

HPC P1 – parallel BFS, DFS

```
#include <iostream>
#include <vector> #include
<queue>
#include <omp.h>

using namespace std;

const int MAX_NODES = 100;
vector<int> graph[MAX_NODES];

// Parallel Breadth First Search (BFS)
void parallelBFS(int start) {    bool
visited[MAX_NODES] = {false};
queue<int> q;    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
int current = q.front();
q.pop();

        #pragma omp parallel for    for (int i = 0;
i < graph[current].size(); ++i) {        int
neighbor = graph[current][i];
        #pragma omp critical
        {
            if (!visited[neighbor]) {
q.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }
}

cout << "BFS Visited Nodes: ";
for (int i = 0; i < MAX_NODES; ++i) {
    if (visited[i]) {
        cout << i << " ";
    }
}
cout << endl;
}
```

```
// Parallel Depth First Search (DFS) void
```

```
parallelDFS(int start, bool visited[]) {
```

```
    visited[start] = true;
```

```
    #pragma omp parallel for    for (int i =  
0; i < graph[start].size(); ++i) {        int  
neighbor = graph[start][i];        if  
(!visited[neighbor]) {  
        parallelDFS(neighbor, visited);  
    }  
}
```

```
}
```

```
int main() {
```

```
graph[0] = {1, 2};
```

```
graph[1] = {0, 3, 4};
```

```
graph[2] = {0, 5, 6};
```

```
graph[3] = {1};
```

```
graph[4] = {1};
```

```
graph[5] = {2};
```

```
graph[6] = {2};
```

```
int start_node = 0;
```

```
// Perform parallel BFS
```

```
parallelBFS(start_node);
```

```
// Perform parallel DFS    bool
```

```
visited[MAX_NODES] = {false};
```

```
parallelDFS(start_node, visited);
```

```
cout << "DFS Visited Nodes: ";
```

```
for (int i = 0; i < MAX_NODES; ++i) {
```

```
    if (visited[i]) {
```

```
        cout << i << " ";
```

```
    }
```

```
}
```

```
cout << endl;
```

```
return 0;
```

```
}
```

P2 - Parallel Bubble Sort and Merge sort using OpenMP.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <omp.h>
#include <chrono>
using namespace std;

void bubble_sort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

void merge(vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[m + 1 + j];
    }
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }
    while (i < n1) {
        arr[k++] = L[i++];
    }
    while (j < n2) {
        arr[k++] = R[j++];
    }
}
```

```

void merge_sort(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

```

```

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
}

```

// Sequential Bubble Sort

```

vector<int> arr_copy = arr;
auto start = chrono::high_resolution_clock::now();
bubble_sort(arr_copy);
auto end = chrono::high_resolution_clock::now();
long long sequential_bubble_time = chrono::duration_cast<chrono::microseconds>(end -
start).count();
cout << "\nSequential Bubble Sort Time: " << sequential_bubble_time << "
microseconds\n";
cout << "Sorted order: ";
for (int num : arr_copy) {
    cout << num << " ";
}
cout << endl;

```

// Parallel Bubble Sort

```

start = chrono::high_resolution_clock::now();
bubble_sort(arr);
end = chrono::high_resolution_clock::now();
long long parallel_bubble_time = chrono::duration_cast<chrono::microseconds>(end -
start).count();
cout << "\nParallel Bubble Sort Time: " << parallel_bubble_time << " microseconds\n";
cout << "Sorted order: ";
for (int num : arr) {
    cout << num << " ";
}
cout << endl;

```

// Sequential Merge Sort

```
arr_copy = arr;
start = chrono::high_resolution_clock::now();
merge_sort(arr_copy, 0, n - 1);
end = chrono::high_resolution_clock::now();
long long sequential_merge_time = chrono::duration_cast<chrono::microseconds>(end -
start).count();
cout << "\nSequential Merge Sort Time: " << sequential_merge_time << "
microseconds\n";
cout << "Sorted order: ";
for (int num : arr_copy) {
    cout << num << " ";
}
cout << endl;
```

// Parallel Merge Sort

```
start = chrono::high_resolution_clock::now();
#pragma omp parallel
{
    #pragma omp single
    merge_sort(arr, 0, n - 1);
}
end = chrono::high_resolution_clock::now();
long long parallel_merge_time = chrono::duration_cast<chrono::microseconds>(end -
start).count();
cout << "\nParallel Merge Sort Time: " << parallel_merge_time << " microseconds\n";
cout << "Sorted order: ";
for (int num : arr) {
    cout << num << " ";
}
cout << endl;

return 0;
}
```

Pr 3 - Implement Min, Max, Sum and Average operations using Parallel Reduction.

