《MPI 编程练习实验报告》

并行第05次作业

题　　目：	MPI 编程练习实验报告
上课时间：	周一下午
授课教师：	孙永谦
姓　　名：	张怡桢
学　　号：	2013747
年　　级：	2020级本科生
日　　期：	2022/12/14

MPI 编程练习实验报告

张怡桢，2013747

南开大学软件学院

摘　要：Massage Passing Interface:是消息传递函数库的标准规范，由MPI论坛开发。 定义：1. 一种新的库描述，不是一种语言。共有上百个函数调用接口，提供与C和Fortran语言的绑定。 2. MPI是一种标准或规范的代表，而不是特指某一个对它的具体实现。 3. MPI是一种消息传递编程模型，并成为这种编程模型的代表和事实上的标准。

关键词：MPI

1　实验内容

1. 实现第5章课件中的梯形积分法的MPI编程熟悉并掌握MPI编程方法，规模自行设定，可探讨不同规模对不同实现方式的影响。
2. 对于课件中“多个数组排序”的任务不均衡案例进行MPI编程实现，规模可自己设定、调整。
3. 附加：实现高斯消去法解线性方程组的MPI编程，规模自己设定。

2　实验环境

MPI实验在华为鲲鹏云服务器三台上实现

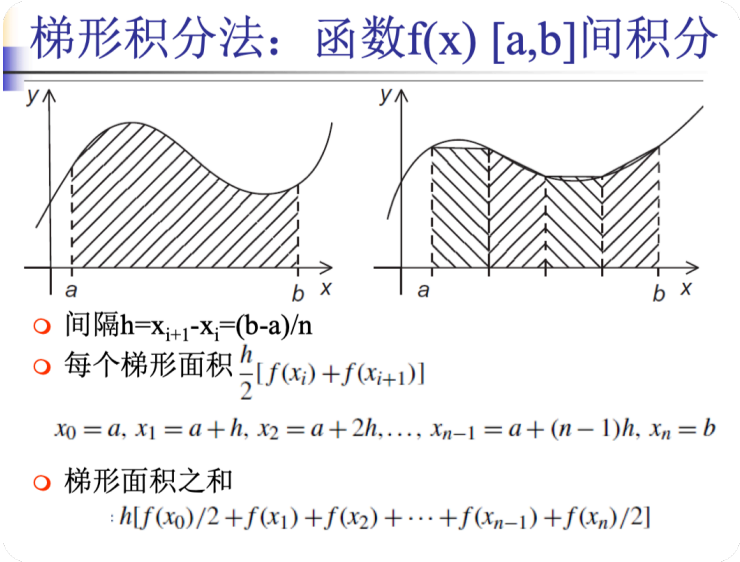
OpenMP以及Pthread使用M1芯片的MacOS实现

3　梯形积分法

3.1　实验内容

参见ppt上的内容

对于下图中给出的梯形积分法，实现并行编程。 每个线程或进程计算 a 到 b 区间中的某一段的梯形面积，最后求取全局和得到结果。



本部分实现第5章课件中的梯形积分法的MPI编程，并设置不同的数据规模进行对照实验，同时讨论不同规模对不同实现方式（包含MPI，Pthread 和 OpenMP）的影响。

3.2 算法设计

3.2.1 MPI实现

实验共用到了 3 台主机的进程进行计算。每个进程根据自己的 my_rank 进程号得到自身的计算任务，完成局部和的计算后，（除 0 号进程外）使用 MPI_Send 将结果发送至 0 号进程； 0 号进程使用 MPI_Recv 阻塞式地接受其他进程传回的结果，并计算全局和。同时，0 号进程 还完成计时、输出结果的任务。具体的实现代码如下：

mph_trap.cpp

```
#include <stdio.h>
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <sys/time.h>
#include <mpi.h>

//待积分的f(x)
double f(double x) {
    return x*x;
}

//计算局部和
double Trap(double a, double b, int count, double h) {
    double my_result, x;
    int i;
    my_result = (f(a) + f(b)) / 2.0;
    //局部求和，避免过多通信
    for (i = 1; i < count; i++) {
        x = a + i * h;
```

```
        my_result += f(x);
    }
    my_result *= h;
    return my_result;
}

//主函数
int main()
{
    int my_rank, comm_sz, name_len, n = 1000;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    double a = 0, b = 5000;
    double h = (b - a) / n;
    double global_result;
    //clock_t start, end;
    //计时
    struct timeval tpstart,tpend;
    double time;

    MPI_Init(NULL, NULL);
    // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    // Get the name of the processor
    MPI_Get_processor_name(processor_name, &name_len);
    // Print off a hello world message
    printf("Trap Caculation from processor %s, rank %d out of %d processors\n",
    processor_name, my_rank, comm_sz);

    //0号进程开始计时
    if (my_rank == 0) {
        //start = clock();
      gettimeofday(&tpstart,NULL);
    }
    int local_n = n / comm_sz;
    int local_a = a + my_rank * local_n * h;
    int local_b = local_a + local_n * h;
    //将未除尽的任务数分配给最后一个进程
    if (my_rank == comm_sz - 1) {
        local_n = n - (comm_sz - 1) * local_n;
        local_b = b;
    }
    //每个进程计算自身分配的任务的局部和
    double local_result = Trap(local_a, local_b, local_n, h);
```

```
    //其他进程将局部和发送给0号进程
    if (my_rank != 0) {
        MPI_Send(&local_result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
    //0号进程收集局部和并求全局和
    else {
        global_result = local_result;
        for (int source = 1; source < comm_sz; source++)
        {
            MPI_Recv(&local_result, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            global_result += local_result;
        }
        //完成计算，0号进程停止计时
        //end = clock();
        //time = (end - start) / CLOCKS_PER_SEC;
      gettimeofday(&tpend,NULL);
      time=(1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec)/1000;

    }
    //0号进程打印出结果
    if (my_rank == 0) {
        printf("划分成小梯形的块数：%d\n", n);
        printf("计算结果是：%.15e\n", global_result);
        printf("总共耗时：%f\n\nms", time);
    }

    MPI_Finalize();
    return 0;
}
```

makefile

makefile文件进行编译

```
EXECS=mpi_trap
MPICC?=mpicc
all: ${EXECS}
mpi_trap: mpi_trap.cpp
  ${MPICC} -o mpi_trap mpi_trap.cpp
clean:
  rm -f ${EXECS}
```

config

给每台主机分配了2个进程（在 config 文件中设置），总共有6个进程共同执行计算任务。

```
ecs-zyz-0001:2
ecs-zyz-0002:2
ecs-zyz-0003:2
```

