Checking if the number of processes is a perfect square and obtaining the square root:

```
int sqrt_size = sqrt(size);
if (sqrt_size * sqrt_size != size) {
  cerr << "Error: The number of processes must be a perfect square." << endl;
  MPI_Finalize();
  return 1;
}
```

c. Initializing random number generation with the current time, and creating the submatrices A, B, and C for each process:

```
srand(time(NULL));
int n = sqrt(size);
int block_size = atoi(argv[1]) / n;
int* A = new int[block_size * block_size];
int* B = new int[block_size * block_size];
int* C = new int[block_size * block_size];
for (int i = 0; i < block_size * block_size; i++) {
  A[i] = rand() % 5;
  B[i] = rand() % 5;
  C[i] = 0;
}
```

d. Creating a Cartesian topology for the processes:

```
int dims[2] = { n, n };
int periods[2] = { 1, 1 };
int reorder = 0;
```

```
MPI_Comm comm_cart;

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &comm_cart);
```

e. Getting the Cartesian coordinates for the current process:

```
int coords[2];

MPI_Cart_coords(comm_cart, rank, 2, coords);
```

f. Performing the initial alignment of the submatrices A and B:

```
int rank_source, rank_dest;

MPI_Cart_shift(comm_cart, 1, coords[0], &rank_source, &rank_dest);

MPI_Sendrecv_replace(A, block_size * block_size, MPI_INT, rank_dest, 0, rank_source, 0, comm_cart, MPI_STATUS_IGNORE);

MPI_Cart_shift(comm_cart, 0, coords[1], &rank_source, &rank_dest);

MPI_Sendrecv_replace(B, block_size * block_size, MPI_INT, rank_dest, 0, rank_source, 0, comm_cart, MPI_STATUS_IGNORE);
```

g. Performing local block multiplication and updating the partial result:

```
for (int i = 0; i < block_size; i++)
  for (int j = 0; j < block_size; j++)
    for (int k = 0; k < block_size; k++)
      C[i * block_size + j] += A[i * block_size + k] * B[k * block_size + j];
```

h. Shifting blocks of A and B, and performing further block multiplications:

```
for (int l = 1; l < n; l++) {
  MPI_Cart_shift(comm_cart, 1, 1, &rank_source, &rank_dest);
  MPI_Sendrecv_replace(A, block_size * block_size, MPI_INT, rank_dest, 0, rank_source, 0, comm_cart, MPI_STATUS_IGNORE);
  MPI_Cart_shift(comm_cart, 0, 1, &rank_source, &rank_dest);
  MPI_Sendrecv_replace(B, block_size * block_size, MPI_INT, rank_dest, 0, rank_source, 0, comm_cart, MPI_STATUS_IGNORE);

// Perform block multiplication and add to the partial result
for (int i = 0; i < block_size; i++)
for (int j = 0; j < block_size; j++)
for (int k = 0; k < block_size; k++)
C[i * block_size + j] += A[i * block_size + k] * B[k * block_size + j];
}
```

In this loop, the code iterates n-1 times to perform additional block multiplications and sum them to the C matrix. For each iteration:

The code shifts the blocks of matrix A horizontally by 1 using MPI_Cart_shift. This function computes the source and destination ranks for the shift operation.

The MPI_Sendrecv_replace function is called to send the block of A to the destination rank and receive a new block from the source rank. The received block replaces the original block.

The same operations are performed for matrix B, but this time the shift is performed vertically.

After shifting the blocks of A and B, the code performs a block multiplication similar to the one done previously, and the result is added to matrix C.

i. Timing the block multiplications:

```
end_time = MPI_Wtime();
double elapsed_time = end_time - start_time;
double max_elapsed_time;
MPI_Reduce(&elapsed_time, &max_elapsed_time, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);
```

MPI_Wtime() is called again to record the end time. The elapsed time is computed by subtracting the start time from the end time. The code then calls MPI_Reduce to find the maximum elapsed time across all processes. This operation is useful in determining the execution time of the parallel algorithm.

j. Synchronize processes before printing the output:

```
MPI_Barrier(MPI_COMM_WORLD);
```

