Cloud Computing

Special Task 2 - Parallel Merge Sort with MPI – Summer Term 2018

Prachi Agrawal, Henry Cocos, David Merkl, Samuel Santos

{agrawal,cocos,merkl,santos}@stud.fra-uas.de

Computer Science
Faculty of Computer Science and Engineering
Frankfurt University of Applied Sciences

July 4th 2018

Contents

1. Theory
2. Implementation
3. Results
4. Conclusion
5. References

Merge Sort Pseudo Code I

Algorithm 1 Pseudocode for Merge Sort [1]

```
function mergesort(List m)
    if length of m <= 1 then
return m
    end if
    left := emptylist
    right := emptylist
    for each x with index i in m do
        if i < (length of m) then
            add x to left
        else
            add x to right
        end if
    end for

    left := mergesort(left)
    right := mergesort(right)
     return merge(left, right)
end function
```

Merge Sort Pseudo Code II

Algorithm 2 Pseudocode for Merge Function [1]

```
function merge(List left, List right)
    result := emptylist
    while left is not empty and right is not empty do
        if first(left) <= first(right) then
            append first(left) to result
        else
            append first(right) to result
        end if
    end while

    while left is not empty do
        append first(left) to result
    end while

    while right is not empty do
        append right to result
    end while
return result
end function
```

Merge Sort Example

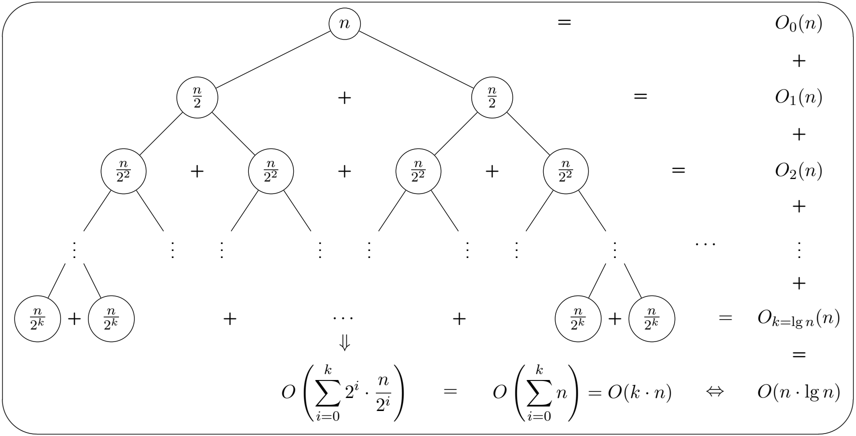

Figure: Recursion Tree Merge Sort [2]

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Source Code MPI Merge Sort I

```
/********** Initialize MPI **********/
int world_rank;
int world_size;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

/********** Divide the array in equal-sized chunks **********/
int size = n/world_size;

/********** Send each subarray to each process **********/
int *sub_array = malloc(size * sizeof(int));
MPI_Scatter(original_array, size, MPI_INT, sub_array, size, MPI_INT, 0,
  MPI_COMM_WORLD);

// Start Timer Sorting Parallel
parallelTime1 = MPI_Wtime();

/********** Perform the mergesort on each process **********/
int *tmp_array = malloc(size * sizeof(int));
mergeSort(sub_array, tmp_array, 0, (size - 1));
```

Listing 1: Merge Sort with MPI [3]

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Source Code MPI Merge Sort II

```c
/********** Gather the sorted subarrays into one **********/
int *sorted = NULL;
if(world_rank == 0) {
  sorted = malloc(n * sizeof(int));
  }

// Start Timer Sorting Parallel
parallelTime2 = MPI_Wtime();

MPI_Gather(sub_array, size, MPI_INT, sorted, size, MPI_INT, 0,
  MPI_COMM_WORLD);

// Sequential Last Merge Sort Start Timer
sequentialTime1 = MPI_Wtime();
/********** Make the final mergeSort call **********/
if(world_rank == 0) {

  int *other_array = malloc(n * sizeof(int));
  mergeSort(sorted, other_array, 0, (n - 1));
```

Listing 2: Merge Sort with MPI [3]

Source Code MPI Merge Sort III

```c
// Sequential Last Merge Sort Stop Timer
sequentialTime2 = MPI_Wtime();

/********** Clean up rest **********/
free(original_array);
free(sub_array);
free(tmp_array);

if(world_rank == 0)
{
// Time of Execution
printf("%i \t %.3f \t\t %f \t %f \t\t %f \n",numProc, (sequentialTime2 - 
  sequentialMasterRead1), (sequentialMasterRead2 - sequentialMasterRead1),
   (parallelTime2 - parallelTime1), (sequentialTime2 - sequentialTime1) );
}
/********** Finalize MPI **********/
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();

}
```