执行语句

```
mpiexec -n 6 -f /home/zhangyizhen/trap/config /home/zhangyizhen/trap/mpi_trap
```

3.2.2 Pthread实现

对于每个线程，先求局部和最后全局求和，避免过多通信。同时注意给全局和加锁保证结果的正确性。

```
//pthread编程线程控制程序
void pthread_trap(double a,double b,int n,double*global_result_p,int thread_count){
    pthread_param params[thread_count];
    pthread_t threads[thread_count];
    pthread_mutex_t amutex;
    pthread_mutex_init(&amutex,NULL);
    for(int i=0;i<thread_count;i++){
        //为每个线程参数赋值
        params[i].a=a;
        params[i].b=b;
        params[i].n=n;
        params[i].global_result_p=global_result_p;
        params[i].thread_count=thread_count;
        params[i].my_rank=i;
        params[i].amutex=&amutex;
        //创建线程
        pthread_create(&threads[i],NULL,pthread_trap,(void*)&params[i]);
    }
    //销毁线程
    for(int i=0;i<thread_count;i++){
        pthread_join(threads[i],NULL);
    }
    //销毁互斥量锁
    pthread_mutex_destroy(&amutex);
}
```

其中每个线程执行的函数如下：

```
//pthread中每个线程调用的程序
void* pthread_trap(void* p){
    //恢复获取参数
    pthread_param* params=(pthread_param*) p;
    double h,x,my_result;
    double local_a,local_b;
    int local_n;
    int my_rank=params->my_rank;
    int thread_count=params->thread_count;
    h=(params->b-params->a)/params->n;
    local_n=params->n/thread_count;
    local_a=params->a+my_rank*local_n*h;
    local_b=local_a+local_n*h;
    my_result=(f(local_a)+f(local_b))/2.0;
    //先局部求和，避免过多通信
    for(int i=1;i<=local_n-1;i++){
        x=local_a+i*h;
        my_result+=f(x);
    }
    my_result=my_result*h;
    //使用互斥量 amutex 上锁保证全局求和正确性
    pthread_mutex_lock(params->amutex);
    *params->global_result_p+=my_result;
    pthread_mutex_unlock(params->amutex);
}
```

3.2.3 OpenMP实现

```
//1:最普通的OpenMP版本
    cout<<"Using OpenMP"<<endl;
    gettimeofday(&tpstart,NULL);
#pragma omp parallel num_threads(thread_count)
    trap(a,b,n,&global_result);
    gettimeofday(&tpend,NULL);
    time=(1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec)/1000;
```

```
//每个线程执行的求和程序
void trap(double a,double b,int n,double*global_result_p){
    double h,x,my_result;
    double local_a,local_b;
```

```
    int local_n;
    int my_rank=omp_get_thread_num();
    int thread_count=omp_get_num_threads();
    h=(b-a)/n;
    local_n=n/thread_count;
    local_a=a+my_rank*local_n*h;
    local_b=local_a+local_n*h;
    my_result=(f(local_a)+f(local_b))/2.0;
    //先局部求和，避免过多通信
    for(int i=1;i<=local_n-1;i++){
        x=local_a+i*h;
        my_result+=f(x);
    }
    my_result=my_result*h;
    //用临界区保证全局求和正确性
#pragma omp critical
    *global_result_p+=my_result;
}
```

3.2.4 OpenMP与Pthread在MacOS上的makefile配置

```
cmake_minimum_required(VERSION 3.23)
project(pthread)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread")

#set(CMAKE_C_COMPILER "/usr/bin/gcc") #这里写你的gcc路径
#set(CMAKE_CXX_COMPILER "/usr/bin/g++") #这里写你的g++路径
set(OPENMP_LIBRARIES "/opt/homebrew/Cellar/libomp/15.0.4/lib") #这里写你的libomp路径，通过
brew安装的默认地址
set(OPENMP_INCLUDES "/opt/homebrew/Cellar/libomp/15.0.4/include")#这里写你的libomp路径

OPTION (USE_OpenMP "Use OpenMP to enamble <omp.h>" ON)

# Find OpenMP
if(APPLE AND USE_OpenMP)
    if(CMAKE_C_COMPILER_ID MATCHES "Clang")
        set(OpenMP_C "${CMAKE_C_COMPILER}")
        set(OpenMP_C_FLAGS "-Xpreprocessor -fopenmp -lomp -Wno-unused-command-line-
argument")
        #注意以上需要增加-Xpreprocessor 以及不能直接-llibomp 在这里不需要前缀lib只需要-lomp即
可，下面相似的地方也是同个道理
```

```
        set(OpenMP_C_LIB_NAMES "libomp" "libgomp" "libiomp5")
        set(OpenMP_libomp_LIBRARY ${OpenMP_C_LIB_NAMES})
        set(OpenMP_libgomp_LIBRARY ${OpenMP_C_LIB_NAMES})
        set(OpenMP_libiomp5_LIBRARY ${OpenMP_C_LIB_NAMES})
    endif()
    if(CMAKE_CXX_COMPILER_ID MATCHES "Clang")
        set(OpenMP_CXX "${CMAKE_CXX_COMPILER}")
        set(OpenMP_CXX_FLAGS "-Xpreprocessor -fopenmp -lomp -Wno-unused-command-line-
argument")
        set(OpenMP_CXX_LIB_NAMES "libomp" "libgomp" "libiomp5")
        set(OpenMP_libomp_LIBRARY ${OpenMP_CXX_LIB_NAMES})
        set(OpenMP_libgomp_LIBRARY ${OpenMP_CXX_LIB_NAMES})
        set(OpenMP_libiomp5_LIBRARY ${OpenMP_CXX_LIB_NAMES})
    endif()
endif()

if(USE_OpenMP)
    find_package(OpenMP REQUIRED)
endif(USE_OpenMP)


if (OPENMP_FOUND)
    include_directories("${OPENMP_INCLUDES}")
    link_directories("${OPENMP_LIBRARIES}")
    set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
endif(OPENMP_FOUND)

add_executable(pthread trap.cpp)
```

