Nathaniel Cantwell

Parallel Bitonic Sort

Contents

1. Task Description

Sorting problems are encountered frequently in computer science and engineering. Hence, finding an efficient parallel algorithm to sort large amounts of data has the potential to greatly reduce computation time in areas where these problems arise. *Bitonic sort* is a parallel sorting algorithm that offers attractive features for parallel implementation. Task 6 requires that we implement bitonic sort in a distributed memory MPI environment with a varying number of processors and variable data size.

1.1. Critical Success Factors

Implementing bitonic sort in a distributed environment required a deep understanding of the algorithm. The lecture slides available on the course website provided examples that shed light on the details of bitonic sort as well as the fundamental *compare and exchange* process used to implement bitonic sort. Additionally, Big Red 2 needed to be available during the testing phase of this task. During the development phase, however, a local MPI runtime environment was used to test the program at before executing the code on Big Red 2.

1.2. Assumptions

While developing this program, it was assumed that the *n* numbers and *m* compute nodes would each be a power of two. This assumption greatly simplifies coordination between the compute nodes and eliminates the case of uneven data distribution. It was also assumed that the *n* numbers could be a sequence of integers generated in the root node. In principle, however, the data being sorted could be any comparable type from any source.

During the sorting process, it was assumed that the *n/m* set of numbers originally received by each process could be sorted in any manner. Following the bitonic sort algorithm, these n/m numbers would be sorted in serial, without the need to communicate between nodes. Thus, any suitable serial algorithm could be chosen to sort them. In practice, a more efficient algorithm, such as heap sort, may be used to increase efficiency; however, in this program, bubble sort was selected as a placeholder for its conceptually simple implementation. This assumption necessarily decreases the performance of the program with fewer nodes.

1.3. Constraints

The number of processors, *m*, was required to be variable and independent of the size of the data, *n*. It was required that Big Red 2 be used to evaluate the performance of the program, and that the program be written for an MPI environment.

2. Methodology and Results

2.1. Methodology

The input data was a shuffled sequence of *n* numbers generated by the root node. In principle, this data could be of any type and could come from any source. Runtime arguments were used to determine the value of *n*, and dynamic memory was used to store the varying amount of data. This allowed tests to be conducted with a simple script on Big Red 2 to vary the number of nodes and the size of the starting array.

The fundamental operation in bitonic sort is the *compare and exchange* operation. Given two processors with two equally-sized, sorted arrays, the compare and exchange operation results in one processor containing numbers less than all the numbers contained by the other processor. In this program, compare and exchange was implemented in the following way: first, each processor sends its own array to the other participating processor in the exchange. Next, each processor merges the two arrays into a single sorted array. Finally, the processor copies its respective half of the array (either the minimum or the maximum) into the memory location of its original starting array. Information about which processor obtains the minimum and maximum partition of the numbers must be available when the function is invoked, so the final step amounts to a simple *if-else* statement. Although some inefficiency is present by exchanging numbers which will not be contained by the other processor after the exchange, the communication complexity does not degrade by more than a constant compared to the ideal scenario in which each processor only sends half of its numbers to the other. Either case is O(*n*).

Implementing bitonic sort involved decomposing the algorithm into iterative steps and describing the relationships between processors to calculate partners for compare and exchange. A *step* refers to an iteration where a single bitonic sequence (contained in multiple processors) is merged into a sorted list in either ascending or descending order. Bitonic sort with *m* processors requires $\log_2(m)$ such steps to sort data contained by these processors. At each step, a processor may be part of some *group* which determines which other processors participate in the sorting process. The group is calculated by dividing the processor's ID by 2^{step}. At the beginning of a step, a group of processors contains a bitonic sequence between them, starting with the degenerate case with two processors, and the bitonic sequence is transformed into a single sorted sequence in a series of *substeps*. The sort order within each group is alternated, which results in a bitonic sequence contained within adjacent groups. Substeps likewise define *subgroups*, which determine communication between processors during the transformation phase. At each substep, a processor participates in compare-and-exchange with its partner on the other half of the subgroup. The partner ID is calculated by adding or subtracting $2^{substep - 1}$ from the processor's own ID, where the choice of addition or subtraction is determined by the half of the subgroup that contains the processor.