Listing 3: Merge Sort with MPI [3]

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Automated Tests - Shell Script

```
# Iterate over all Problem Sizes
for PROBLEM_SIZE in 1000 10000 100000 1000000 10000000 100000000 1000000000 10000000000 100000000000 1000000000000 10000000000000
do

# Print Header for Result File
echo -e "#| Num of Proc | Total Time Elapsed | File Read Part on Master | Parallel Part | Sequential Last Merge Call |" >> "$RESULT_DIR""$RESULT_FILE_NAME"_"$PROBLEM_SIZE.txt"

# Iterate over all Number of Processes
for NUM_PROC in 1 2 4 8 16 32 64 128 256 512
do

# Check if Number of Processes gets bigger than Number of Nodes
if [ "$NUM_PROC" -gt 128 ]
then
  NUM_NODE=128
else
  NUM_NODE=$NUM_PROC
fi

# Prompt Test Run on Console and Write to File
echo "Array Size: $PROBLEM_SIZE"
echo -e "Number of Processes: $NUM_PROC\nNumber of Nodes: $NUM_NODE\nProblem Size: $PROBLEM_SIZE\n"
echo -e "Number of Processes: $NUM_PROC\nNumber of Nodes: $NUM_NODE\nProblem Size: $PROBLEM_SIZE\n" >> "$LOG_DIR""$LOGFILE_NAME"_"$PROBLEM_SIZE.txt"

# Execute MPI with Parameters
mpirun -np $NUM_PROC --hostfile "$HOSTFILE_DIR"cluster_$NUM_NODE.txt $MPI_EXEC_NAME $PROBLEM_SIZE $NUM_PROC >> "$RESULT_DIR""$RESULT_FILE_NAME"_"$PROBLEM_SIZE.txt"


done
echo -e "Successful Test Run for $PROBLEM_SIZE !!!\n\n"
echo -e "Successful Test Run for $PROBLEM_SIZE !!!\n\n" >> "$LOG_DIR""$LOGFILE_NAME"_"$PROBLEM_SIZE.txt"

done

echo -e "All Test runs commpleted successfully!!!"
```

Listing 4: Shell Script for automated Tests

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Automated Test Runs

The Test runs were automated with the following Parameters:

- 1.000 entries
- 10.000 entries
- 100.000 entries
- 1.000.000 entries
- 10.000.000 entries
- 100.000.000 entries

The Test runs with 100.000.000 Entries produced errors!!!
Assumption is that Memory Size is not sufficient!!!

Memory

4 Bytes x 100.000.000 $\approx$ 400 Mbytes
2 x 400 Mbytes $\approx$ 800 Mbytes
Raspberry Pi 3 has only 1 Gbytes

KFURT
ERSITY
IENCES

Automated Tests - Result File

```
#| Num of Proc | Total Time Elapsed | File Read Part on
    Master | Parallel Part | Sequential Last Merge Call |
1      0.051      0.000231      0.000848       0.000724
2      0.074      0.000230      0.000391       0.000741
4      0.080      0.000230      0.000180       0.000761
8      0.094      0.000229      0.000097       0.000837
16     0.115      0.000229      0.000042       0.000801
32     0.220      0.000383      0.000024       0.000817
64     0.273      0.000232      0.000015       0.000842
128      0.511      0.000382      0.000011       0.000845
256      0.748      0.000384      0.000011       0.000841
512      1.947      0.000386      0.000008       0.001059
```

Listing 5: Example Results File for 1000 Entries

Automated Tests - Log File

```
Number of Processes: 1
Number of Nodes: 1
Problem Size: 1000

Number of Processes: 2
Number of Nodes: 2
Problem Size: 1000

Number of Processes: 4
Number of Nodes: 4
Problem Size: 1000

Number of Processes: 8
Number of Nodes: 8
Problem Size: 1000

Number of Processes: 16
Number of Nodes: 16
Problem Size: 1000
```

