MSc in High Performance Computing Coursework for MPP

Coursework Assignment

Exam Number: B129230

1. Introduction

Message-passing programming is mainly based on MPI to implement parallel programs. MPI is a library of function or subroutines calls which can specify parallelism in C and Fortran. The purpose of this report is to describe the deign and implementation of two-dimensional domain decomposition and non-blocking communications for parallel image processing.

2. Experiments

2.1 Experimental Setup

The whole program which contains I/O, communication, computing several parts executes on the ‘Cirrus’ back end by using PBS(portable batch system) to submit.

2.1.1 Computing Environment.

‘Cirrus’ is a SGI ICE XA system with 280 compute nodes 10, 080 cores, utilizing Infiniband interconnect. Each node provides 36 physical cores which use Intel(R) Xeon(R) E5-2695 v4, clocked at 2.1GHZ, and contains 256GB RAM per node.

This program is compiled with mpicc provided by ‘HPE MPT (Message Passing

Toolkit) 2.16'. The optimization level is set to -O3.

2.1.2 Experimental Steps

This experiment selects 2 nodes 72 cores as computing environment to count average time of computing and I/O separately to draw graphs from 0 core to 64 cores.

2.1.3 PBS(portable batch system).

This experiment uses PBS script to submit jobs, control resources, required module and number of processes. The following code is an important part of PBS script:

```
# Select 2 full node 72 cores
#PBS -l select=2:ncpus=36
# Exclusive Node Access and scatter to the full set of assigned compute cores
#PBS -l place=scatter:excl
# using 64 MPI processes with 36 MPI processes on each compute node
mpiexec_mpt -n 64 -ppn 32 ./image
```

2.2 Parallel Processing

2.2.1 Preparation

The whole parallel processing is divided into four sections: preparation, input, computing and output.

During preparation, the program initializes MPI library at the first and then declares various variables such as two-dimensional array by using arralloc. In addition, the author also creates Cartesian topology in current section in order to adapt to different number of cores division by using MPI_Dims_create().

```
/*Cartesian topology */
ndims = 2;
dims[0] = 0;
dims[1] = 0;           //in order to set by MPI_Dims_create() automatically;
period[0] = 1;         // top and down are periodic.
period[1] = 0;
reorder = 0;
disp = 1;              //shift by 1
MPI_Dims_create(size, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, period, reorder, &comm2d);
MPI_Comm_rank(comm2d, &rank);
MPI_Cart_shift(comm2d, 1, disp, &left, &right);
MPI_Cart_shift(comm2d, 0, disp, &down, &top);
```

2.2.2 Input

In the input section, there is a helper function ‘pgmread’ which is included in pgmio.c to read the whole image file to master buffer. Then using MPI_Bcast() broadcasts the whole image to each MPI process. After broadcast, each MPI process copies each small image, which is part of the whole image, to its own buffer according to corresponding position. The corresponding positions are decided by rank number and topology division. The difficulty here is to design an algorithm to copy small images of the full images to its own buffer:

```
/* store the image from edge[i][j] to small part edge2[i][j] */
int row = rank / dims[1];
int column = rank % dims[1];
int MP = M / dims[1];
int NP = N / dims[0];
Loop over i = MP*column + 1, MP*(column + 1) + 1 ;
          j = NP*row + 1, NP*(row +1) + 1;
        edge2[i - MP* column][j - NP*row] = edge[i][j];
end loop;
```

Till now, each process should be able to handle its own small image, such as assignment of small edge buffer and small old image buffer. In addition, it is necessary to set fixed boundary conditions on the left and right sides. It is noteworthy that initialization of left and right processes are different:

```
/* Set fixed boundary conditions on the left and right sides */
loop over j = NP*row + 1, NP*(row + 1) +1;
        val = boundaryval(j, N);                        // compute sawtooth value
if (left == MPI_PROC_NULL)
       old2[0][j - NP*row] = (int)(255.0*(1.0 - val));
if( right == MPI_PROC_NULL)
       old2[1 + MP][j - NP*row] = (int)(255.0*val);
end loop;
```

2.2.3 Computing

During the computing section, it is the best way to create derived data to send rows because of discontinuous memory:

```
/* design a new datatype to send the uncontinuous row*/
MPI_Type_vector(MP, 1, NP + 2, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
```

After creating derived data successfully, using non-blocking communication to swap halo. In order to get better performance and take advantage of non-blocking

communication, the author designs to compute the inner image during non-blocking communication. After computing the inner image and finishing halo swap, the program starts to compute the small image boundaries. In addition, the program also computes the average pixel value at regular intervals and quit iterations when global delta is smaller than 0.1 during the computing section.

