Fast Parallel Sorting Algorithms

Each location with the final k bits set to zero and the other (log n) - k bits matching the appropriate bits of an active processor's address will be marked after k iterations. This means that any one of two thousand computers will flag each such spot. A technique for designing algorithms that takes into account space, time, and processing power is provided. For the first time, Muller and Preparata demonstrated a network that could sort n numbers in O time (log n). When first introduced, parallel bucket-sorting algorithms minimise the number of processors and the amount of time utilised at the expense of increased space requirements. However, these algorithms are unique in that the space requirements are more than the processor time requirements. The model is SIMD-based, hence it takes O(log n) time to sort n integers using n (parallel) processors and remove duplicates. There will be a value to be sorted for each processor, and since there may be two processors with the same value, only one of them should be operational at any given time.

It has been decided that m memory locations will be used, one for each container. Each region will have the same size, n, as the number of numbers in the input set. Processors are assigned integer values from 0 to n-1, and if one of their "Buddy" processors with a higher rank is already running, the current processor will stop running as well. Each location with the final k bits set to zero and the other (log n) - k bits matching the appropriate bits of an active processor's address will be marked after k iterations. This means that any one of two thousand computers will flag each such spot. The execution of this algorithm takes time T=O(log n), needs space S=O(mn), and can run on n processors.

The relative order of the input integers will be preserved by a different algorithm we have, and it will return the rankings. The algorithm picks a single instance of each digit and adds it to the total as many times as the digit appears there. We observe that Algorithm 2 needs S = O(mn) space, T = O(log n + log m) time, and n processors.

Memory locations are set to zero before these algorithms are run. Memory fetch conflicts can be avoided by prohibiting access to the same area at the same time. Because of this, if either of the two processes for which the location could be important is running, it will be initialised.

To solve this problem, the third technique employs a parallel bucket sort with n3/2 processors and O (log n). This algorithm divides the input integers into n12 groups, with each group being processed by a set of n processors. Then, we apply bucket sort to each group, followed by binary search to each group, which yields a count for each 'jth' element, and finally, we use this count value to finish bucket sort.

The fourth approach, parallel bucket sort, employs n4/3 processors and has an O (n squared) complexity (log n). This algorithm divides the input numbers into n23 groups, each of which contains n13 elements/sectors. Next, perform a binary search within each sector to locate the count value for each 'jth' element, and finally, utilise that count value to finish the bucket sort. As a result, we can find a way to sort n numbers using n 1+1/k processors that takes O(k log n) time.

Implementation:

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<time.h>
#include<math.h>
#include<mpi.h>
#include<io.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
#include <Windows.h>
#include <stdint.h>

#define WIN32_LEAN_AND_MEAN

int world_size;
long n;
int *vector_serial;
int *vector_parallel;
int *temp;
int *pivots;
int *local_vector_parallel;
int local;
int my_rank;
int *local_arr;
int *bucket;
int num;
int *recv_bucket;

struct node {
    int value;
    struct node *next;
};
```

```c
typedef struct node node;

struct bucket {
    int size;
    struct node *linkedList;
};
typedef struct bucket bucket;

typedef struct timeval {
    long tv_sec;
    long tv_usec;
} timeval;

int gettimeofday(struct timeval* tp, struct timezone* tzp){
    static const uint64_t EPOCH =
     ((uint64_t)116444736000000000ULL);

    SYSTEMTIME  system_time;
    FILETIME    file_time;
    uint64_t    time;

    GetSystemTime(&system_time);
    SystemTimeToFileTime(&system_time, &file_time);
    time = ((uint64_t)file_time.dwLowDateTime);
    time += ((uint64_t)file_time.dwHighDateTime) << 32;

    tp->tv_sec = (long)((time - EPOCH) / 10000000L); tp-
    >tv_usec = (long)(system_time.wMilliseconds * 1000);
    return 0;
}

