# Parallel Merge Sort

Christopher Zelenka
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000

Zelenka.Chris@gmail.com

## ABSTRACT
In this paper, I present the differences in efficiency on merge sort and it's parallelization. In order to study the differences in the efficiency, I have done research on how to implement parallelization on merge sort and compare the two. I use an i7-3700 CPU that contains 4 cores and run Ubuntu on a Virtual Machine. MPI is implemented to help with the communication between the processors.

## 1. INTRODUCTION
Sorting is the process of reordering a sequence of elements and creating the same set of elements ordered according to an attribute. During a sorting phase, many elements are left idle for certain periods of time, while other elements are being put into their correct position. With the use of parallel sorting, less elements are left idle, increasing the efficiency of the sorting algorithm. However, not every non-parallel sorting algorithm can be easily implemented into a parallel sorting algorithm. Merge sort is one of the easiest algorithms to implement parallel sorting, due to the nature of divide and conquer, allowing us to compare between the parallel and non-parallel algorithms based on time. With this, we can have a better representation and understanding of how powerful parallel sorting can be.

## 2. Merge Sort

### 2.1 What is it?
Merge sort was invented by John von Neumann in 1945, Neumann an applied mathematician, physicist, and polymath. Not much is documented on his invention of merge sort. Merge sort is a divide and conquer type algorithm, allowing us to easily implement a parallel sorting algorithm. Divide and conquer algorithms are those that divide a problem into many subsets and attempt to solve each subset, in the case of merge sort, merge sort would constantly divide every subset until there is only a single element in each subset and to solve the subsets, merge sort will start to merge each subset together, placing each element to their according position.

### 2.2 Problems
There is a problem with the non-parallel algorithm of merge sort that can be optimized from parallel sorting, which is the idle time of the halves that merge sort does not step into until later. Merging allows for any sorted array of elements to be merged with another sorted array, this allows for parallel programing to try and push out two sorted array of elements as fast as possible and have get them to merge.

### 2.3 Efficiency
Efficiency plays a huge role in sorting algorithms, it tells us how fast an algorithm is able to complete it's given task. This is usually measured by how many times the algorithm has to access an element within a given set of N elements. Not every set of N elements is able to produce the same time of completion, so we try and measure both the worst run time and the best run times.

**Table 1. Sequential MergeSort Efficiency**

| **Worst Case** | **Best Case** | **Averge Case** |
|---|---|---|
| O(n log n) | O(n log n) | O(n log n) |

Merge sort's efficiency is consistent for any case that it runs through and is always stable.

### 2.4 Parallel algorithm
With the use of MPI, implementing the parallel sorting algorithm for merge sort was very straight forward. MPI allows for the processes to communicated with one another and send / receive data or information. I chose to use MPI mainly for these reasons, this would allow me send a subset to a child process and allow for it to run a sequential merge sort and return the subset back as an ordered set. The parent process would simply receive the ordered set of elements from the child process and merge it with other ordered sets of elements. This elements the amount of idle time depending on how many processes are created at the start of the program.

## 3. Pseudo Code

### 3.1 Sequential
This is the merge sort algorithm, in which the first and second halves of an array are each sorted recursively and then merged.

**Listing1. Mergesort Pseudocode**

```
mergesort(int[] a, int left, int right):
        IF number of elements <= 1
                return
```

middle = (left + right) / 2

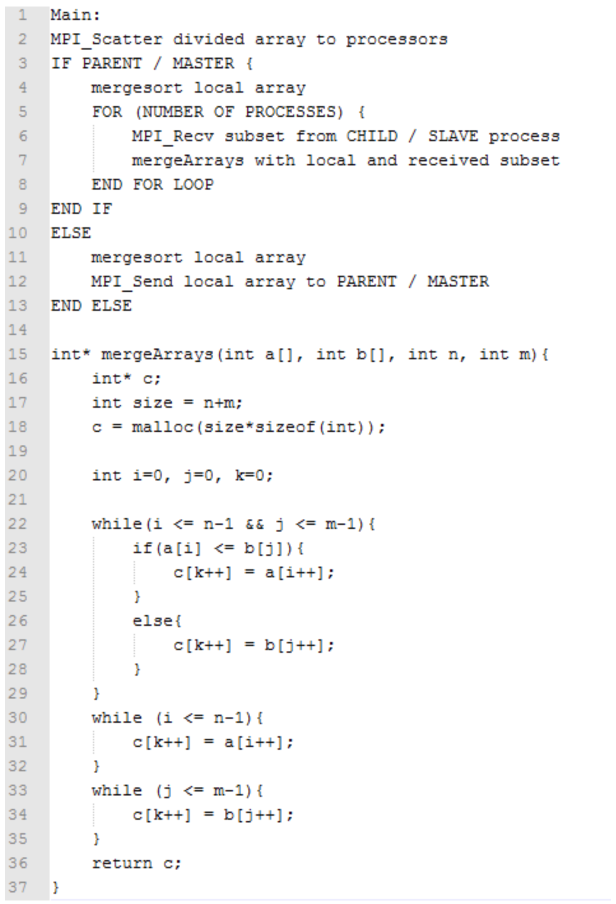
mergesort(a, left, middle-1)