Timing the runtime of the code involved calling an MPI_Barrier to synchronize the processors before recording the time of day. The difference between these times determined the total runtime of the algorithm. Random number generation, environment initialization, and the initial

MPI_Scatter were omitted from the section of code that was timed. The root processor appends a record with the number of processors, data size, and runtime results to a log file.

2.2. Results

The program resulted in a sorted sequence of integers with a varying number of processors and varying array size. A comparison of execution times is shown in Figure 1. Unlike many parallel algorithms, bitonic sort displays clear speed advantages with multiple nodes. With n = 1024, the computation time using eight nodes is slightly longer than the time using four nodes. This, I believe, clearly highlights the communication overhead present when using bitonic sort, as the steps require more communication with eight nodes. With larger starting arrays, however, the computational complexity has a greater effect on the final timing result, so the advantage of using more nodes is clear. This is most prominent in the case where n = 131072; however, this result comes with the caveat that the serial sorting was done using bubble sort. Thus, this is not necessarily as fair of a comparison as it might be using a faster serial sorting algorithm.

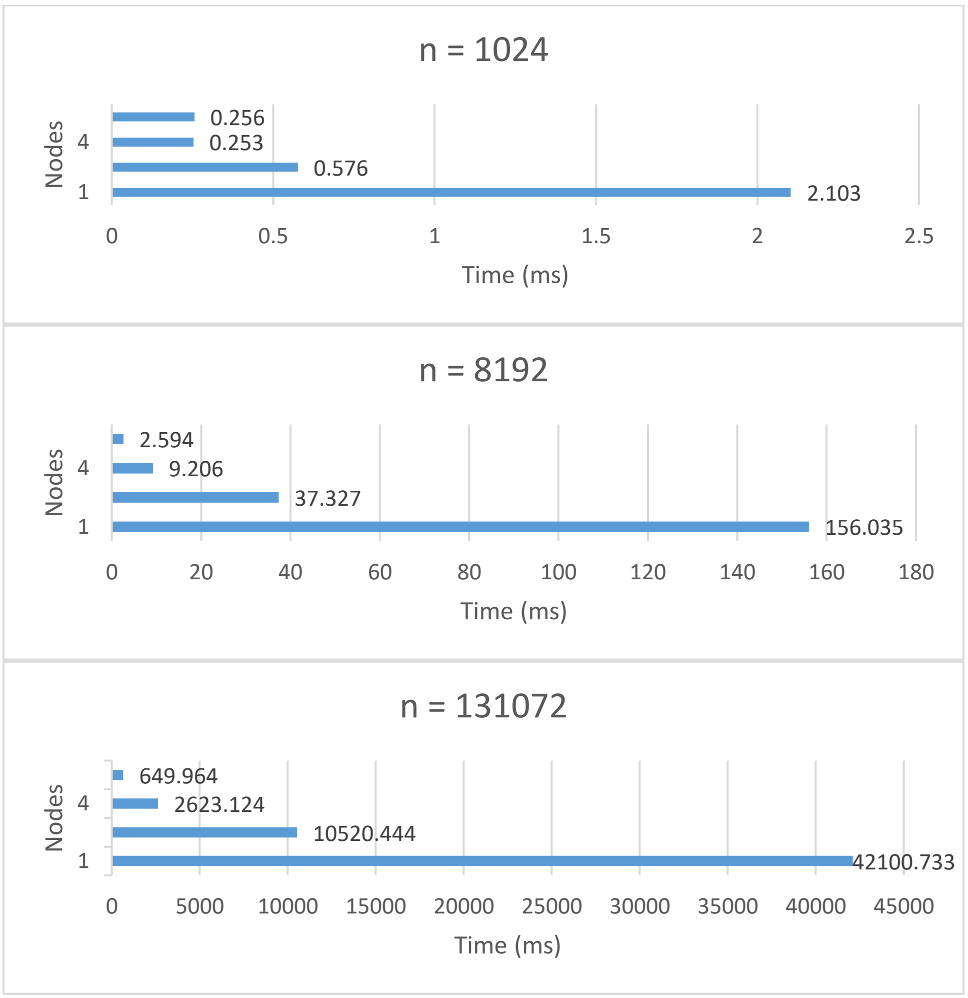

Figure 1: Execution time results for a varying number of nodes and various data sizes.

Speedup was calculated for each of the processors with each value for *n* (Table 1). This calculation was performed by dividing the serial time for each value of *n* by the parallel time for that same value of *n*. The largest gain in speedup occurred between n = 1024 and n = 8192 in each case for the number of nodes. The highest amount of speedup was observed with eight nodes and n = 131072. As mentioned before, this may be due not only to the advantages obtained by parallelism, but also the inefficiencies present in the serial sorting function.

	2 nodes	4 nodes	8 nodes
1024	3.65	8.31	8.21
8192	4.18	16.95	60.15
131072	4.00	16.05	64.77

Table 1: Speedup calculations for each recorded number of processors and array sizes

In all cases for the number of nodes, the greatest jump in speedup is seen between n = 1024 and n = 8192. As discussed previously, 1024 is a small enough problem size such that the overhead involved in parallelizing the task accounts for a significant portion of the runtime. With larger values of *n*, the computation done by each processor requires more time than the communication between processors. In effect, the granularity of parallelism becomes coarser as *n* increases. Bitonic sort nonetheless requires $O(\log^2(n))$ communication steps between nodes, which may be unacceptable in certain environments.