3.3 实验设计与数据分析

实验计算的梯形面积为——函数:f(x)=x*x，a=0，b=5000的面积部分。

通过调整梯形积分 法划分成小梯形的个数规模，来比较不同编程方式的异同。

下面对三种编程模式设置规模n=100，1000，2000，5000这几个参数来实现计算并计算时间进行对比。

3.3.1 mpi实现

规模：n=100

```
[zhangyizhen@ecs-zyz-0001 ~]$ mpiexec -n 6 -f /home/zhangyizhen/trap/config /home/zhangyizhen/trap/mpi_trap

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
Trap Caculation from processor ecs-zyz-0003, rank 4 out of 6 processors
Trap Caculation from processor ecs-zyz-0002, rank 2 out of 6 processors
Trap Caculation from processor ecs-zyz-0003, rank 5 out of 6 processors
Trap Caculation from processor ecs-zyz-0002, rank 3 out of 6 processors
Trap Caculation from processor ecs-zyz-0001, rank 0 out of 6 processors
Trap Caculation from processor ecs-zyz-0001, rank 1 out of 6 processors
划分成小梯形的块数： 100
计算结果是： 4.166875000000000e+10
总共耗时： 5.000000

ms[zhangyizhen@ecs-zyz-0001 ~]$
```

规模：n=1000

```
[zhangyizhen@ecs-zyz-0001 trap]$ mpiexec -n 6 -f /home/zhangyizhen/trap/config /home/zhangyizhen/trap/mpi_trap

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
Trap Caculation from processor ecs-zyz-0003, rank 4 out of 6 processors
Trap Caculation from processor ecs-zyz-0002, rank 2 out of 6 processors
Trap Caculation from processor ecs-zyz-0003, rank 5 out of 6 processors
Trap Caculation from processor ecs-zyz-0002, rank 3 out of 6 processors
Trap Caculation from processor ecs-zyz-0001, rank 1 out of 6 processors
Trap Caculation from processor ecs-zyz-0001, rank 0 out of 6 processors
划分成小梯形的块数： 1000
计算结果是： 4.166668750000000e+10
总共耗时： 10.000000

ms[zhangyizhen@ecs-zyz-0001 trap]$
```


规模：n=2000

```
[zhangyizhen@ecs-zyz-0001 trap]$ mpiexec -n 6 -f /home/zhangyizhen/trap/config /home/zhangyizhen/trap/mpi_trap

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
Trap Caculation from processor ecs-zyz-0001, rank 0 out of 6 processors
Trap Caculation from processor ecs-zyz-0003, rank 4 out of 6 processors
Trap Caculation from processor ecs-zyz-0002, rank 2 out of 6 processors
Trap Caculation from processor ecs-zyz-0001, rank 1 out of 6 processors
Trap Caculation from processor ecs-zyz-0003, rank 5 out of 6 processors
Trap Caculation from processor ecs-zyz-0002, rank 3 out of 6 processors
划分成小梯形的块数： 2000
计算结果是： 4.165936106562500e+10
总共耗时： 8.000000

ms[zhangyizhen@ecs-zyz-0001 trap]$
```

规模：n=5000

```
[zhangyizhen@ecs-zyz-0001 trap]$ mpiexec -n 6 -f /home/zhangyizhen/trap/config /home/zhangyizhen/trap/mpi_trap

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
Trap Caculation from processor ecs-zyz-0003, rank 4 out of 6 processors
Trap Caculation from processor ecs-zyz-0002, rank 2 out of 6 processors
Trap Caculation from processor ecs-zyz-0003, rank 5 out of 6 processors
Trap Caculation from processor ecs-zyz-0002, rank 3 out of 6 processors
Trap Caculation from processor ecs-zyz-0001, rank 1 out of 6 processors
Trap Caculation from processor ecs-zyz-0001, rank 0 out of 6 processors
划分成小梯形的块数： 5000
计算结果是： 4.166666750000000e+10
总共耗时： 4.000000

ms[zhangyizhen@ecs-zyz-0001 trap]$
```

3.3.2 Pthread实现

规模：n=100

```
Using Pthread
With n = 100 trapezoids, our estimate
of the integral from 0.000000 to 5000.000000 = 4.166875000000000e+10
cost time: 0.235ms
```

规模：n=1000

```
Using OpenMP
With n = 1000 trapezoids, our estimate
of the integral from 0.000000 to 5000.000000 = 4.166668750000000e+10
cost time: 0.319ms
```

规模：n=2000

```
Using OpenMP
With n = 2000 trapezoids, our estimate
of the integral from 0.000000 to 5000.000000 = 4.166667187500000e+10
cost time: 0.347ms
```

规模：n=5000

```
Using Pthread
With n = 5000 trapezoids, our estimate
of the integral from 0.000000 to 5000.000000 = 4.166666750000000e+10
cost time: 0.33ms
```

3.3.3 OpenMP实现

规模：n=100

```
/Users/zhanghaha/Program/CLionProjects/pthread/cmake-build-debug/pthread
Using OpenMP
With n = 100 trapezoids, our estimate
of the integral from 0.000000 to 5000.000000 = 4.16687500000000e+10
cost time: 0.475ms
```

