HW 2: Basic Parallel Computing Data Reduction and API Calls

Adam Ross
adro4510@colorado.edu

Abstract

The Paper addresses basic MPI message passing, it's effects on a program and some of the nuances of moving an algorithm between serial and simple parallel computing with MPI. I first discusses the blocking nature of MPI_RECV and the effects this has on program flow. Then I compare the differences and benefits of post processing all to one data reduction versus post processing distributed data reduction which leads to how I conceptually compare the overhead and idle time of each method.

1. Introduction

We have a very basic knowledge of MPI at this point and mostly need some experience working with the code itself. Coding with the fundamental building blocks of MPI teaches through experience the top level nature of the MPI library. I examine the OpenMPI RECV call and the semantics about it. I then begin to look at MPI data reduction algorithms and the motivation for further investigation.

2. Overview

The OpenMPI Recv api call is blocking and as such, in order for our program not to block, there must be a send for every receive. Specifically there must be a send before every single receive for our program to not block on some level. This information directs thought to the manner in which we call send and receive and as a repercussion the source and destination management for each of these calls. It needs to be carefully done as not to put too much load on a single process to compile the data and not too complex that we cannot manage our calls and block the program as a whole. Edge cases were also a delicate case that needed though. If your MPI code did not handle it properly then the code would not work.

For example the required trapezoid integral estimation program calculated the sources and destination for the MPI send and receive calls as such:

```
source = my_rank + 1;
dest = (my_rank == 1 ? 0 : (my_rank - 1));
```

This means if we have a single process my_rank = 0. The dest will be calculated (0+1)mod1 = 0. The source will be calculated (0==0) -> True -> (1 - 1) = 0. We will send and receive from the same process which works as intended.

Each process starting from the highest ranked process would send its data to the process one rank below it and sum the data being passed through. This would occur until process 0 summed the final result which was the value of our integral estimation. This enables us to spread the aggregation and receiving work out, increasing efficiency of the program.

3. Verification

A program was written that included a send only after a receive call. The program hung and accomplished nothing. The program was then rewritten to send first with destination in a wrapping fashion to the process rank above it. This ran fine and did not block.

Another program was then written to do a calculation and send data to the process one rank below it to sum data down to the first process rank. This program started with all but one process to be blocking on receive where a single send would trigger a receive send chain to sum the data and distribute the actual summation and receiving work and overhead.

The final program written performed Simpson's rule given an interval, a more complex operation that requires more per process management and more complicated edge cases. It is verified by computing the edge cases.

Code Output:

Only one interval across 10 processes (low interval and odd interval amount)

```
mpiexec -np 10 parallel_simpson -i 1 -v
An odd interval was given, which basic simpson's rule does not work with. Adding 1
interval.
[ 1 ] [ 4 ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ 1 ]
The parallel sum for Simpson's rule from 0.000000 to 1.000000 is 0.333333
```

Many intervals across one process

```
mpiexec -np 1 parallel_simpson -i 100 -v
[ 1 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4
2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 2
4 2 4 2 4 2 4 2 4 2 4 2 4 2 4 1 ]
The parallel sum for Simpson's rule from 0.000000 to 1.000000 is 0.333333
```

Many intervals across moderate processes

```
mpiexec -np 8 parallel_simpson -i 100 -v
[ 1 4 2 4 2 4 2 4 2 4 2 4 2 ] [ 4 2 4 2 4 2 4 2 4 2 4 2 4 ] [ 2 4 2 4 2 4 2 4 2 4 2 4
2 ] [ 4 2 4 2 4 2 4 2 4 2 4 2 4 ] [ 2 4 2 4 2 4 2 4 2 4 2 4 ] [ 2 4 2 4 2 4 2 4 2 4 2
4 ] [ 2 4 2 4 2 4 2 4 2 4 2 4 ] [ 2 4 2 4 2 4 2 4 2 4 2 4 1 ]
The parallel sum for Simpson's rule from 0.000000 to 1.000000 is 0.333333
```

Many intervals across many processes

```
mpiexec -np 32 parallel_simpson -i 100 -v
[ 1 4 2 4 ] [ 2 4 2 4 ] [ 2 4 2 4 ] [ 2 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4
] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2
4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ]
[ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 ] [ 2 4 2 ] [ 4 2 4 1 ]
The parallel sum for Simpson's rule from 0.000000 to 1.000000 is 0.333333
```