```
#include <iostream>
#include <vector>
#include <omp.h>
#include <time.h>
using namespace std;
int main() { const int
size = 1000;
vector<int> data(size);
srand(time(0));
for (int i = 0; i < size; ++i) { data[i]
= rand() % 100;
}
int min_value = data[0];
#pragma omp parallel for reduction(min:min_value)
for (int i = 0; i < size; ++i) { if (data[i] <
min_value) { min_value = data[i];
} }
int max_value = data[0];
#pragma omp parallel for reduction(max:max_value)
for (int i = 0; i < size; ++i) { if (data[i] > max_value)
{ max_value = data[i];
} }
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < size; ++i) {
sum += data[i];
}
float average = 0.0;
#pragma omp parallel for reduction(+:average) for
(int i = 0; i < size; ++i) {
average += static_cast<float>(data[i]) / size;
}
cout << "Minimum value: " << min_value << endl;
cout << "Maximum value: " << max_value << endl;
cout << "Sum of values: " << sum << endl; cout <<
"Average of values: " << average << endl; return 0;
```

Pr – 4 CUDA

HPC/4/matrixMul.cpp

```
1  #include <cmath>
2  #include <cstdlib>
3  #include <iostream>
4
5  #define checkCudaErrors(call) \
6      do { \
7          cudaError_t err = call; \
8          if (err != cudaSuccess) { \
9              printf("CUDA error at %s %d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
10             exit(EXIT_FAILURE); \
11         } \
12     } while (0)
13
14  using namespace std;
15
16  // Matrix multiplication Cuda
17  __global__ void matrixMultiplication(int *a, int *b, int *c, int n) {
18      int row = threadIdx.y + blockDim.y * blockIdx.y;
19      int col = threadIdx.x + blockDim.x * blockIdx.x;
20      int sum = 0;
21
22      if (row < n && col < n)
23          for (int j = 0; j < n; j++) {
24              sum = sum + a[row * n + j] * b[j * n + col];
25          }
26
27      c[n * row + col] = sum;
28  }
29
30  int main() {
31      int *a, *b, *c;
32      int *a_dev, *b_dev, *c_dev;
33      int n = 10;
34
35      a = new int[n * n];
36      b = new int[n * n];
37      c = new int[n * n];
38      int *d = new int[n * n];
39      int size = n * n * sizeof(int);
40      checkCudaErrors(cudaMalloc(&a_dev, size));
41      checkCudaErrors(cudaMalloc(&b_dev, size));
42      checkCudaErrors(cudaMalloc(&c_dev, size));
43
44      // Array initialization
45      for (int i = 0; i < n * n; i++) {
46          a[i] = rand() % 10;
47          b[i] = rand() % 10;
48      }
49
50      cout << "Given matrix A is =>\n";
51      for (int row = 0; row < n; row++) {
52          for (int col = 0; col < n; col++) {
53              cout << a[row * n + col] << " ";
54          }
55          cout << "\n";
56      }
57      cout << "\n";
58
59      cout << "Given matrix B is =>\n";
60      for (int row = 0; row < n; row++) {
61          for (int col = 0; col < n; col++) {
62              cout << b[row * n + col] << " ";
63          }
64          cout << "\n";
65      }
66      cout << "\n";
67
68      cudaEvent_t start, end;
69
70      checkCudaErrors(cudaEventCreate(&start));
```

