HW 3:

Adam Ross
adro4510@colorado.edu

Abstract

This paper discusses and analyzes the underlying mechanisms of the Parallel Broadcast and Reduce operations with additional relation to machine organization and optimization. First I will discuss the correctness and implementation of my broadcast and reduce algorithms. Then I will analyze optimization of these algorithms including timing, network topology context and algorithm bit transversal.

1. Introduction

Often in parallel programming it is needed to aggregate data between processing nodes or maybe it is needed that all nodes synchronize the same data at some point in processing. It is worth investigating the algorithms that accomplish these tasks and exploring the resulting subtleties as to optimize performance. I begin by supporting the correctness of my implementation by picking apart the procedure and output. Latency calculations are then done for each of the procedures and then related to network topology.

2. Overview

To accomplish broadcast and reduce we use a fan in and fan out methods. To reduce we use fan in to aggregate the data to a single process and the opposite for broadcast to make all the processes have the same data. The fan transversal can be done with bit-masks in a high to low or low to high fashion that dictates data flow between processes which is displayed in the following 4 diagrams.

Fig 2.1 High to Low bit transversal Broadcast

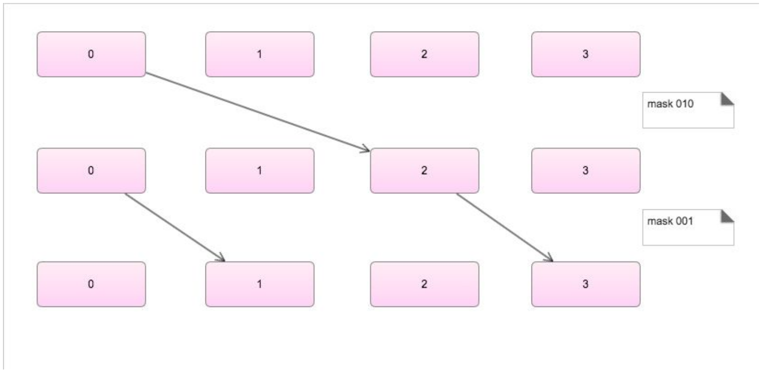


Fig 2.2 Low to High bit transversal Broadcast

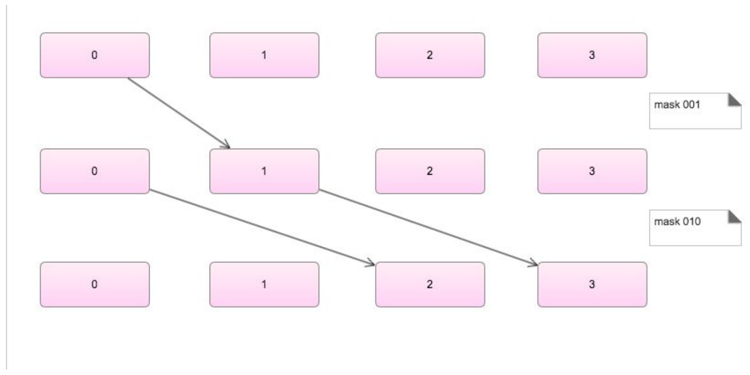


Fig 2.3 High to Low bit transversal Reduce

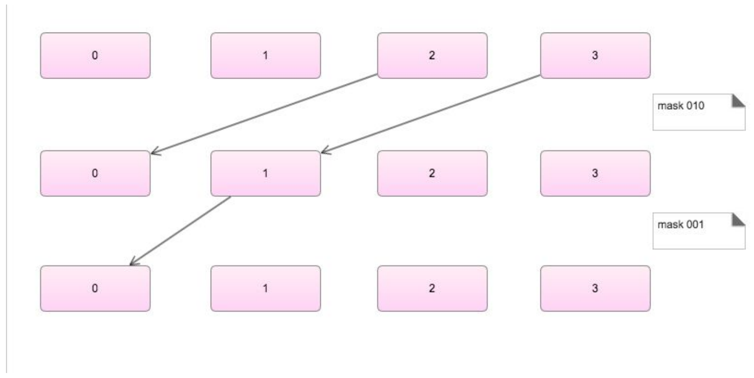
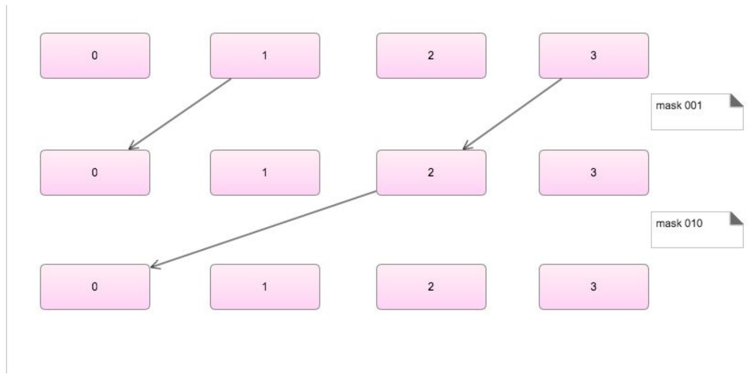


