REPORT: Optimising the communication pattern of the Manager-worker version of the Mandelbrot calculation program

Duc Dat Hoang

High-performance Computing ECMM461

1 Introduction

This report details the modifications made to a Mandelbrot set calculation program that employs the Message Passing Interface (MPI) for parallelization. The implementation of the program utilizes a manager-worker pattern to distribute computational tasks among MPI processes, specifically focusing on the calculation of the Mandelbrot set in a two-dimensional array representing points in the complex plane. Additionally, the report presents a comparative analysis of the original and modified programs' performance, focusing on their parallel scaling characteristics. By measuring and plotting the parallel speed-up relative to the number of MPI processes for both versions, we assess the impact of the modifications on the program's efficiency. Through this analysis, we aim to quantify the performance improvements of the modified program.

P/s: For details on the file structure and instructions on how to run the program, please refer to the README.md file or visit the following link: `https://github.com/dathd6/Mandelbrot-MPI`.

2 Description of modifications to the program

```
27  /* Task 1.1: Initial variables
28   *    - Create buffer space for sending and receiving data
29   *    - Missing data value (negative value)
30   * */
31  float buffer[N_IM + 3];
32  const int MISSING_DATA_VALUE = -1;
```

Figure 1: Allocate *buffer* space to facilitate data transmission and reception. In the complex plane, one column is designated for the imaginary axis (`N_IM`), resulting in a buffer size of `N_IM + 3`. Within this buffer, the entry at position `N_IM + 1` signifies the current rank of the process, while the entry at `N_IM + 2` serves as a placeholder for either the missing data indicator or the present value i which show the column of values that buffer are stored. Concurrently, the missing data value is set to -1.

```
40  /* Set to true to write out results*/
41  const bool doIO = false;    // [Courswework] Set to TRUE
42  const bool verbose = false; // [Coursework] Set to TRUE
```

Figure 2: When conducting the test for task 1, assign the doIO value as TRUE. For task 2, change the doIO value to FALSE.

```
85  /* Task 1.5: do_communication function is no longer required
86   * */
87  // void do_communication(int myRank) {
88  //
89  //   MPI_Group worldGroup, workerGroup;
90  //   MPI_Comm workerComm;
91  //   int zeroArray = {0};
272   /* Communicate results so rank 1 has all the values */
273   // Task 1.5: this is no longer required
274   // do_communication(myRank);
```

Figure 3: Comment out the call to ‘do_communication‘ as this is no longer required

```
230  // Worker Processes
231  else {
232    /* Task 1.2 */
233    buffer[N_IM + 1] = myRank; // Store current process to buffer
234    buffer[N_IM + 2] = MISSING_DATA_VALUE; // Task 1.3: initial missing data value
235    while (true) {
236      // Send request for work
237      MPI_Send(&buffer, N_IM + 3, MPI_INT, 0, 100 + myRank, MPI_COMM_WORLD);
238      // Receive i value to work on
239      MPI_Recv(&i, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
240      if (i == endFlag) {
241        break;
242      } else {
243        calc_vals(i);
244      }
245    } // while(true)
246  }   // else worker process
```

Figure 4: **Worker Processes** - At the beginning of the operation, each parallel process will initialize a buffer. This buffer will contain the process's rank and a placeholder for missing data, signifying that the buffer's values have yet to be computed. Each worker process will then transmit a buffer array of size `N_IM + 3` to the manager process to store the values in nIter.

```
50  void calc_vals(int i) {
62    buffer[N_IM + 2] = i; // Set value column value to buffer
63    /* Loop over imaginary axis */
64    for (j = 0; j < N_IM + 1; j++) {
65      z0 = z_Re[i] + z_Im[j] * I;
66      z = z0;
67      /* Iterate up to a maximum number or bail out if mod(z) > 2 */
68      k = 0;
69      while (k < maxIter) {
70        buffer[j] = k; // Store all value of current column i in buffer
71        if (cabs(z) > 2.0)
72          break;
73        z = z * z + z0;
74        k++;
75      }
76    }
```

Figure 5: **Worker Processes** - Each worker, while computing the column of values for the current i, assigns the value of i to the position `N_IM + 2` in the buffer to signify the calculation has been implemented. The previous code updates the nIter array at line 70, while the modified version updates the buffer to transfer to the manager.

```
204     // Hand out work to worker processes
205     for (i = 0; i < N_RE + 1; i++) {
206       // Receive request for work
207       /* Task 1.4: Receiving the buffer array to store in the correct column of the nIter */
208       MPI_Recv(&buffer, N_IM + 3, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status); // Receive the message
209       nextProc = buffer[N_IM + 1];          // Next process
210       int c_values = buffer[N_IM + 2];      // Unpack the column of values
211       if (c_values != MISSING_DATA_VALUE) { // Task 1.3: If there are no data -> manager process discard the data
212         for (j = 0; j < N_IM + 1; j++) {
213           nIter[c_values][j] = buffer[j]; // stores it in the correct column of the nIter array
214         }
215       }
216       // Send i value to requesting process
217       MPI_Send(&i, 1, MPI_INT, nextProc, 100, MPI_COMM_WORLD);
218     }
```