// Returns 0 on success and 1 on failure
int serialsort(int size, int unsorted[], int temp1[]){
    if(!(mergeSort(0, size -1, unsorted, temp1))){
```

```
        return 0;
    }else{
        return 1;
    }
}

// Serial mergesort
int mergeSort(int s, int end, int unsorted[], int temp1[]){
    if(s >= end){
        return 0;
    }
    int middle = ((end + s) / 2); mergeSort(s,
    middle, unsorted, temp1);
    mergeSort(middle+1, end, unsorted, temp1);
    merge(s, middle, end, unsorted, temp1);
    return 0;
}

int merge(int s, int middle, int end, int unsorted[], int
     temp1[]){
    int first = s;
    int second = middle+1;
    int tempIndex = s;
    while(first <= middle && second <= end){
        if(unsorted[first] < unsorted[second]){
            temp1[tempIndex] = unsorted[first];
            first++;
            tempIndex++;
        } else {
            temp1[tempIndex] = unsorted[second];
            second++;
            tempIndex++;
        }
    }
```

```
    while(first <= middle){
        temp1[tempIndex] = unsorted[first];
            first++;
            tempIndex++;
    }
    while(second <= end){
        temp1[tempIndex] = unsorted[second];
            second++;
            tempIndex++;
    }
    int i;
    for(i = s; i <= end; i++){
        unsorted[i] = temp1[i];
    }
    return 0;
}

// Verify that the serial mergesort is
correct int validateSerial(){
    int i;
    for(i = 0; i < n-1; i++){ if(vector_serial[i]
        > vector_serial[i+1]){
            printf("Serial sort
            unsuccesful.\n"); return 1;
        }
    }
    return 0;
}

// Verification of parallel bucket sort array is
correct int validateParallel(){
    int i;
    for(i = 0; i < n-1; i++){ if(vector_serial[i]
        != vector_parallel[i]){
```

```
            printf("Parallel sort unsuccesful.\n");
            return 1;
        }
    }
    return 0;
}

void printArray(int arr[], int size){
    int i;
    for(i = 0; i < size; i++){
        printf("%d\t", arr[i]);
    }
    printf("\n");
    return;
}

// Creates the pivots for each process bucket
sort int createPivots(){
    // Process 0 computes pivots
    int s = (int) 10 * world_size *
    log2(n); int *samples;
    int *samples_temp;
    int i, random, index;
    int *samplesIndexSet;

    if(s > n){
        s = n;
        samples = (int *) malloc(sizeof(int) * s);
        samples_temp = (int *) malloc(sizeof(int) * s);
        memcpy(samples, vector_parallel, s*sizeof(int));
    } else {
        samples = (int *) malloc(sizeof(int) * s);
        samples_temp = (int *) malloc(sizeof(int) * s);
        samplesIndexSet = (int *)malloc(sizeof(int)*s);
```

```
        // Floyd sampling without
        replacement index = 0;
        for(i = n - s; i < n; i++){
            random = rand() % i;
            if(samplesIndexSet[random] == 0){
                samples[index] = vector_parallel[random];
                samplesIndexSet[random] = 1;
            } else {
                samples[index] = vector_parallel[i];
                samplesIndexSet[i] = 1;
            }
            index++;
        }
        free(samplesIndexSet);
    }
    serialsort(s, samples, samples_temp);
    for(i = 0; i < world_size - 1; i++){
        pivots[i] = samples[((i+1) * s) / world_size];
    }
    free(samples);
    free(samples_temp);
    return 0;
}

// Divide the values received from Process 0 into buckets
     to send
// to other processes
int divideIntoBuckets(){
    int i;
    int *tempbucket = (int *) malloc(sizeof(int) * local);
    serialsort(local, local_vector_parallel, tempbucket);
    free(tempbucket);
    bucket = (int *) malloc(sizeof(int) *
    world_size); int bucketNum = 0;
```

