

Assignment 1 - SC9505 - Jared Wogan

Question 1 - MISD

The program will generate an array of data on the main process. Next, the main process will send the data to every other process in the pool. Each process is then responsible for calculating a unique moment determined by the rank in the pool.

Code:

```
C++  
/* MISD Program Example  
 * Generates a large vector of numbers on the main MPI process  
 * Which is subsequently distributed to all other processes  
 * Each process then computes the moment corresponding to it's rank  
 */  
  
#include <iostream>  
#include <cmath>  
#include <algorithm>  
#include <numeric>  
#include <vector>  
#include <mpi.h>  
  
constexpr unsigned long int N = 100'000'000U;  
  
int main(int argc, char** argv) {  
    int size, rank;  
  
    MPI_Init(&argc, &argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Status status;  
  
    long double moment = 0.0;  
    std::vector<long double> numbers(N);  
  
    // Generate some data  
    // std::fill(numbers.begin(), numbers.end(), 1);
```

```

iota(numbers.begin(), numbers.end(), 1);
std::transform(numbers.begin(), numbers.end(), numbers.begin(), []
(long double &x) { return x / N; });

// Send it to every process
MPI_Bcast(numbers.data(), N, MPI_LONG_DOUBLE, 0, MPI_COMM_WORLD);

// Compute the moment
for (int i = 0; i < N; i++) {
    moment += pow(numbers[i], rank+1);
}
moment /= N;

// Print the result
printf("Process %d has calculated the %d moment to be %.6Lf\n", rank,
rank+1, moment);

MPI_Finalize();
}

```

The screenshot shows a Visual Studio Code interface with a terminal window. The terminal output displays the results of calculating moments for 16 processes, all of which are 1.000000. The program is run using 'mpirun -n 8 ./1-misd' and 'mpirun --oversubscribe -n 16 ./1-misd'.

```

jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 25 time mpirun -n 8 ./1-misd
Process 4 has calculated the 5 moment to be 1.000000
Process 5 has calculated the 6 moment to be 1.000000
Process 1 has calculated the 2 moment to be 1.000000
Process 2 has calculated the 3 moment to be 1.000000
Process 6 has calculated the 7 moment to be 1.000000
Process 3 has calculated the 4 moment to be 1.000000
Process 0 has calculated the 1 moment to be 1.000000
Process 7 has calculated the 8 moment to be 1.000000
2.11user 1.19system 0:00.71elapsed 450%CPU (0avgtext+0avgdata 91572maxresident)k
216inputs+16496outputs (2242major+16663minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 25 time mpirun --oversubscribe -n 16 ./1-misd
Process 2 has calculated the 3 moment to be 1.000000
Process 8 has calculated the 9 moment to be 1.000000
Process 6 has calculated the 7 moment to be 1.000000
Process 3 has calculated the 4 moment to be 1.000000
Process 1 has calculated the 2 moment to be 1.000000
Process 14 has calculated the 15 moment to be 1.000000
Process 7 has calculated the 8 moment to be 1.000000
Process 13 has calculated the 14 moment to be 1.000000
Process 15 has calculated the 16 moment to be 1.000000
Process 11 has calculated the 12 moment to be 1.000000
Process 0 has calculated the 1 moment to be 1.000000
Process 5 has calculated the 6 moment to be 1.000000
Process 10 has calculated the 11 moment to be 1.000000
Process 4 has calculated the 5 moment to be 1.000000
Process 12 has calculated the 13 moment to be 1.000000
Process 9 has calculated the 10 moment to be 1.000000
8.46user 8.57system 0:01.53elapsed 1111%CPU (0avgtext+0avgdata 94232maxresident)k
216inputs+16496outputs (2429major+3767minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 25

```

1. In the first example, the data we are calculating the various moments of is an array full of 1s. As a base case, it is clear the code is working as the sum of N ones raised to any power

and subsequently divided by N should indeed return one.

```

jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 25 time mpiCC -O2 -o 1-misd ./MISD.cpp
0.25user 0.05system 0:00.50elapsed 608CPU (0avgtext+0avgdata 169596maxresident)k
216inputs+16496outputs (2067major+3444minor)pagefaults 0swaps

jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 25 time mpiexec -n 8 ./1-misd
Process 5 has calculated the 6 moment to be 0.142857
Process 0 has calculated the 1 moment to be 0.500000
Process 6 has calculated the 7 moment to be 0.125000
Process 1 has calculated the 2 moment to be 0.333333
Process 7 has calculated the 8 moment to be 0.111111
Process 3 has calculated the 4 moment to be 0.200000
Process 4 has calculated the 5 moment to be 0.166667
Process 2 has calculated the 3 moment to be 0.250000
4.48user 2.41system 0:01.16elapsed 589%CPU (0avgtext+0avgdata 171564maxresident)k
216inputs+16496outputs (2239major+18102minor)pagefaults 0swaps

jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 25 time mpiexec -n 16 ./1-misd
Process 3 has calculated the 4 moment to be 0.200000
Process 7 has calculated the 8 moment to be 0.111111
Process 6 has calculated the 7 moment to be 0.125000
Process 11 has calculated the 12 moment to be 0.076923
Process 0 has calculated the 1 moment to be 0.500000
Process 5 has calculated the 6 moment to be 0.142857
Process 10 has calculated the 11 moment to be 0.083333
Process 13 has calculated the 14 moment to be 0.066667
Process 2 has calculated the 3 moment to be 0.250000
Process 15 has calculated the 16 moment to be 0.058824
Process 9 has calculated the 10 moment to be 0.090909
Process 8 has calculated the 9 moment to be 0.100000
Process 4 has calculated the 5 moment to be 0.166667
Process 1 has calculated the 2 moment to be 0.333333
Process 12 has calculated the 13 moment to be 0.071429
Process 14 has calculated the 15 moment to be 0.062500
19.76user 17.21system 0:02.85elapsed 1296%CPU (0avgtext+0avgdata 174260maxresident)k
216inputs+16496outputs (2422major+36808minor)pagefaults 0swaps

jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 25

```

2) In this second example, I have run the code with floating point numbers instead for a variety of different thread counts. I can be confident the code works by considering the value obtained for the first moment. The array contains the numbers 1 through N all divided by N . The sum of the first N integers is $\frac{N(N+1)}{2}$, and dividing by N^2 in total yields approximately $\frac{1}{2}$, which is what we obtained from the code.

Question 2 - SIMD

The program generates an array of data on the main process which is then distributed in equal chunks to the other threads. If the number of data points in the generated array is not divisible by the number of processes, the program will pad the array with zeros before sending the equally sized chunks. Each thread then calculates the 4th moment of it's share of the data. Finally, the main node collects each intermediate value from the threads and prints the 4th moment of the entire dataset.