MPI_Barrier is called to ensure that all processes reach this point before proceeding. This is important because the root process (rank 0) will print the output, and it needs to wait for all other processes to finish their computations.

k. Gathering the blocks of matrices A, B, and C in the root process:

```
int* gathered_A = nullptr;
int* gathered_B = nullptr;
int* gathered_C = nullptr;
if (rank == 0) {
  gathered_A = new int[n * n * block_size * block_size];
  gathered_B = new int[n * n * block_size * block_size];
  gathered_C = new int[n * n * block_size * block_size];
}
```

```
MPI_Gather(A, block_size * block_size, MPI_INT, gathered_A, block_size * block_size, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Gather(B, block_size * block_size, MPI_INT, gathered_B, block_size * block_size, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Gather(C, block_size * block_size, MPI_INT, gathered_C, block_size * block_size, MPI_INT, 0, MPI_COMM_WORLD);
```

In this section, the blocks of matrices A, B, and C are gathered in the root process (rank 0). Memory is allocated for the gathered matrices only in the root process. The MPI_Gather function is called to gather the blocks from all processes into the root process.

l. Printing the matrices and the elapsed time:

```
if (rank == 0) {
  cout << "Matrix A:" << endl;
  print_complete_matrix(gathered_A, block_size, n);
  cout << "Matrix B:" << endl;
  print_complete_matrix(gathered_B, block_size, n);
  cout << "Matrix C:" << endl;
  print_complete_matrix(gathered_C, block_size, n);

  cout << "Time taken: " << max_elapsed_time << " seconds" << endl;
}
```

If the current process is the root process, the code prints the gathered matrices A, B, and C using the print_complete_matrix function. It also prints the maximum elapsed time taken to perform the block multiplications.

m. Clean up:

```
delete[] A;
delete[] B;
delete[] C;
if (rank == 0) {
  delete[] gathered_A;
  delete[] gathered_B;
  delete[] gathered_C;
}
MPI_Finalize();
```

Finally, the code deallocates the memory allocated for the matrices and gathered matrices (if the process is the root process). The MPI_Finalize function is called to complete the MPI environment.

Screenshots:

Input:

```
D:\VisualStudioProjects\MPI_Part\x64\Debug>mpiexec -n 4 MPI_Part 4
```

Output:

```
Matrix A:
3 0  3 0
3 4  3 4

3 0  3 0
3 4  3 4

Matrix B:
1 1  1 1
0 3  0 3

1 1  1 1
0 3  0 3

Matrix C:
6 6  6 6
6 30  6 30

6 6  6 6
6 30  6 30

Time taken: 0.0003281 seconds
```

Time taken: 0.0003281 seconds.

Conclusion:

I provided two implementations of the parallel Cannon's multiplication algorithm, one using OpenMP and the other using MPI. Both implementations were executed with 4 processes (or threads) and matrices of size 4x4.

The OpenMP implementation completed the matrix multiplication in 9.20007e-06 seconds (9.2 microseconds), while the MPI implementation took 0.0003281 seconds (328.1 microseconds).

Based on these results, the OpenMP implementation is significantly faster than the MPI implementation for the given problem size and the number of processes. The main reason behind this difference in performance is the nature of the parallelism used in each approach.

OpenMP is a shared-memory parallel programming model, which means that all threads have access to the same memory space. This allows for efficient communication and synchronization between threads with minimal overhead. In this case, the overhead of creating and synchronizing threads is relatively low, making the OpenMP implementation faster for small problem sizes.

On the other hand, MPI is a distributed-memory parallel programming model, which means that each process has its own separate memory space. Communication between processes is done through message passing, which has a higher overhead compared to shared-memory communication. In the case of small problem sizes, the overhead of communication and synchronization in MPI can outweigh the benefits of parallelism, leading to longer execution times compared to the OpenMP implementation.

In conclusion, for small problem sizes and a limited number of processes, the OpenMP implementation is likely to be faster due to its lower communication and synchronization overhead. However, as the problem size and the number of processes increase, the performance advantage of the MPI implementation may become more apparent, especially in cases where the problem cannot fit into the memory of a single compute node or when using a distributed computing environment.