Depending on the scenario, bitonic sort may make sense in a real-world implementation. If the data is meant to end up in each of the nodes after the sorting process, bitonic sort provides this result without the need to redistribute the data, as with parallel merge sort. Bitonic sort also requires each node to perform a uniform amount of work to sort the given data. In distributed machines where communication between nodes is a significant bottleneck, however, bitonic sort might not make sense due to the large amount of communication required between nodes to perform compare and exchange. In this case, an algorithm such as parallel merge sort may prove more attractive due to the decreased amount of information exchange between nodes. For extremely large datasets, bitonic sort also offers the potential advantage of sorting data entirely in memory rather than using a method to sort the data on secondary storage. Each compute node need only hold 2n/m elements rather than all *n*, as each would in a serial environment.

2.3. Summary & Lessons Learned

Bitonic sort is a relatively efficient and potentially attractive parallel sorting algorithm in certain computing environments. It offers significant performance gains for large amounts of data, and clearly defines communication between each participating node at every step. The algorithm may be less efficient for environments where communication between nodes is greatly constrained but may be more appealing in higher-bandwidth networks due to the uniform distribution of work among nodes.

With respect to implementation, I would have liked to test the program with alternative serial sorting methods. Comparing these methods within the context of bitonic sort would demonstrate the effect of the bitonic sorting algorithm itself rather than the inefficiencies of the serial code supporting the function of the parallel code. Further implementation considerations include wrapping the bitonic sort function and iteratively applying it to larger groups of nodes. This would offer flexibility over the approach I took, which requires the node size to be fixed within the program. Another aspect I would have liked to explore is the comparison between bitonic sort and parallel merge sort in a variety of environments. Big Red 2 is a high-performance machine with a sophisticated interconnect between compute nodes. In an environment where network bandwidth is a major consideration, I believe merge sort may perform better due to decreased communication costs compared to bitonic sort. This would also highlight the practical considerations of algorithm design and analysis, which may necessitate accepting inefficiencies for performance gains in bottlenecked portions of the supporting environment.