Fig 2.4 Low to High bit transversal Reduce



The Code Below describes the logic for a high to low bit transversal Broadcast.

```
for (int stage = 0; stage < log2(p); stage++) {
    current_mask = 1 << stage;
    if (my_rank < current_mask) {
        dest = my_rank | current_mask;
        // send
        MPI_Send(buffer, count + 1, MPI_DOUBLE, dest, 0, comm);
    } else if ((my_rank >= current_mask) && (my_rank < (current_mask * 2))) {
        // recv
        source = my_rank - current_mask;
        MPI_Recv(buffer, count + 1, MPI_DOUBLE, source, 0, comm, &status);
    }}
```

This and the other three logical operations can be derived from the diagrams above to solve for the operations of each of the transversal and fan in and fan out combinations.

3. Verification

The output below is from each of the 4 operations to verify the correct behaviour as described in the diagrams above. The reduce operation performs a sum as the aggregation step.

Fig 3.1 - High to Low Broadcast(process 0 given data = 3, p > 0 given 0):

```
My rank is 0 - Stage is: 0 - Sending to 2
My rank is 1 - Stage is: 1 - Receiving from 0
My rank is 2 - Stage is: 0 - Receiving from 0
My rank is 3 - Stage is: 1 - Receiving from 2
My rank is 0 - Stage is: 1 - Sending to 1
+ My rank is 0 and the buffer contained 3.000000
+ My rank is 1 and the buffer contained 3.000000
My rank is 2 - Stage is: 1 - Sending to 3
+ My rank is 2 and the buffer contained 3.000000
+ My rank is 3 and the buffer contained 3.000000
```

Fig 3.2 - Low to High Broadcast(process 0 given data = 3, p > 0 given 0):

```
My rank is 0 - Stage is: 0 - Sending to 1
My rank is 0 - Stage is: 1 - Sending to 2
My rank is 1 - Stage is: 0 - Receiving from 0
My rank is 1 - Stage is: 1 - Sending to 3
My rank is 2 - Stage is: 1 - Receiving from 0
My rank is 3 - Stage is: 1 - Receiving from 1
+ My rank is 0 and the buffer contained 3.000000
+ My rank is 1 and the buffer contained 3.000000
+ My rank is 2 and the buffer contained 3.000000
+ My rank is 3 and the buffer contained 3.000000
```

Fig 3.3 - High to Low Reduce(all processes given data = 3):

```
My rank is 0 - Stage is: 0 - Receiving from 2
My rank is 1 - Stage is: 0 - Receiving from 3
My rank is 1 - Stage is: 1 - Sending to 0
My rank is 2 - Stage is: 0 - Sending to 0
+ My rank is 2 and the total is 3.000000
My rank is 3 - Stage is: 0 - Sending to 1
+ My rank is 3 and the total is 3.000000
My rank is 0 - Stage is: 1 - Receiving from 1
+ My rank is 0 and the total is 12.000000
+ My rank is 1 and the total is 6.000000
```

Fig 3.4 - Low to High Reduce(all processes given data = 3):

```
My rank is 0 - Stage is: 0 - Receiving from 1
My rank is 0 - Stage is: 1 - Receiving from 2
My rank is 1 - Stage is: 0 - Sending to 0
+ My rank is 1 and the total is 3.000000
My rank is 2 - Stage is: 0 - Receiving from 3
My rank is 2 - Stage is: 1 - Sending to 0
My rank is 3 - Stage is: 0 - Sending to 2
+ My rank is 3 and the total is 3.000000
+ My rank is 0 and the total is 12.000000
+ My rank is 2 and the total is 6.000000
```

4. Analysis

Let us now assume sending a single message takes 0.7 us, below is a table of each algorithm's expected execution time is.

Algorithm	High to Low Broadcast	Low to High Broadcast	High to Low Reduce	Low to High Reduce	All Reduce	All to One All Reduce
4 nodes	1.4 us	1.4 us	1.4 us	1.4 us	2.8 us	4.2 us

* I know these are completely wrong. I have run out of time and need to submit.

