Projet de Fin d'Études

Tutor : Stéphane VIALLE

Silin WANG (SIR)

25/03/2015

CentraleSupélec

Contents

Introduction

This project can be divided into two parts. In the first part, we parallelize a simple program by using library *OpenMP*, then we measure the performance of this parallel program on *Xeon-phi* in different configurations such like "*regular for*", "*irregular for*", "*schedule*" and "*offload*". In the second part, I realize an algorithm named *Artificial Neural Network* in language C++ which can recognize the different gestures from the database. Then I tried to parallelize this program by using *OpenMP* and observe its performance on *Xeon-phi*.

Background knowledge

Xeon-phi

Intel Many Integrated Core Architecture or *Intel MIC* is a coprocessor computer architecture developed by Intel incorporating, and Xeon-phi is the brand of this computer. In this project, the machine we use contains two parts. One is named *phi-host* which has 6 physical cores, the other one is named *mic* which has 57 physical cores.

OpenMP

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on most processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, Mac OS X, and Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

In our project, there is a clause named *Schedule* that we use in most of our program. It is used to control the manner in which loop iterations are distributed over the threads, which can have a major impact on the performance of a program. The syntax is *schedule(kind, chunk_size)*.

There are two different kind of *schedule* we use in our program. The first one is *static*. In this kind, iterations are divided into chunks of size *chunk_size*. The chunks are assigned to the threads statically in a round-robin manner, in the order of the thread number. The last chunk to be assigned may have a smaller number of iterations.

The second one is *dynamic*. In this kind, the iterations are assigned to threads as the threads request them. The thread executes the chunk of iterations, then requests another chunk until there are no more chunks to work on.

PART I

In this part, we measure the performance of our parallelize program on *Xeon-phi* by using different configuration suck like "*regular for*", "*irregular for*", "*schedule*" and "*offload*".

Regular for

In this section, we use a "*regular for*" program to execute on machine mic. "*Regular for*" means the iteration of the for loop is regular. We observe the performance of this program by using different number of threads from 1 to 240. Then we obtain the curve shown below.

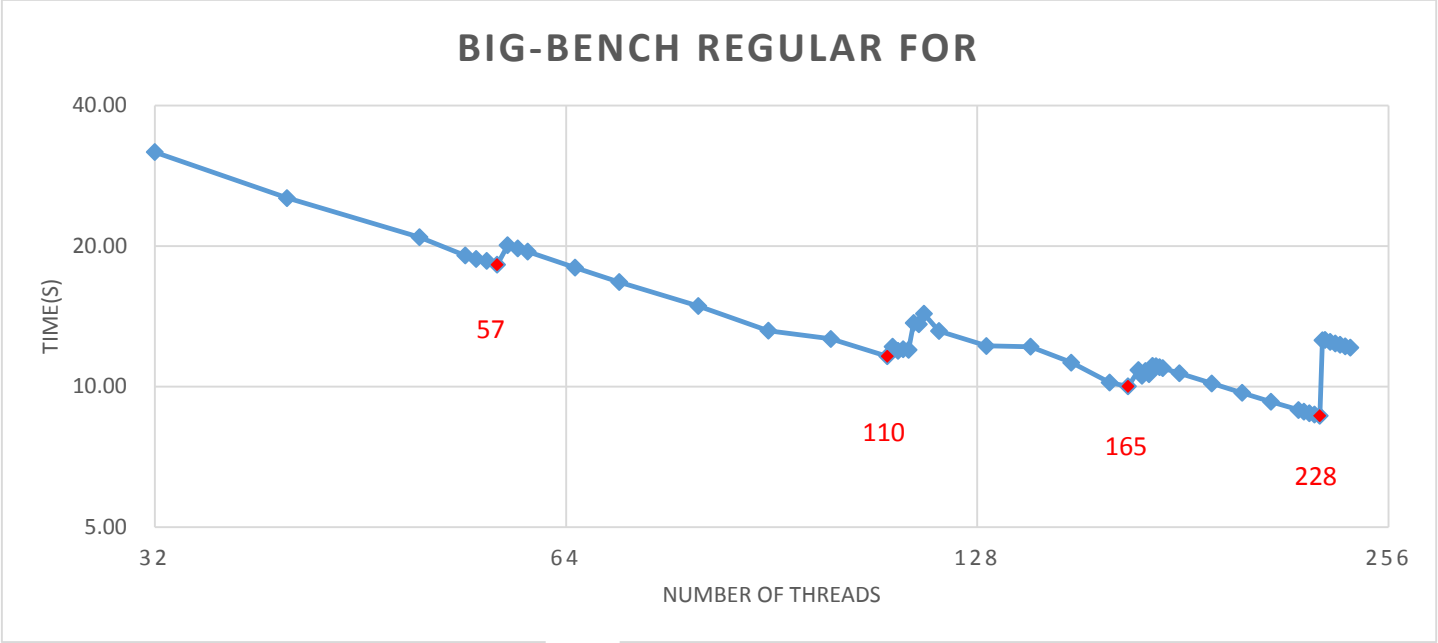

Figure 1 Curve of "*regular for*"

As we can see in figure 1, there are 4 local minimums in this curve (the points in red) when the number of threads is 57, 110, 165 and 228. We can find that those 4 number are equal or around the multiples of 57. Among them, 57 and 228 are exactly equal to 57*1 and 57*4. And 110 and 165 are around 57*2 and 57*3.

As we know, the number of physical cores of mic is 57. So we can infer that the performance of the points with the number of threads equal or around the multiples of the number of physical cores are better than other points.

The best performance is when number of threads equals to 228 (57*4), the execution time is 8.65 seconds.

Irregular for

In this section, we use an "ir*regular for*" program to execute on machine mic. "*irregular for*" means the iteration of the for loop is irregular. In this case, we add the clause *schedule* in our program to control the manner in which loop iterations are distributed over the threads. We use different kinds of *schedule*, then we observe their performance.

Without schedule and static default

Firstly, we measure the performance of our program without the clause *schedule* and *static default*. *Static default* means that the iteration space is divided into chunks that are approximately equal in size.