3. Appendices

```
/*
 * bitonic.c
 *  A C program written for an MPI environment which implements Bitonic
 *  Sort on an array of 'n' integers using 'm' processors. It is assumed
 *  that both 'n' and 'm' are powers of 2. The integers are generated within
 *  the program using a random number generator.
 *
 * Author: Nathaniel Cantwell
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include "mpi.h"

#define MASTER 0
#define SIZE 1024
#define TASKS 4  // Number of tasks/processors
#define STEPS 2  // log2(TASKS), the number of "steps" required for bitonic
sort.

void comp_exchange_arr(int myid, int minid, int maxid, int arr[], int size);
void comp_exchange_min(int myid, int maxid, int arr[], int size);
void comp_exchange_max(int myid, int minid, int arr[], int size);
void sort(int arr[], int size);
void generate_random_arr(int arr[], int size);
void quit(int code);
int timedif(struct timeval start, struct timeval end);
void parallel_print(int taskid, int numTasks, int arr[], int size);

int main(int argc, char* argv[]) {
    // Seed random number generator
    srand(42);

    // Variables to store processor task id and number of tasks
    int numTasks, taskid, i;

    // Size and array length variable
    int size, loc_size, len, loc_len;

    // File pointer to result file
    FILE* resfp;
    char resfilename[50] = "result.csv";

    // Filename and pointer to timing file
    FILE* timefp;
    char timefname[] = "/gpfs/home/n/a/nadacant/BigRed2/Tasks/Task6/timing.csv";

    // Arrays to hold the integers, both in the master and in the slaves
    int *start_arr;
    int *local_arr;
    int *end_arr;

    // Structs for timing
    struct timeval start, end;
    int microsec;

    // Initialize MPI environment
    MPI_Init(&argc, &argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &numTasks);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

// Get size from argument list or use default
if (argc < 2) {
   if (taskid == MASTER)
      printf("No size parameter given. Using default %d\n", SIZE);
   size = SIZE;
} else {
   size = atoi(argv[1]);
   if (size <= 0) {
      if (taskid == MASTER)
         printf("Invalid size argument. Using default %d\n", SIZE);
      size = SIZE;
   }
}

if (taskid == MASTER) printf("size = %d\n", size);

// Generate the random sequence in the master array
int badfp = 0;
if (taskid == MASTER) {
   // Construct result filename and open result file
   sprintf(resfilename, "res_%dnodes_%dsize.csv", numTasks, size);

   // Open result file
   resfp = fopen(resfilename, "w");
   if (resfp == NULL) {
      printf("Error opening result file.\n");
      badfp = 1;
   }

   // Open timing file
   timefp = fopen(timefname, "a");
   if (timefp == NULL) {
      printf("Error opening timing file.\n");
      badfp = 1;
   }

   // Allocate space for array
   start_arr = (int*) malloc(size*sizeof(int));
   generate_random_arr(start_arr, size);

   // Print original array to file
   for (i = 0; i < size-1; i++)
      fprintf(resfp, "%d,", start_arr[i]);
   fprintf(resfp, "%d\n", start_arr[size-1]);

   // Allocate space for result array
   end_arr = (int*) malloc(size*sizeof(int));
}

// Broadcast status of file opening to nodes
MPI_Bcast(&badfp, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

// Check for failure
if (badfp) quit(1);

// Allocate space for local_arr
loc_size = size/numTasks;
local_arr = (int*) malloc(loc_size*sizeof(int));

// Scatter array to all tasks
```

```
   MPI_Scatter(start_arr, loc_size, MPI_INT, local_arr, loc_size, MPI_INT,
MASTER, MPI_COMM_WORLD);

   /*** Bitonic Sort section ***/

   int group;             // Current "group" based on "step"
   int subgroup;          // Current subgroup based on substep
   int step, maxsteps;    // Current "step" in bitonic sort and maximum step