Knowing a network's topography should very much influence algorithm design. We are two consider two situations: 1. Our nodes are set up in a 2d Torus Network. 2. Our nodes are set up in a Fat Tree Network. Given the nature of a 2d torus network where the rows and columns of node are connected the bit transversal is less relevant because the mesh is fairly strongly connected, you have at most sqrt(n) hops to any given node. Comparatively, a fat tree network is much better suited to a high to low broadcast and high to low reduce. This is because the opposite algorithms result in more cross tree messages, and fewer local which is less ideal.

Bit traversal order does matter in some situations given the layout of your network if you are pushing for complete optimization. It makes sense to match the opposite bit transversal broadcast and reduce methods together because they are organized similarly, just reverse of each other. I.e. if you follow 001 -> 010 -> 100 forward to reduce it makes sense to traverse backwards during the broadcast to follow the same path.

```c
/* RossAdam_HW3.c
 *
 * Broadcast and Reduce methods
 * with high to low and low to high
 * bit mask transversal.
 *
 * All_Reduce method compound from the above.
 * All_Reduce method using an all to one alogrithm.
 *
 * Input: none.
 * Output: none.
 *
 *
 */
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

double My_Broadcast(double *buffer, int count, MPI_Comm comm, int lo_hi, int p, int my_rank)
;
double My_Reduce(double *buffer, int count, MPI_Comm comm, int lo_hi, int p, int my_rank);
double My_Compound_All_Reduce(double *buffer, int count, MPI_Comm comm, int lo_hi, int p, in
t my_rank);
double My_All_Reduce(double *buffer, int count, MPI_Comm comm, int lo_hi, int p, int my_rank
);


main(int argc, char* argv[]) {
    int         my_rank;       /* rank of process      */
    int         p;             /* number of processes  */
    int         tag = 0;       /* tag for messages     */
    double      to_sum;        /* storage for message  */

    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank  */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    //MPI_COMM_WORLD
    //if (my_rank == 0) {
    to_sum = 3.0;
    //}

    /* Comment or un comment to run specific programs */
    //double a = My_Broadcast(&to_sum, 1, MPI_COMM_WORLD, 1, p, my_rank);
    //double b = My_Broadcast(&to_sum, 1, MPI_COMM_WORLD, 0, p, my_rank);
    //double c = My_Reduce(&to_sum, 1, MPI_COMM_WORLD, 1, p, my_rank);
    //double d = My_Reduce(&to_sum, 1, MPI_COMM_WORLD, 0, p, my_rank);
    //double e = My_Compound_All_Reduce(&to_sum, 1, MPI_COMM_WORLD, 0, p, my_rank);
    //double f = My_All_Reduce(&to_sum, 1, MPI_COMM_WORLD, 0, p, my_rank);

    /* Shut down MPI */
    MPI_Finalize();
} /* main */


// low to high bit transversal
// 001 -> 010 -> 100
//  1      2      4

// hight to low bit transversal
// 100 -> 010 -> 001
//  4      2      1

double My_Broadcast(double *buffer, int count, MPI_Comm comm, int lo_hi, int p, int my_rank)
 {
    int current_mask;
    int dest;
    int source;
    MPI_Status  status;
    //printf("+ My rank is %d and the buffer contains %f\n", my_rank, *buffer);
    if (lo_hi != 0) { // hi
        for (int stage = 0; stage < log2(p); stage++) {
            current_mask = (p / 2) >> stage;
            dest = my_rank | current_mask;
            if ((my_rank % current_mask == 0) && (my_rank != dest)) {
                printf("My rank is %d - Stage is: %d - Sending to %d\n", my_rank, stage, des
t);
                //send
                MPI_Send(buffer, count + 1, MPI_DOUBLE, dest, 0, comm);
            } else if ((my_rank + current_mask) % (2 * current_mask) == 0) {
                //recv
                source = my_rank ^ current_mask;
                printf("My rank is %d - Stage is: %d - Receiving from %d\n", my_rank, stage,
 source);
                MPI_Recv(buffer, count + 1, MPI_DOUBLE, source, 0, comm, &status);
            }
        }
    } else { // lo
        for (int stage = 0; stage < log2(p); stage++) {
            current_mask = 1 << stage;
            if (my_rank < current_mask) {
                dest = my_rank | current_mask;
                printf("My rank is %d - Stage is: %d - Sending to %d\n", my_rank, stage, des
t);
                // send
                MPI_Send(buffer, count + 1, MPI_DOUBLE, dest, 0, comm);