Code:

```
/* This program calculates the 4th moment of an array of numbers.
 * The data is generated on the main process. Then, the data is
 * split amongst the processes in even chunks. Each process then
 * calculates it's portion of the moment. The results are finally
 * collected on the main process and the final moment is calculated.
 */

#include <iostream>
#include <cmath>
#include <numeric>
#include <algorithm>
#include <vector>
#include <mpi.h>

constexpr unsigned int N = 100'000'000U;
constexpr int n_moment = 4;

int main(int argc, char** argv) {
    int size, rank;
    double moment = 0.0;
    double global_moment = 0.0;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int padded_N = N % size == 0 ? N : N + size - N % size;
    std::vector<double> numbers;
```

```

// Generate some data
if (rank == 0) {
    numbers.reserve(padded_N); // Pad the vector to be a multiple of
the number of processes
    numbers.resize(padded_N);
    iota(numbers.begin(), numbers.end(), 1);
    std::transform(numbers.begin(), numbers.end(), numbers.begin(),
[])(double &x) { return x / N; });
} else {
    numbers.reserve(padded_N / size); // The other processes only
need to reserve space for their part of the vector
    numbers.resize(padded_N / size);
}

// Send data to every process
MPI_Scatter(numbers.data(), padded_N / size, MPI_DOUBLE,
numbers.data(), padded_N / size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

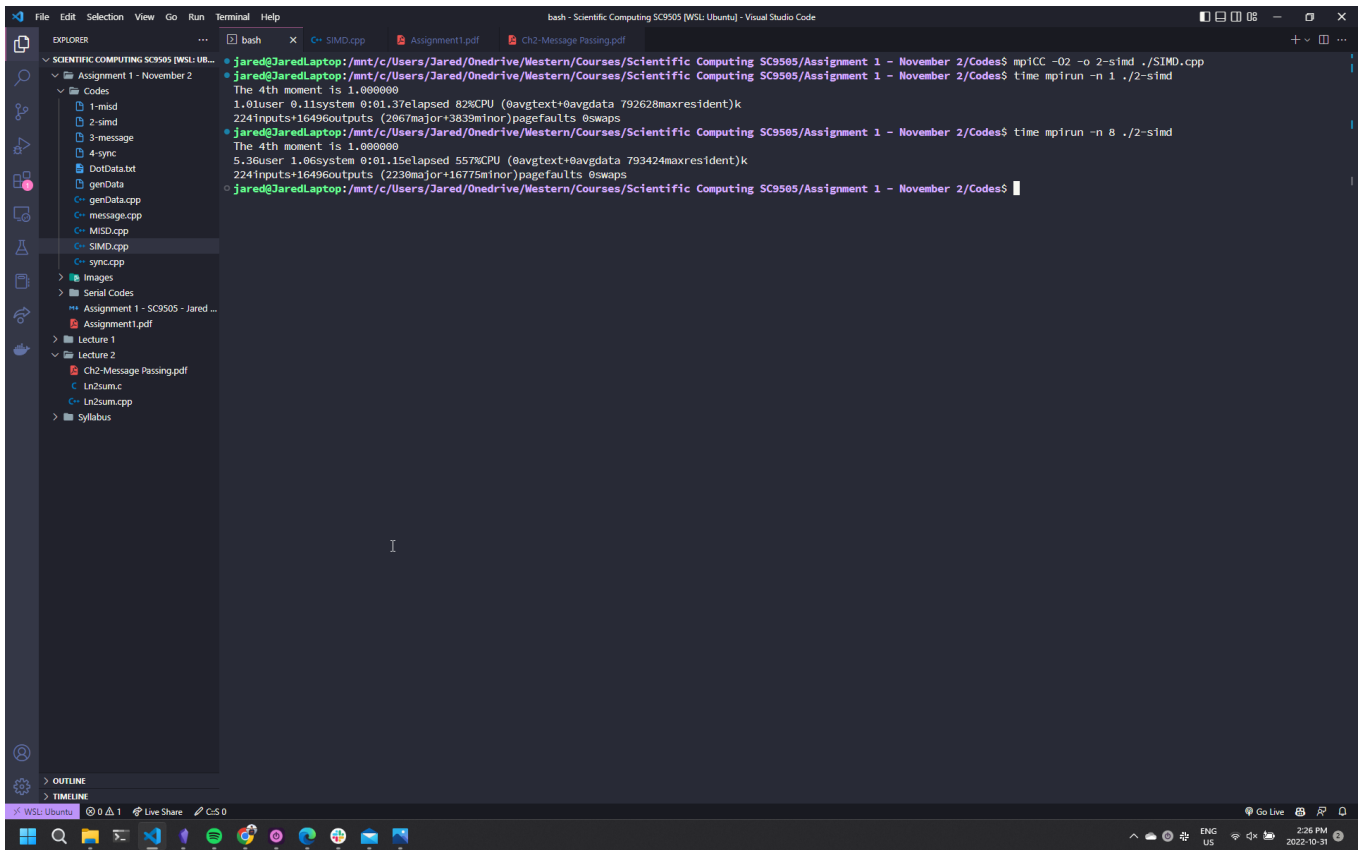
// Compute the moment
for (int i = 0; i < padded_N / size; i++) {
    moment += pow(numbers[i], n_moment);
}
moment /= N;

// Collect each process' moment
MPI_Reduce(&moment, &global_moment, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

// Print the result
if (rank == 0) {
    printf("The 4th moment is %f\n", global_moment);
}

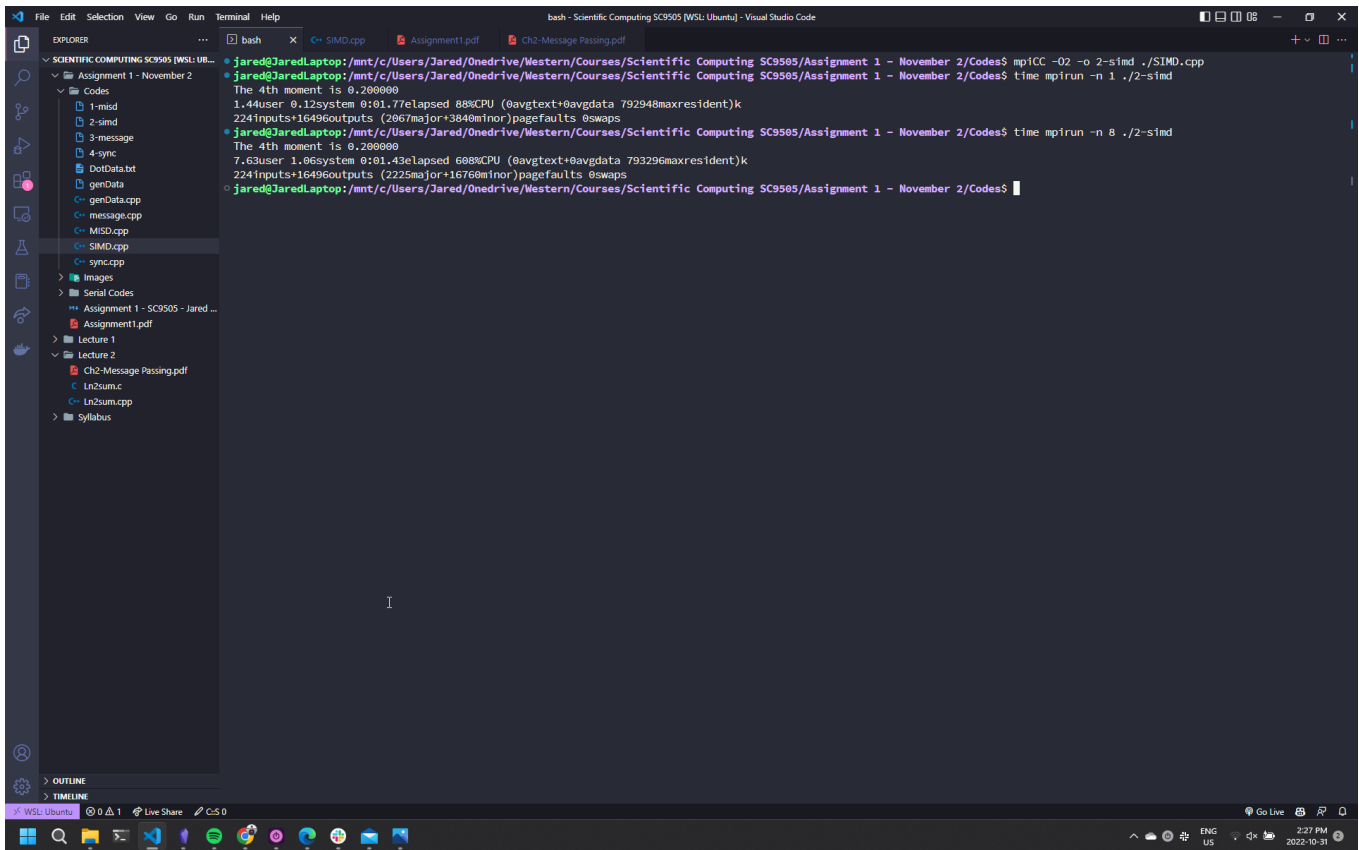
MPI_Finalize();
}

