Petre Tiberiu-Adrian, Ionita Alexandru-Mihail

Group 1241B

#Ex1

#include <mpi.h>

#include <stdio.h>

int main(int argc, char\*\* argv) {

int data[6] = {1, 2, 3, 4, 5, 6};

int local\_sum = 0, total\_sum = 0;

int local\_data[3];

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Scatter(data, 3, MPI\_INT, local\_data, 3, MPI\_INT, 0, MPI\_COMM\_WORLD);

for (int i = 0; i < 3; i++) {

local\_sum += local\_data[i];

}

MPI\_Reduce(&local\_sum, &total\_sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

printf("Total sum is: %d\n", total\_sum);

}

MPI\_Finalize();

return 0;

}

#Ex2

float data[10] = {10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0};

float average = 0.0;

MPI\_Init(&argc, &argv);

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

if (rank == 0) {

float sum = 0.0;

for (int i = 0; i < 10; i++) {

sum += data[i];

}

average = sum / 10;

MPI\_Send(&average, 1, MPI\_FLOAT, 1, 0, MPI\_COMM\_WORLD);

printf("Process 0 sent average: %.2f\n", average);

} else if (rank == 1) {

MPI\_Recv(&average, 1, MPI\_FLOAT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process 1 received average: %.2f\n", average);

}

#Ex3

int A[2][3] = {{1, 2, 3}, {4, 5, 6}};

int B[2][3] = {{6, 5, 4}, {3, 2, 1}};

int result[2][3];

int local\_A[3], local\_B[3], local\_result[3];

MPI\_Init(&argc, &argv);

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Scatter(A, 3, MPI\_INT, local\_A, 3, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(B, 3, MPI\_INT, local\_B, 3, MPI\_INT, 0, MPI\_COMM\_WORLD);

for (int i = 0; i < 3; i++) {

local\_result[i] = local\_A[i] + local\_B[i];

}

MPI\_Gather(local\_result, 3, MPI\_INT, result, 3, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

printf("Resulting Matrix:\n");

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 3; j++) {

printf("%d ", result[i][j]);

}

printf("\n");

}

}

#Homwork\_Ex1

#include <mpi.h>

#include <stdio.h>

#include <time.h>

int main(int argc, char\*\* argv) {

int data[6] = {1, 2, 3, 4, 5, 6};

int local\_data[3], local\_sum = 0, total\_sum = 0;

double start\_time, end\_time

MPI\_Init(&argc, &argv);

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

if (rank == 0 || rank == 1) {

int offset = rank \* 3;

for (int i = 0; i < 3; i++) {

local\_data[i] = data[offset + i];

local\_sum += local\_data[i];

}

MPI\_Send(&local\_sum, 1, MPI\_INT, 2, 0, MPI\_COMM\_WORLD);

}

if (rank == 2) {

start\_time = MPI\_Wtime();

int sum1, sum2;

MPI\_Recv(&sum1, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Recv(&sum2, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

total\_sum = sum1 + sum2;

end\_time = MPI\_Wtime();

printf("Final Sum: %d\n", total\_sum);

printf("Processing Time: %f seconds\n", end\_time - start\_time);

}

MPI\_Finalize();

return 0;

}

#Homework\_Ex2

int A[2][2] = {{1, 2}, {3, 4}};

int B[2][2] = {{5, 6}, {7, 8}};

int result[2][2];

int local\_result[2];

int rank;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

if (rank == 0 || rank == 1) {

for (int j = 0; j < 2; j++) {

local\_result[j] = A[rank][0] \* B[0][j] + A[rank][1] \* B[1][j];

}

MPI\_Gather(local\_result, 2, MPI\_INT, result, 2, MPI\_INT, 0, MPI\_COMM\_WORLD);

}

if (rank == 0) {

printf("Result of Matrix Multiplication:\n");

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

printf("%d ", result[i][j]);

}

printf("\n");