```
    for(i = 0; i < local; i++){
        // Determine bucket end
        if(local_vector_parallel[i] >= pivots[bucketNum]){
            while(local_vector_parallel[i] >=
     pivots[bucketNum]){
                bucket[bucketNum] = i;
                bucketNum++;
                if(bucketNum == world_size - 1){
                    break;
                }
            }
        }
        if(bucketNum == world_size - 1){
            break;
        }
    }
    while(bucketNum < world_size){
        bucket[bucketNum] = local;
        bucketNum++;
    }
    free(pivots);
    return 0;
}

// Send the buckets to other processes for them to sort later
int sendBuckets(){
    // allocate memory for an array of vals to sort
    local_arr = (int *) malloc(sizeof(int)*local * 2);
    recv_bucket = (int *) malloc(sizeof(int)*world_size);
    int myArrSize = local * 2;
    //  check if index greater than size
    MPI_Status status;
    int i = 0;
```

```
    int index = 0;
    for(i = 0; i < world_size; i++){
        int sendcount, numElems, s;
        // Determine number of elems to send and where they
        s if(i == 0){
            sendcount = bucket[0] -
            0; s = 0;
        } else {
            sendcount = bucket[i] - bucket[i-
            1]; s = bucket[i-1];
        }

        if(i == my_rank){
            // If looking at own bucket, add values to local_arr
            memcpy(&local_arr[index], &local_vector_parallel[s],
     sizeof(int)*sendcount);
            index += sendcount;

        } else{
            // Send values in bucket i to process i and
     receive values from process i
            MPI_Sendrecv(&local_vector_parallel[s], sendcount,
     MPI_INT, i, 123,
                &local_arr[index], local, MPI_INT, i, 123,
     MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_INT,
            &numElems); index += numElems;
            // Reallocate memory if local_arr is full
            if(index > myArrSize){
                printf("Reallocating memory\n");
                local_arr = (int *) realloc(local_arr,
     sizeof(int)*local);
            }
        }
```

```
        recv_bucket[i] = index;
    }
    return 0;
}

// Merge k sorted arrays into one
int kWayMerge(int k, int unsorted[], int temp1[]){
    int *s = (int *) malloc(sizeof(int) * k); int
    i;
    for(i = 0; i < k; i++){
        if(i == 0){
            s[0] = 0;
        } else {
            s[i] = recv_bucket[i -1];
        }
    }
    int tempIndex = 0;
    int min, minProc;
    int valueLeft = 0;

    while(valueLeft == 0){
        min = 10000;
        minProc = -1;
        // Find the minimum value from all k
        arrays for(i = 0; i < k; i++){
            if(s[i] < recv_bucket[i]){
                if(unsorted[s[i]] < min){
                    min = unsorted[s[i]];
                    minProc = i;
                }
            }
        }
        // Check if no more elements
        if(minProc == -1){
```

```
            valueLeft = -1;
        } else {
            // Add to temp array with the value
            found temp1[tempIndex] =
            unsorted[s[minProc]]; tempIndex++;
            s[minProc]++;
        }
    }
    for(i = 0; i <= recv_bucket[k-1];
        i++){ unsorted[i] = temp1[i];
    }
    free(s);
    return 0;
}

int main(int argc, char* argv[]){
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    pivots = (int *) malloc(sizeof(int) * world_size-1);

    // Process 0
    if( my_rank == 0 ) {
        // Get n from standard input
        printf("Enter the size of the
        array:\n"); scanf("%ld", &n);
        while(n % world_size != 0){
            printf("Please enter an array size in factors of
     the number of processes:\n");
            scanf("%ld", &n);
        }
        struct timeval  tv1, tv2;
```

```
    // Allocate memory for global arrays
    vector_serial = (int *) malloc(sizeof(int) * n);
    vector_parallel = (int *) malloc(sizeof(int) *
    n); temp = (int *) malloc(sizeof(int) * n);
    int i;