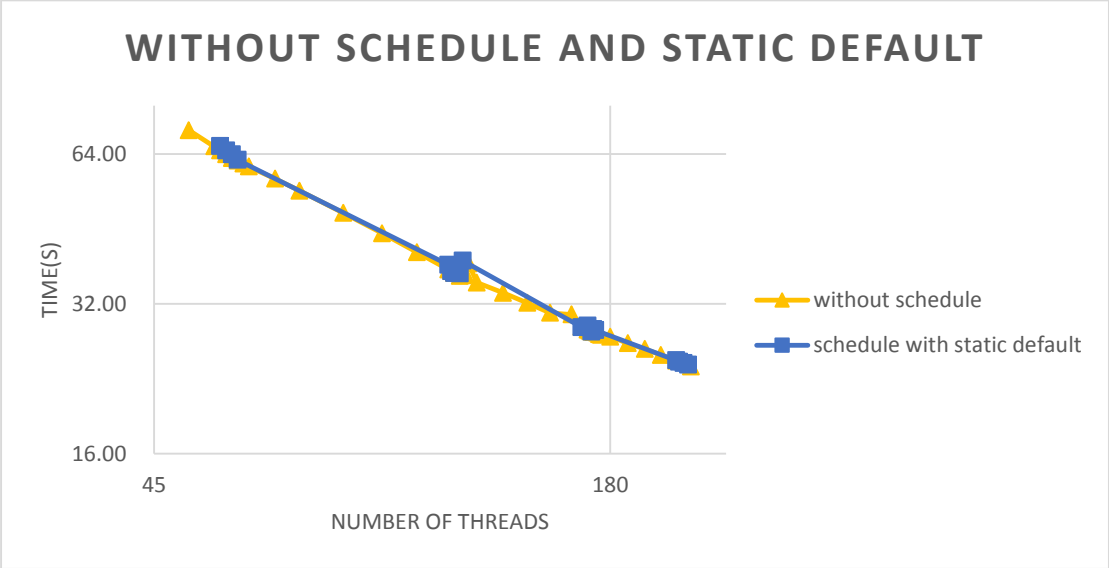

Figure 2 Curve of irregular for without schedule and static default

As we can see in figure 2, the performance of this two different configurations are almost the same. They both improve linearly when the number of threads increases.

With schedule (57 threads)

In this part, we parallelize our program with the *static schedule* and *dynamic schedule* when the number of threads is 57. Then we measure and compare the performance of them. The reason why we chose 57 as the number of threads is that the number of physical cores of mic is 57. The performance are shown below.

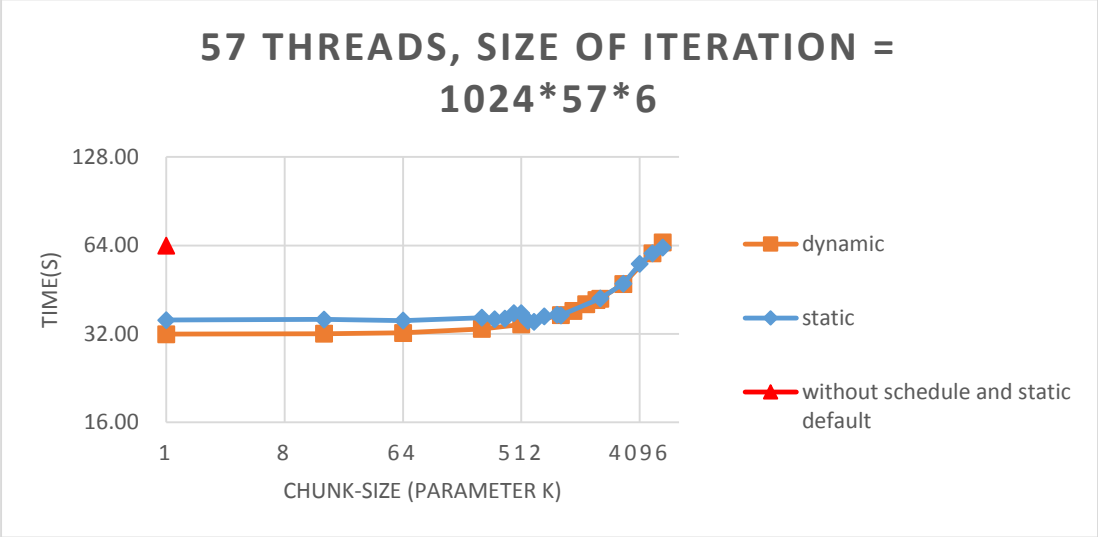

Figure 3 Curve of irregular for with static and dynamic schedule when number of threads is 57

As we can see in figure 3, the performance of *static schedule* and *dynamic schedule* are almost the same, they both reduce when the number of *chunk_size* increases. When the *chunk_size* is less than 512, the performance of *dynamic schedule* is a little better than *static schedule*.

Compared to the performance we measure in last part (without *schedule* and *static default*), we can find that the performance of the program with *static* and *dynamic schedule* when number of threads is 57 are better.

The best performance for both *static schedule* and *dynamic schedule* are when chunk_size equals to 1, the execution time is 31.9 and 35.62 seconds.

With schedule (228 threads)

In this part, we parallelize our program with the *static schedule* and *dynamic schedule* when the number of threads is 228. Then we measure and compare the performance of them. Because 228 is the multiple of 57 (57*4), so we chose 228 as the number of threads.

As we can see in the figure 4 shown below, the performance of static and dynamic schedule both reduce when the number of the chunk_size increases. When the chunk_size is less than 64, their performance are almost the same. But after this point, the performance of static schedule is better than dynamic schedule.

Compared to the performance we measure in last part (without *schedule* and *static default*), we can find that the performance of the program with *static* and *dynamic schedule* when number of threads is 57 are better.

The best performance for both *static schedule* and *dynamic schedule* are when

chunk_size equals to 1, the execution time are both 16.75. And this result is better than the best performance with the number of threads equal to 57.