2.2.4 Output

When it comes to the output section, the author creates another derived data to send the small images from other ranks to rank 0. Rank 0 receives each small image from other MPI processes to make up the full image. The difficulty here is how to design an algorithm to let all small images to make up a full image according to specified rank numbers.

```
/* receive each small image from different processes to make up the full image */
for ( k = 1; k < size; k++)
{
    krow = k / dims[1];
    kcolumn = k % dims[1];
    MPI_Irecv(&localold1[0][0], 1, newtype2, MPI_ANY_SOURCE, k, comm2d, &request6);
    MPI_Wait(&request6, &status6);
    loop over i = MP*kcolumn +1, MP*(column + 1) + 1
              j = NP*krow + 1, NP*(krow +1 ) + 1
             buf[i - 1][j - 1] = localold1[i - MP*kcolumn][j - NP*krow];
    end loop;
}
```

2.3 Result and Performance Analysis

2.3.1 Output Images and Result tests

Running the program with different input images will get the following images:
Imagenew256x192:

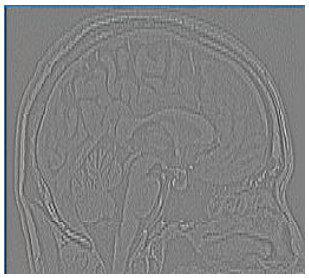

Image1: 1 iteration

Image2: 869 iterations

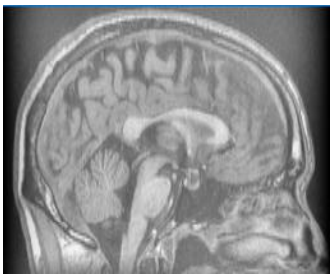

Image3: 1500 iterations

Average pixel of Image 1, 2 and 3 are 254.504, 227.902 and 220.582 respectively. 869 iterations stop when delta is smaller than 0.1 and check frequency is 30.

Imagenew512x384:

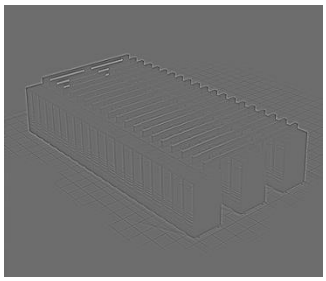

Image4: 1 iteration

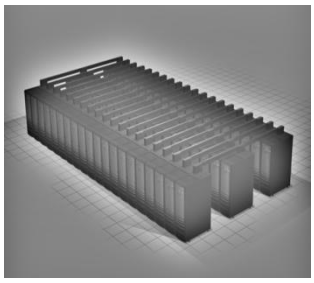

Image5: 1439 iterations

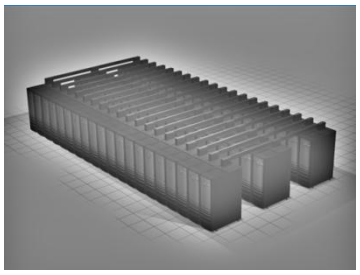

Image6: 1500 iterations

Average pixel of Image 4, 5 and 6 are 254.986, 252.849 and 252.778 respectively. 1439 iterations stop when delta is smaller than 0.1 and check frequency is 30.

Imagenew768x768:

Image7: 1 iteration

Image8: 1229 iterations

Image9: 1500 iterations

Average pixel of Image 7, 8 and 9 are 254.944, 250.568 and 250.074 respectively. 1229 iterations stop when delta is smaller than 0.1 and check frequency is 30.

In addition, no matter running on 1 process or multiple processes, it will have the same image and global average pixels as long as running on the same number of iterations, which means the image are the same.

2.3.2 Performance Analysis

The following performance analysis is all on 30 frequency situations to check global pixels.