规模：n=1000

```
Using Pthread
With n = 1000 trapezoids, our estimate
of the integral from 0.000000 to 5000.000000 = 4.16666875000000e+10
cost time: 0.26ms
```

规模：n=2000

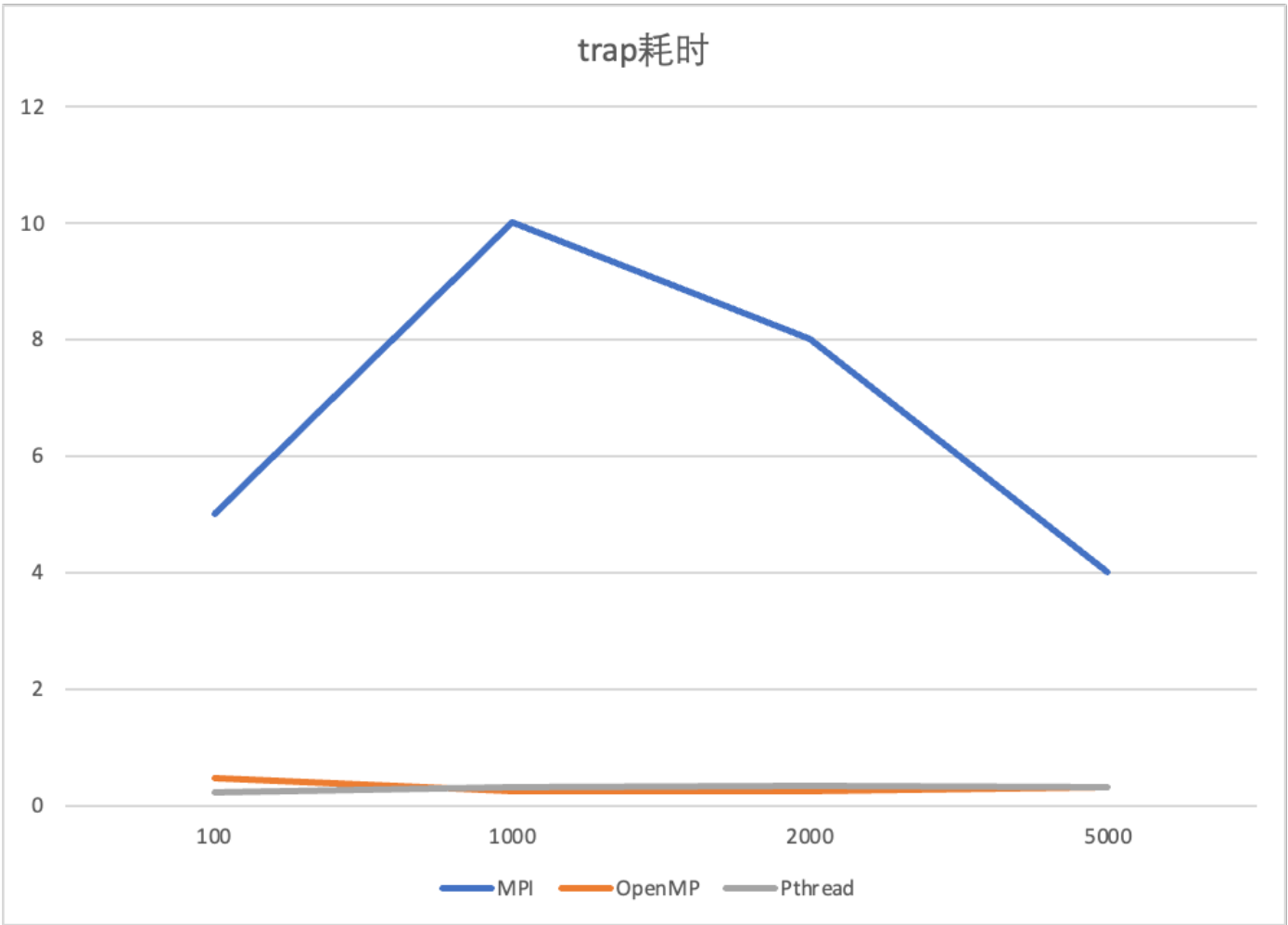
```
Using Pthread
With n = 2000 trapezoids, our estimate
of the integral from 0.000000 to 5000.000000 = 4.16666718750000e+10
cost time: 0.248ms
```

规模：n=5000

```
Using OpenMP
With n = 5000 trapezoids, our estimate
of the integral from 0.000000 to 5000.000000 = 4.16666675000000e+10
cost time: 0.329ms
```

3.3.4 对比分析

规模	MPI	OpenMP	Pthread
100	5	0.475	0.235
1000	10	0.26	0.319
2000	8	0.248	0.347
5000	4	0.329	0.33