}

}

#Cuda\_Ex1

#include <stdio.h>

#include <cuda.h>

#define N 3

\_\_global\_\_ void matrixMul(float \*A, float \*B, float \*C, int width) {

int row = threadIdx.y;

int col = threadIdx.x;

float sum = 0;

for (int k = 0; k < width; ++k)

sum += A[row \* width + k] \* B[k \* width + col];

C[row \* width + col] = sum;

}

int main() {

float h\_A[N\*N] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

float h\_B[N\*N] = {9, 8, 7, 6, 5, 4, 3, 2, 1};

float h\_C[N\*N];

float \*d\_A, \*d\_B, \*d\_C;

cudaMalloc((void\*\*)&d\_A, N\*N\*sizeof(float));

cudaMalloc((void\*\*)&d\_B, N\*N\*sizeof(float));

cudaMalloc((void\*\*)&d\_C, N\*N\*sizeof(float));

cudaMemcpy(d\_A, h\_A, N\*N\*sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, N\*N\*sizeof(float), cudaMemcpyHostToDevice);

dim3 threadsPerBlock(N, N);

matrixMul<<<1, threadsPerBlock>>>(d\_A, d\_B, d\_C, N);

cudaMemcpy(h\_C, d\_C, N\*N\*sizeof(float), cudaMemcpyDeviceToHost);

printf("Matrix C (Result):\n");

for (int i = 0; i < N\*N; ++i) {

printf("%f ", h\_C[i]);

if ((i + 1) % N == 0) printf("\n");

}

cudaFree(d\_A); cudaFree(d\_B); cudaFree(d\_C);

return 0;

}

#Cuda\_Ex2

#include <cuda\_runtime.h>

#include <cusolverDn.h>

void printMatrix(int n, const float\* A) {

for (int i = 0; i < n \* n; i++) {

printf("%8.4f ", A[i]);

if ((i + 1) % n == 0) printf("\n");

}

}

int main() {

const int N = 3;

float h\_A[N \* N] = {

1, 2, 3,

0, 1, 4,

5, 6, 0

};

float \*d\_A, \*d\_B, \*d\_work;

int \*d\_info, \*d\_pivots;

int workspace\_size, info\_h = 0;

float h\_I[N \* N], h\_inv[N \* N];

for (int i = 0; i < N \* N; ++i)

h\_I[i] = (i % N == i / N) ? 1.0f : 0.0f;

cudaMalloc((void\*\*)&d\_A, N \* N \* sizeof(float));

cudaMalloc((void\*\*)&d\_B, N \* N \* sizeof(float));

cudaMalloc((void\*\*)&d\_info, sizeof(int));

cudaMalloc((void\*\*)&d\_pivots, N \* sizeof(int));

cudaMemcpy(d\_A, h\_A, N \* N \* sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_I, N \* N \* sizeof(float), cudaMemcpyHostToDevice);

cusolverDnHandle\_t handle;

cusolverDnCreate(&handle);

cusolverDnSgetrf\_bufferSize(handle, N, N, d\_A, N, &workspace\_size);

cudaMalloc((void\*\*)&d\_work, workspace\_size \* sizeof(float));

cusolverDnSgetrf(handle, N, N, d\_A, N, d\_work, d\_pivots, d\_info);

cusolverDnSgetrs(handle, CUBLAS\_OP\_N, N, N, d\_A, N, d\_pivots, d\_B, N, d\_info);

cudaMemcpy(h\_inv, d\_B, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);

cudaMemcpy(&info\_h, d\_info, sizeof(int), cudaMemcpyDeviceToHost);

if (info\_h == 0) {

printf("Inverse Matrix:\n");

printMatrix(N, h\_inv);

} else {

printf("Matrix inversion failed. Info = %d\n", info\_h);

}

cudaFree(d\_A); cudaFree(d\_B); cudaFree(d\_work);

cudaFree(d\_pivots); cudaFree(d\_info);

cusolverDnDestroy(handle);

return 0;

}