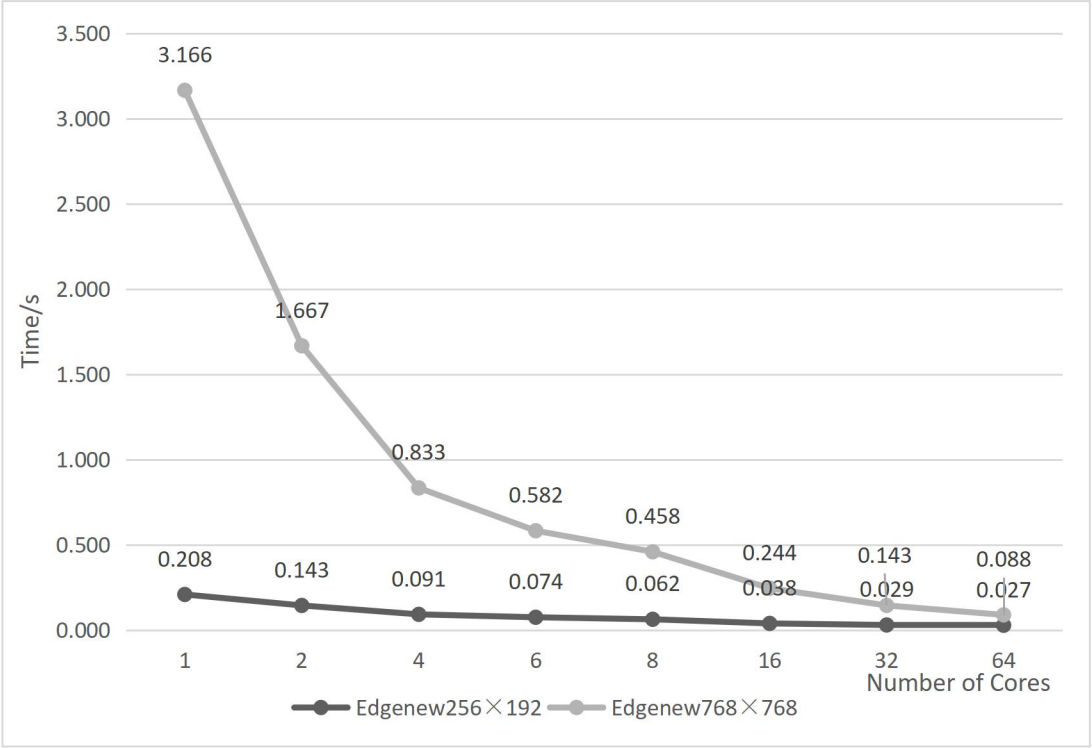

Figure 1: Computing Time of edgenew256x192 and edgenew768x768

According to figure 1, edgenew768x768 have a very noticeable change on computing time with the increase of number of cores.