```



```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ g++ -O2 -o 2-simd ./SIMD.cpp
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 1 ./2-simd
The 4th moment is: 1.000000
1.00user 0.11system 0:01.37elapsed 82%CPU (0avgtext+0avgdata 792628maxresident)k
224inputs+16496outputs (2867major+3839minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 8 ./2-simd
The 4th moment is: 1.000000
5.36user 1.06system 0:01.15elapsed 557%CPU (0avgtext+0avgdata 793424maxresident)k
224inputs+16496outputs (2230major+16775minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

1. As a first test, I have created an array of all ones which we will calculate the 4th moment of. the code properly returns 1 as the the calculated value for the 4th moment of all ones, in accordance with the MISD problem.



```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ mpicc -O2 -o 2-simd ./SIMD.cpp
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 1 ./2-simd
The 4th moment is 0.200000
1.44user 0.12system 0:01.77elapsed 88%CPU (0avgtext+0avgdata 792948maxresident)k
224inputs+16496outputs (2867major+3840minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 8 ./2-simd
The 4th moment is 0.200000
7.63user 1.06system 0:01.43elapsed 608%CPU (0avgtext+0avgdata 793296maxresident)k
224inputs+16496outputs (2225major+16760minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

2) As another test, I have filled the vector with the numbers 1 through N all divided by N , giving numbers between 0 and 1. The program outputs the an answer of 0.2 for the forth moment, which again agrees with the result from the `MISD` problem.

Increasing the number of threads from 1 to 8 in this problem with $N = 100,000,000$ results in a speedup of $\sim 1.24x$,

Question 3 - Message Passing (Dot Product)

The program will read a chunk of data on the main thread and send it to one of the worker threads. Once each worker thread has been saturated with an equal amount of data, the main thread reads the remaining (potentially larger) chunk of data. Each thread is then responsible for calculating the dot product of the two arrays it received. The main process then collects each partial dot product, and prints out the collective result.

Code:

```
/* This program calculates the dot product of two vectors read from a
file.
* The main process reads the file and distributes the data to each
process.
* Each process can then begin calculating while the main process reads
the
* chunk of data (the main process is responsible for the last chunk).
* The results are collected on the main process and the final dot
product
* is calculated as the sum of the partial dot products.
*/

#include <iostream>
#include <fstream>
#include <vector>
#include <mpi.h>

class MPI_stuff
{
public:
    int size;
    int rank;

    MPI_stuff(int &argc, char** &argv)
    {
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    }
}
```



```

~MPI_stuff()
{
    MPI_Finalize();
}
};

// Get the number of rows of data in the file from first row of file
int GetNumberElements(std::ifstream &fin) {
    int n;
    if (fin.is_open())
        fin >> n;
    else { // no file to read from
        std::cout << "Input File not found\n";
        MPI_Abort(MPI_COMM_WORLD, -1);
    }
    return n;
}

// Read and distribute n elements of a two column data file accross MPI
Processes
void ReadArrays(std::ifstream &fin, std::vector<long double> &a,
std::vector<long double> &b, int size, const MPI_stuff &the_mpi) {
    MPI_Status status;

    if (the_mpi.rank == 0) {
        if(fin.is_open()) {
            // Main process reads data and immediately sends to another
process
            for (int i = 1; i < the_mpi.size; i++) {
                for(int j = 0; j < size; j++) {
                    fin >> a[j] >> b[j];
                }
                MPI_Send(&a[0], size, MPI_LONG_DOUBLE, i, 10,
MPI_COMM_WORLD);
                MPI_Send(&b[0], size, MPI_LONG_DOUBLE, i, 20,
MPI_COMM_WORLD);
            }
            // After other processes are saturated, read the remaining
data
            int j = 0;

```

```

        while(!fin.eof()) {
            fin >> a[j] >> b[j];
            j++;
        }
    } else { // fp null means no file to read from
        std::cout << "Input File not found\n";
        MPI_Abort(MPI_COMM_WORLD, -1);
    }
} else {
    MPI_Recv(&a[0], size, MPI_LONG_DOUBLE, 0, 10, MPI_COMM_WORLD,
&status);
    MPI_Recv(&b[0], size, MPI_LONG_DOUBLE, 0, 20, MPI_COMM_WORLD,
&status);
}
}

// Dot product of two vectors owned and fully stored on Boss node
float DotProduct(std::vector<long double> &a, std::vector<long double>
&b, int size) {

    // Work out the sum on each process
    long double sum=0, Gsum=0;
    for(int i = 0; i < a.size(); i++) {
        sum += a[i] * b[i];
    }

    // Collect results in Gsum
    MPI_Reduce(&sum, &Gsum, 1, MPI_LONG_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
    return Gsum;
}

int main(int argc, char** argv) {
    MPI_stuff the_mpi(argc, argv);

    std::ifstream fin;
    int n = 0;
    if(the_mpi.rank == 0) {
        fin.open("DotData.txt");
        n = GetNumberElements(fin);
    }
}

```

```

    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    int size = n / the_mpi.size;

    std::vector<long double> a, b;
    if(the_mpi.rank == 0){
        // Main process may require a slightly larger vector if n % size
        != 0
        a.reserve(size + n%the_mpi.size);  a.resize(size +
n%the_mpi.size);
        b.reserve(size + n%the_mpi.size);  b.resize(size +
n%the_mpi.size);
    } else {
        a.reserve(size);  a.resize(size);
        b.reserve(size);  b.resize(size);
    }
    ReadArrays(fin, a, b, size, the_mpi);

    float adotb = DotProduct(a, b, size);
    if(the_mpi.rank == 0) {
        std::cout << "The inner product is " << adotb << std::endl;
    }

    return 0;
}

```

```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ ./genData 10000000 100000
The inner product is 1e+07
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 1 ./3-message
224inputs+16496outputs (2867major+4181minor)pagefaults 8swaps
The inner product is 1e+07
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 8 ./3-message
224inputs+16504outputs (2226major+20913minor)pagefaults 8swaps
The inner product is 1e+07
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

1. This first example demonstrates the code should be working. I have first generated a file containing two vectors, each filled with 10^7 ones. The dot product of the two vectors is then exactly what we would expect. It should also be noted that the runtime with one thread and eight threads are nearly identical; this suggests that the program is I/O bottlenecked.

