HW 5

Code  
#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <string.h>

// Function to compare integers (for qsort)

int compare(const void \*a, const void \*b)

{

return (\*(int \*)a - \*(int \*)b);

}

void printArray(int arr[], int size)

{

for (int i = 0; i < size; i++)

printf("%d ", arr[i]);

}

int main(int argc, char \*argv[])

{

int rank, size;

int \*data = NULL;

int \*sorted\_data = NULL;

if (argc != 2)

{

printf("Usage: %s <n>\n", argv[0]);

return 1;

}

int n = atoi(argv[1]);

MPI\_Init(&argc, &argv);

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

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

if (rank == 0)

{

data = (int \*)malloc(n \* sizeof(int));

// Initialize or read data

printf("Unsorted Data: ");

for (int i = 0; i < n; i++)

{

data[i] = rand() % 1000;

printf("%d ", data[i]);

}

printf("\n");

}

// Broadcast the size of the problem to all processes

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Allocate memory for scattered data in all processes

int \*scattered\_data = (int \*)malloc(n / size \* sizeof(int));

// Synchronize all processes

MPI\_Barrier(MPI\_COMM\_WORLD);

// Scatter data to each processor

MPI\_Scatter(data, n / size, MPI\_INT, scattered\_data, n / size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Sort scattered data individually on each processor

qsort(scattered\_data, n / size, sizeof(int), compare);

// get size^2 samples; size samples from each process

int \*samples = (int \*)malloc(size \* sizeof(int));

for (int i = 0; i < size; i++)

{

samples[i] = scattered\_data[i \* n / (size \* size)];

}

// Gather all samples to process 0

int \*all\_samples = NULL;

if (rank == 0)

{

all\_samples = (int \*)malloc(size \* size \* sizeof(int));

}

MPI\_Gather(samples, size, MPI\_INT, all\_samples, size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// merge sort all samples

if (rank == 0)

{

qsort(all\_samples, 0, size \* size - 1, compare);

}

// broadcast pivots to all processes

int \*pivots = (int \*)malloc(size - 1);

if (rank == 0)

{

for (int i = 0; i < size - 1; i++)

{

pivots[i] = all\_samples[(i + 1) \* size + size / 2 - 1];

}

}

MPI\_Bcast(pivots, size - 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// send data to processes based on pivots

// Initialize sendcounts, displs, and recvcounts to 0

int \*sendcounts = (int \*)calloc(size, sizeof(int));

int \*displs = (int \*)calloc(size, sizeof(int));

int \*recvcounts = (int \*)calloc(size, sizeof(int));

// Calculate sendcounts and displs

for (int i = 0, j = 0; i < n / size; i++)

{

if (j < size - 1 && scattered\_data[i] > pivots[j])

{

sendcounts[j] = i - displs[j];

displs[++j] = i;

}

}

sendcounts[size - 1] = n / size - displs[size - 1];

// Use MPI\_Alltoall to get recvcounts

MPI\_Alltoall(sendcounts, 1, MPI\_INT, recvcounts, 1, MPI\_INT, MPI\_COMM\_WORLD);

// Calculate total and displs for MPI\_Alltoallv

int total = 0;

for (int i = 0; i < size; i++)

{

total += recvcounts[i];

if (i > 0)

{

displs[i] = displs[i - 1] + sendcounts[i - 1];

}

}

int \*sorted\_sublist = (int \*)malloc(total \* sizeof(int));

// Use MPI\_Alltoallv to exchange data

MPI\_Alltoallv(scattered\_data, sendcounts, displs, MPI\_INT, sorted\_sublist, recvcounts, displs, MPI\_INT, MPI\_COMM\_WORLD);

// merge sort all received partitions

qsort(sorted\_sublist, 0, total - 1, compare);

// Print sorted data

printf("\n");

printArray(sorted\_sublist, total);

// Send sorted data back to process 0

// MPI\_Gather(sorted\_sublist, total, MPI\_INT, data, total, MPI\_INT, 0, MPI\_COMM\_WORLD);

// // Print sorted data

// if (rank == 0)

// {

// printf("Sorted Data: ");

// for (int i = 0; i < n; i++)

// {

// printf("%d ", data[i]);

// }

// printf("\n");