```

71     checkCudaErrors(cudaEventCreate(&end));
72
73     checkCudaErrors(cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice));
74     checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));
75
76     dim3 threadsPerBlock(n, n);
77     dim3 blocksPerGrid(1, 1);
78
79     // GPU Multiplication
80     checkCudaErrors(cudaEventRecord(start));
81     matrixMultiplication<<<blocksPerGrid, threadsPerBlock>>>(a_dev, b_dev, c_dev, n);
82
83     checkCudaErrors(cudaEventRecord(end));
84     checkCudaErrors(cudaEventSynchronize(end));
85
86     float time = 0.0;
87     checkCudaErrors(cudaEventElapsedTime(&time, start, end));
88
89     checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));
90
91     // CPU matrix multiplication
92     int sum = 0;
93     for (int row = 0; row < n; row++) {
94         for (int col = 0; col < n; col++) {
95             sum = 0;
96             for (int k = 0; k < n; k++) sum = sum + a[row * n + k] * b[k * n + col];
97             d[row * n + col] = sum;
98         }
99     }
100
101     cout << "CPU product is ⇒\n";
102     for (int row = 0; row < n; row++) {
103         for (int col = 0; col < n; col++) {
104             cout << d[row * n + col] << " ";
105         }
106         cout << "\n";
107     }
108     cout << "\n";
109
110     cout << "GPU product is ⇒\n";
111     for (int row = 0; row < n; row++) {
112         for (int col = 0; col < n; col++) {
113             cout << c[row * n + col] << " ";
114         }
115         cout << "\n";
116     }
117     cout << "\n";
118
119     int error = 0;
120     int _c, _d;
121     for (int row = 0; row < n; row++) {
122         for (int col = 0; col < n; col++) {
123             _c = c[row * n + col];
124             _d = d[row * n + col];
125             error += _c - _d;
126             if (0 ≠ (_c - _d)) {
127                 cout << "Error at (" << row << ", " << col << ") ⇒ GPU: " << _c << ", CPU: " << _d
128                     << "\n";
129             }
130         }
131     }
132     cout << "\n";
133
134     cout << "Error : " << error;
135     cout << "\nTime Elapsed: " << time;
136
137     return 0;
138 }
139
140 /*
141
142 OUTPUT:
143
144 Given matrix A is ⇒

```



```
145 3 7 3 6 9 2 0 3 0 2
146 1 7 2 2 7 9 2 9 3 1
147 9 1 4 8 5 3 1 6 2 6
148 5 4 6 6 3 4 2 4 4 3
149 7 6 8 3 4 2 6 9 6 4
150 5 4 7 7 7 2 1 6 5 4
151 0 1 7 1 9 7 7 6 6 9
152 8 2 3 0 8 0 6 8 6 1
153 9 4 1 3 4 4 7 3 7 9
154 2 7 5 4 8 9 5 8 3 8
155
156 Given matrix B is =>
157 6 5 5 2 1 7 9 6 6 6
158 8 9 0 3 5 2 8 7 6 2
159 3 9 7 4 0 6 0 3 0 1
160 5 7 5 9 7 5 5 7 4 0
161 8 8 4 1 9 0 8 2 6 9
162 0 8 1 2 2 6 0 1 9 9
163 9 7 1 5 7 6 3 5 3 4
164 1 9 9 8 5 9 3 5 1 5
165 8 8 0 0 4 4 6 1 5 6
166 1 8 7 1 5 7 3 8 1 9
167
168 CPU product is =>
169 190 278 145 132 190 136 200 169 161 167
170 186 355 156 157 207 209 185 164 210 246
171 191 335 233 179 196 257 220 227 174 232
172 191 319 172 156 167 218 182 186 165 186
173 276 433 239 205 229 305 251 252 193 257
174 233 378 222 181 218 240 231 216 180 226
175 232 430 221 155 255 274 187 203 193 328
176 248 319 178 137 201 217 233 171 165 236
177 267 379 184 141 231 276 259 247 218 301
178 252 477 239 204 282 302 239 261 245 334
179
180 GPU product is =>
181 190 278 145 132 190 136 200 169 161 167
182 186 355 156 157 207 209 185 164 210 246
183 191 335 233 179 196 257 220 227 174 232
184 191 319 172 156 167 218 182 186 165 186
185 276 433 239 205 229 305 251 252 193 257
186 233 378 222 181 218 240 231 216 180 226
187 232 430 221 155 255 274 187 203 193 328
188 248 319 178 137 201 217 233 171 165 236
189 267 379 184 141 231 276 259 247 218 301
190 252 477 239 204 282 302 239 261 245 334
191
192
193 Error : 0
194 Time Elapsed: 0.018144
195
196 */
197
```

HPC/4/matrixVectorMul.cpp

