

CSCE 435 Group 16 project

0. Group number: 16

1. Group members:

1. Anjali Kumar
2. Shawn Mathen
3. Ashrita Vadlapatla
4. Robert Eads

2. Project topic (e.g., parallel sorting algorithms)

Parallel Sorting Algorithms

2a. Communication Method

Our team's primary method of communication will be GroupMe with Slack as a secondary method.

2b. Brief project description (what algorithms will you be comparing and on what architectures)

Each of the selected sort algorithms, Bubble, Merge, Quick, & Sample will be run in parallel using MPI and CUDA separately.

2c. Pseudocode for each parallel algorithm

Algorithm 1: Bubble Sort

MPI

1. Distribute the data among processes using `MPI_Scatter`.
2. Each process performs a local sequential Bubble Sort on its portion of the data.
3. Exchange data with neighboring processes using `MPI_Send` and `MPI_Recv` for each iteration of the Bubble Sort.
4. Perform multiple iterations of the Bubble Sort while exchanging and comparing adjacent elements to sort the entire dataset.
5. Repeat the Bubble Sort and data exchange steps for the required number of iterations (in parallel) to ensure a fully sorted dataset.
6. Verify the correctness of the sorted data.

CUDA

1. Each CUDA thread loads a portion of the data into shared memory.
2. Perform a local sequential Bubble Sort within each thread's shared

memory.

3. Synchronize threads to ensure all shared memory sorting is complete.
4. No need for pivot or reduction steps, as Bubble Sort doesn't require pivot-based partitioning.
5. Each thread can individually compare and swap adjacent elements in its partition as part of the Bubble Sort.
6. No offset calculations are needed in Bubble Sort.
7. There is no need for a scatter operation, as Bubble Sort is an in-place sorting algorithm.
8. Continue the Bubble Sort for the required number of iterations, ensuring all elements are correctly sorted.
9. There's no merging step in Bubble Sort, as it's an exchange-based sorting algorithm that works directly on the data in place.

General Pseudocode

```
begin BubbleSort(list)

    for all elements of list
        if list[i] > list[i+1]
            swap(list[i], list[i+1])
        end if
    end for

    return list

end BubbleSort
```

MPI Pseudocode

```
void parallelBubbleSort(vector<int> &localData) {
    int myRank, numProcesses;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);

    int localN = localData.size();

    for (int i = 0; i < localN - 1; i++) {
        for (int j = 0; j < localN - i - 1; j++) {
            if (localData[j] > localData[j + 1]) {
                swap(localData[j], localData[j + 1]);
            }
        }
    }
}
```

CUDA Pseudocode

```

__global__ void bubbleSort(int *array, int size) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < size) {
        for (int i = 0; i < size - 1; i++) {
            for (int j = 0; j < size - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
}

```

Algorithm 2: Merge Sort

MPI

1. Distribute data among processes
2. Each process performs a local sequential merge sort
3. Pairwise merging of sorted sublists using MPI_Send and MPI_Recv
4. Recursively perform merging until the data is fully sorted
5. Verify correctness

CUDA

1. Each CUDA thread loads a portion of the data into shared memory
2. Perform a local sequential merge sort within each thread's shared memory
3. Use CUDA parallel reduction to find the pivot elements
4. Broadcast to all threads
5. Each thread partitions its data around pivot
6. Calculate the offsets for each partition
7. CUDA scatter to move elements to their respective partitions
8. Recursively sort each partition using CUDA parallel sort
9. Perform parallel merge or merge in shared memory, depending on the size of the partitions

Pseudocode

Parallel: The following parallel Merge Sort algorithms will be implemented using MPI.

```

function parallel_merge_sort(arr, num_threads)
    if length(arr) <= 1
        return arr

    mid = length(arr) // 2
    left = arr[0...mid-1]
    right = arr[mid...]

    if num_threads > 1
        left_thread = start_thread(parallel_merge_sort(left,
num_threads/2))
        right_thread = start_thread(parallel_merge_sort(right,
num_threads/2))
        join_thread(left_thread)
        join_thread(right_thread)

    else
        left = merge_sort(left)
        right = merge_sort(right)

    return parallel_merge(left, right)