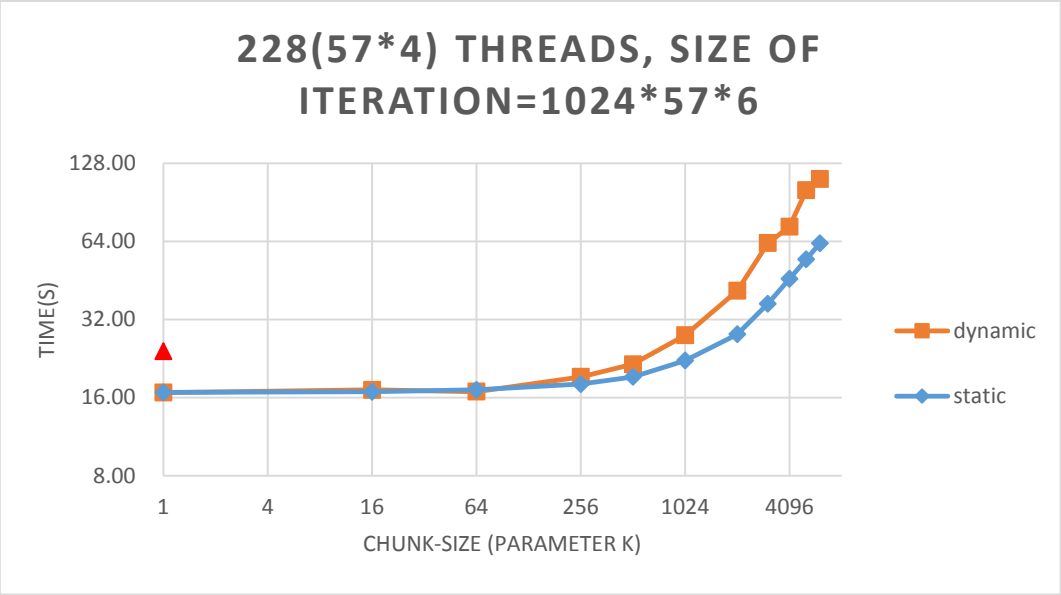

Figure 4 Curve of irregular for with static and dynamic schedule when number of threads is 228

Offload

In this part, we use the method *offload* to test the performance of our parallelize program. Before this part, when we want to compare the performance of our program on *phi-host* and *mic*. We have to run the program on *phi-host* and *mic* separately. That means we need to execute one time on *phi-host*, then on *mic*. But when we use offload, we just need to run the program once for obtaining both the performance on *phi-host* and *mic*.

We measure the performance with *static* and *dynamic schedule* both on *phi-host* and *mic*.

On phi-host:

On phi-host, we measure the performance of *static* and *dynamic schedule* with the number of threads equal to 6 and 24.

As we can see in the figure 5 and 6 below, the performance of *static and dynamic schedule* on *phi-host* are nearly same. The change of *chunk_size* does not have an obvious influence on the performance. When the *chunk_size* is less than 512, the performance are barely change. After this point, the performance improve a little bit.

For both *static* and *dynamic schedule*, the performance when the number of threads

equal to 24 are better than the number of threads equal to 6.

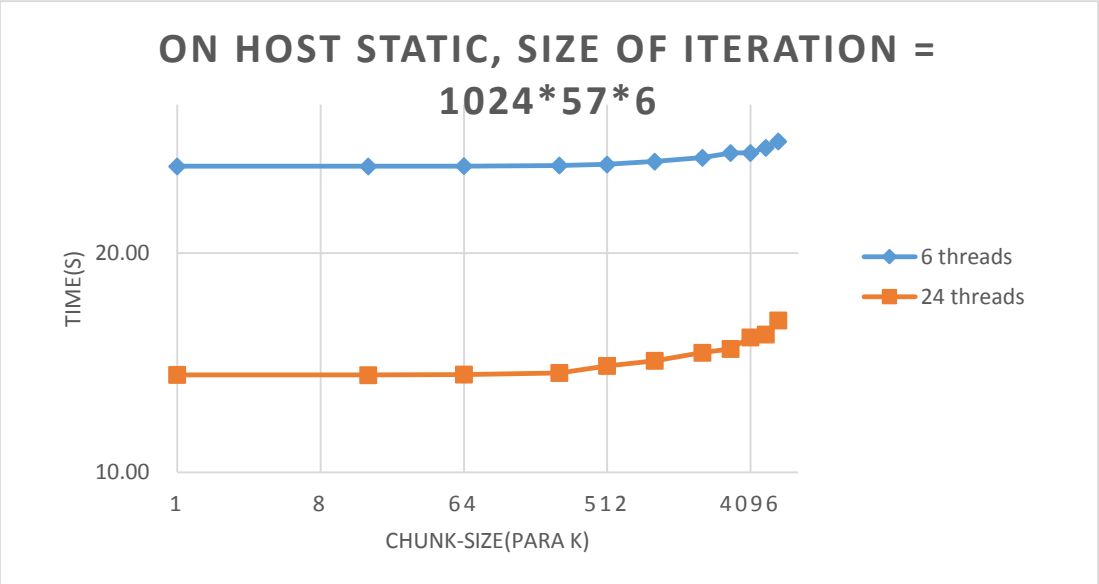

Figure 5 Curve of offload on host with static schedule when number of threads equal to 6 and 24