            } else if ((my_rank >= current_mask) && (my_rank < (current_mask * 2))) {

                // recv
                source = my_rank - current_mask;
                printf("My rank is %d - Stage is: %d - Receiving from %d\n", my_rank, stage,
 source);
                MPI_Recv(buffer, count + 1, MPI_DOUBLE, source, 0, comm, &status);
            }
        }
    }
    printf("+ My rank is %d and the buffer contained %f\n", my_rank, *buffer);
    return *buffer;
}

double My_Reduce(double *buffer, int count, MPI_Comm comm, int lo_hi, int p, int my_rank) {
    int current_mask;
    int dest;
    int source;
    double total = *buffer;
    MPI_Status  status;

    //printf("+ My rank is %d and the buffer contains %f\n", my_rank, *buffer);

    if (lo_hi != 0) { // hi
        for (int stage = 0; stage < log2(p); stage++) {
            current_mask = (p / 2) >> stage;
            if ((my_rank >= current_mask) && (my_rank < current_mask * 2)) {
                dest = my_rank ^ current_mask;
                printf("My rank is %d - Stage is: %d - Sending to %d\n", my_rank, stage, des
t);
                // send
                MPI_Send(&total, count + 1, MPI_DOUBLE, dest, 0, comm);
            } else if (my_rank < current_mask) {
                // recv
                source = my_rank + current_mask;
                printf("My rank is %d - Stage is: %d - Receiving from %d\n", my_rank, stage,
 source);
                MPI_Recv(buffer, count + 1, MPI_DOUBLE, source, 0, comm, &status);
                total = total + *buffer;
            }
        }
    } else { // lo
        for (int stage = 0; stage < log2(p); stage++) {
            current_mask = 1 << stage;
            if ((my_rank - current_mask) % ((stage + 1) * 2) == 0) {
                dest = my_rank ^ current_mask;
                printf("My rank is %d - Stage is: %d - Sending to %d\n", my_rank, stage, des
t);
                // send
                MPI_Send(&total, count + 1, MPI_DOUBLE, dest, 0, comm);
            } else if ((my_rank % (current_mask * 2)) == 0) {
                // recv
                source = my_rank + current_mask;
                printf("My rank is %d - Stage is: %d - Receiving from %d\n", my_rank, stage,
 source);
                MPI_Recv(buffer, count + 1, MPI_DOUBLE, source, 0, comm, &status);
                total = total + *buffer;
            }
        }
    }
    printf("+ My rank is %d and the total is %f\n", my_rank, total);
    return total;

}

double My_Compound_All_Reduce(double *buffer, int count, MPI_Comm comm, int lo_hi, int p, in
t my_rank) {
    double red = My_Reduce(buffer, count, comm, lo_hi, p, my_rank);
    double a =  My_Broadcast(&red, count, comm, lo_hi, p, my_rank);
    printf("- MY rank is %d, and the number I have is %f\n", my_rank, a);
    return a;
}

// That is, have every processor send to one processor, which receives using a loop, and the
n broadcast to each using MPI_Send from a loop

double My_All_Reduce(double *buffer, int count, MPI_Comm comm, int lo_hi, int p, int my_rank
) {
    int dest;
    int source;
    double total = 0;
    MPI_Status  status;

    // Reduce
    if (my_rank != 0) {
        dest = 0;
        //printf("My rank is %d, sending to %d\n", my_rank, dest);
        MPI_Send(buffer, count + 1, MPI_DOUBLE, dest, 0, comm);
    } else {
        total = total + * buffer;
        for (int i = 1; i < p; i++) {
            source = i;
            //printf("My rank is %d, receiving from %d\n", my_rank, source);
            MPI_Recv(buffer, count + 1, MPI_DOUBLE, source, 0, comm, &status);
            total = total + *buffer;
        }
        printf("- My rank is %d and the reduced value I have is %f\n",my_rank, total);
    }

    // Broadcast
    if (my_rank != 0) {
        source = 0;
        //printf("My rank is %d, receiving from %d\n", my_rank, source);
        MPI_Recv(buffer, count + 1, MPI_DOUBLE, source, 0, comm, &status);
        total = *buffer;
    } else {
        for (int i = 1; i < p; i++) {
            //printf("My rank is %d, sending to %d\n", my_rank, i);
            MPI_Send(&total, count + 1, MPI_DOUBLE, i, 0, comm);
        }
    }

    printf("+ My rank is %d, and the value I have is %f\n", my_rank, total);

    return total;
}

void Print(char* message, int my_rank) {
    printf("My rank is %d: %s\n",my_rank, message);
}
```