mergesort(a, middle, right)

merge both halves

## 3.2 Parallel

The parallel algorithm utilizes the sequential algorithm, allowing for the parallel algorithm to run P times faster, where P is the number of processors being used.



```
Main:
MPI_Scatter divided array to processors
IF PARENT / MASTER {
    mergesort local array
    FOR (NUMBER OF PROCESSES) {
        MPI_Recv subset from CHILD / SLAVE process
        mergeArrays with local and received subset
    END FOR LOOP
END IF
ELSE
    mergesort local array
    MPI_Send local array to PARENT / MASTER
END ELSE

int* mergeArrays(int a[], int b[], int n, int m){
    int* c;
    int size = n+m;
    c = malloc(size*sizeof(int));

    int i=0, j=0, k=0;

    while(i <= n-1 && j <= m-1){
        if(a[i] <= b[j]){
            c[k++] = a[i++];
        }
        else{
            c[k++] = b[j++];
        }
    }
    while (i <= n-1){
        c[k++] = a[i++];
    }
    while (j <= m-1){
        c[k++] = b[j++];
    }
    return c;
}
```

**Figure 1. Parallel pseudo code**

## 4. MPI

MPI, known as Message Passing Interface, is the main tool used in the parallel merge sort algorithm to produce parallelization. MPI allows the user to set up how many processes to start with, along with a large library of functions and tools as listed below.

## 4.1 Utilizing MPI

Since MPI allows for the communication of processes, this allows the algorithm to split up the work necessary in order for the processes to run evenly. The following functions were used from the MPI header file:

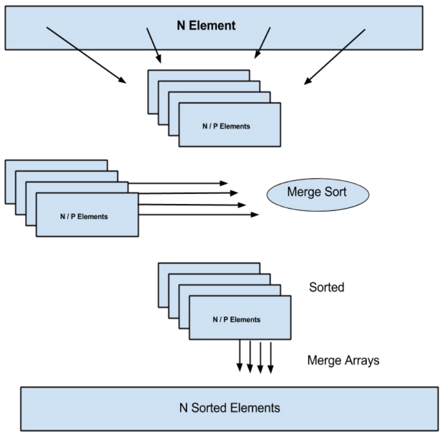


***Figure 2. Parallel merge sort in action***

### *1. MPI_Scatter*

The MPI_Scatter function is able to scatter elements in an array into every process. This function is used to scatter the array that should be sorted into the four processes. The scatter splits the elements in the array evenly between every process, allowing the process to all have the same amount of work load. After each process receives their portion of the array of elements that was scattered, each process runs a sequential merge sort on the given array and quickly send it back to the parent process.

### *2. MPI_Recv*

When MPI_Recv is called, the process is put on wait until a specified source sends over data. The source can however be changed so that it can receive from any incoming source, allowing for any process that completes in timely order to send over an ordered array. MPI_Recv is only called in the parent process, once data has been received, the parents process would then merge the data received into the data the parent process is currently holding. This repeats for however many other processes there are.

### *3. MPI_Send*

This function is only used in the child processes. After the child process completes the sequential merge sort on the array it was given, the child would then push the completed array to the parent process. After the parent receives the completed array, the child processes task is complete.

## 5. Code

## 5.1 Main function

Shown below in Listing 5.1 is the main function. The main handles all of the parallel coding with MPI. All the variables are first initialized and MPI is also initialized. After everything is initialized, the array of random numbers is created and quickly divided and scattered to the amount of processes inputted by the user. The parent acts as a collector, collecting all the child processes sorted arrays and merging them to the global array.