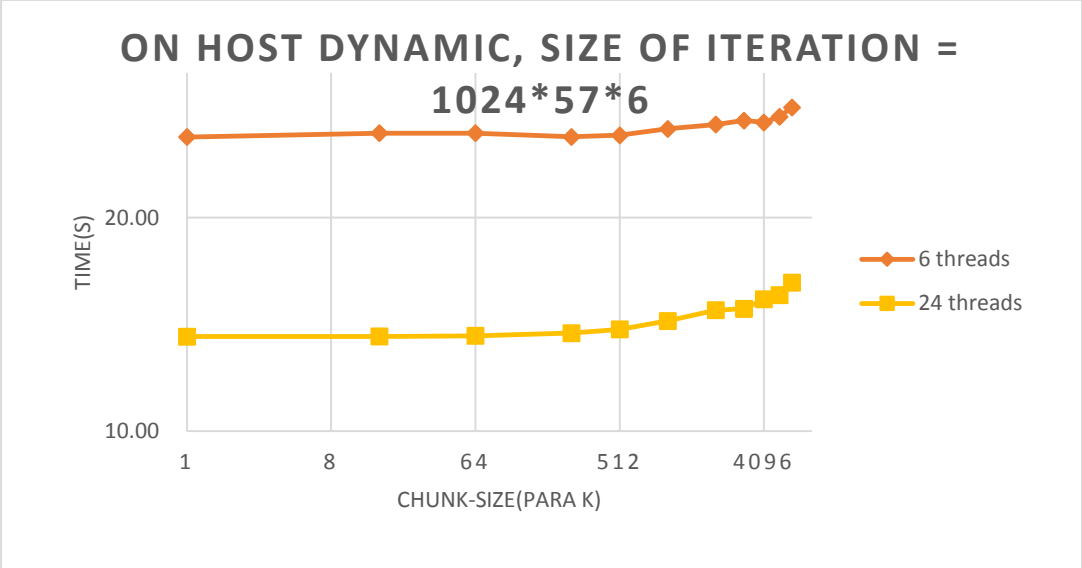

Figure 6 Curve of offload on host with dynamic schedule when number of threads equal to 6 and 24

On mic:

On mic, we measure the performance of *static* and *dynamic schedule* with the number of threads equal to 228.

We can observe in figure 7 that the performance of *static schedule* on mic reminds unchanged when the *chunk_size* is less than 2048. After that, the performance improves firstly and obtains the best performance at the point 4096. The execution time is 700.45 seconds. Then it becomes worse.

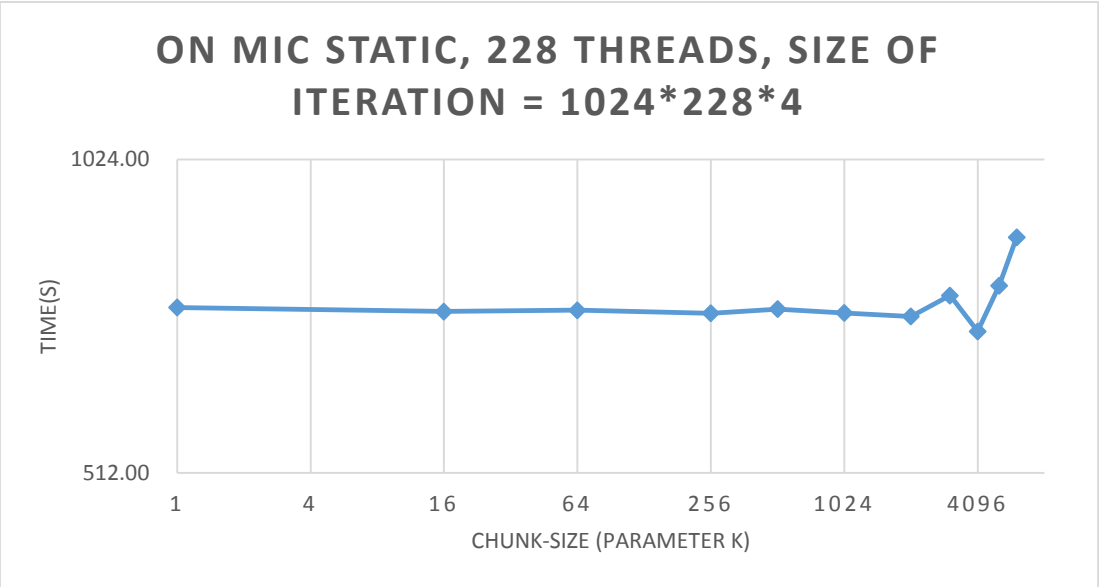

Figure 7 Curve of offload on mic with static schedule when number of threads equal to 228

As we can see in the figure 8, the curve of *dynamic schedule* is strange. The execution time is so big when the *chunk_size* is small. The performance improves so much when the *chunk_size* is from 1 to 8. Between 8 and 256, it reminds unchanged. After 256, then it reduce rapid. The best performance is when *chunk_size* equals to 8, the execution time is 501.40 seconds. It is better than the best performance of *static schedule* on mic.

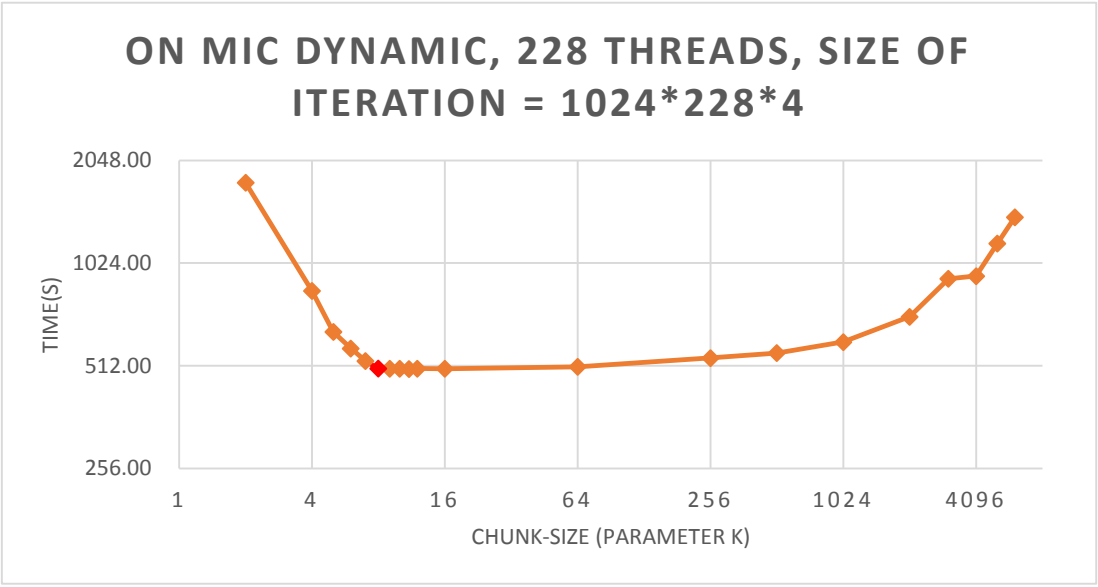

Figure 8 Curve of offload on mic with dynamic schedule when number of threads equal to 228

PART Ⅱ

In this part, I realize an *Artificial Neural Network* which can recognize two different gestures. Then I execute this program on *Phi-host* and make sure it has a good recognition accuracy. After that, I parallelize this program by using *OpenMP* and observe the performance of it.