```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
EXPLORER
SCIENTIFIC COMPUTING SC9505 [WSL: Ubuntu]
  Assignment 1 - November 2
    Codes
      1-mid
      2-mid
      3-message
      4-sync
      DotData.txt
      genData
      genData.cpp
      message.cpp
      SIMD.cpp
      SIMD.cpp
      sync.cpp
    Images
    Serial Codes
    Assignment 1 - SC9505 - Jared ...
    Assignment1.pdf
    Lecture 1
    Lecture 2
    Ch2-Message Passing.pdf
    Ln2sum.c
    Ln2sum.cpp
    Syllabus
  OUTLINE
  TIMELINE

jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 1 ./3-message
The inner product is 2.49877e+06
3.20user 0.26system 0:05.02elapased 70%CPU (0avgtext+0avgdata 323624maxresident)k
224inputs+16504outputs (2068major+4181minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 8 ./3-message
The inner product is 2.49877e+06
31.10user 0.75system 0:06.77elapased 470%CPU (0avgtext+0avgdata 52540maxresident)k
224inputs+16504outputs (2227major+20962minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

2) In the second example, I have instead generated two vectors, each filled with 10^7 random numbers between 0 and 1. The result looks reasonable and given the first test, we can trust the program is working.

In this exercise, we parallelized the dot product and tried to minimize the amount of memory being used. The main process never contains the full array of data, it only reads in sequential chunks before sending it off to the worker process. In doing so, we have introduced an I/O bottleneck which currently defeats the purpose of parallelization altogether. For this program to be effective, the reading operation of the text file will need to become more efficient.

Question 4 - Synchronization

This program demonstrates some of the commands that can be used to control synchronization in MPI. The base code uses blocking send and receive instructions while dividing a list of tasks between threads. We can then investigate the effects of different synchronization commands such as `MPI_Barrier` and `MPI_Isend / MPI_Irecv` by passing an argument in the command line.

Code:

```
/* This program distributes a number of tasks between various threads.
 * The main process listens for all other processes to announce what task
 * each
 * process is responsible for. There is one process that takes longer to
 * send in
 * this confirmation that the rest. We explore various methods of
 * speeding up the
 * the program as a whole by reducing wait times.
 */

#include <iostream>
#include <mpi.h>

class MPI_stuff
{
public:
    int size;
    int rank;

    MPI_stuff(int &argc, char** &argv)
    {
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    }

    ~MPI_stuff()
    {
        MPI_Finalize();
    }
}
```

```

};

// Calculates Pi
void waste_your_time(int N = 1'000'000'000U) {
    double h = 1.0/N, sum = 0.0, x, pi;
    for (long i = 0; i < N; i++) {
        x = i*h;
        sum += 4.0 / (1.0 + x*x);
    }
    pi = h * sum;
    printf("%.12f\n", pi);
}

int main(int argc, char** argv) {
    MPI_stuff the_mpi(argc, argv);
    MPI_Status status;
    MPI_Request req_recv10, req_send10;

    // mode = 0: Base code
    // mode = 1: Barrier between loops
    // mode = 2: Unique send/recv tags
    // mode = 3: Non-blocking send/recvs with Wait statements
    // mode = 4: Non-blocking send/recvs, recvs seperated from main loop,
with Barrier between loops
    // mode = 5: Same as 4, but no Barrier between loops
    int loops = 2, mode = 0;
    int task, taskx, node, tag = 10;

    if (argc == 3) {
        loops = atoi(argv[1]);
        mode = atoi(argv[2]);
    }

    for (int i = 0; i < loops; i++) {
        for (task = 1 + 10*i; task <= 10*(1 + i); task++) {

            node = task - the_mpi.size * (task / the_mpi.size);
            // 2nd Modification: Force the blocking behaviour of sends by
making the tags unique
            if (mode == 2) {

```

```

        tag = 10 + node;
    }

    // Sending Data
    if (node == the_mpi.rank && node != 0) {    //Check to see if
this is my task to do
        if (node == 2) waste_your_time();
        // 3rd Modification: Use non blocking sends and recieving
to avoid waiting
        // 4th Modification: Also use non-blocking routines
        if (mode == 3 || mode == 4 || mode == 5) {
            MPI_Isend(&task, 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
&req_send10); //Tell Boss I am the one
        } else {
            MPI_Send(&task, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
//Tell Boss I am the one
        }
    }

    // Recieving Data
    // Note: Modes 4 and 5 move this section of the loop to an
independent loop
    if (the_mpi.rank == 0 && mode != 4 && mode != 5) {
        if (node == 0) {
            printf("Processor %d will compute %d\n", 0, task);
        } else {
            // 3rd Modification: Use non blocking sends and
recieving to avoid waiting
            if (mode == 3) {
                req_recv10 = MPI_REQUEST_NULL;
                MPI_Irecv(&taskx, 1, MPI_INT, MPI_ANY_SOURCE,
tag, MPI_COMM_WORLD, &req_recv10);
                MPI_Wait(&req_recv10, &status);
            } else {
                MPI_Recv(&taskx, 1, MPI_INT, MPI_ANY_SOURCE, tag,
MPI_COMM_WORLD, &status);
            }
            printf("Processor %d will compute %d\n",
status.MPI_SOURCE, taskx);
        }
    }