**Listing 1. Main function**

```
int main(int argc, char* argv[]){
	int i, a_size = NUM, local_size;
	int numtasks, rank, dest, source, rc, count, tag=1, j;
	int a[NUM];
	int global[NUM];
	int* comp;

	MPI_Status Stat;
	MPI_Request req;
	MPI_Init(&argc,&argv);
	MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);

	//Local array for every process
	int local[(NUM/numtasks)];

	srand(time(NULL));

	//Setup array with random numbers
	for(i=0; i<NUM; i++)
		a[i] = rand()%100000;

	//Scatter and split array evenly for each process
	MPI_Scatter(a, NUM/numtasks, MPI_INT, local, NUM/numtasks, MPI_INT, 0, MPI_COMM_WORLD);

	local_size = NUM/numtasks;

	if(rank == 0){//Parent Process
		clock_t begin, end;
		double time_spent;
		begin = clock();
		//Sequential merge sort
		mergesort(local, 0, local_size-1);
		//Push sorted local array to global array
		l2g(global, local, local_size);
		int j, recv_size = local_size;
		int buff[recv_size];
		for(j=0; j<numtasks-1; j++){
			//Receive sorted array from child process
			MPI_Recv(buff, recv_size, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &Stat);
			//Merge received array and global array together
			comp = mergeArrays(global, buff, local_size, recv_size);
			local_size += recv_size;
			//Pointer to Array
			p2a(global, comp, local_size);
		}
		end = clock();
		time_spent = (double)(end-begin) / CLOCKS_PER_SEC;
		int k;
		//for (k=0; k<local_size;k++)
			//printf("%d\n", comp[k]);
		printf("Time spent (Parallel) : %f\n", time_spent);

		begin = clock();
		//Time sequential merge sort on same set of numbers
		mergesort(a, 0, NUM-1);
		end = clock();
		time_spent = (double)(end-begin) / CLOCKS_PER_SEC;
		printf("Time spent (Non-Parallel): %f\n", time_spent);

	}
	else{//Child process
```

```
                //Sequential mergesort the given array from
scatter
                mergesort(local, 0,  local_size-1);
                //send the sorted array to the parent process
                MPI_Send(local, local_size, MPI_INT, 0, 0,
MPI_COMM_WORLD);
        }
        MPI_Finalize();
}
```

## 5.2 Merge sort

**Listing2. Merge Sort code**

```
//merge sort
void mergesort(int array[], int left, int right){
        if(left < right) {
                int middle = (left + right) / 2;
                mergesort(array, left, middle);
                mergesort(array, middle+1, right);
                merge(array, left, middle, right);
        }
}

//merge
void merge(int array[], int left, int middle, int right){
        int temp[NUM];
        int i = left, j = middle+1, k = 0;

        while(i <= middle && j <= right){
                if(array[i] <= array[j])
                        temp[k++] = array[i++];
                else
                        temp[k++] = array[j++];
        }
        while (i <= middle)
                temp[k++] = array[i++];
        while(j <= right)
                temp[k++] = array[j++];

        k--;
        while(k >= 0) {
                array[left + k] = temp[k];
                k--;
        }
}
```

## 5.3 Merge Array

A child processes task is to sort the array that it was given and send the sorted array back to the parent so that the parent can perform a merge between the newly sorted array being sent over to the parents local array (In the end would contain the completed array sorted).

**Listing3. Merge Array function**

```
//Merge Arrays
//Since the child sends a sorted Array, this function will create a
new array and merge the 2 passed in array of local array and
received array.
int* mergeArrays(int a[], int b[], int n, int m){
        int* c;
        int size = n+m;
        c = malloc(size*sizeof(int));

        int i=0, j=0, k=0;

        while(i <= n-1 && j <= m-1){
                if(a[i] <= b[j]){
                        c[k++] = a[i++];
                }
                else{
                        c[k++] = b[j++];
                }
        }
        while (i <= n-1){
                c[k++] = a[i++];
        }
        while (j <= m-1){
                c[k++] = b[j++];
        }
        return c;
}
```

When looking closely to the mergeArray function in Listing5.2, it looks very similar to the merge function used by mergesort.