4. Conclusion

It is clear that in order to optimize highly parallel processing we must also manage and distribute MPI calls and their overhead as well as the aggregation work itself. As was said in class there is basicly no way to completely get rid of all serialized work to achieve a true parallel processing.

```c
/* RossAdam_parallel_simpson.c -- Parallel Simpson's Rule
 *
 *
 * Input: -i # or --intervals # = number of intervals
 *        -v or --verbose = print additional information
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 *
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* We'll be using MPI routines, definitions, etc. */
#include "mpi.h"

int main(int argc, char** argv) {
    int n;
    float a = 0.0;   /* Left endpoint             */
    float b = 1.0;   /* Right edndpoint            */
    float delta_x;
    float sum = 0.0;

    int found;
    int indx;
    int verbose;

    int bins;
    int remain;
    int sum_a;

    int         my_rank;   /* My process rank           */
    int         p;         /* The number of processes   */
    float       total;     /* Total sum                 */
    int         source;    /* Process sending sum       */
    int         dest;      /* Message cascade to 0      */
    int         tag = 0;
    MPI_Status  status;

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    float f(float x);

    /* Pasre arguements and make sure our values work */
    if (argc != 3 && argc != 4) {
        if (my_rank == 0) {
            printf("Too many or too few arguements. Expecting -i(or --intervals) # and optio
nally -v(or --verbose)\n");
        }
        return 1;
    }
    /* Check if we have -i or optionally if we have -v */
    found = 0;
    indx = 0;
    verbose = 0;
    for (int i = 1; i < argc; i++) {
        if (strcmp("-i", argv[i]) == 0 || strcmp("--intervals", argv[i]) == 0) {
            indx = i;
            found = 1;
        }
        if (strcmp("-v", argv[i]) == 0 || strcmp("--verbose", argv[i]) == 0) {
            verbose = 1;
        }
    }

    /* If we are not specified intervals */
    if (found == 0) {
        if (my_rank == 0) {
            printf("No intervals specified. Quitting.\n");
        }
        return 1;
    }

    /* set n to intervals specified */
    n = atoi(argv[indx + 1]);
    if (n <= 0){
        if (my_rank == 0) {
            printf("You cannot have negative or zero intervals.\n");
        }
        return 1;
    } else if (n % 2) { // Simpson's rule cannot operate with odd intervals
        if (my_rank == 0) {
            printf("An odd interval was given, which basic simpson's rule does not work with
. Adding 1 interval.\n");
        }
        n++;
    }
    /* Done with house keeping */

    /* Individual process setup */
    bins = (int) (n / p);
    remain = n % p;

    /* Build coefficient array */
    int size_ca = (n + 1);
    int coef_array[size_ca];
    for (int i = 0; i < size_ca; i++) {
        if (i % 2) {
            coef_array[i] = 4;
        } else {
            coef_array[i] = 2;
        }
    }
    coef_array[0] = 1;
    coef_array[n] = 1;

    /* Build array of boundery coefficient index values for each process */
    int bound[p + 1];
    sum_a = 0;
    for (int i = 1; i < (p + 1); i++) {
        if (i < (remain + 1)) {
            sum_a += bins + 1;
            bound[i] = sum_a;
        } else {
            sum_a += bins;
            bound[i] = sum_a;
        }
    }
    bound[0] = 0;
    bound[p] += 1;

    /* If we have been passed -v print information */
    if (verbose && my_rank == 0) {
        for (int i = 0; i < p; i++) {
            printf("[ ");
            for (int j = bound[i]; j < bound[i+1]; j++) {
                printf("%d ", coef_array[j]);
            }
            printf("] ");
        }
        printf("\n");
    }

    /* Do the actual Simpson's rule work */
    delta_x = (b - a) / n;
    for (int i = bound[my_rank]; i < bound[my_rank + 1]; i++) {
        sum += (coef_array[i] * f(i * delta_x));
    }

    /* Fold the data back down to process 0 */
    source = my_rank + 1;
    dest = (my_rank == 1 ? 0 : (my_rank - 1));
    if (p != 1) {
        if (my_rank == (p - 1)) {
            MPI_Send(&sum, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
        } else {
            MPI_Recv(&total, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);
            total = total + sum;
            if (dest != -1) {
                MPI_Send(&total, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
            }
        }
    } else {
        total = sum;
    }

    /* Do the final Simpson's rule multiplication and print*/
    if (my_rank == 0) {
        total = total * (delta_x / 3.0);
        printf("The parallel sum for Simpson's rule from %f to %f is %f\n", a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();

} /*  main  */

float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
```

```c
/* RossAdam_serial_simpson.c -- Serial Simpson's Rule
 *
 *
 * Input: -i # or --intervals # = number of intervals
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 *
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int n;
    float a = 0.0;   /* Left endpoint              */
    float b = 1.0;   /* Right edndpoint             */
    float delta_x;
    float sum = 0.0;
    float current_x = 0.0;

    float f(float x);

    if (argc != 3) {
        printf("Too many or too few arguements. Expecting -i(or --intervals) #");
        return 1;
    } else if (!strcmp("-i", argv[1]) && !strcmp("--intervals", argv[1])) {
        return 1;
    }
    n = atoi(argv[2]);
    delta_x = (b - a) / n;

    sum += f(a);
    for (int i=1; i <= (n-1); i++) {
        current_x += delta_x;
        if (i % 2) { // odd
            sum += 4 * f(current_x);
        } else { // even
            sum += 2 * f(current_x);
        }
    }
    sum += f(b);

    sum = sum * (delta_x / 3);

    printf("of the integral from %f to %f = %f\n", a, b, sum);

} /*  main  */

float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
```

```c
/* RossAdam_HW2-1.c
 *
 *  Written by Adam Ross
 *
 * Send a message in wrapping fashion to rank + 1.
 *    Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by each process.
 *
 *
 */

#include <stdio.h>
#include <string.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    int         my_rank;       /* rank of process      */
    int         p;             /* number of processes  */
    int         source;        /* rank of sender       */
    int         dest;          /* rank of receiver     */
    int         tag = 50;       /* tag for messages     */
    char        message[100];  /* storage for message  */
    MPI_Status  status;        /* return status for    */
                               /* receive              */

    /* 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);

    /* Create message */
    sprintf(message, "Greetings from process %d!", my_rank);
    dest = (my_rank + 1) % p;
    source = (my_rank == 0 ? (p - 1) : (my_rank - 1));
    /* Use strlen+1 so that '\0' gets transmitted */
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
    printf("Rank is: %d, Source is: %d, Dest is %d, Message recieved is: %s\n", my_rank, sou
rce, dest, message);

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