```



```

    }
}

// 4th Modification: Seperate the receiving so the main process
does not get stuck waiting
if (the_mpi.rank == 0 && (mode == 4 || mode == 5)) {
    for (task = 1 + 10*i; task <= 10*(1 + i); task++) {

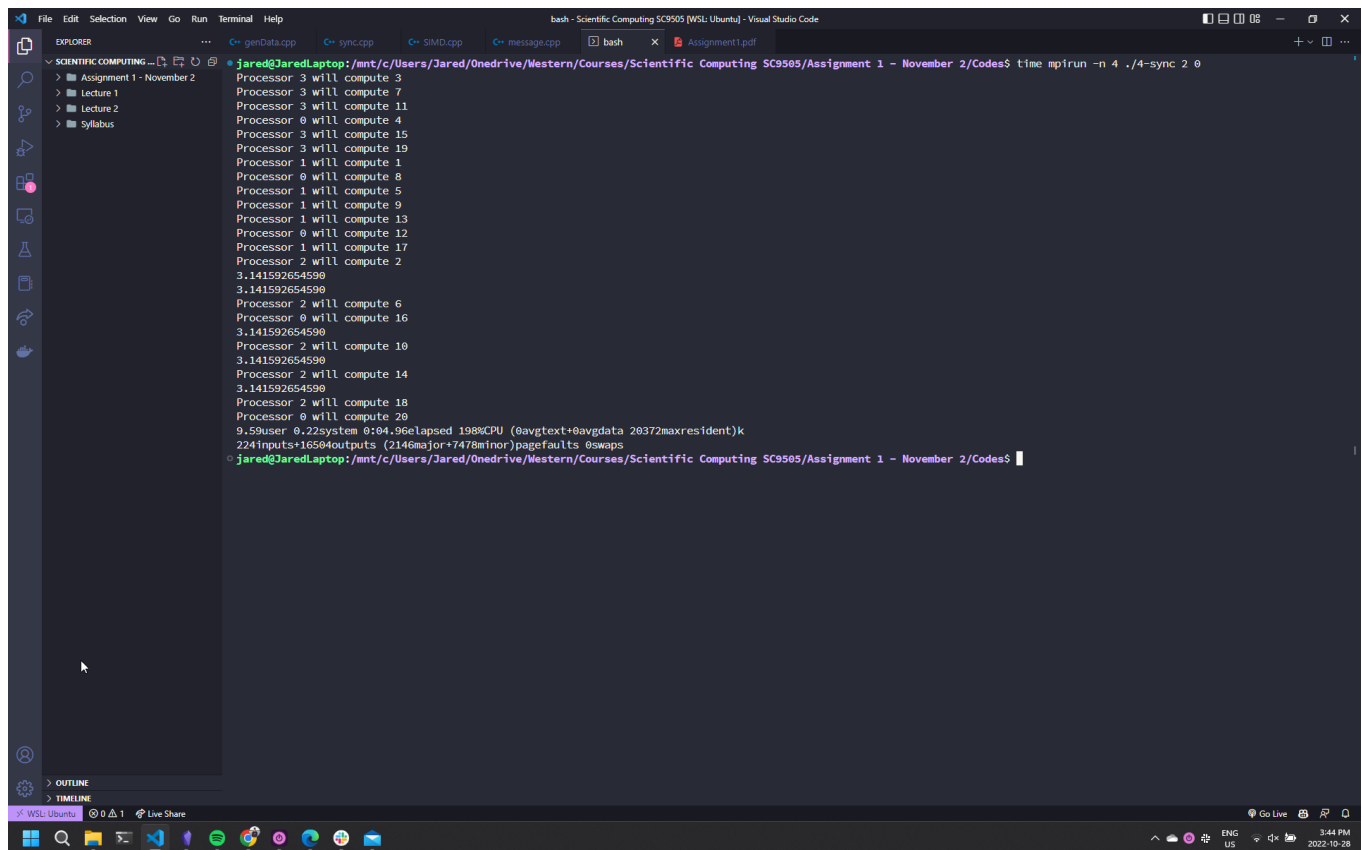
        node = task - the_mpi.size * (task / the_mpi.size);

        if (node == 0) {
            printf("Processor %d will compute %d\n", 0, task);
        } else {
            req_recv10 = MPI_REQUEST_NULL;
            MPI_Irecv(&taskx, 1, MPI_INT, MPI_ANY_SOURCE, tag,
MPI_COMM_WORLD, &req_recv10);
            MPI_Wait(&req_recv10, &status);
            printf("Processor %d will compute %d\n",
status.MPI_SOURCE, taskx);
        }
    }
}

// 1st Modification: Require each group of 10 tasks to finish
before moving on
// 4th Modification: Also prevent advancement to avoid unwanted
behaviour from the Isend/Irecv
// (since they are open and receiving from any source)
if (mode == 1 || mode == 4) {
    MPI_Barrier(MPI_COMM_WORLD);
}
}
return 0;
}

```

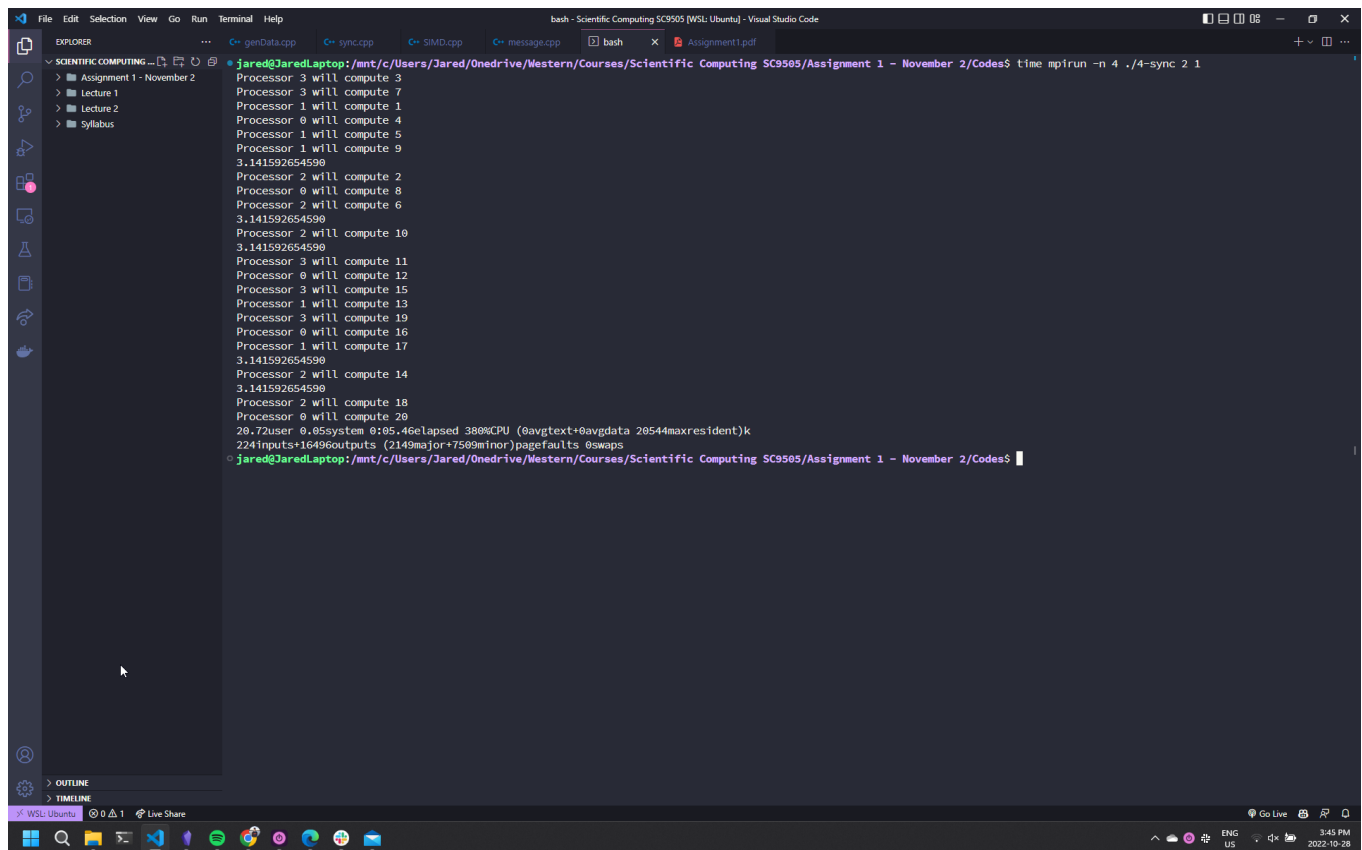
Base Code:



```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 4 ./4-sync 2 0