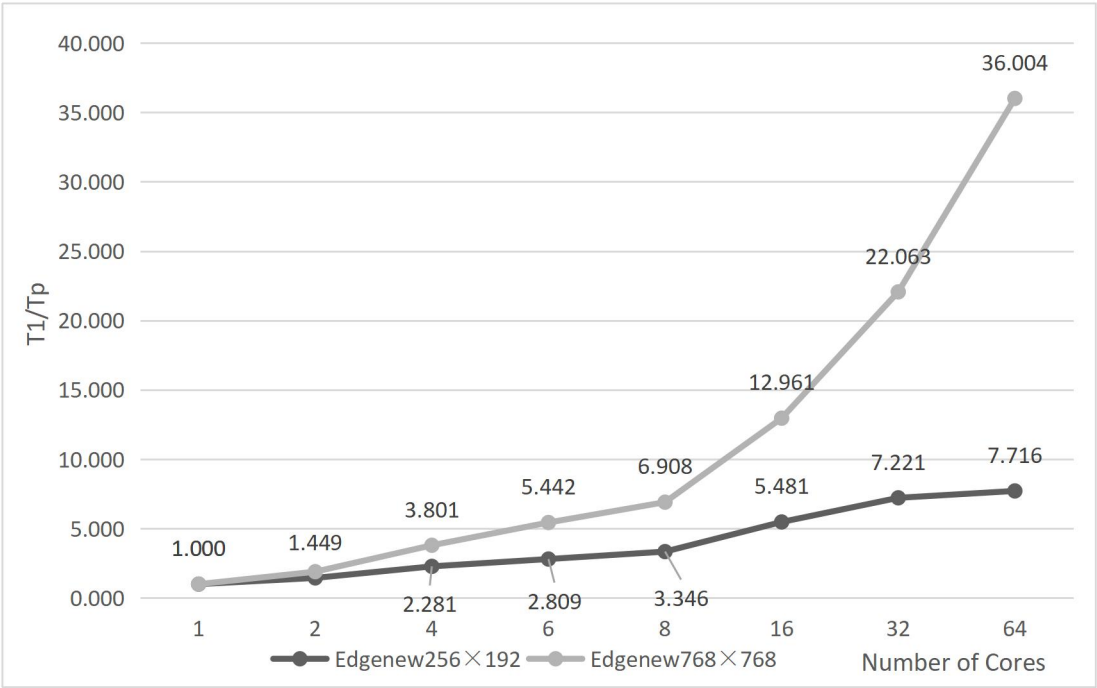

Figure 2: Speedup of edgenew256x192 and edgenew768x768

According to figure 2, with the increase of number of cores, speedup of edgenew768x768 has the huge improvement compared with edgenew256x192. Because the speedup of edgenew256x192 just improve around 7.7 times but edgenew768x768 improves around 36 times when number of cores are 64.

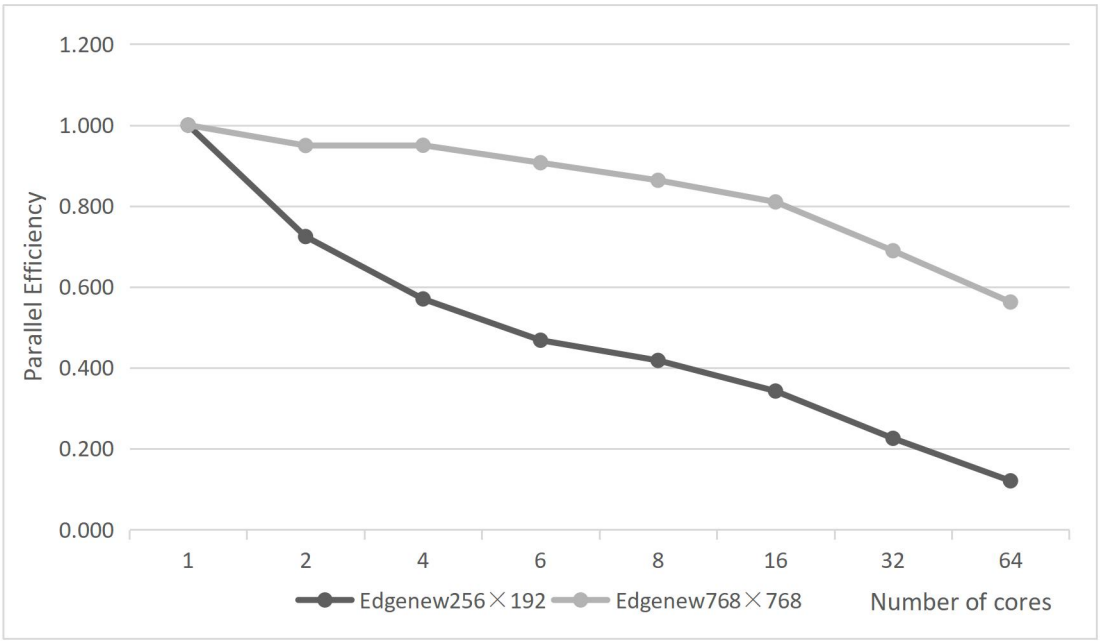

Figure 3: Parallel efficiency of edgenew256x192 and edgenew768x768

As the Figure 3 shown, parallel efficiency of edgenew256x192 reduces very quickly compared with big edgenew768x768.