## 5.4 Helper functions

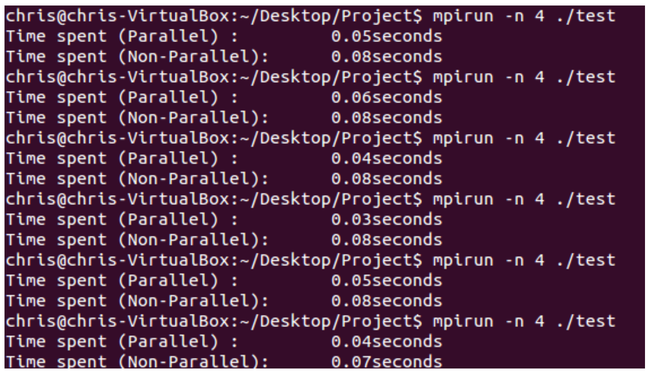
Since C is a low-level programming language, helper functions are created to help me handle arrays. In C, once and array is initialized with a given size (besides an array pointer) that given array can never change in size. These helper functions help me overcome these obstacles.

**Listing2. Helping functions**

```
//Pointer to Array

//mergeArray sends back a pointer to an array, this function
pushes everything from the pointer to the global array

void p2a(int a[], int* b, int size){

        int i;

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

                a[i] = b[i];

        }

}


//Local to Global

//Pushes local contents into the global array, only used for parents
initial push of sorted data into global

void l2g(int a[], int b[], int size){

        int i;

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

                a[i]=b[i];

}
```

## 6. Running the code

The parallel merge sort algorithm was test and ran on a virtual machine Ubuntu with an i7-3770 CPU containing 4 cores. The algorithm shows the potential of the parallel implementation of merge sort. With 400,000 elements being sorted, the parallel sorting algorithm on average ran twice as fast as the non-parallel algorithm.



```
chris@chris-VirtualBox:~/Desktop/Project$ mpirun -n 4 ./test
Time spent (Parallel) :         0.05seconds
Time spent (Non-Parallel):      0.08seconds
chris@chris-VirtualBox:~/Desktop/Project$ mpirun -n 4 ./test
Time spent (Parallel) :         0.06seconds
Time spent (Non-Parallel):      0.08seconds
chris@chris-VirtualBox:~/Desktop/Project$ mpirun -n 4 ./test
Time spent (Parallel) :         0.04seconds
Time spent (Non-Parallel):      0.08seconds
chris@chris-VirtualBox:~/Desktop/Project$ mpirun -n 4 ./test
Time spent (Parallel) :         0.03seconds
Time spent (Non-Parallel):      0.08seconds
chris@chris-VirtualBox:~/Desktop/Project$ mpirun -n 4 ./test
Time spent (Parallel) :         0.05seconds
Time spent (Non-Parallel):      0.08seconds
chris@chris-VirtualBox:~/Desktop/Project$ mpirun -n 4 ./test
Time spent (Parallel) :         0.04seconds
Time spent (Non-Parallel):      0.07seconds
```

**Figure 4. Timing between parallel and sequential**

## 7. Optimization

After running many tests on different sets of elements with the parallel merge sort, I notice that my code is flawed, there are moments of idle time in the child processes, waiting for the parent process to receive an ordered array of elements. This can be handled by allowing the child processes to communicate with one another and have them merge both of their sorted arrays together then have that sent over to the parent process. In doing so, this would greatly increase the performance of the parallel merge sort algorithm. I believe that running on a 4-core machine and having the program run at 4 processes, the run-time of the parallel merge sort should be cut down by a quarter of the non-parallel merge sorts run-time.

## 8. Conclusion

In summary, parallel sorting of any kind is very efficient, even with an algorithm that allows for a process to idle for a little bit, the increase in efficiency is dramatic.

In summary, parallel sorting is very effective. Even with a code that allows for a processor to idle, the increase in efficiency is dramatic with a powerful computer. Parallel sorting is not limited to just merge sorting or any other sequential sorting that are based on divide and conquer. In my research I was able to find a list of many other sorting methods using parallel sorting

## 9. REFERENCES

[1] A Specimen of Parallel Programming: Parallel Merge Sort Implementation.

http://penguin.ewu.edu/~trolfe/ParallelMerge/ParallelMerge.html

[2] 11.4MergeSort
http://www.mcs.anl.gov/~itf/dbpp/text/node127.html

[3] Parallel Merge Sort

http://www.drdobbs.com/parallel/parallel-merge-sort/229400239

[4] Parallel Sorting Algorithms

http://einstein.drexel.edu/courses/PHYS405/sort_algorithms/sort_algorithm.html