Techniques d'optimisation de la parallélisation TP4

YANG Yizhi

April 2024

1 Introduction

In this lab, we will apply MPI to realize the parallelization and analyse the performance speed-up.

2 Ping Pong

As a context, here we introduce the different MPI communication modes:

Standard mode communication, also known as blocking communication, involves synchronous data transfer between processes. In this mode, the sender process is blocked until the message is received by the receiver process. While straightforward to use, standard mode communication can lead to potential performance issues, especially in scenarios where processes spend a significant amount of time waiting for messages.

Buffered mode communication allows the sender process to continue execution immediately after buffering the message for transmission, without waiting for the receiver to receive it. This mode can reduce the blocking time for the sender, potentially improving overall performance. Buffering may introduce additional memory overhead, as it requires space to store the messages until they are transmitted.

Ready mode communication allows the sender process to determine if the receiver process is ready to receive a message before initiating communication. This mode can help prevent deadlocks by allowing processes to check the readiness of their communication partners before sending messages. Ready mode communication is less commonly used compared to standard and buffered modes due to its specific use case and potential performance implications.

Synchronous mode communication ensures that the sender and receiver processes synchronize their communication. In this mode, the sender process blocks until the message is received by the receiver process, and vice versa. Synchronous communication is useful in scenarios where strict synchronization between processes is required. Synchronous communication can introduce additional overhead, as it requires coordination between processes to synchronize the message exchange.

2.1 Set up

We firstly implement a simple ping from rank 0 to rank 1 with a call of sleep(2). During the measure in using OpenMPI 2.0.1, we observed a gap of execution time at the message size between 4000 and 4096 Bytes, which is caused of the fact that MPI applied different communication modes for different message sizes. For a small message size, the communication mode is buffered, as increasing the message size over a certain limit, the communication mode will be synchronous. At that moment, the timer measures actually both sent and reception process, including the two seconds of "sleep".

For my implementation, the gap appears at the message size of 65400 Bytes.

```
...
65390 0.001388
65400 2.00794
...
```

Then we remove the call of sleep(), this gap disappears.

```
...
65390 0.00175
65400 0.003919
...
```

To measure the actual sending time of a message, we need to apply the synchronous mode of communication.

We complete this ping-pong implementation. The measured time corresponds to a message exchange. In fact, we put the timer at the process of rank 0 which sends firstly a message to the process of rank 1 and in reverse, waits for its response; meanwhile, the process 1 receives firstly the message from process 0 and replies to it: the timer will thus measure the whole round of sent and reception between the two processes. That's the reason why the measured time corresponds to a message exchange.

2.2 First measures

We measured several times the message exchange duration for message sizes of multiples of 32 Bytes. A part of results is shown below:

```
...
96 5.5e-05
96 5.6e-05
96 6e-05
96 0.000154
96 5.2e-05
128 0.000137
128 5.5e-05
128 5.3e-05
128 6.2e-05
128 0.000421
...
```

We added a barrier before the message exchange phase. This step is supposed to bring us a better accuracy cause the barrier guarantees the synchronization of all processes involved before they start sending or receiving messages. This eliminates any variability in the timing caused by differences in the execution speed of individual processes.

We rerun the same measures and compare the same part of measures, now we can observe a much better accuracy since for the same message sizes, the results keep a same order of magnitudes.

```
...
96 2.3e-05
96 1.3e-05
96 1.7e-05
96 2.3e-05
96 1.9e-05
128 2.3e-05
128 1.8e-05
128 2.1e-05
128 1.4e-05
128 2.1e-05
...
```

However, the previous changes may not be sufficient because there might be some difference between two sequential executions. To reduce the potential errors, we need to repeat the measures and calculate the mean value and/or the median.

2.3 Repetitions

We rewrite the program to print the average time of message exchange. In this procedure, we need to discount the first several measures of performance, because the warming-up effect may invoke some errors.

```
...
96 4.4e-06
128 2.35e-06
...
3168 2.97e-05
3200 2.9e-05
...
```

2.4 Impact of message size

For small message sizes, the dominant factor influencing communication performance is latency. Local communication benefits from low-latency communication within a single node. As the message size increases, the impact of bandwidth becomes more significant. Infiniband communication has a higher bandwidth compared to local communication interfaces. This fact leads to a reduction in the performance gap.

The choice of sending distinct sets of data separately or grouping them together depends on the communication mode. For a local communication, sending messages of small size is more efficient, in contrast, for an inter-node communication, it's better to group messages together.