Listing 6: Example Log File for 1000 Entries

Results

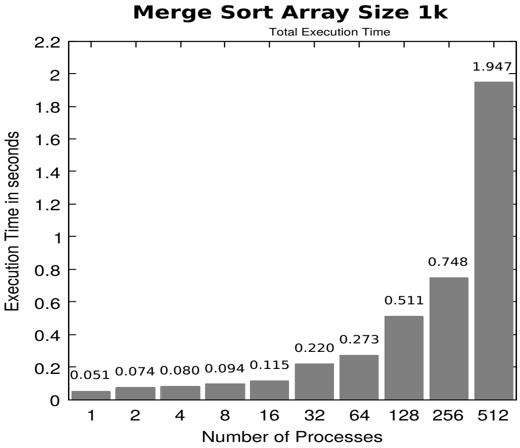

Figure: Plot for 1 Thousand Entries

Results

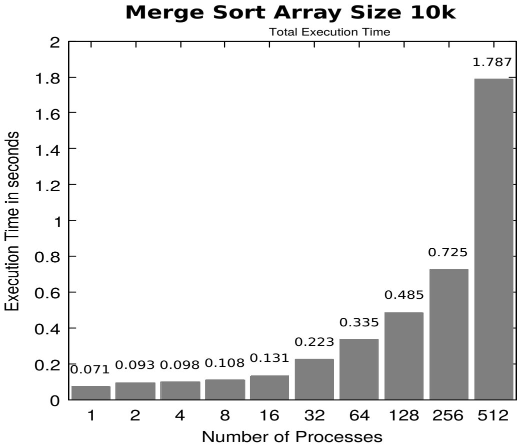

Figure: Plot for 10 Thousand Entries

Results

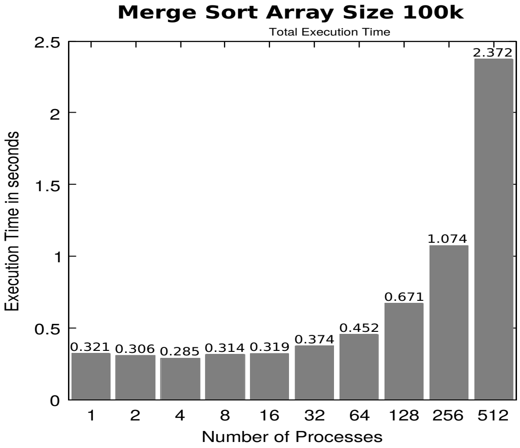

Figure: Plot for 100 Thousand Entries

Results

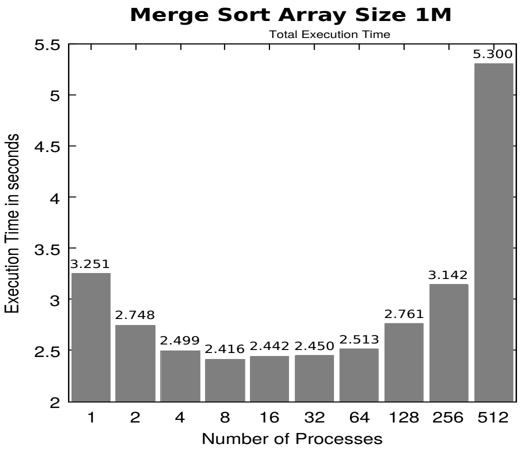

Figure: Plot for 1 Million Entries

Results

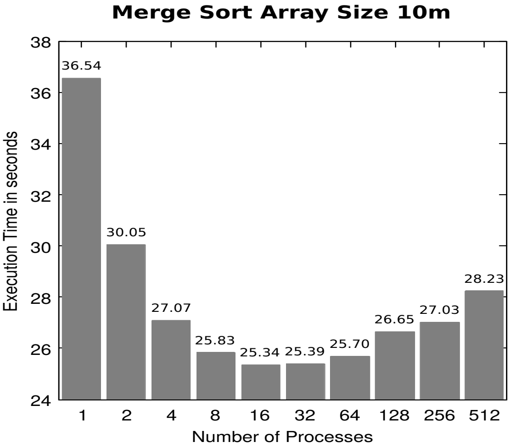

Figure: Plot for 10 Million Entries

Conclusion

Figure 6 demonstrates the following things:

- The Problem is scaling
- Increasing the Number of Processes on small problems leads to communication overhead
- Increasing the Problem Size leads to better performance

References I

[1] "Merge Sort Wikipedia," [Accessed: July 4, 2018]. [Online]. Available: https://en.wikipedia.org/wiki/Merge_sort

[2] "Merge Sort Recursion Tree," [Accessed July 4, 2018]. [Online]. Available: http://www.texample.net/tikz/examples/merge-sort-recursion-tree/

[3] "Merge Sort MPI Implementation," [Accessed: July 4, 2018]. [Online]. Available: https://github.com/racorretjer/Parallel-Merge-Sort-with-MPI