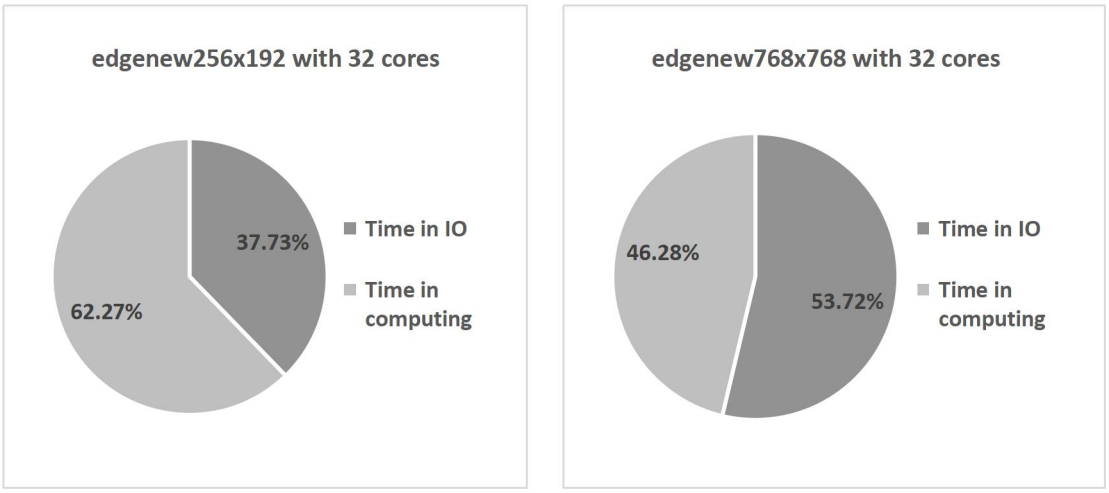

Figure 4: Time in IO and Computing of edgenew256x192 and edge768x768 with 32 cores

According to Figure 4, with the increase of image size, computing time is becoming smaller from 62.27% to 46.28% but IO time becomes bigger from 37.73% to 53.72%.

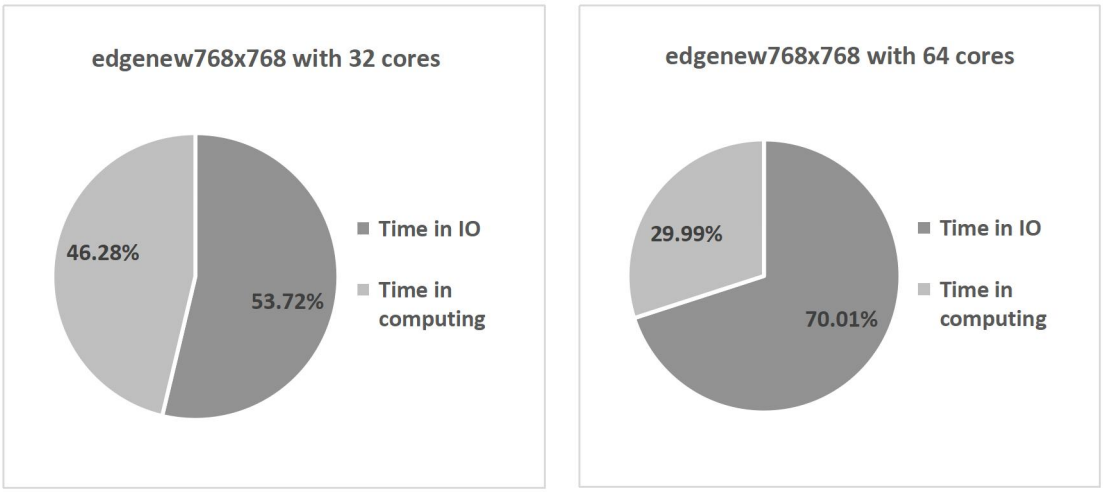

Figure 5: Time in IO and Computing of edge768x768 with 64 cores

According to Figure 5, with the increase of number of cores from 32 to 64, IO nearly occupies from 53.72% of the whole time to 70.01% , which means the more cores, the more IO proportion.

Lastly, the author also analyses the different check frequencies from 30 to 100. According to comparison, it does not have a very big influence on IO time and computing time. Because with the increase of check frequency, it means the computing will execute more iterations to meet the requirement that delta is smaller than 0.1. More iterations offset the reduce time of MPI_Allreduce() communication time.

3. Conclusion

According to these experiments, code and data analysis, the program can run on any cores without problems as long as the size of image can be divided exactly by cores.

On the other hand, this program has a better parallel efficiency when handling the big image, although IO time will become a bigger overhead with the increase of image size and number of cores. Hence, the future optimization is to reduce the IO overheads in order to get maximal performance.