Processor 3 will compute 3
Processor 3 will compute 7
Processor 3 will compute 11
Processor 0 will compute 4
Processor 3 will compute 15
Processor 3 will compute 19
Processor 1 will compute 1
Processor 0 will compute 8
Processor 1 will compute 5
Processor 1 will compute 9
Processor 1 will compute 13
Processor 0 will compute 12
Processor 1 will compute 17
Processor 2 will compute 2
3.141592654590
3.141592654590
Processor 2 will compute 6
Processor 0 will compute 16
3.141592654590
Processor 2 will compute 10
3.141592654590
Processor 2 will compute 14
3.141592654590
Processor 2 will compute 18
Processor 0 will compute 20
9.59user 0.22system 0:04.96elapsed 198%CPU (0avgtext+0avgdata 20372maxresident)k
224inputs+16504outputs (2146major+7478minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

This is the code provided in lecture, adapted to C++. I get a similar output compared to the example output found in the lecture notes. I have implemented the `waste_your_time()` as a function that calculates π (calculating $\arctan(1)$ using a finite integral approximation).

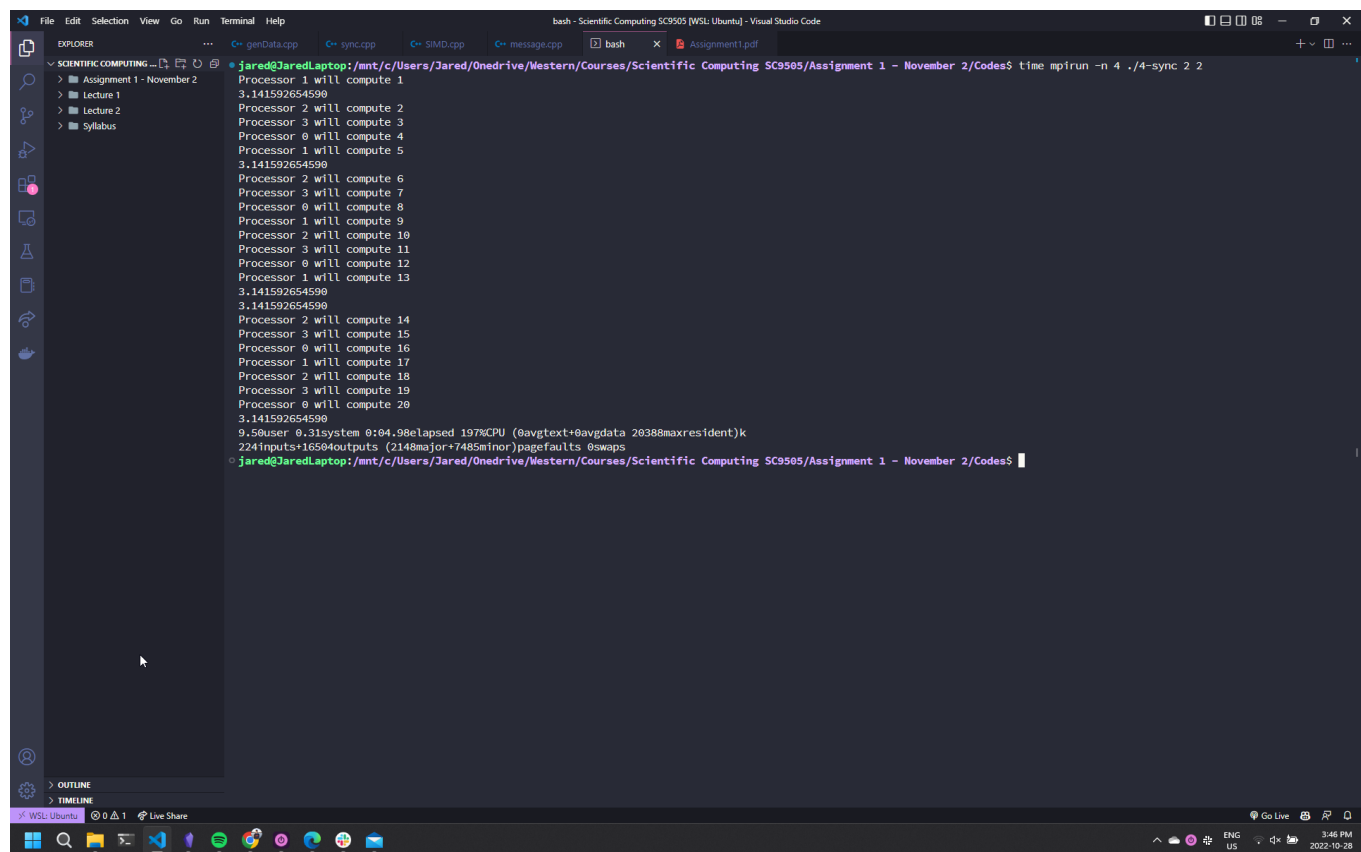
Modification 1:



```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 4 ./4-sync 2 1
Processor 3 will compute 3
Processor 3 will compute 7
Processor 1 will compute 1
Processor 0 will compute 4
Processor 1 will compute 5
Processor 1 will compute 9
3.141592654590
Processor 2 will compute 2
Processor 0 will compute 8
Processor 2 will compute 6
3.141592654590
Processor 2 will compute 10
3.141592654590
Processor 3 will compute 11
Processor 0 will compute 12
Processor 3 will compute 15
Processor 1 will compute 13
Processor 3 will compute 19
Processor 0 will compute 16
Processor 1 will compute 17
3.141592654590
Processor 2 will compute 14
3.141592654590
Processor 2 will compute 18
Processor 0 will compute 20
20.72user 0.05system 0:05.46elapsed 380%CPU (0avgtext+0avgdata 20544maxresident)k
224inputs+16496outputs (2149major+7509minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

The first modification introduced a `MPI_Barrier()` between loops. This requires that all tasks in each block (here each block contains 10 tasks) finish before the next block of tasks can begin being evaluated. This can be useful if the second set of tasks depends on the output of the first 10 tasks.

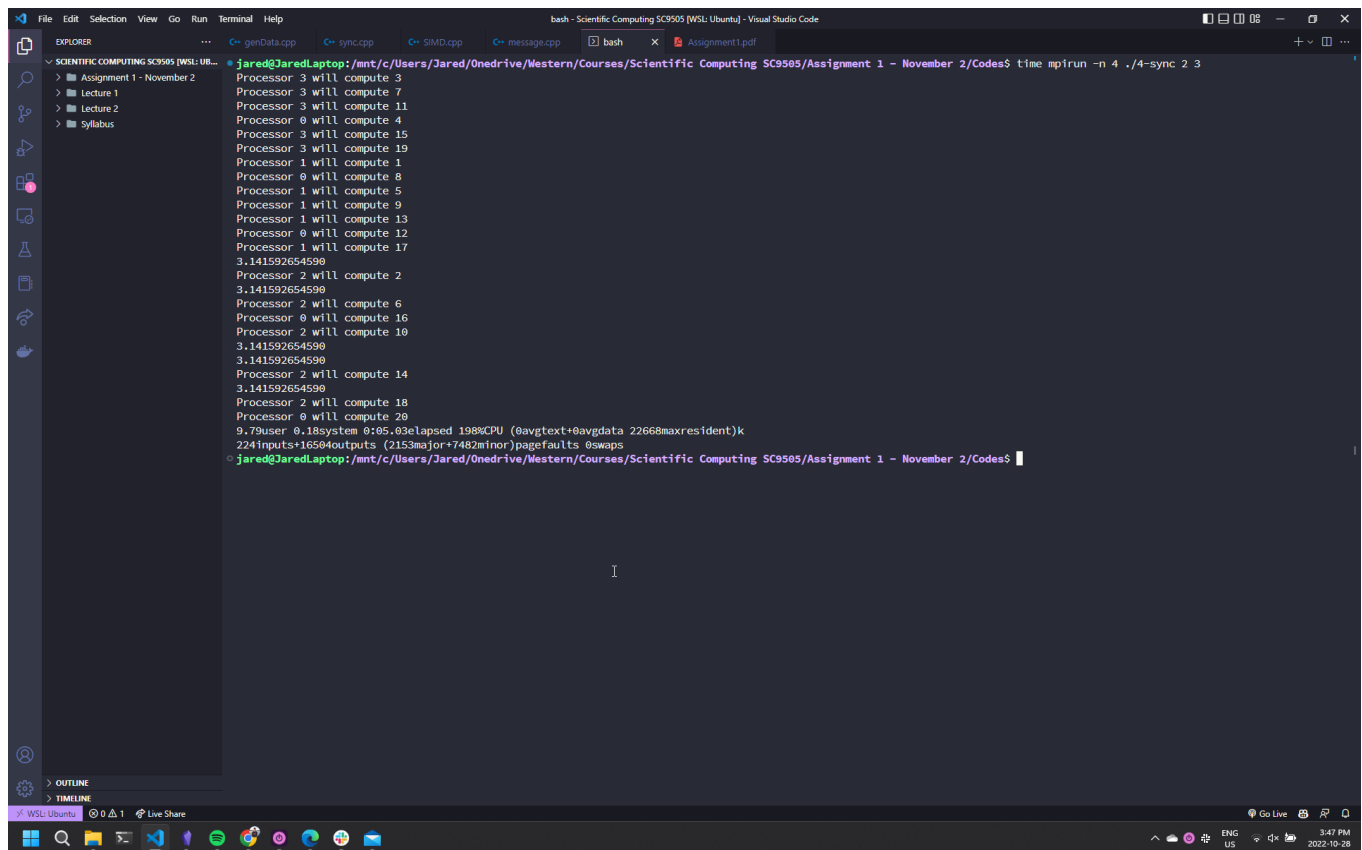
Modification 2:



```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 4 ./4-sync 2 2
Processor 1 will compute 1
3.141592654590
Processor 2 will compute 2
Processor 3 will compute 3
Processor 0 will compute 4
Processor 1 will compute 5
3.141592654590
Processor 2 will compute 6
Processor 3 will compute 7
Processor 0 will compute 8
Processor 1 will compute 9
Processor 2 will compute 10
Processor 3 will compute 11
Processor 0 will compute 12
Processor 1 will compute 13
3.141592654590
3.141592654590
Processor 2 will compute 14
Processor 3 will compute 15
Processor 0 will compute 16
Processor 1 will compute 17
Processor 2 will compute 18
Processor 3 will compute 19
Processor 0 will compute 20
3.141592654590
9.50user 0.31system 0:04.98elapsed 197%CPU (0avgtext+0avgdata 20388maxresident)k
224inputs+16504outputs (2148major+7485minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

The second modification uses unique tag identifiers for the communications. Since we are using the blocking send / receive functions, this forces each task to be completed in order (essentially making parallelization pointless).

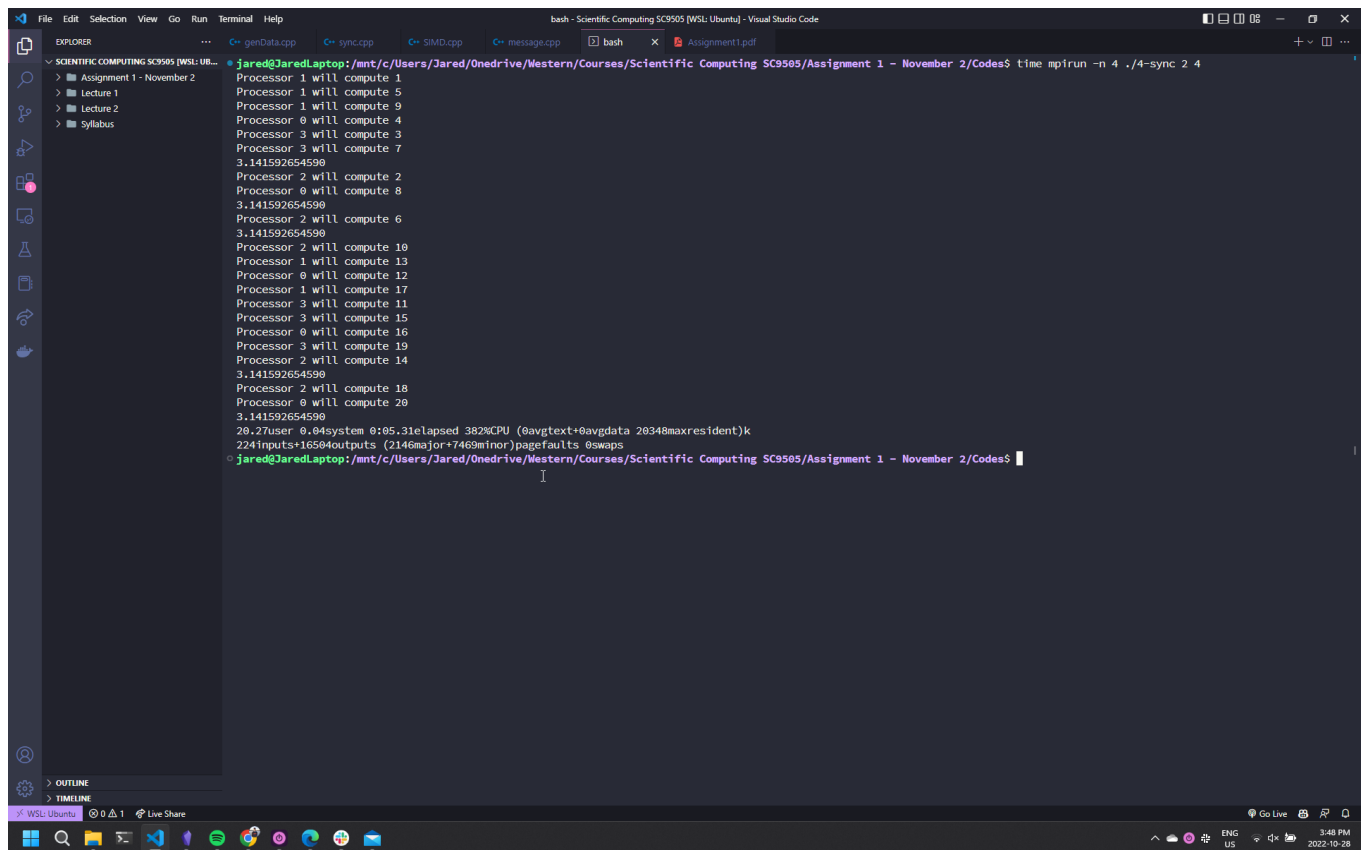
Modification 3:



```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 4 ./4-sync 2 3
Processor 3 will compute 3
Processor 3 will compute 7
Processor 3 will compute 11
Processor 0 will compute 4
Processor 3 will compute 15
Processor 3 will compute 19
Processor 1 will compute 1
Processor 0 will compute 8
Processor 1 will compute 5
Processor 1 will compute 9
Processor 1 will compute 13
Processor 0 will compute 12
Processor 1 will compute 17
3.141592654590
Processor 2 will compute 2
3.141592654590
Processor 2 will compute 6
Processor 0 will compute 16
Processor 2 will compute 10
3.141592654590
3.141592654590
Processor 2 will compute 14
3.141592654590
Processor 2 will compute 18
Processor 0 will compute 20
9.79user 0.18system 0:05.03elapsd 198%CPU (0avgtext+0avgdata 22668maxresident)k
224inputs+16504outputs (2153major+7482minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

Our third modification replaces the blocking send and receive instructions with instantaneous alternatives followed by a wait statement. On the sending process, we don't actually care when the message is received, so this allows each sending process to continue onto the next task immediately. The receiving process will have to wait for the message before using the to-be received data.