根据上述图表可以看出，几种实现方式的性能差异较为明显。

可能是因为OpenMP以及Pthread使用的是本地的电脑完成，所以处理速度快，效率高；

而MPI 方法的执行时间出现较长的现象，MPI使用远程服务器以及三个服务器之间的通信完成，存在网络延迟问题。

可见，在网络较差的情况下，使用MPI算法，并不能得到更大的优势。

并且耗时并没有随着梯形划分规模n的增大而增大，推测应该是数据规模太小了，程序多线程处理以及网络延时的耗时影响完全覆盖了规模问题。

3.4 实验出现的问题与解决办法

3.4.1 计时问题

精度太低

使用clock() 函数

头文件：

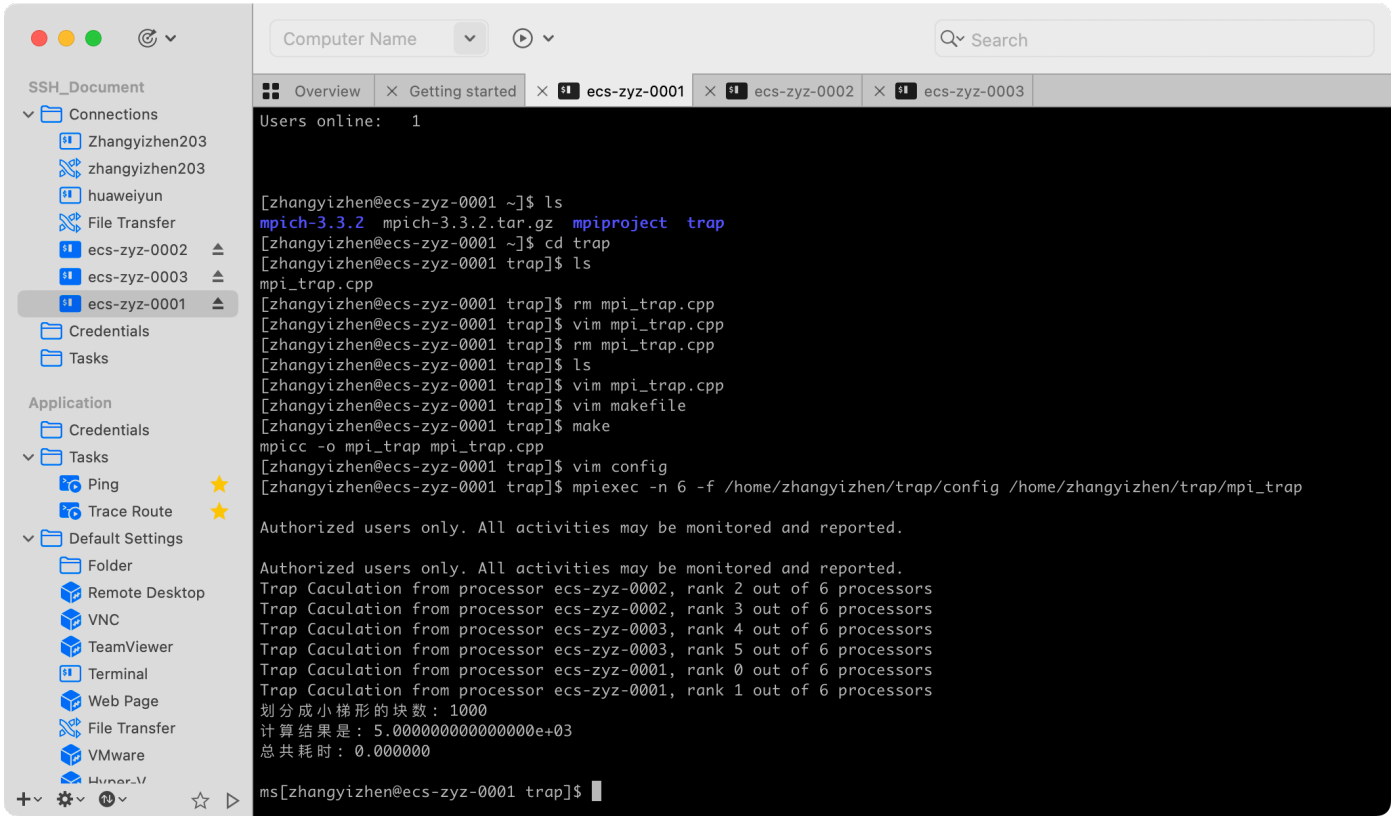
clock()函数，返回“自程序启动到调用该函数，CPU时钟的计时单元数（clock tick）”

每过1ms，计数值+1

精度：1毫秒

```
start = clock(); ......

end = clock(); time = (end - start) / CLOCKS_PER_SEC;
```



在实验过程中出现耗时为0的情况，该计时函数精度太低！！！

解决办法

使用gettimeofday（）函数

头文件：<sys/time.h>

精度：1us

```
struct timeval tpstart,tpend;
gettimeofday(&tpstart,NULL);
gettimeofday(&tpend,NULL);
double timeuse = 1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
printf("used time:%fus\n",timeuse);
```

该精度可以满足实验的耗时要求。

3.5 实验总结

通过本实验，我熟悉并掌握 MPI 编程方法，在探讨不同规模对不同实现方式的影响，了解了多台主机共同完成同一计算任务的思想。本实验中 MPI耗时比Pthread、OpenMP方法的性能差异较为明显，这应该是设备性能问题以及网络延时问题带来的差异。

4 多个数组排序（MPI）

4.1 实验内容

对于课件中“多个数组排序”的任务不均衡案例进行MPI编程实现，规模可自己设定、调整。

对 ARR_NUM 个长度为 ARR_LEN 的一维数组进行排序，使用MPI编程。

4.2 算法设计

4.2.1 数组不均衡初始化

初始化 ARR_NUM 行，ARR_LEN 列的数组。

当行数 i 属于 0~2499 时, ratio = 0，此时 2500 行内完全升序排列；当

行数 i 属于 2500~4999 时, ratio = 32 此时随机有 2500 * 1/4 行升序， 2500 * 3/4 行降序;

当 行数 i 属于 5000~7499 时，ratio = 64，此时随机有 2500 * 1/2 行升序，2500*1/2 行降序；

当行数 i 属于 7500~9999 时，ratio = 128，此时 2500 行完全降序。此时数组可分为四块，数组负载不均衡。

```
// 初始化待排序数组，使得
// 第一段：完全升序
// 第二段：1/4逆序，3/4升序
// 第三段：1/2逆序，1/2升序
// 第四段：完全逆序
void init(void){
    int ratio;
    srand(unsigned(time(nullptr)));
    for (int i = 0; i < ARR_NUM; i++){
        arr[i].resize(ARR_LEN);
        if(i < seg)ratio = 0;
        else if(i < seg * 2)ratio = 32;
        else if(i < seg * 3)ratio = 64;
        else ratio = 128;
        if((rand() & 127) < ratio){
            for(int j = 0; j < ARR_LEN; j++){
                arr[i][j] = ARR_LEN - j;
            }
```

```
        }
        else{
            for(int j = 0; j < ARR_LEN; j++){
                arr[i][j] = j;
            }
        }
    }
}
```