Artificial Neural Network

The algorithm I used is a classic Artificial Neural Network like the picture shown below. It consists of three part, Input layer, hidden layer and output layer.

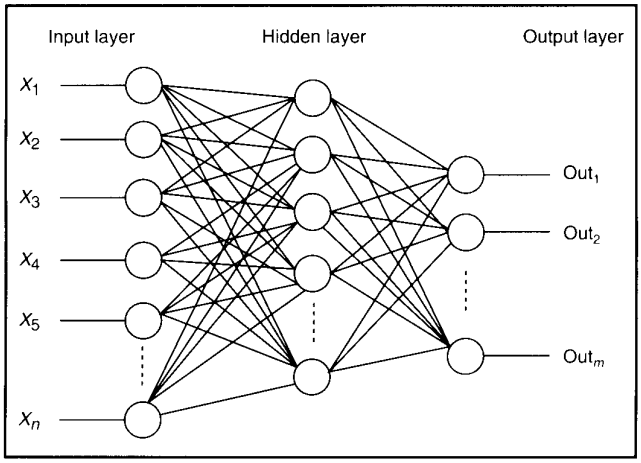

Figure 9 Artificial Neural Network

In each layer, there are some neurons. For each neuron, there are several inputs and an output. Each input has a unique weight.

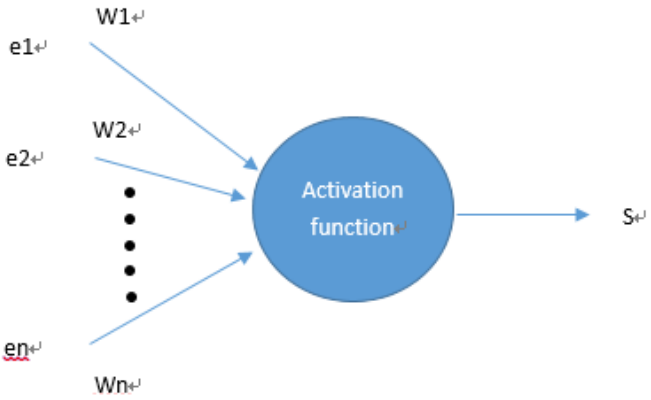

Figure 10 mathematical modeling of the neuron

Like the modeling shown in figure 10, the output of each neuron can be calculated

according to the expression below:

$$s = f\left(\sum_{i=1}^{n} w_i e_i\right)$$

Equation 1 Output of a neuron

In this expression, s, w_i and e_i represent output, each weights and input of a neuron. And f is the function of activation. In my case, I use sigmoid function as the activation function.

$$S(t) = \frac{1}{1 + e^{-t}}$$

Equation 2 Sigmoid function

Programming

I use language C++ to realize this algorithm. Firstly, I create a *Neuron Network* class, it contains layers, neurons, weights, inputs and outputs. After that, I create the important methods for the network such like *Propagation()*, *BackPropagation()*, *CalError()*, *PutWeights()* and *GetWeights()*.

Propagation()

This method is to calculate the outputs of the *Neuron Network*. When input layer receive input vectors, it calculate the outputs of each neuron of each layer until the output layer. Then it accumulate the outputs of the neurons of the penultimate layer, and obtain output vectors.

CalError()

This method is to calculate the error of the weights of each neuron. The weights of each neuron is initialized randomly. So in purpose to obtain the outputs that we desire, we need to correct these weighs.

BackPropagation()

This method is to calculate the new value of each weight based on the errors calculated by *CalError()*.

PutWeights()

This method is to update the value of each weight in the whole *Neural Network*.

GetWeights()

This method is to obtain the actual value of each weight in the whole *Neural Network.*

Training and testing

To recognize the two different gestures shown in the picture below, we need to train the *Neural Network* we build above.

Figure 11 Two gestures

For each gesture, I have taken 200 photos. This means there are 200 vectors to represent a gesture, and between each vector there are slight differences. Because I have 2 different gestures, so in all I have 400 vectors to form our input dataset.

Taking into account that we have training and testing 2 parts, I need to distribute these vectors. In my case, I use 100 vectors of gesture 1 plus 100 vectors of gesture 2 for training, and use the other 200 vectors for testing.

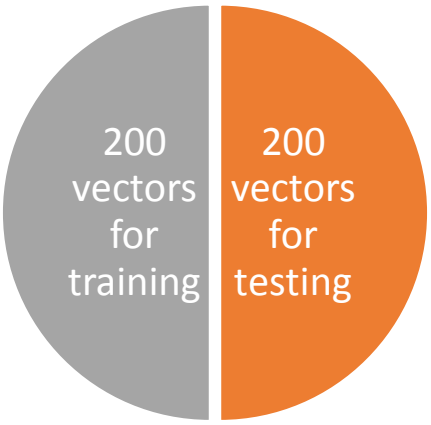

Figure 12 Distribution of input vectors

In the training part, I create a loop with 60000 iteration times to train the 200 input vectors. In each iteration, it randomly picks a vector among these 200 vectors and trains the *Neural Network* with this vector. Statistically, every vector will be pick 300 times to train the network. The process of each iteration is shown below.