    // Fill the arrays with the same random
    numbers srand(time(NULL));
    for(i = 0; i < n; i++){
        int random = rand() % 100;
        vector_serial[i] = random;
    }

    // Copy first array to second array
    memcpy(vector_parallel, vector_serial, sizeof(int)*n);
    memcpy(temp, vector_serial, sizeof(int)*n);

    // Perform the serial mergesort
    gettimeofday(&tv1, NULL);
    serialsort(n, vector_serial,
    temp); gettimeofday(&tv2, NULL);
    double serialTime = (double) (tv2.tv_usec - tv1.tv_usec)
/ 1000000 +

    validateSerial();
    gettimeofday(&tv1, NULL);
    createPivots();

    // Broadcast n and pivots to other procs
    MPI_Bcast(&n, 1, MPI_LONG, 0, MPI_COMM_WORLD);
    MPI_Bcast(pivots, world_size - 1, MPI_INT, 0,
MPI_COMM_WORLD);
```

```
        // Distribute vector_parallel to different
    processes with block distribution
        local = n / world_size;
        local_vector_parallel = (int *)malloc(sizeof(int)
    * local);
        MPI_Scatter(vector_parallel, local, MPI_INT,
    local_vector_parallel, local,
            MPI_INT, 0, MPI_COMM_WORLD);

// BODY OF ALG:

        divideIntoBuckets();
        sendBuckets();
        int *temp2 = (int *)malloc(sizeof(int)*num);

        kWayMerge(world_size, local_arr, temp2);
        free(temp2);
        memcpy(&vector_parallel[0], &local_arr[0],
    sizeof(int)*num);
        int index = num;
        MPI_Status status;
        // Receive all the pieces from the
        procs for(i = 1; i < world_size; i++){
            MPI_Recv(&vector_parallel[index], n, MPI_INT, i, 0,
    MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_INT,
            &num); index += num;
        }

        gettimeofday(&tv2, NULL);
        double parallelTime = (double) (tv2.tv_usec -
    tv1.tv_usec) / 1000000 +
            (double) (tv2.tv_sec - tv1.tv_sec);
        validateParallel();
```

```
    double speedup = serialTime / parallelTime;
    double efficiency = speedup / world_size;
    printf("Number of processes: %d\n", world_size);
    printf("Array Size: %ld\n", n);
    printf("Serial merge sort execution time: %e\n",
 serialTime);
    printf("Parallel bucket sort execution time:
 %e\n", parallelTime);
    printf("Speedup: %e\n", speedup);
    printf("Efficiency: %e\n", efficiency);

    free(vector_serial);
    free(vector_parallel);
    free(temp);
    free(local_arr);
  }
  else {
  // Other processes
    // Broadcast to recieve n
    MPI_Bcast(&n, 1, MPI_LONG, 0, MPI_COMM_WORLD);
    MPI_Bcast(pivots, world_size - 1, MPI_INT, 0,
 MPI_COMM_WORLD);

    // Distribute vector_parallel to different
 processes with block distribution
    local = n / world_size; // local is number of elems per
 proc
    local_vector_parallel = (int *)malloc(sizeof(int)
 * local);
    MPI_Scatter(vector_parallel, local, MPI_INT,
 local_vector_parallel, local,
        MPI_INT, 0, MPI_COMM_WORLD);
```

```
// BODY OF ALG:
        divideIntoBuckets()
        ;
        sendBuckets();
        int *temp2 = (int *)malloc(sizeof(int)*num);
        kWayMerge(world_size, local_arr, temp2);
        // Send sorted array to Process 0
        MPI_Send(local_arr, num, MPI_INT, 0, 0, MPI_COMM_WORLD);

        free(local_arr);
        free(temp2);
    }
    free(bucket);
    free(local_vector_parallel);
    MPI_Finalize();
    return 0;
}
```