4.2.2 mpi代码实现

实验共用到了 3 台主机的进程完成数组排序。根据进程号连续划分任务，每个进程完成排序之后，将自己执行任务的耗时传送给 0 号进程，最终 0 号进程接受各个进程完成排序任务的 时间并输出。具体的实现代码如下：

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <algorithm>
#include <mpi.h>
#include <sys/time.h> // for gettimeofday()
using namespace std;

//数组数量
const int ARR_NUM = 1000;
//每个数组的长度
const int ARR_LEN = 300;
//进程数量/主机数量
const int PROCESS_NUM = 3;
//每个进程处理的数组数量
const int seg = 50;
vector<int> arr[ARR_NUM];

struct timeval tpstart,tpend;

// 初始化待排序数组，使得
// 第一段：完全升序
// 第二段：1/4逆序，3/4升序
// 第三段：1/2逆序，1/2升序
// 第四段：完全逆序
void init(void){
    int ratio;
    srand(unsigned(time(nullptr)));
    for (int i = 0; i < ARR_NUM; i++){
        arr[i].resize(ARR_LEN);
```

```
        if(i < seg)ratio = 0;
        else if(i < seg * 2)ratio = 32;
        else if(i < seg * 3)ratio = 64;
        else ratio = 128;
        if((rand() & 127) < ratio){
            for(int j = 0; j < ARR_LEN; j++){
                arr[i][j] = ARR_LEN - j;
            }
        }
        else{
            for(int j = 0; j < ARR_LEN; j++){
                arr[i][j] = j;
            }
        }
    }
}

void doTask(int begin) {
    for (int i = begin; i < min(begin + seg, ARR_NUM); ++i) {
        sort(arr[i].begin(), arr[i].end());
    }
}

//主函数
int main() {
    int my_rank, comm_sz,name_len;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    //初始化数组
    init();

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    // Get the name of the processor
    MPI_Get_processor_name(processor_name, &name_len);
    // Print off a hello world message
    printf("From processor %s, rank %d out of %d processors\n",
    processor_name, my_rank, comm_sz);

    MPI_Status status;
    bool done = false;
    int current_task = 0;//当前的任务
    int ready;
    double time = 0;
```

```
    gettimeofday(&tpstart,NULL);
    //其他进程将本进程的运行时间发送给0号进程
    if (my_rank != 0) {
         while (!done) {
            int begin;
            MPI_Send(&ready, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
            MPI_Recv(&begin, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            if (status.MPI_TAG == 1) {
                break;
            }
            doTask(begin);
        }
    }
    //0号进程打印各个进程的运行时间
    if (my_rank == 0) {
         while (current_task < ARR_NUM) {
            //接受任何一个线程的状态
            MPI_Recv(&ready, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
            MPI_Send(&current_task, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
            current_task += seg;
        }
        done = true;
        printf("数组数量: %d;数组长度:%d \n",ARR_NUM,ARR_LEN);
        cout << "all work done!!!!!" << endl;
        //唤醒所有线程
        for (int i = 1; i < comm_sz; ++i) {
            MPI_Recv(&ready, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);

            MPI_Send(&current_task, 1, MPI_INT, status.MPI_SOURCE, 1, MPI_COMM_WORLD);
        }
        gettimeofday(&tpend,NULL);
        time += (1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec)/1000;
        cout <<" time: " << time << "ms" << endl;

    }


    MPI_Finalize();



    return 0;
}
```

4.2.3 执行设置

`makefile`

makefile文件进行编译

```
EXECS=mpi_array
MPICC?=mpicc
all: ${EXECS}
mpi_array: mpi_array.cpp
  ${MPICC} -o mpi_array mpi_array.cpp -lstdc++
clean:
  rm -f ${EXECS}
```

`config`

给每台主机分配了2个进程（在 config 文件中设置），总共有6个进程共同执行计算任务。

```
ecs-zyz-0001:2
ecs-zyz-0002:2
ecs-zyz-0003:2
```

`执行语句`

```
mpiexec -n 6 -f /home/zhangyizhen/array/config /home/zhangyizhen/array/mpi_array
```

4.3 实验设计与数据分析

1. 使用 init 函数生成分布不均的数组；
2. 使用与实验一（梯形积分法）中相同的计时方法。
3. 改变数组规模，分别处理 1000，1500，2000，2500，3000 个长度为 300 的数组， 比较在不同规模下各个进程的时间差异，从而理解任务负载不均衡对效率的影响。

`ARR_NUM=1000`

```
[zhangyizhen@ecs-zyz-0001 array]$ mpiexec -n 6 -f /home/zhangyizhen/array/config /home/zhangyizhen/array/mpi_array

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
From processor ecs-zyz-0003, rank 4 out of 6 processors
From processor ecs-zyz-0001, rank 0 out of 6 processors
数组数量： 1000;数组长度:300
all work done!!!!!
 time: 7ms
From processor ecs-zyz-0003, rank 5 out of 6 processors
From processor ecs-zyz-0002, rank 2 out of 6 processors
From processor ecs-zyz-0001, rank 1 out of 6 processors
From processor ecs-zyz-0002, rank 3 out of 6 processors
[zhangyizhen@ecs-zyz-0001 array]$
```

ARR_NUM=1500

```
[zhangyizhen@ecs-zyz-0001 array]$ mpiexec -n 6 -f /home/zhangyizhen/array/config /home/zhangyizhen/array/mpi_array

