CSCE 435 Group project

0. Group number: 12

12

1. Group members:

1. Christion Bradley
2. Warren Wu
3. Sua Bae
4. Kunal Jain

- communication via: iMessage & Discord

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

Parallel sorting is incredibly important for processing large amounts of data efficiently on distributed systems/multicore processors.

This project focuses on various parallel sorting algorithms and their implementations, with the goal of understanding how they can be optimized for perfomance on parallel architechture. Specifically, we will compare the parallel performance, implementation complexity, and memory usage of Bitonic Sort, Sample Sort, Merge Sort, and Radix Sort using the Message Passing Interface.

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

- Bitonic Sort: Bitonic sort repeatedly creates bitonic sequences, which first monotonically increases and then monotonically decreases, then merges them into a fully sorted sequence through performing pairwise comparisons and swaps between elements. This sorting algorithm will be using MPI on TAMU Grace Cluster by dividing the array into multiple processes, locally sorting the subarrays within each process, then finally merging them in a parallel fashion, until the entire array is sorted.
- Sample Sort: An irregular algorithm similar to parallelized bucket sort except there are p-1 pivot points. Sample sort first splits an array of size n into different processes that is then sorted locally. From there the "splitters" are found and the local sequence is split into different sequences according to the splitters. Lastly, for each of the sequences they are merged locally
- Merge Sort: For a list of size n, Merge Sort recursively splits the list into n sublists, where each sublist contains a single element and is therefore sorted. These sublists are then combined into larger lists where smaller elements appear before larger elements. This is done until one list remains, which will be the final sorted list. We will be implementing this algorithm on the TAMU Grace Cluster using MPI to communicate between processes.
- Radix Sort: Radix sort sorts numbers digit by digit starting from the least to most significant digit. In parallelized radix sort, the array will be distributed among subprocesses, with each process performing the sorting for one or more digits. After sorting based on the current digit, the processes will communicate with each other to exchange data to ensure that the elements are properly distributed for the next iteration. This algorithm will also be implemented using MPI and ran on the Grace Cluster.

2b. Pseudocode for each parallel algorithm

```
Bitonic Sort:

void bitonic_merge(int a[], int low, int cnt, int dir)
{
  if (cnt > 1)
  {
    int k = cnt / 2;
    for (int i = low; i < low+k; i++)
      if (dir == (a[i] > a[i+k])) {
        int temp = arr[i];
        arr[i] = arr[i + k];
        arr[i + k] = temp;
      }
    bitonic_merge(a, low, k, dir);
    bitonic_merge(a, low+k, k, dir);
  }
}

void bitonic_sort(int a[], int low, int cnt, int dir)
{
  if (cnt > 1)
  {
    int k = cnt / 2;
    bitonic_sort(a, low, k, 1);
    bitonic_sort(a, low+k, k, 0);
    bitonic_merge(a, low, cnt, dir);
  }
}

int main() {
  int rank, size, n_workers, n;
  int* arr = array to be sorted;
  int* workers_arr = NULL;
  int* final_list = NULL;

  MPI_Init();
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &n_workers);

  if (rank == 0) {
    n = sizeof(arr) / sizeof(arr[0]);
  }

  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  int local_size = n / n_workers;
  workers_arr = (int*)malloc(local_size * sizeof(int));

  MPI_Scatter(arr, local_size, MPI_INT, workers_arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD);
  bitonic_sort(workers_arr, 0, local_size, 1);
  if (rank == 0) {
      final_list = (int*)malloc(n * sizeof(int));
  }

  MPI_Gather(workers_arr, local_size, MPI_INT, final_list, local_size, MPI_INT, 0,
MPI_COMM_WORLD);

  if (rank == 0) {
    bitonic_sort(final_list, 0, n, 1);
    free(final_list);
  }

  free(workers_arr);

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

```
Merge Sort:

void merge(int arr[], int temp[], int left, int right) {
  int b = left;
  int l = left;
  int middle = (left + right) / 2;
  int r = middle + 1;

  while (l <= middle && r <= right) {
    if (arr[l] <= arr[r]) {
      temp[b++] = arr[l++];
    } else {
      temp[b++] = arr[r++];
    }
  }

  while (l <= middle) {
    temp[b++] = arr[l++];
  }

  while (r <= right) {
    temp[b++] = arr[r++];
  }

  for (int i = left; i <= right; i++) {
    arr[i] = temp[i];
  }
}

void merge_sort(int arr[], int temp[], int left, int right) {
  if (left < right) {
    int middle = (left + right) / 2;
    merge_sort(arr, temp, left, middle);
    merge_sort(arr, temp, middle + 1, right);
    merge(arr, temp, left, right);
  }
}
void merge_sort(int* arr, int* temp, int left, int right){
  if (left < right) {
  middle = (left + right) // 2;
  merge_sort(arr1, arr2, left, middle);
  merge_sort(arr1, arr2, middle + 1, right);
  merge(arr1, arr2, left, right);
}
}
int main() {
  arr = array of elements to be sorted;
  MPI_Init();
  int rank, n_workers;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &n_workers);