Modification 4:

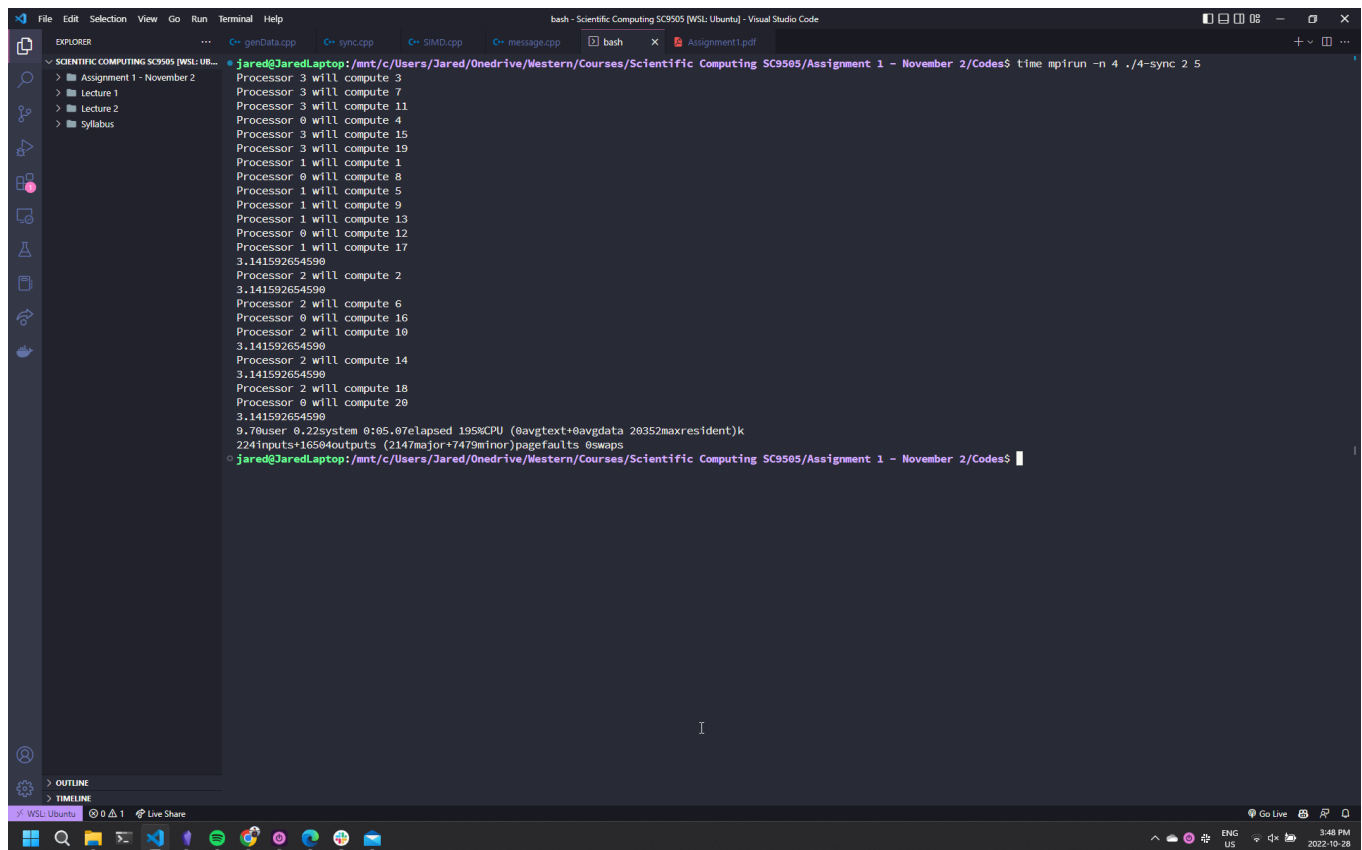


```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
EXPLORER
  SCIENTIFIC COMPUTING SC9505 [WSL: Ubuntu]
    Assignment 1 - November 2
    Lecture 1
    Lecture 2
    Syllabus
  genData.cpp
  sync.cpp
  SIMD.cpp
  message.cpp
  bash
  Assignment1.pdf

jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 4 ./4-sync 2 4
Processor 1 will compute 1
Processor 1 will compute 5
Processor 1 will compute 9
Processor 0 will compute 4
Processor 3 will compute 3
Processor 3 will compute 7
3.141592654590
Processor 2 will compute 2
Processor 0 will compute 8
3.141592654590
Processor 2 will compute 6
3.141592654590
Processor 2 will compute 10
Processor 1 will compute 13
Processor 0 will compute 12
Processor 1 will compute 17
Processor 3 will compute 11
Processor 3 will compute 15
Processor 0 will compute 16
Processor 3 will compute 19
Processor 2 will compute 14
3.141592654590
Processor 2 will compute 18
Processor 0 will compute 20
3.141592654590
20.27user 0.04system 0:05.31elapsd 382%CPU (@avgtext+@avgdata 20348maxresident)k
224inputs+16504outputs (2146major+7469minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

This modification is similar to the third one, except we have moved the receiving instructions into a completely independent loop. This allows the main receiving process and the worker processes to loop independently. Here, we also have a `MPI_Barrier` inserted between each subgroup of (here 10) tasks.

Modification 5:



```
bash - Scientific Computing SC9505 [WSL: Ubuntu] - Visual Studio Code
EXPLORER
  SCIENTIFIC COMPUTING SC9505 [WSL: Ubuntu]
    Assignment 1 - November 2
    Lecture 1
    Lecture 2
    Syllabus
  bash
  Assignment1.pdf

jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$ time mpirun -n 4 ./4-sync 2 5
Processor 3 will compute 3
Processor 3 will compute 7
Processor 3 will compute 11
Processor 0 will compute 4
Processor 3 will compute 15
Processor 3 will compute 19
Processor 1 will compute 1
Processor 0 will compute 8
Processor 1 will compute 5
Processor 1 will compute 9
Processor 1 will compute 13
Processor 0 will compute 12
Processor 1 will compute 17
3.141592654590
Processor 2 will compute 2
3.141592654590
Processor 2 will compute 6
Processor 0 will compute 16
Processor 2 will compute 10
3.141592654590
Processor 2 will compute 14
3.141592654590
Processor 2 will compute 18
Processor 0 will compute 20
3.141592654590
9.70user 0.22system 0:05.07elapsed 195%CPU (0avgtext+0avgdata 20352maxresident)k
224inputs+16504outputs (2147major+7479minor)pagefaults 0swaps
jared@JaredLaptop: /mnt/c/Users/Jared/OneDrive/Western/Courses/Scientific Computing SC9505/Assignment 1 - November 2/Codes$
```

Finally as a bonus change, I have removed the `MPI_Barrier` between each subgroup of tasks. This will allow the worker processes to run through as many tasks as it can without waiting for each subgroup to be fully received first. This could be more efficient if the next group of tasks is independent of the previous group, and so on. One must be careful when receiving in this case as the receive calls are set to accept from any source.