   int substep;        // Smaller steps, where the two bitonic sequences are
merged into a larger sorted sequence
   int upperhalf;         // Determines if the processor is in the upper or
lower half of the group
   int partner;        // Partner for compare-exchange

   // Start timing sorting
   MPI_Barrier(MPI_COMM_WORLD);
   gettimeofday(&start, NULL);

   // Sort the local array serially
   sort(local_arr, loc_size);

   // Calculate maximum number of steps
   maxsteps = log2(numTasks);

   // Iterate through large steps (dictates group size)
   for (step = 1; step <= maxsteps; step++) {
      // Determine current group. Group size is 2^step
      group = (int)(taskid/pow(2, step));

      // Iterate through smaller steps (exchange within group, merge bitonic
sequences)
      for (substep = step; substep > 0; substep--) {
         // Determine if processor is in upper or lower half of subgroup
(partner less than or greater than)
         subgroup = (int)(taskid/pow(2, substep));
         upperhalf = (int)( (taskid - subgroup*pow(2.0, substep)) >= pow(2.0,
substep-1) );

         // Determine partner based on upperhalf parameter
         MPI_Barrier(MPI_COMM_WORLD);
         if (upperhalf) {
            partner = taskid - pow(2.0, substep-1);
         } else {
            partner = taskid + pow(2.0, substep-1);
         }

         // Determine if sorting order should be min or max
         if (group % 2 == 0) {   // Min-order case
            // Lower partner gets min in the min-order case
            if (taskid < partner)
               comp_exchange_min(taskid, partner, local_arr, loc_size);
            else
               comp_exchange_max(taskid, partner, local_arr, loc_size);
         } else {            // Max-order case
            // Higher partner gets min in the max-order case
            if (taskid > partner)
               comp_exchange_min(taskid, partner, local_arr, loc_size);
            else
               comp_exchange_max(taskid, partner, local_arr, loc_size);
         }

         // Synchronize
         MPI_Barrier(MPI_COMM_WORLD);
```

```
        }
    }

    // Gather array in master
    MPI_Gather(local_arr, loc_size, MPI_INT, end_arr, loc_size, MPI_INT, MASTER,
MPI_COMM_WORLD);

    // Get end time and take difference
    MPI_Barrier(MPI_COMM_WORLD);
    gettimeofday(&end, NULL);
    microsec = timedif(start, end);

    // Print for debugging
    //if (taskid == MASTER) printf("After:\n");
    //parallel_print(taskid, numTasks, local_arr, loc_size);

    // Output sorted array to file
    if (taskid == MASTER) {
        // Print result array to file
        for (i = 0; i < size-1; i++)
            fprintf(resfp, "%d,", end_arr[i]);
        fprintf(resfp, "%d\n", end_arr[size-1]);

        // Print timing information
        printf("time (ms): %lf\n", (double)microsec/1000.0);
        fprintf(resfp, "%lf\n", (double)microsec/1000.0);
        fprintf(timefp, "%d,%d,%lf\n", numTasks, size, (double)microsec/1000.0);

        // Close result file
        fclose(resfp);

        // Close timing file
        fclose(timefp);
    }

    // Free dynamic memory
    free(local_arr);
    if (taskid == MASTER) {
        free(start_arr);
        free(end_arr);
    }

    // Free MPI resources
    MPI_Finalize();
    return 0;
}

// Wrapper function to do compare and exchange and keep the minimum
void comp_exchange_min(int myid, int maxid, int arr[], int size) {
    comp_exchange_arr(myid, myid, maxid, arr, size);
}

// Wrapper function to do compare and exchange and keep the maximum
void comp_exchange_max(int myid, int minid, int arr[], int size) {
    comp_exchange_arr(myid, minid, myid, arr, size);
}

// Compare and exchange an array of numbers. Generic function
