High Performance Computing

2023 Fall

Lab 7. Message Passing Interface

Yongzhuo Ma

October 13, 2023

Chapter 1

Introduction to the Environment

1.1 Host Machine

Item	Value
OS version	macOS Sonoma 14.0
Apple clang version	15.0.0
CPU	Apple M2 Max
CPU Frequency	3.54 - 3.70 GHz
CPU Cores	12
Memory	64 GB
Memory Bandwidth	400 GB/s

1.2 Virtual Machine

Item	Value
Virtualization	Parallels
OS version	Ubuntu 22.04.3 LTS
gcc version	11.4.0
CPU	Apple M2 Max
CPU Frequency	3.69 GHz
CPU Cores	12
Memory	32 GB

1.3 Dependencies

Item	Version
MPICH	4.0

Chapter 2

DGEMM with MPICH

In this chapter, I will share my experience on the DGEMM implement using MPICH.

Steps

1.Prepare MPI environment.
2.Break the matrices into smaller blocks
3.Message Sending and Receiving

2.1 Prepare MPI environment

mpi.h should be included.
Define the range of MPI, the calculation part should be in it.

```
1 int thread_num,thread_id;
2 MPI_Status status;
3 MPI_Init(&argc,&argv);
4 MPI_Comm_rank(MPI_COMM_WORLD, &thread_id);
5 MPI_Comm_size(MPI_COMM_WORLD,&thread_num);
6 ⋮
7 MPI_Finalize();
```

2.2 Break the matrices

We will break the matrix by rows evenly.

2.3 Message Sending and Receiving

Mind that we have $thread\_num - 1$ calculating threads, the last one is the main thread. After deciding the blocks to process, we just need to send the data to the calculating thread. To send a series of data, the following parameters should be confirmed.

Item	Value
data source	(double*)
data count	(count)
data type	MPI_DOUBLE
destination thread	(Calc thread)
tag	(User defined int)
communicator channel	MPI_COMM_WORLD

Here is an example.
MPI_Send(&A[row_s*k],k*pblocks, MPI_DOUBLE, i+1, 1, MPI_COMM_WORLD)

This means: Starting from the value **A[row_s*k]**, and **k×pblocks double** data will be sent to thread **i+1** through **the world communicator** as tag **1**. After sending the message, we should receive the data from the calculating thread.
To receive, we should confirm the followings,

Item	Value
data destination	(double*)
data count	(count)
data type	MPI_DOUBLE
source thread	(Main thread)
tag	(User defined before)
communicator channel	MPI_COMM_WORLD
syncing status	(MPI_Status*)

Here is an example.
MPI_Recv(a,k*pblocks,MPI_DOUBLE,0,1,MPI_COMM_WORLD,&status);

This means: A series of **double** data from thread **0** and tagged **1** will be written into **a[0]** to **a[count-1]** through **the world communicator** according to the status of different threads saved in **status**.
After calculating, we should return the result parts to the main thread.

Problem and solution

1. Send the result directly back to matrix C will get the wrong answer.

Solution:

Allocate new matrix to save the result will solve this problem

Code

L7 8x8.c — Edited

L7 8x8 › main(argc, argv)

```c
int main(int argc, char** argv){
    int m = 8,n = 8,k = 8;
    int thread_num, thread_id, source;
    MPI_Status  status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&thread_id);
    MPI_Comm_size(MPI_COMM_WORLD,&thread_num);
    int pblocks = m/(thread_num-1);
        if(thread_id == 0)
        {
            for(int i=0;i<thread_num-1;i++)
            {
                int row_s= i * pblocks;
                int row_e= row_s + pblocks;
                MPI_Send(&A[row_s*k],k*pblocks, MPI_DOUBLE, i+1, 1, MPI_COMM_WORLD);
                MPI_Send(B, n*k, MPI_DOUBLE, i+1, 2, MPI_COMM_WORLD);
                MPI_Send(&C[row_s*n], pblocks*n, MPI_DOUBLE, i+1, 3,
                    MPI_COMM_WORLD);
            }
            for(int i=0;i<thread_num-1;i++)
            {
                MPI_Recv(&D
                    [i*pblocks*n],pblocks*n,MPI_DOUBLE,i+1,4,MPI_COMM_WORLD
                    ,&status);
            }
            for (int i = 0;i<m;i++)
            {
                for (int j = 0;j<n;j++)
                    printf("%lf ",D[i*n+j]);
                printf("\n");}
            }
        else
        {
                double* a= (double*) malloc(sizeof(double)*k*pblocks);
                double* b= (double*) malloc(sizeof(double)*k*n);
                double* c= (double*) malloc(sizeof(double)*pblocks*n);
                MPI_Recv(a,k*pblocks,MPI_DOUBLE,0,1,MPI_COMM_WORLD,&status);
                MPI_Recv(b,k*n,MPI_DOUBLE,0,2,MPI_COMM_WORLD,&status);
                MPI_Recv(c,pblocks*n,MPI_DOUBLE,0,3,MPI_COMM_WORLD,&status);
                for (int i=0;i<1;i++)
                for (int row = 0 ;row< pblocks;row++)
                    for(int col = 0;col<n;col++)
                        for (int ind = 0;ind<k;ind++)
                            c[row*n+col]+=a[row*k+ind]*b[ind*n+col];
                MPI_Send(c,pblocks*n,MPI_DOUBLE,0,4,MPI_COMM_WORLD);
        }
    MPI_Finalize();
}
```

Results

$$\boldsymbol{A} \times \boldsymbol{B} + \boldsymbol{C}$$

```
Reference 4x4:
-1.297094 -1.394872 -0.577658 0.137014
0.487135 -1.748927 -0.322982 0.274266
1.027899 -2.371782 -0.915003 -0.682657
-0.572809 0.082546 -0.001052 3.719102
L7 4x4:
-1.297094 -1.394872 -0.577658 0.137014
0.487135 -1.748927 -0.322982 0.274266
1.027899 -2.371782 -0.915003 -0.682657
-0.572809 0.082546 -0.001052 3.719102
Reference 8x8:
-1.404979 -1.235186 1.427717 0.127163 2.824722 1.899560 -0.239773 -0.218375
1.099094 -0.116566 0.404765 -2.818233 5.003082 4.961088 1.687088 -0.153294
0.066527 2.939972 -0.031972 -0.127371 2.422058 2.442030 3.383406 -2.500426
3.136486 2.072512 -0.793920 -0.061983 2.596720 -0.114813 2.297156 -1.964893
0.660703 3.123916 -2.851094 3.102350 -1.113017 -4.027380 0.254342 -1.468646
1.391210 2.018252 -0.349725 2.678988 -0.571531 -2.986300 2.481400 -1.105977
-2.751630 -1.706770 1.382435 -0.342028 1.762590 3.534226 0.921025 0.176807
0.544367 -0.063625 -2.069692 0.809548 -0.877884 -2.453056 -0.263956 -0.792012
L7 8x8:
-1.404979 -1.235186 1.427717 0.127163 2.824722 1.899560 -0.239773 -0.218375
1.099094 -0.116566 0.404765 -2.818233 5.003082 4.961088 1.687088 -0.153294
0.066527 2.939972 -0.031972 -0.127371 2.422058 2.442030 3.383406 -2.500426
3.136486 2.072512 -0.793920 -0.061983 2.596720 -0.114813 2.297156 -1.964893
0.660703 3.123916 -2.851094 3.102350 -1.113017 -4.027380 0.254342 -1.468646
1.391210 2.018252 -0.349725 2.678988 -0.571531 -2.986300 2.481400 -1.105977
-2.751630 -1.706770 1.382435 -0.342028 1.762590 3.534226 0.921025 0.176807
0.544367 -0.063625 -2.069692 0.809548 -0.877884 -2.453056 -0.263956 -0.792012
```

The results are consistent.

This chart shows that MPI(12 cores) did worse than OpenMP, but MPI will not decreases that fast than OpenMP when size reaches 2900. Optimized MPI is 12 times faster than Naive one.