```c
/* RossAdam_trapezoid.c -- Parallel Trapezoidal Rule, first version
 *
 * Modified By Adam Ross
 *
 * Input: None.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 * Notes:
 *    1.  f(x), a, b, and n are all hardwired.
 *    2.  The number of processes (p) should evenly divide
 *        the number of trapezoids (n = 1024)
 *
 * See Chap. 4, pp. 56 & ff. in PPMPI.
 */
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include "mpi.h"


main(int argc, char** argv) {
    int         my_rank;   /* My process rank           */
    int         p;         /* The number of processes   */
    float       a = 0.0;   /* Left endpoint             */
    float       b = 1.0;   /* Right endpoint            */
    int         n = 1024;  /* Number of trapezoids      */
    float       h;         /* Trapezoid base length     */
    float       local_a;   /* Left endpoint my process  */
    float       local_b;   /* Right endpoint my process */
    int         local_n;   /* Number of trapezoids for  */
                           /* my calculation            */
    float       integral;  /* Integral over my interval */
    float       total;     /* Total integral            */
    int         source;    /* Process sending integral  */
    int         dest;  /* All messages go to 0      */
    int         tag = 0;
    MPI_Status  status;

    /* Calculate local integral  */
    float Trap(float local_a, float local_b, int local_n, float h);

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    // MY CODE
    source = my_rank + 1;
    dest = (my_rank == 1 ? 0 : (my_rank - 1));

    // printf("my rank is: %d, and my destination is: %d\n", my_rank, dest);
    if (p != 1) {
        if (my_rank == (p - 1)) {
            MPI_Send(&integral, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
        } else {
            MPI_Recv(&total, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);
            total = total + integral;
            if (dest != -1) {
                MPI_Send(&total, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
            }
        }
    } else {
        total = integral;
    }
    // END MY CODE

    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n", n);
        printf("of the integral from %f to %f = %f\n", a, b, total);
    }

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


float Trap(
          float  local_a   /* in */,
          float  local_b   /* in */,
          int    local_n   /* in */,
          float  h         /* in */) {

    float integral;   /* Store result in integral  */
    float x;
    int i;

    float f(float x); /* function we're integrating */

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /*  Trap  */


float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
```