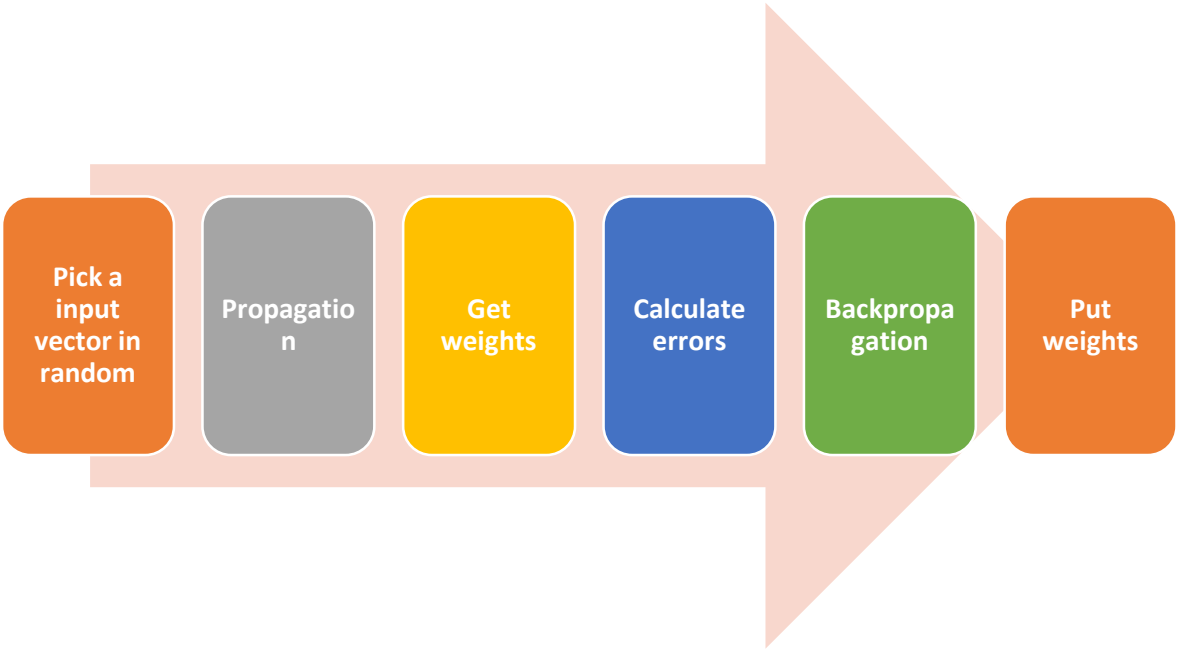

Figure 13 Process of each iteration

Parallelization

In this part, I use *OpenMP* to parallelize my program. Because of the structure of the classic *Artificial Neural Network*, I can just parallelize the program by neurons. There are 10 neurons in one layer. That means I can just parallelize the program with small number of threads.

In my case, I parallelize the part of *Propagation*. Firstly, I create a small region *#pragma*

omp parallel, but the result comes out that this parallelization cannot improve the performance of the program. After taking the advice of my teacher Mr.Vialle, I create a larger *#pragma omp parallel* region. This time, the performance are increased by the parallelization.

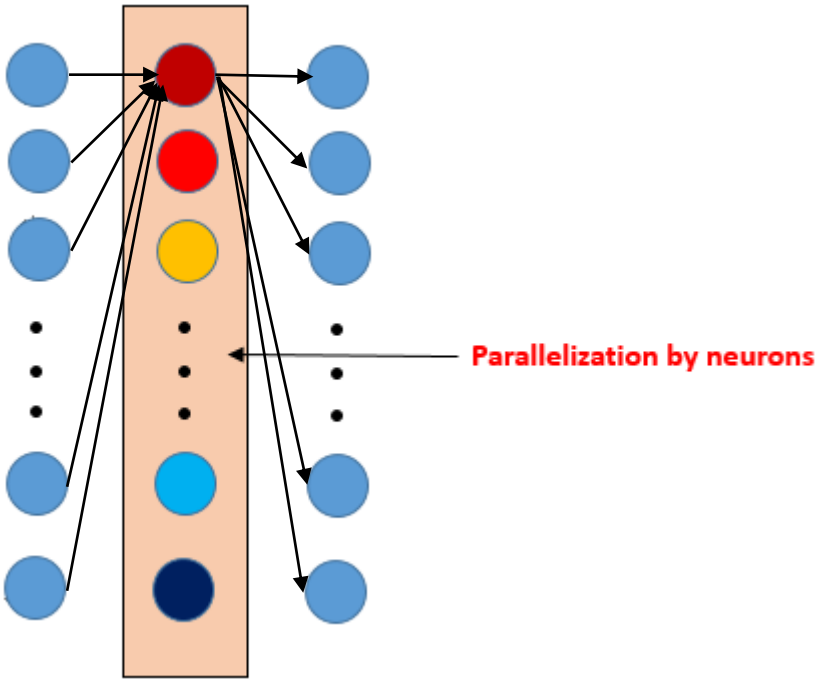

Figure 14 Parallelization

Result

Here is the result of the parallelize program on *Phi-host*, I measure the performance with the number of threads from 1 to 10. The curve shows that the parallelization actually increases the performance of the program, but the improvement is not so obvious. The best performance is 4.52 seconds when the number of threads is 2.

We find that the improvement curve is a little strange, and we want to know the reasons for that. So I measured the execution time of the different parts in the parallel region (*#pragma omp parallel*). The result is shown in the figure 16.

I have created three part in the parallel region, two *#pragma omp single* and one *#pragma omp for*. We can find that the performance of the *omp_for* part improves when the number of threads increases. But the performance of the two *omp_single* part reduce when the number of threads augments. And the *omp_single* part 2 is worse than the *omp_single* part 1.