Figure 6: **Manager Process** - The manager receives a buffer array of size `N_IM + 3`. It checks the value at index `N_IM + 2` within the buffer to determine if the data for the current buffer has been computed. If the data is calculated, the manager updates the nIter array for the current i to match the received buffer.

```
218     // Tell all the worker processes to finish (once for each worker process = nProcs-1)
219     for (i = 0; i < nProcs - 1; i++) {
220       // Receive request for work
221       /* Task 1.4: Get the process rank from buffer and tell the worker to finish its process
222        *           Receiving the last buffer from the array to store in the correct column of the nIter
223        * */
224       MPI_Recv(&buffer, N_IM + 3, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status); // Receive the message
225       nextProc = buffer[N_IM + 1];          // Next process
226       int c_values = buffer[N_IM + 2];      // Unpack the column of values
227       if (c_values != MISSING_DATA_VALUE) { // Task 1.3: If there are no data -> manager process discard the data
228         for (j = 0; j < N_IM + 1; j++) {
229           nIter[c_values][j] = buffer[j]; // stores it in the correct column of the nIter array
230         }
231       }
232       // Send endFlag to finish
233       MPI_Send(&endFlag, 1, MPI_INT, nextProc, 100, MPI_COMM_WORLD);
234     }
```

Figure 7: **Manager Process** - The manager collects the final buffer array from each worker process, which was sent in the last iteration but has not yet been received. This buffer is then stored in the corresponding position i of the nIter array. The manager instructs all worker processes to terminate by extracting the rank from the buffer and sending an endFlag to each worker process, thereby stopping their respective while loops.

```
252     /* Write out results */
253     if (doIO && myRank == 0) { /* Task 1.6: Write out the results from the manager process */
254       if (verbose) {
255         printf("Writing out results from process %d \n", myRank);
256       }
257       write_to_file("mandelbrot.dat");
258     }
```

Figure 8: Write out the results from the manager process instead of the rank 1 process.

3 Scaling test

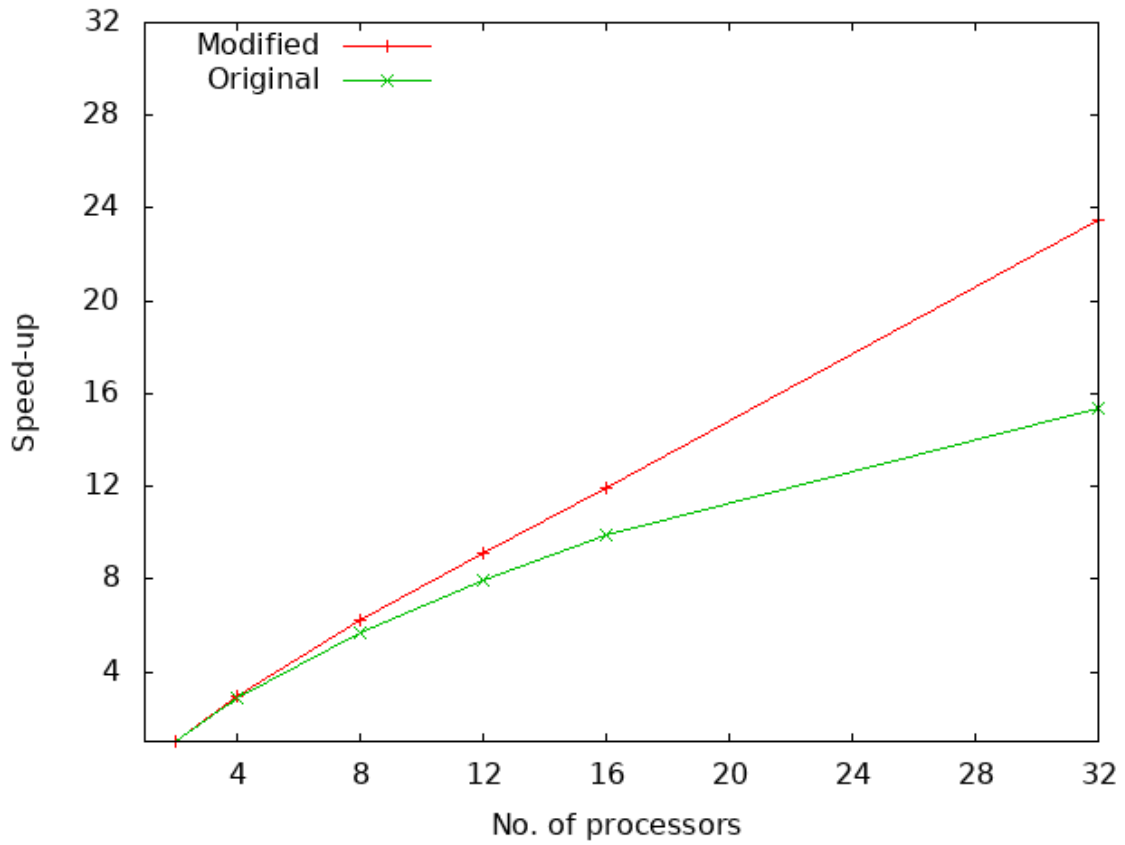

Figure 9: Scaling test plot show parallel-speed up against number of MPI processes for both the original program (green line) and modified version (red line).

4 Discussion

According to figure 9, The modified version clearly perform better than original program at every point measured. Additionally, the slope of the modified version's line is steeper than that of the original version, especially noticeable from around 8 processors onward. This suggests that the modified version is benefiting more efficiently from the added processors. This could be explained by the previous approach leads to the transmission of excess data, as each worker sends the entire computational domain, including regions it did not compute. To address this inefficiency, we modified the communication pattern within the program. Rather than waiting until the end of all computations to aggregate results, worker processes now send their computed column of values to the manager immediately after each is completed. This change aims to reduce the data transmitted between processes, potentially enhancing the program's overall parallel efficiency.