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
From processor ecs-zyz-0003, rank 4 out of 6 processors
From processor ecs-zyz-0003, rank 5 out of 6 processors
From processor ecs-zyz-0002, rank 2 out of 6 processors
From processor ecs-zyz-0002, rank 3 out of 6 processors
From processor ecs-zyz-0001, rank 0 out of 6 processors
From processor ecs-zyz-0001, rank 1 out of 6 processors
数组数量： 1500;数组长度:300
all work done!!!!!
 time: 11ms
[zhangyizhen@ecs-zyz-0001 array]$
```

ARR_NUM=2000

```
[zhangyizhen@ecs-zyz-0001 array]$ mpiexec -n 6 -f /home/zhangyizhen/array/config /home/zhangyizhen/array/mpi_array

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
From processor ecs-zyz-0001, rank 0 out of 6 processors
From processor ecs-zyz-0002, rank 2 out of 6 processors
From processor ecs-zyz-0003, rank 4 out of 6 processors
From processor ecs-zyz-0001, rank 1 out of 6 processors
From processor ecs-zyz-0002, rank 3 out of 6 processors
From processor ecs-zyz-0003, rank 5 out of 6 processors
数组数量： 2000;数组长度:300
all work done!!!!!
 time: 19ms
[zhangyizhen@ecs-zyz-0001 array]$
```

ARR_NUM=2500

```
[zhangyizhen@ecs-zyz-0001 array]$ mpiexec -n 6 -f /home/zhangyizhen/array/config /home/zhangyizhen/array/mpi_array

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
From processor ecs-zyz-0003, rank 4 out of 6 processors
From processor ecs-zyz-0002, rank 2 out of 6 processors
From processor ecs-zyz-0001, rank 0 out of 6 processors
From processor ecs-zyz-0003, rank 5 out of 6 processors
From processor ecs-zyz-0002, rank 3 out of 6 processors
From processor ecs-zyz-0001, rank 1 out of 6 processors
数组数量： 2500;数组长度:300
all work done!!!!!
 time: 20ms
[zhangyizhen@ecs-zyz-0001 array]$
```

ARR_NUM=3000

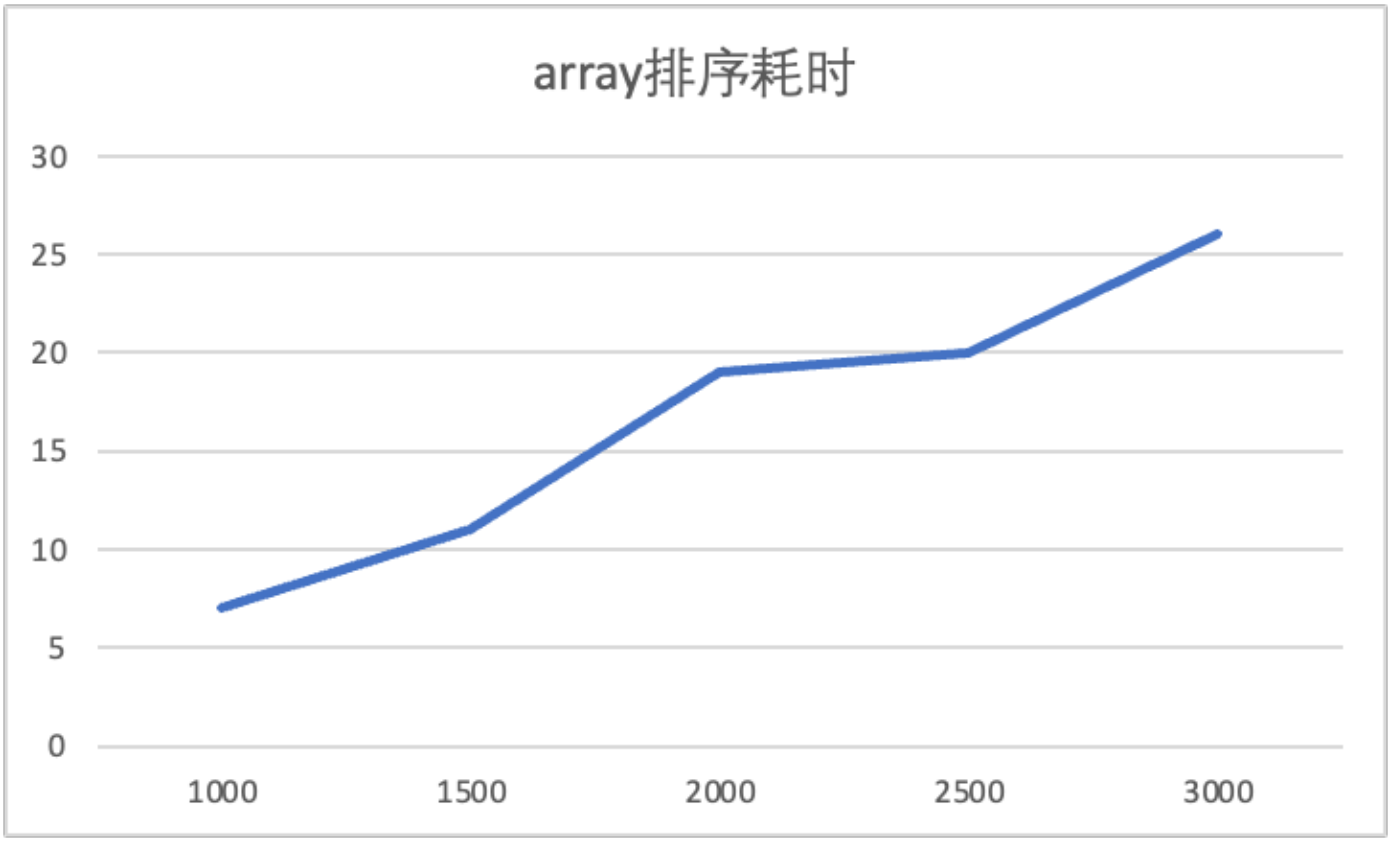
```
[zhangyizhen@ecs-zyz-0001 array]$ mpiexec -n 6 -f /home/zhangyizhen/array/config /home/zhangyizhen/array/mpi_array