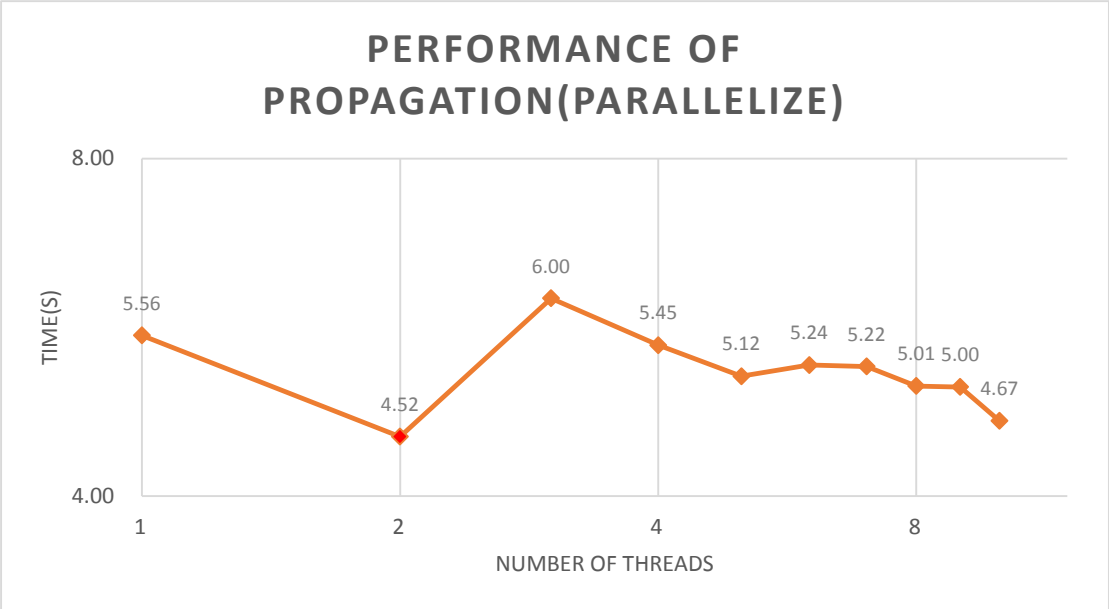

Figure 15 Performance of the parallelize program

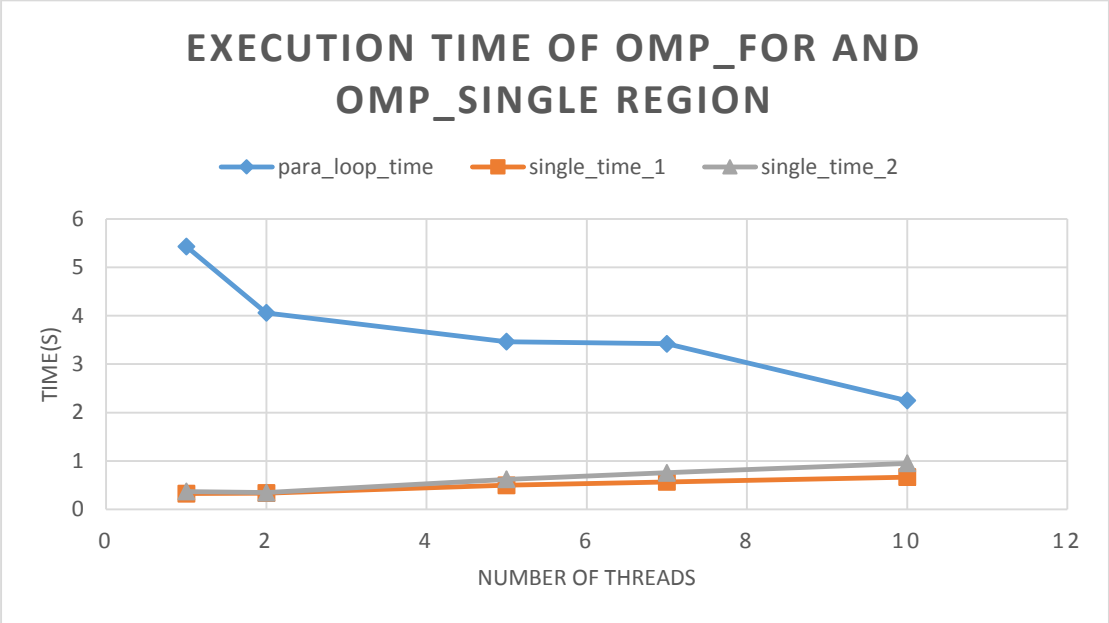

Figure 16 Execution time of omp_for and omp_single region

Then I measure the percentage of the execution time of propagation and backpropagation in the global execution time. We can see in the figure 17 below, the percentage is about 55%. That means that the propagation is the biggest time-consuming part in the whole training process. And it's much bigger than the percentage of the backpropagation which is about 20%.

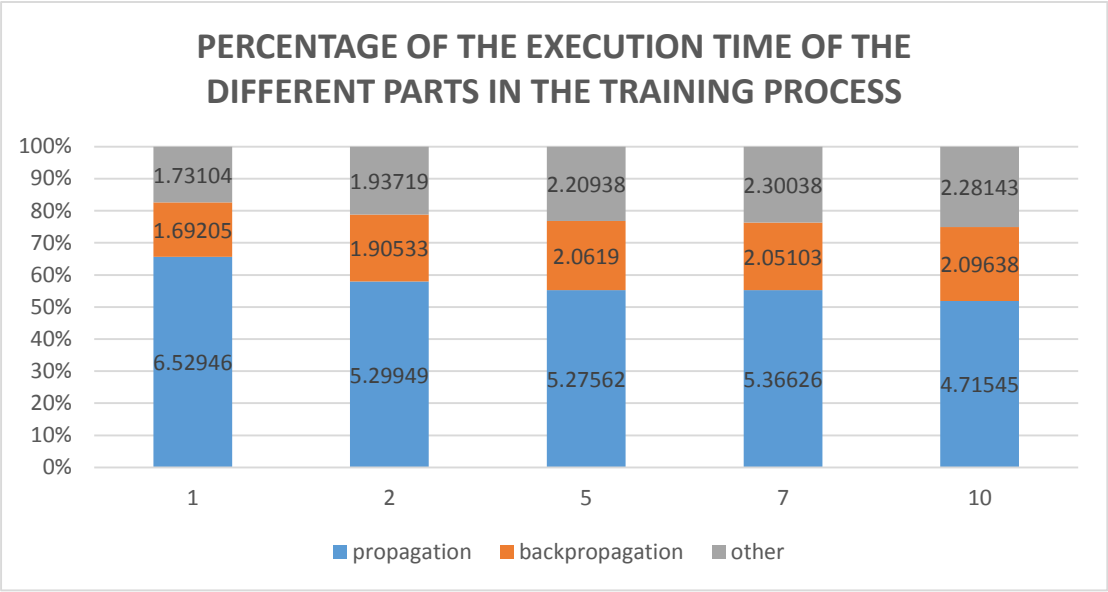

Figure 17 Percentage of the execution time of the different parts in the training process

Conclusion

Through this project, I learn how to use library OpenMP to parallelize a program. Especially in the second part, I have spent much time to parallelize the *Neural Network* and improve my parallelization to really increase the performance of the parallelize program. It's an important learning experience for me.