  int size = arr.size() / n_workers;
  int *workers_arr = malloc(size * sizeof(int));
  MPI_Scatter(arr, size, MPI_INT, workers_arr, size, MPI_INIT, 0, MPI_COMM_WORLD);

  int *w_arr = malloc(size * sizeof(int));
  mergeSort(workers_arr, w_arr, 0, size - 1);

  if (rank == 0) {
    final_list = malloc(arr.size() * sizeof(int));
   }

  MPI_Gather(worker_arr, size, MPI_INT, final_list, size, MPI_INT, 0,
MPI_COMM_WORLD);

  if (rank == 0) {
    int* helper_arr malloc(arr.size() * sizeof(int));
    mergeSort(final_list, helper_arr, 0, arr.size() - 1);
    print(final_list);
    free(final_list); free(helper_arr);
  }

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();

}
```

```
Sample Sort

int main() {
    arr = array of elements to be sorted;
    MPI_Init();
    MPI_Comm_rank(...);
    MPI_Comm_size(...);

    int localSize = arr.size() / size;
    localData = sequence of elements of size localSize;
    MPI_Scatter(arr, localSize, MPI_INT, localData, localSize, MPI_INT, 0,
MPI_COMM_WORLD);

    sort(localData);

    localSortedData[arr.size()];
    MPI_Gather(localData, arr.size(), MPI_INT, localSortedData, arr.size(),
MPI_INT, 0, MPI_COMM_WORLD);

    splitter[size-1];
    for i=0 to size-1 {
        splitter[i] = localData[arr.size()/(size*size) * (i+1)];
    }

    allSplitters[size*(size-1)];
    MPI_Gather(splitter, size-1, MPI_INT, allSplitters, size-1, MPI_INT, 0,
MPI_COMM_WORLD);

    sort(allSplitters);
    for i=0 to size-1 {
        splitter[i] = allSplitters[(size-1)*(i+1)];
    }
    MPI_Bcast(spliiter, size-1, MPI_INT, 0, MPI_COMM_WORLD);

    buckets[size];
    for i=0 to localSize {
        int bucketIndex = 0;
        while (bucketIndex < size-1 and localData[i] > allSplitters[bucketIndex])
{
            bucketIndex++;
        }
        buckets[bucketIndex].append(localData[i]);
    }

    bucketBuffer[arr.size()+size]
    MPI_Alltoall(buckets, localSize+1, MPI_INT, bucketBuffer, localSize+1,
MPI_INT, MPI_COMM_WORLD);

    localBucket[2*arr.size()/size];
    counter = 1;
    for i=0 to size {
        incrementer = 1;
        for j=0 to bucketBuffer[(arr.size()/size + 1)*i] {
            localBucket[counter] = bucketBuffer[(arr.size()/size +
1)*i+incrementer];
            incrementer++;
            counter++
        }
    }
    localBucket[0] = counter-1;
    sort(localBucket);

    sortedArr[arr.size()];
    MPI_Gather(localBucket, arr.size(), MPI_INT, sortedArr, arr.size() , MPI_INT,
0, MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```

```
Radix Sort:

int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

void countSort(int arr[], int n, int exp) {
    int* output = (int*)malloc(n * sizeof(int));
    int count[10] = {0};

    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (int i = 0; i < n; i++)
        arr[i] = output[i];

    free(output);
}

void radixSort(int arr[], int n) {
    int m = getMax(arr, n);

    for (int exp = 1; m / exp > 0; exp *= 10) {
        countSort(arr, n, exp);
    }
}
```

```
int main(int argc, char** argv) {
    int rank, size, n;
    int* arr = NULL;
    int* local_arr = NULL;

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

    if (rank == 0) {
        n = /* size of the array */;
        arr = (int*)malloc(n * sizeof(int));
    }

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    int local_size = n / size;
    local_arr = (int*)malloc(local_size * sizeof(int));

    MPI_Scatter(arr, local_size, MPI_INT, local_arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD);

    radixSort(local_arr, local_size);

    MPI_Gather(local_arr, local_size, MPI_INT, arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        radixSort(arr, n);
    }

    free(local_arr);
    if (rank == 0)
        free(arr);

    MPI_Finalize();
    return 0;
}
```

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

- Input sizes, Input types: We will measure the time in seconds that it takes for each algorithm to sort a given array of elements. We will sort arrays of consisting of integers with size 128, 1024, and 8192 to determine the speed of each algorithm.
- Strong scaling (same problem size, increase number of processors/nodes) We will increase the number of processes by powers of two: 2, 4, 8, 16, 32, and 64 processes. If an algorithm parallelizes well, then increasing the number of processes should continue to decrease the computation time.

- Weak scaling (increase problem size, increase number of processors) We will increase the size of the array (128, 1024, and 8192) alongside increasing the number of processes (2, 4, 8, 16, 32, 64) in order to determine how well these algorithms can be parallelized.