// }

// free memory

free(samples);

free(all\_samples);

free(pivots);

free(sendcounts);

free(displs);

free(recvcounts);

free(sorted\_sublist);

free(scattered\_data);

MPI\_Finalize();

printf("\n");

return 0;

}

* 1. asdfA piece of paper with writing on it

     Description automatically generated
  2. asdfA notebook with writing on it

     Description automatically generated
  3. asdfA piece of paper with writing on it

     Description automatically generated

1. Dot Product Program Cuda
   1. Benchmark results:

aaryandehade@bizon-workstation:~/parallel-computing/dot$ make test

./dot 128 256

Num Blocks: 128, Threads Per Block: 256, CUDA time: 0.000132225 seconds

CUDA dot product: 7.94094e+06

CPU time: 0.00154298 seconds

CPU dot product: 7.94094e+06

./dot 100 256

Num Blocks: 100, Threads Per Block: 256, CUDA time: 0.000124067 seconds

CUDA dot product: 7.92719e+06

CPU time: 0.00152448 seconds

CPU dot product: 7.92719e+06

./dot 256 128

Num Blocks: 256, Threads Per Block: 128, CUDA time: 0.000159063 seconds

CUDA dot product: 7.9479e+06

CPU time: 0.00151392 seconds

CPU dot product: 7.9479e+06

./dot 200 128

Num Blocks: 200, Threads Per Block: 128, CUDA time: 0.000149192 seconds

CUDA dot product: 7.93259e+06

CPU time: 0.0015405 seconds

CPU dot product: 7.93259e+06

./dot 64 256

Num Blocks: 64, Threads Per Block: 256, CUDA time: 0.000114663 seconds

CUDA dot product: 7.91789e+06

CPU time: 0.005046 seconds

CPU dot product: 7.91789e+06

./dot 512 64

Num Blocks: 512, Threads Per Block: 64, CUDA time: 0.000227638 seconds

CUDA dot product: 7.93311e+06

CPU time: 0.00151611 seconds

CPU dot product: 7.93311e+06

./dot 32 32

Num Blocks: 32, Threads Per Block: 32, CUDA time: 0.000124275 seconds

CUDA dot product: 7.92887e+06

CPU time: 0.00152396 seconds

CPU dot product: 7.92887e+06

./dot 1024 32

Num Blocks: 1024, Threads Per Block: 32, CUDA time: 0.000414984 seconds

CUDA dot product: 7.91552e+06

CPU time: 0.00153838 seconds

CPU dot product: 7.91552e+06

./dot 16 2048

Num Blocks: 16, Threads Per Block: 2048, CUDA time: 8.6708e-05 seconds

CUDA dot product: 0

CPU time: 0.00152405 seconds

CPU dot product: 7.93271e+06

./dot 2048 16

Num Blocks: 2048, Threads Per Block: 16, CUDA time: 0.00114207 seconds

CUDA dot product: 7.93888e+06

CPU time: 0.00152266 seconds

CPU dot product: 7.93888e+06

* 1. Code:

1. #include <iostream>
2. #include <vector>
3. #include <cmath>
4. #include <chrono>
5. #include <random>
6. #include <cstdlib>
7. #if !defined(\_\_CUDA\_ARCH\_\_) || \_\_CUDA\_ARCH\_\_ >= 600
8. #else
9. \_\_device\_\_ double atomicAdd(double\* address, double val) {
10. unsigned long long int\* address\_as\_ull =
11. (unsigned long long int\*)address;
12. unsigned long long int old = \*address\_as\_ull, assumed;
13. do {
14. assumed = old;
15. old = atomicCAS(address\_as\_ull, assumed,
16. \_\_double\_as\_longlong(val +
17. \_\_longlong\_as\_double(assumed)));
18. } while (assumed != old);
19. return \_\_longlong\_as\_double(old);
20. }
21. #endif
22. // CUDA Kernel to compute dot product
23. \_\_global\_\_ void dotProduct(double \*a, double \*b, double \*result, int n) {
24. int index = threadIdx.x + blockIdx.x \* blockDim.x;
25. int stride = blockDim.x \* gridDim.x;
26. \_\_shared\_\_ double temp[256];
27. temp[threadIdx.x] = 0;
28. for (int i = index; i < n; i += stride) {
29. temp[threadIdx.x] += a[i] \* b[i];
30. }
31. \_\_syncthreads();
32. // Reduction in shared memory
33. for (int i = blockDim.x / 2; i > 0; i >>= 1) {
34. if (threadIdx.x < i) {
35. temp[threadIdx.x] += temp[threadIdx.x + i];
36. }
37. \_\_syncthreads();
38. }
39. // Write the final sum to global memory
40. if (threadIdx.x == 0) {
41. atomicAdd(result, temp[0]);
42. }
43. }
44. // CPU function to compute dot product
45. double dotProductCPU(std::vector<double>& a, std::vector<double>& b) {
46. double result = 0.0;
47. for (int i = 0; i < a.size(); ++i) {
48. result += a[i] \* b[i];
49. }
50. return result;
51. }
52. int main(int argc, char \*argv[]) {
53. if (argc != 3) {
54. std::cerr << "Usage: " << argv[0] << " <numBlocks> <threadsPerBlock>" << std::endl;
55. return 1;
56. }
57. int numBlocks = atoi(argv[1]);
58. int threadsPerBlock = atoi(argv[2]);
59. int n = pow(2, 18);
60. std::vector<double> a(n); // Initialize vector 'a' with random values
61. std::vector<double> b(n); // Initialize vector 'b' with random values
62. // Fill vectors with random values
63. std::random\_device rd;
64. std::mt19937 gen(rd());
65. std::uniform\_real\_distribution<double> dis(1.0, 10.0); // Generate random values between 1 and 10
66. for (int i = 0; i < n; ++i) {
67. a[i] = dis(gen);
68. b[i] = dis(gen);
69. }
70. double \*d\_a, \*d\_b, \*d\_result;
71. double result = 0.0;
72. cudaError\_t cudaStatus;
73. cudaStatus = cudaMalloc(&d\_a, n \* sizeof(double));
74. if (cudaStatus != cudaSuccess) {
75. std::cerr << "cudaMalloc failed for d\_a!" << std::endl;
76. return 1;
77. }
78. cudaStatus = cudaMalloc(&d\_b, n \* sizeof(double));
79. if (cudaStatus != cudaSuccess) {
80. std::cerr << "cudaMalloc failed for d\_b!" << std::endl;
81. return 1;
82. }
83. cudaStatus = cudaMalloc(&d\_result, sizeof(double));
84. if (cudaStatus != cudaSuccess) {
85. std::cerr << "cudaMalloc failed for d\_result!" << std::endl;
86. return 1;
87. }
88. // Copy input data to device memory
89. cudaMemcpy(d\_a, a.data(), n \* sizeof(double), cudaMemcpyHostToDevice);
90. cudaMemcpy(d\_b, b.data(), n \* sizeof(double), cudaMemcpyHostToDevice);
91. // Benchmark CUDA implementation
92. auto start = std::chrono::high\_resolution\_clock::now();
93. dotProduct<<<numBlocks, threadsPerBlock>>>(d\_a, d\_b, d\_result, n);
94. cudaStatus = cudaDeviceSynchronize();
95. if (cudaStatus != cudaSuccess) {
96. std::cerr << "cudaDeviceSynchronize returned error code " << cudaStatus << " after launching dotProduct kernel!" << std::endl;
97. return 1;
98. }
99. auto end = std::chrono::high\_resolution\_clock::now();
100. std::chrono::duration<double> duration = end - start;
101. std::cout << "Num Blocks: " << numBlocks << ", Threads Per Block: " << threadsPerBlock << ", CUDA time: " << duration.count() << " seconds" << std::endl;
102. // Copy result back to host and print
103. cudaMemcpy(&result, d\_result, sizeof(double), cudaMemcpyDeviceToHost);
104. std::cout << "CUDA dot product: " << result << std::endl;
105. // Benchmark CPU implementation
106. start = std::chrono::high\_resolution\_clock::now();
107. result = dotProductCPU(a, b);
108. end = std::chrono::high\_resolution\_clock::now();
109. duration = end - start;
110. std::cout << "CPU time: " << duration.count() << " seconds" << std::endl;
111. std::cout << "CPU dot product: " << result << std::endl;
112. // Free device memory
113. cudaFree(d\_a);
114. cudaFree(d\_b);
115. cudaFree(d\_result);
116. return 0;
117. }