void comp_exchange_arr(int myid, int minid, int maxid, int arr[], int size) {
    // Dynamically-sized partner's array
    int* partner_arr = (int*) malloc(size*sizeof(int));

    // Dynamically-sized array to hold the merged arrays
    int* merged_arr = (int*) malloc(2*size*sizeof(int));

    // Status of recv
    MPI_Status stat;

    // Counters for merge
    int i, j, k;

    // Make arrays common to both. Alternate send/recv calls.
    if (myid == minid) {
        MPI_Send(arr, size, MPI_INT, maxid, 0, MPI_COMM_WORLD);
        MPI_Recv(partner_arr, size, MPI_INT, maxid, 0, MPI_COMM_WORLD, &stat);
    } else {
        MPI_Recv(partner_arr, size, MPI_INT, minid, 0, MPI_COMM_WORLD, &stat);
        MPI_Send(arr, size, MPI_INT, minid, 0, MPI_COMM_WORLD);
    }

    // Merge the two arrays into one large array
    j = 0;
    k = 0;
    for (i = 0; i < 2*size; i++) {
        // Check if the indices have hit the limit. Otherwise, take the minimum.
        if (k == size) {
            merged_arr[i] = arr[j++];
        } else if (j == size) {
            merged_arr[i] = partner_arr[k++];
        } else if (arr[j] < partner_arr[k]) {
            merged_arr[i] = arr[j++];
        } else {
            merged_arr[i] = partner_arr[k++];
        }
    }

    // Take the small half if I am minid, or the large half if I am maxid
    if (myid == minid) {
        // Copy the first half (minimum half) of the array
        for (i = 0; i < size; i++)
            arr[i] = merged_arr[i];
    } else {
        // Copy the second half (maximum half) of the array
        for (i = 0; i < size; i++)
            arr[i] = merged_arr[i+size];
    }

    // Free memory for temp arrays
    free(partner_arr);
    free(merged_arr);
}

// Serially sort an array passed by reference. In this case, use bubble sort
void sort(int arr[], int size) {
    int i, swap, swap_occurred;

    // Keep looping/sorting until no swaps occur
    swap_occurred = 1;
    while (swap_occurred) {
        // No swaps have occurred yet
        swap_occurred = 0;

        // Loop through all adjacent pairs of elements in the list
        for (i = 0; i < size - 1; i++) {
            // Swap the ith and (i+1)th element to the correct order
            if (arr[i] > arr[i+1]) {
                swap = arr[i+1];
                arr[i+1] = arr[i];
                arr[i] = swap;

                // Swap occurred, set flag to true
                swap_occurred = 1;
            }
        }
    }
}

// Generates a random array of size 'size' by shuffling the sequence of numbers
//  from zero to (size-1)
void generate_random_arr(int arr[], int size) {
    int i, swap_index;
    int swap;

    // Generate the sequence from zero to 'size'
    for (i = 0; i < size; i++)
        arr[i] = i;

    // Shuffle the array from end to beginning
    for (i = size-1; i > 0; i--) {
        swap_index = rand() % i;
        swap = arr[i];
        arr[i] = arr[swap_index];
        arr[swap_index] = swap;
    }
}

// Function to quit execution with finalize
void quit(int code) {
    MPI_Finalize();
    exit(code);
}

// Calculates the time difference between two different struct timeval
structures. Returns the number of microseconds.
int timedif(struct timeval start, struct timeval end) {
    return (end.tv_sec*1000000 + end.tv_usec) - (start.tv_sec*1000000 +
start.tv_usec);
}

// Debug function to print a group of arrays in order
void parallel_print(int taskid, int numTasks, int arr[], int size) {
    int i, j;

    // Initial synchronization step
    MPI_Barrier(MPI_COMM_WORLD);
    for (j = 0; j < numTasks; j++) {
        if (taskid == j) {
            printf("Processor %d: ", taskid);
            for (i = 0; i < size; i++)
                printf("%d ", arr[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
}
```