```
1  #include <time.h>
2
3  #include <cmath>
4  #include <cstdlib>
5  #include <iostream>
6
7  #define checkCudaErrors(call) \
8      do { \
9          cudaError_t err = call; \
10         if (err != cudaSuccess) { \
11             printf("CUDA error at %s %d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
12             exit(EXIT_FAILURE); \
13         } \
14     } while (0)
15
16 using namespace std;
17
18 __global__ void matrixVectorMultiplication(int *a, int *b, int *c, int n) {
19     int row = threadIdx.x + blockDim.x * blockIdx.x;
20     int sum = 0;
21
22     if (row < n)
23         for (int j = 0; j < n; j++) {
24             sum = sum + a[row * n + j] * b[j];
25         }
26
27     c[row] = sum;
28 }
29
30 int main() {
31     int *a, *b, *c;
32     int *a_dev, *b_dev, *c_dev;
33     int n = 10;
34
35     a = new int[n * n];
36     b = new int[n];
37     c = new int[n];
38     int *d = new int[n];
39     int size = n * sizeof(int);
40     checkCudaErrors(cudaMalloc(&a_dev, size * size));
41     checkCudaErrors(cudaMalloc(&b_dev, size));
42     checkCudaErrors(cudaMalloc(&c_dev, size));
43
44     for (int i = 0; i < n; i++) {
45         for (int j = 0; j < n; j++) {
46             a[i * n + j] = rand() % 10;
47         }
48         b[i] = rand() % 10;
49     }
50
51     cout << "Given matrix is =>\n";
52     for (int row = 0; row < n; row++) {
53         for (int col = 0; col < n; col++) {
54             cout << a[row * n + col] << " ";
55         }
56         cout << "\n";
57     }
58     cout << "\n";
59
60     cout << "Given vector is =>\n";
61     for (int i = 0; i < n; i++) {
62         cout << b[i] << ", ";
63     }
64     cout << "\n\n";
65
66     cudaEvent_t start, end;
67
68     checkCudaErrors(cudaEventCreate(&start));
69     checkCudaErrors(cudaEventCreate(&end));
70 }
```

```

71     checkCudaErrors(cudaMemcpy(a_dev, a, size * size, cudaMemcpyHostToDevice));
72     checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));
73
74     dim3 threadsPerBlock(n, n);
75     dim3 blocksPerGrid(1, 1);
76
77     checkCudaErrors(cudaEventRecord(start));
78     matrixVectorMultiplication<<<blocksPerGrid, threadsPerBlock>>>(a_dev, b_dev, c_dev, n);
79
80     checkCudaErrors(cudaEventRecord(end));
81     checkCudaErrors(cudaEventSynchronize(end));
82
83     float time = 0.0;
84     checkCudaErrors(cudaEventElapsedTime(&time, start, end));
85
86     checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));
87
88     // CPU matrixVector multiplication
89     int sum = 0;
90     for (int row = 0; row < n; row++) {
91         sum = 0;
92         for (int col = 0; col < n; col++) {
93             sum = sum + a[row * n + col] * b[col];
94         }
95         d[row] = sum;
96     }
97
98     cout << "CPU product is =>\n";
99     for (int i = 0; i < n; i++) {
100         cout << d[i] << ", ";
101     }
102     cout << "\n\n";
103
104     cout << "GPU product is =>\n";
105     for (int i = 0; i < n; i++) {
106         cout << c[i] << ", ";
107     }
108     cout << "\n\n";
109
110     int error = 0;
111     for (int i = 0; i < n; i++) {
112         error += d[i] - c[i];
113         if (0 != (d[i] - c[i])) {
114             cout << "Error at (" << i << ") => GPU: " << c[i] << ", CPU: " << d[i] << "\n";
115         }
116     }
117
118     cout << "Error: " << error;
119     cout << "\nTime Elapsed: " << time;
120
121     return 0;
122 }
123
124 /*
125
126 OUTPUT:
127
128 Given matrix is =>
129 3 6 7 5 3 5 6 2 9 1
130 7 0 9 3 6 0 6 2 6 1
131 7 9 2 0 2 3 7 5 9 2
132 8 9 7 3 6 1 2 9 3 1
133 4 7 8 4 5 0 3 6 1 0
134 3 2 0 6 1 5 5 4 7 6
135 6 9 3 7 4 5 2 5 4 7
136 4 3 0 7 8 6 8 8 4 3
137 4 9 2 0 6 8 9 2 6 6
138 9 5 0 4 8 7 1 7 2 7
139
140 Given vector is =>
141 2, 8, 2, 9, 6, 5, 4, 1, 4, 2,
142
143 CPU product is =>
144 220, 147, 190, 201, 168, 171, 245, 235, 234, 210,

```

