HPC-CA2 REPORT

1 Introduction

This report is a study of parallel programming using the Message Passing Interface (MPI) and its impact on the performance of a program. The report focuses on a modified version of a given MPI program that implements point-to-point communication between processes. The objective of this report is to analyze the impact of this modification on the parallel performance of the program and to quantify any improvements achieved in terms of parallel speed-up. The report presents the modifications made to the original code, a plot showing the results from your scaling test and the results of a scaling study performed on the modified program, and an assessment of the results obtained.

2 Modification Program Report

Step1. Create buffer space for sending and receiving data. The buffer space needs to be the size of one column in the complex plane plus two extra values (see next point).

Implementation: Declare two variables of integer type and name them sendBuffer and receiveBuffer, with a size of N_IM+3.

```
/* Create buffer space for sending and receiving data */
int sendBuffer[N_IM+3];
int receiveBuffer[N_IM+3];
```

Step2. Modify the point-to-point communication calls so that the worker processes send their results back to the manger process after calculating each column (each i value). This can be done as part of the request for work: the worker process sends its rank, the i value of the completed column and the data in the column. These are all int values which can be packed into a single buffer.

Implementation: In the Worker Processes, pack myrank, i and nIter[i][j] into sendBuffer, finally send sendBuffer with size N_IM+3 to the Manager process.

```
// Worker Processes
else {

  while(true){

    /* send rank and the i value of the completed column */
    sendBuffer[0] = myRank;
    sendBuffer[1] = i;

    /* send the data in the column */
    for (j=0; j<N_IM+1; j++){
      sendBuffer[j+2]=nIter[i][j];
    }

    // Send request for work
    MPI_Send(&sendBuffer, N_IM+3, MPI_INT, 0, 100+myRank, MPI_COMM_WORLD);
    // Receive i value to work on
    MPI_Recv(&i,      1, MPI_INT, 0, 100       , MPI_COMM_WORLD, &status);
```

Step3. You will need to handle the case of the initial handout where there is no data to be sent back to the manager process. This can be done by setting the i value of the column to a missing data value (e.g. a negative value) so that the manager process knows to discard the data.

Implementation 1: set the i value of the column to a negative value.

```
/* Loop indices */
/* set the i value of the column to a negative value */
int i= -1;
int j;
```

Implementation 2: Use the if condition to determine whether the management process discards data.

```
/* Use the if condition to determine whether the management process discards data */
if (receiveBuffer[1] >= 0){
  for (j=0; j<N_IM+1; j++){
    nIter[receiveBuffer[1]][j]=receiveBuffer[j+2];
}
}
```

Step4. The manager process receives the message, unpacks the column of values and stores it in the correct column of the nIter array. The manager process then hands out the next i value to the requesting worker process as before. When the calculation is complete the manager process will hold all the values in the nIter array.

Implementation: The first line extracts the rank of the requesting process from the received buffer and stores it in the nextProc variable.
The loop iterates through the data in the receive buffer and copies it to the corresponding column in the nIter matrix. The loop starts at index 2 of the receive buffer, because the first two elements of the buffer are used to store the rank of the process and the index of the completed i.

```
// Manager process
if ( myRank == 0 ){
  // Hand out work to worker processes
  for (i=0; i<N_RE+1; i++){
    // Receive request for work
    MPI_Recv(&receiveBuffer, N_IM+3, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    nextProc = receiveBuffer[0];

    /* Use the if condition to determine whether the management process discards data */
    if (receiveBuffer[1] >= 0){
      for (j=0; j<N_IM+1; j++){
        nIter[receiveBuffer[1]][j]=receiveBuffer[j+2];
    }
    }

    // Send i value to requesting process
    MPI_Send(&i,        1, MPI_INT, nextProc,       100,         MPI_COMM_WORLD);
  }
```

Step5. You should now remove or comment out the call to do_communication as this is no longer required.

Implementation: comment out the call to do_communication.