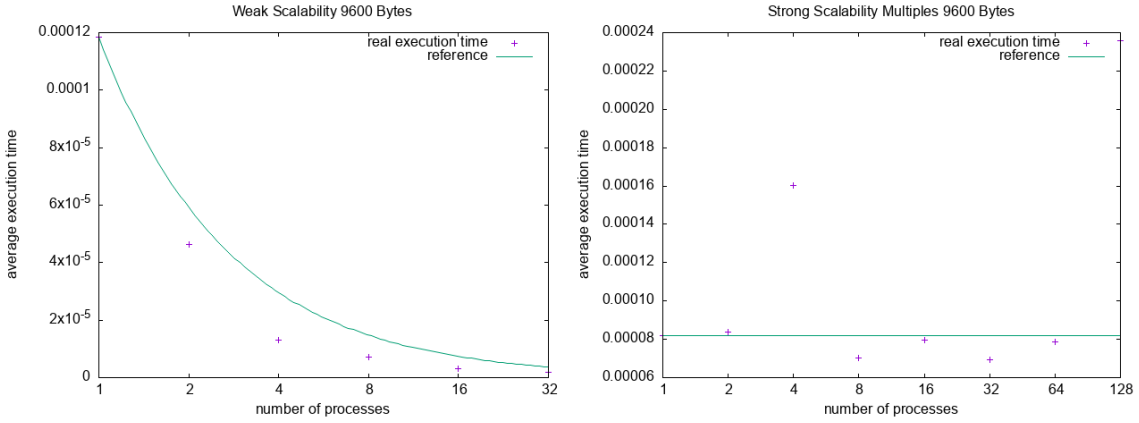
3 Experimental evaluation of scalability

3.1 Definitions

Both strong scalability and weak scalability are used to assess the efficiency of a parallel algorithm. Strong scalability refers to keeping the problem size constant and checking if the amount of work decreases as $\frac{1}{n}$ when increasing the number of computing units by n times. In contrast, weak scalability increases the problem size proportionally when increasing the number of processors and verifies if the amount of work per processor remains constant.

3.2 Evaluation results

Here represents the results of Scalability Evaluation of the finite elements method.



(a) Weak Scalability

(b) Strong Scalability

Figure 1: Scalability Evaluation

4 Conclusion

By investigating the impact of different communication modes and optimizing our ping-pong implementation, we were able to accurately measure message exchange duration and identify factors influencing performance. Furthermore, our analysis of scalability demonstrated the importance of considering both strong and weak scalability when assessing the efficiency of parallel algorithms.

5 Appendix

```c
// ping-pong
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <unistd.h>


// add barrier
int main(int argc, char **argv)
{
    int rank, size;
    char *buffer1, *buffer2;
    MPI_Status status;

    double t0, t1;

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

    if (argc != 2)
    {
        if (rank == 0)
            printf("Usage: %s <message_size>\n", argv[0]);
        MPI_Finalize();
        return 1;
    }

    size = atoi(argv[1]);
    buffer1 = (char *)malloc(size * sizeof(char));
    buffer2 = (char *)malloc(size * sizeof(char));

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0)
    {
        t0 = MPI_Wtime();
        MPI_Send(buffer1, size, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(buffer2, size, MPI_CHAR, 1, 10, MPI_COMM_WORLD, &status);
        t1 = MPI_Wtime();
        printf("%d\t%g\n", size, t1 - t0);
    }
    else if (rank == 1)
    {
        MPI_Recv(buffer1, size, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Send(buffer2, size, MPI_CHAR, 0, 10, MPI_COMM_WORLD);
    }

    free(buffer1);
    free(buffer2);

    MPI_Finalize();

    return 0;
}
```

```c
// FEM
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void fem_iter(int n, double *x_1, double *x_2)
{
    x_2[0] = (x_1[0] + x_1[1]) / 2;
    for (int i = 1; i < n - 1; i++)
    {
        x_2[i] = (x_1[i - 1] + 2 * x_1[i] + x_1[i + 1]) / 4;
    }
    x_2[n - 1] = (x_1[n - 2] + x_1[n - 1]) / 2;
}

int main(int argc, char **argv)
{
    int rank, p;
    int n = atoi(argv[1]);
    double *x_1, *x_2;
    double *local_x_1, *local_x_2;
    MPI_Status status;
    double start_fem, end_fem;
    double avg_time;

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

    int local_n = n / p;

    local_x_1 = (double *)malloc(local_n * sizeof(double));
    local_x_2 = (double *)malloc(local_n * sizeof(double));

    if (rank == 0)
    {
        x_1 = (double *)malloc(n * sizeof(double));
        x_2 = (double *)malloc(n * sizeof(double));
        // Initialize x_1 array
        for (int i = 0; i < n; i++)
        {
            x_1[i] = i;
        }
    }

    // Distribute x_1 array among processes
    MPI_Scatter(x_1, local_n, MPI_DOUBLE, local_x_1, local_n, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);

    for(int i = 0; i < 5; i++)
    {
        fem_iter(local_n, local_x_1, local_x_2);
    }

    // Perform FEM iteration and measure its execution time
    start_fem = MPI_Wtime();
    for (int i = 0; i < 20; i++)
    {
```

```
        fem_iter(local_n, local_x_1, local_x_2);
    }
    end_fem = MPI_Wtime();
    avg_time = (end_fem - start_fem) / 20.0;

    // Exchange boundary values between neighboring processes
    if (rank != 0)
    {
        MPI_Send(&local_x_2[0], 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&local_x_2[-1], 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &status
    );
    }
    if (rank != p - 1)
    {
        MPI_Send(&local_x_2[local_n - 1], 1, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD
    );
        MPI_Recv(&local_x_2[local_n], 1, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD, &
    status);
    }

    MPI_Gather(local_x_2, local_n, MPI_DOUBLE, x_2, local_n, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);

    if (rank == 0)
    {
        printf("%d\t%g\n", p, avg_time);
    }

    free(local_x_1);
    free(local_x_2);
    if (rank == 0)
    {
        free(x_1);
        free(x_2);
    }
    MPI_Finalize();
    return 0;
}
```