Annex

Regular for

```
Omp_set_num_threads(NbThreads);
for(int step = 0; step < thestep; step++)
{
    #pragma omp parallel
    {
        #pragma omp for
        for(size_t i=0;i<thesize;i++)
            tab[i]=i;
        #pragma omp for
        for(size_t i=0;i<thesize;i++)
            tab[i]=pow(pow(0.99,tab[i]),pow(0.99,tab[i]));
    }
```

```
}
```

Irregular for

```
Omp_set_num_threads(NbThreads);
for(int step=0;step<thestep;step++)
{
    #pragma omp parallel
    {
       #pragma omp for
       for(size_t i=0;i<thesize;i++)
          double res=0;
          for(int j=0;j<i+1;j++)
             res+=pow(tab[j],i);
          tab[i]=res;
    }
}
```

Offload

```
#pragma offload_transfer target(mic:0) in(mytab:length(SIZE) free_if(0) align(ALIGN)) nocopy(myres :length(SIZE) alloc_if(0) free_if(0))
smct=emtt=omp_get_wtime();
dmtt+=emtt-smtt;
#pragma offload target(mic:0) nocopy(mytab : alloc_if(0) free_if(0)) nocopy(myres : alloc_if(0) free_if(0))
 {
    compute(mytab,SIZE,myres,mnt);
 }
 smtt=emct=omp_get_wtime();
 dmct+=emct-smct;
 #pragma offload_transfer target(mic:0) nocopy(mytab : alloc_if(0) free_if(1)) out(myres : length(SIZE) alloc_if(0) free_if(1))
 emtt = omp_get_wtime();
 dmtt = emtt - smtt;
```

Neural Network

```
class NeuralNet
{
```

```
   private:
            int    NumInputs;
            int    NumOutputs;
            int    NumHiddenLayers;
            int    NeuronsPerHiddenLyr;
            //storage for each layer of neurons including the output layer
            vector<NeuronLayer> vecLayers;
   public:
          NeuralNet();
          void CreateNet();
          //gets the weights from the NN
          vector<vector<vector<double> > > GetWeights()const;
          //gets the weight
          vector<vector<double> > GetLayerWeights(int layer_num) const;
          //replaces the weights with new ones
          void PutWeights(const vector<vector<vector<double> > > &weights);
          //calculates the outputs from a set of inputs
          vector<vector<double>> Propagation(vector<double> inputs, int
num_threads);
          //sigmoid response curve
          inline double Sigmoid(double activation, double response);
          //calculate the error (desired output value - real output value) for each
neuron
         vector<vector<double>> CalError(const vector<double>
         &outputOfLastLayer, const vector<double> &targets,
         const vector<vector<vector<double> > > &weights);

          //calculate the new value of weights after training
         vector<vector<vector<double> > > BackPropagation(const
         vector<vector<vector<double> > > &weights,
         const vector<vector<double> > &outputs,
         const double eta, const vector<double> &firstLayerInputs,
         const vector<vector<double> > &errors,
         int num_threads);
};
```

Parallelization

```
vector<vector<double> > NeuralNet::Propagation(vector<double> inputs, int
num_threads)
{
   //stores the resultant outputs from each layer
```

```
vector<double> outputsPerLayer;
//stores the resultant outputs from each layer
vector<vector<double> > outputs;
//for measure the time
Double t0, t1;
//first check that we have the correct amount of inputs
if (inputs.size() != NumInputs)
{
   //just return an empty vector if incorrect.
   return outputs;
}
//For each layer....
omp_set_num_threads(num_threads);
#pragma omp parallel
{
   for (int i=0; i<NumHiddenLayers + 1; ++i)
   {
      #pragma omp single
      {
         if ( i > 0 )
         {
            inputs = outputsPerLayer;
         }
         outputsPerLayer.clear();
         outputsPerLayer.resize(vecLayers[i].NumNeurons);
      }
      If(omp_get_thread_num() == 0){
         t1 = omp_get_wtime();
         sing_time_1 += t1 – t0;
         t0 = t1;
      }
      //for each neuron
      #pragma omp for
      for (int j=0; j<vecLayers[i].NumNeurons; ++j)
      {
         double netinput = 0;
         int NumInputs = vecLayers[i].vecNeurons[j].NumInputs;
         vector<vector<double> > weightsPerLayer = GetLayerWeights(i);
         vector<double> weightsPerNeuron = weightsPerLayer[j];
         //for each weight
         for (int k=0; k<NumInputs; k++)
         {
            //sum the weights x inputs
            //add in the bias
```

```
                if(k == (NumInputs - 1))
                    netinput += weightsPerNeuron[NumInputs-1] * dBias;
                else
                    netinput += weightsPerNeuron[k] * inputs[k];
            }
             //we can store the outputs from each layer as we generate them.
             outputsPerLayer[j] = Sigmoid(netinput, dActivationResponse);
        }
        If(omp_get_thread_num() == 0){
            t1 = omp_get_wtime();
            para_loop_time += t1 – t0;
            t0 = t1;
        }
        #pragma omp single
        {
            outputs.push_back(outputsPerLayer);
        }
        If(omp_get_thread_num() == 0){
            t1 = omp_get_wtime();
            sing_time_2 += t1 – t0;
            t0 = t1;
        }

     }
   }
    return outputs;
}
```