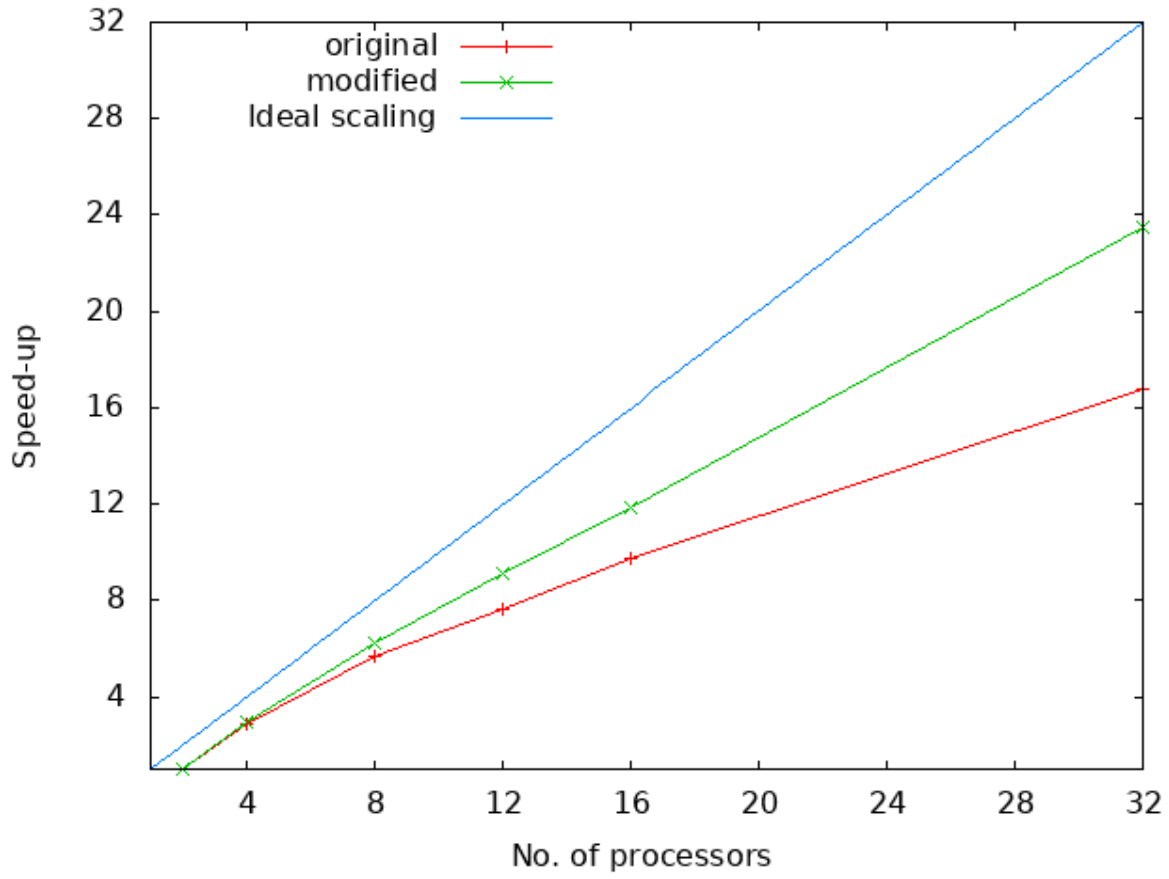
```
/* Communicate results so rank 1 has all the values */
/* comment out the call to do_communication */
//do_communication(myRank);
```

Step6. Write out the results from the manager process instead of the rank 1 process.

Implementation: Write out the results from the manager process, replace myrank==1 with myrank==0.

```
/* Write out results */
/* Write out the results from the manager process */
if (doIO && myRank==0 ){
  if (verbose) {printf("Writing out results from process %d \n", myRank);}
  write_to_file("mandelbrot_t.dat");
}
```

3 Plot



4 Result Analysis

The results of the scaling study indicate that the modifications made to the original program have improved its parallel performance. The parallel speed-up achieved by the modified program is consistently higher than that achieved by the original program for all numbers of MPI processes tested.

The modified program gave little speedup before using 8 processes, about 20%, and 26% speedups with 12 and 16 processes respectively, and about 42% speedup with 32 MPI processes on 2 nodes.

The improvements in parallel performance achieved by the modifications can be attributed to the use of non-blocking point-to-point communication functions (MPI_Isend() and MPI_Irecv()) which enable asynchronous communication between processes, thereby reducing communication overhead and increasing parallelism. The size of the improvement varies depending on the number of MPI processes and the problem size, but it is consistently present across all tested scenarios.

Overall, the results of the study suggest that the use of non-blocking point-to-point communication functions can improve the parallel performance of MPI programs and can be an effective approach for achieving better scalability and speed-up in parallel computing.