```

The following is pseudocode for a CUDA implementation

```

__global__ void mergeCUDA(int* arr, int left, int mid, int right) {
    // Sequential merge
}

__global__ void mergeSortParallelCUDA(int* arr, int size) {
    if (size <= 1) return;

    int mid = size / 2;

    int* left = arr;
    int* right = arr + mid;

    mergeSortParallelCUDA<<<1, 1>>>(left, mid);
    mergeSortParallelCUDA<<<1, 1>>>(right, size - mid);

    mergeCUDA(arr, 0, mid - 1, size - 1);
}

```

Algorithm 3: Quick Sort

MPI

1. Distribute the data among processes using MPI_Scatter.
2. Each process applies the quicksort algorithm to its portion of the data.
3. Communicate with neighboring processes to partition and exchange data using MPI_Send and MPI_Recv.
4. Recursively apply quicksort on the subarrays, ensuring elements are correctly placed relative to the pivot.
5. Repeat the quicksort and data exchange steps for the required number of iterations (in parallel) to ensure a fully sorted dataset.
6. Verify the correctness of the sorted data.

CUDA

1. Each CUDA thread loads a portion of the data into shared memory
2. Apply quicksort within shared memory.
3. Use CUDA parallel reduction for pivot selection.
4. Broadcast pivot elements to all threads.
5. Partition data around pivot.
6. Calculate offsets for each partition.
7. Use CUDA scatter to move elements to partitions.
8. Recursively apply quicksort to each partition.
9. Optionally perform parallel merging (depending on partition size).

```
int partition(int arr[], int low, int high) {

    int pivot = arr[high];
    int i = (low-1);

    for (int j = low; j <= high; j++) {
        if (arr[j] < pivot) {
            i++;
            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);
    }
}
```

Algorithm 4: Sample Sort

MPI

```

FOR number of splitters selected by each process
    Push back randomly selected value from local data to sample vector
Allocate memory for all sampled splitters
MPI_Allgather sampled splitters to each process
Sort sampled splitters and choose bucket bounds from throughout sampled
splitters
Evaluate local elements and place into send buffers
Calculate required buffer size from each process
FOR number of processes
    MPI_Gather the buffer sizes from each process on each process
Allocate receive buffer for incoming data
FOR number of processes
    MPI_Gatherv to send data to the correct process
Sort with sequential quicksort

```

CUDA

```

*All steps are performed with CUDA kernal with the expection of data
transfer*
Sample and select splitters to use
Sort samples
Gather required info about each bucket
Group elements into buckets within an array
Sort the data within each bucket
Transfer data from GPU to CPU

```

2d. Citations

- https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm
- <https://compucademy.net/algorithmic-thinking-with-python-part-3-divide-and-conquer-strategy/#:~:text=There%20is%20a%20really%20clever%20trick%20that,the%20same%20type%20as%20the%20original%20problem.>
- <https://teivah.medium.com/parallel-merge-sort-in-java-e3213ae9fa2c>
- <https://www.geeksforgeeks.org/quick-sort/>
- <https://cse.buffalo.edu/faculty/miller/Courses/CSE702/Nicolas-Barrios-Fall-2021.pdf>
- <https://en.wikipedia.org/wiki/Samplesort>
- <https://www.geeksforgeeks.org/cpp-program-for-quicksort/#>
- <https://pushkar2196.wordpress.com/2017/04/19/mergesort-cuda-implementation/>

2e. Evaluation plan - what and how will you measure and compare

Each algorithm will be run with the input types of sorted, reverse sorted, and randomized. The input sizes are planned to be 2^{16} , 2^{20} , and 2^{24} . Both weak and strong scaling will be analyzed and compared against the other algorithms being tested. The number of threads in a block on the GPU will be [64, 128, 512, 1024].