**Fast Parallel Sorting Algorithms**

After the kth iteration, a mark will be present at each location whose last k bits are zeros and whose other (log n) - k bits coincide with the corresponding bits of the address of a processor active in the same area. Thus, each such location will be marked iff any of 2 k processors. When an algorithm design technique is presented, there is a three-way tradeoff between space, time, and processors. Muller and Preparata were the first to exhibit a network capable of sorting n numbers in time O(log n). Parallel bucket-sorting algorithms, when first presented at the expense of greater space requirements, the number of processors and the amount of time used are both reduced but algorithms are unusual in that the space requirements are greater than the processor time requirements. The model is based on SIMD, so n numbers are sorted using n (parallel) processors in time O(log n) and duplicates are discarded. Each processor will have a value to be sorted and because there can be two processors with the same value only one processor should be active and all others will deactivate.

We shall have m areas of memory, one for each bucket. Each area will be of size n, the number of input numbers to be sorted. Each processor is given a location ‘i’ from 0 to n-1, if another processor ‘Buddy’ of higher rank is active, the current processor will deactivate. After the kth iteration, a mark will be present at each location whose last k bits are zeros and whose other (log n) - k bits coincide with the corresponding bits of the address of a processor active in the same area. Thus, each such location will be marked iff any of 2 k processors. This algorithm requires space S=O(mn), time T=O(log n), and the use of n processors.

We have another algorithm that will give the actual ranking of input numbers, keeping the same relative order. This algorithm sorts one representative of each number and then adds the number into that position as many times as the count of that number.It is noted that Algorithm 2 requires space S = O(mn), time T = O(log n + log m), and the use of n processors.

In these algorithms, the area of memory is initialized to zero. If we disallow simultaneous access to the same location, a memory fetch conflict may happen. To solve this issue a location will thus be initialized if either of the two processes to which it might be relevant is active.

The third algorithm uses parallel bucket sort using n^3/2 processors and O(log n). In this algorithm, we partition input numbers into n^½ groups, and n processors are assigned to a group. Then apply bucket sort to each group, then apply binary search to each group which will give a count for each ‘jth’ element then use that count value to complete bucket sort.

The fourth algorithm uses parallel bucket sort using n^4/3 processors and O(log n). In this algorithm, we partition input numbers into n^⅔ groups, each having n^⅓ elements/sector. Then within each sector do a binary search to find the count value for every ‘jth’ element then use that count value to complete the bucket sort. Hence we can obtain an algorithm to sort n numbers in O(k log n) time that uses n 1+1/k processors.

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;

};

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;

}