```
145
146 GPU product is  $\Rightarrow$ 
147 220, 147, 190, 201, 168, 171, 245, 235, 234, 210,
148
149 Error: 0
150 Time Elapsed: 0.014336
151
152 */
153
```

HPC/4/vectorAdd.cpp

```
1  #include <cstdlib>
2  #include <iostream>
3
4  #define checkCudaErrors(call) \
5      do { \
6          cudaError_t err = call; \
7          if (err != cudaSuccess) { \
8              printf("CUDA error at %s %d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
9              exit(EXIT_FAILURE); \
10         } \
11     } while (0)
12
13 using namespace std;
14
15 // VectorAdd parallel function
16 __global__ void vectorAdd(int *a, int *b, int *result, int n) {
17     int tid = threadIdx.x + blockIdx.x * blockDim.x;
18     if (tid < n) {
19         result[tid] = a[tid] + b[tid];
20     }
21 }
22
23 int main() {
24     int *a, *b, *c;
25     int *a_dev, *b_dev, *c_dev;
26     int n = 1 << 4;
27
28     a = new int[n];
29     b = new int[n];
30     c = new int[n];
31     int *d = new int[n];
32     int size = n * sizeof(int);
33     checkCudaErrors(cudaMalloc(&a_dev, size));
34     checkCudaErrors(cudaMalloc(&b_dev, size));
35     checkCudaErrors(cudaMalloc(&c_dev, size));
36
37     // Array initialization..You can use Randon function to assign values
38     for (int i = 0; i < n; i++) {
39         a[i] = rand() % 1000;
40         b[i] = rand() % 1000;
41         d[i] = a[i] + b[i]; // calculating serial addition
42     }
43     cout << "Given array A is ⇒\n";
44     for (int i = 0; i < n; i++) {
45         cout << a[i] << ", ";
46     }
47     cout << "\n\n";
48
49     cout << "Given array B is ⇒\n";
50     for (int i = 0; i < n; i++) {
51         cout << b[i] << ", ";
52     }
53     cout << "\n\n";
54
55     cudaEvent_t start, end;
56
57     checkCudaErrors(cudaEventCreate(&start));
58     checkCudaErrors(cudaEventCreate(&end));
59
60     checkCudaErrors(cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice));
61     checkCudaErrors(cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice));
62     int threads = 1024;
63     int blocks = (n + threads - 1) / threads;
64     checkCudaErrors(cudaEventRecord(start));
65
66     // Parallel addition program
67     vectorAdd<<<blocks, threads>>>(a_dev, b_dev, c_dev, n);
68
69     checkCudaErrors(cudaEventRecord(end));
70     checkCudaErrors(cudaEventSynchronize(end));
```

```

71
72     float time = 0.0;
73     checkCudaErrors(cudaEventElapsedTime(&time, start, end));
74
75     checkCudaErrors(cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost));
76
77     // Calculate the error term.
78
79     cout << "CPU sum is ⇒\n";
80     for (int i = 0; i < n; i++) {
81         cout << d[i] << ", ";
82     }
83     cout << "\n\n";
84
85     cout << "GPU sum is ⇒\n";
86     for (int i = 0; i < n; i++) {
87         cout << c[i] << ", ";
88     }
89     cout << "\n\n";
90
91     int error = 0;
92     for (int i = 0; i < n; i++) {
93         error += d[i] - c[i];
94         if (0 ≠ (d[i] - c[i])) {
95             cout << "Error at (" << i << ") ⇒ GPU: " << c[i] << ", CPU: " << d[i] << "\n";
96         }
97     }
98
99     cout << "\nError : " << error;
100    cout << "\nTime Elapsed: " << time;
101
102    return 0;
103 }
104
105 /*
106
107 OUTPUT:
108
109 Given array A is ⇒
110 383, 777, 793, 386, 649, 362, 690, 763, 540, 172, 211, 567, 782, 862, 67, 929,
111
112 Given array B is ⇒
113 886, 915, 335, 492, 421, 27, 59, 926, 426, 736, 368, 429, 530, 123, 135, 802,
114
115 CPU sum is ⇒
116 1269, 1692, 1128, 878, 1070, 389, 749, 1689, 966, 908, 579, 996, 1312, 985, 202, 1731,
117
118 GPU sum is ⇒
119 1269, 1692, 1128, 878, 1070, 389, 749, 1689, 966, 908, 579, 996, 1312, 985, 202, 1731,
120
121
122 Error : 0
123 Time Elapsed:  0.017408
124
125 */
126

```