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
From processor ecs-zyz-0002, rank 2 out of 6 processors
From processor ecs-zyz-0003, rank 4 out of 6 processors
From processor ecs-zyz-0002, rank 3 out of 6 processors
From processor ecs-zyz-0003, rank 5 out of 6 processors
From processor ecs-zyz-0001, rank 0 out of 6 processors
From processor ecs-zyz-0001, rank 1 out of 6 processors
数组数量： 3000;数组长度:300
all work done!!!!!
 time: 26ms
[zhangyizhen@ecs-zyz-0001 array]$
```

4.4 实验总结

arr_num	耗时/ms
1000	7
1500	11
2000	19
2500	20
3000	26



由上面的数据可以看到，随着ARR_NUM的增大，MPI排序算法需要的用时也增大了，符合我的预期。

通过以上的实验，我对于array排序在多线程，多进程中的实现有了更深的了解，同时对于多服务器实现MPI进行了掌握。

5 高斯消去（MPI）

5.1 实验内容

附加：实现高斯消去法解线性方程组的MPI编程。

对于给定的线性方程组 Ax=b，使用 MPI 编程，完成对于线性方程组的高斯消元。

5.2 算法设计

5.2.1 mpi实现

每个进程完成部分行的消元计算任务，然后使用 barrier 功能，使得所有进程均完成第 k 行 的校园后再计算下一行的消元。程序实现的具体代码如下：

```
#include <stdio.h>
#include <mpi.h>
#include <ctime>
#include <sys/time.h>
#include <algorithm>


const int n = 1024;//固定矩阵规模，控制变量
const int maxN = n + 1; // 矩阵的最大值
float a[maxN][maxN];
float temp[maxN][maxN];//用于暂时存储a数组中的变量，控制变量唯一
int next_task = 0;
int seg;
int line = 0;//记录当前所依赖的行数
struct timeval startTime, stopTime;// timers


/**
 * 根据第i行的元素，消除j行的元素
 * @param i 根据的行数
 * @param j 要消元的行数
 */
void OMP_elimination(int i, int j) {
    //求出相差倍数
    float temp = a[j][i] / a[i][i];
    //遍历这一行的所有值，将i后面的数值依次减去相对应的值乘以倍数
    for (int k = i + 1; k <= n; ++k) {
        a[j][k] -= a[i][k] * temp;
    }
    //第i个为0
    a[j][i] = 0.00;
}


//用于矩阵改变数值,为防止数据溢出,随机数的区间为100以内的浮点数
void change() {
    srand((unsigned) time(NULL));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= n; j++) {
```

```
            a[i][j] = (float) (rand() % 10000) / 100.00;
        }
    }
}


int main(int argc, char *argv[]) {
    change();
    int rank, thread_num;
    gettimeofday(&startTime, NULL);
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &thread_num);
    MPI_Status status;
    int ready;
    bool done = false;
    if (rank == 0) {
        printf("size : %d\n", n);
        for (line = 0; line < n - 1; ++line) {
            next_task = line + 1;
            seg = (n - next_task) / (thread_num - 1) + 1;
            for (int i = 1; i < thread_num; i++) {
                int task = (i - 1) * seg + next_task;
                MPI_Send(&task, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            }
            //等待所有线程
            for (int j = 1; j < thread_num; ++j) {
                MPI_Recv(&ready, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
            }
        }
        printf("all work done!!!!!\n");
        done = true;
        gettimeofday(&stopTime, NULL);
        double trans_mul_time =
                (stopTime.tv_sec - startTime.tv_sec) * 1000 + (stopTime.tv_usec -
startTime.tv_usec) * 0.001;
        printf("time: %lf ms\n", trans_mul_time);
    } else {
        while (!done) {
            int task;
            MPI_Recv(&task, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            int min = task + seg < n ? task + seg : n;
            for (int i = task; i < min; ++i) {
                OMP_elimination(line, i);
            }
```

```
            MPI_Send(&ready, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
    return 0;
}
```

5.2.2 执行设置

makefile

makefile文件进行编译

```
EXECS=mpi_gause
MPICC?=mpicc
all: ${EXECS}
mpi_gause: mpi_gause.cpp
  ${MPICC} -o mpi_gause mpi_gause.cpp -lstdc++
clean:
  rm -f ${EXECS}
```

config

给每台主机分配了2个进程（在 config 文件中设置），总共有6个进程共同执行计算任务。

```
ecs-zyz-0001:2
ecs-zyz-0002:2
ecs-zyz-0003:2
```

执行语句

```
mpiexec -n 6 -f /home/zhangyizhen/gause/config /home/zhangyizhen/gause/mpi_gause
```

5.3 实验设计与数据分析

改变矩阵规模，分别处理矩阵规模 n=512，1024，2048的高斯消去， 比较在不同规模下各个进程的时间差异，从而理解任务负载不均衡对效率的影响。

```
[zhangyizhen@ecs-zyz-0001 gause]$ make
mpicc -o mpi_gause mpi_gause.cpp -lstdc++
[zhangyizhen@ecs-zyz-0001 gause]$ mpiexec -n 6 -f /home/zhangyizhen/gause/config /home/zhangyizhen/gause/mpi_gause

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
size : 512
all work done!!!!!
time: 405.299000 ms
```

```
[zhangyizhen@ecs-zyz-0001 gause]$ mpiexec -n 6 -f /home/zhangyizhen/gause/config /home/zhangyizhen/gause/mpi_gause

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
size : 1024
all work done!!!!!
time: 513.106000 ms
```

```
[zhangyizhen@ecs-zyz-0001 gause]$ mpiexec -n 6 -f /home/zhangyizhen/gause/config /home/zhangyizhen/gause/mpi_gause

Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
size : 2048
all work done!!!!!
time: 776.379000 ms
```

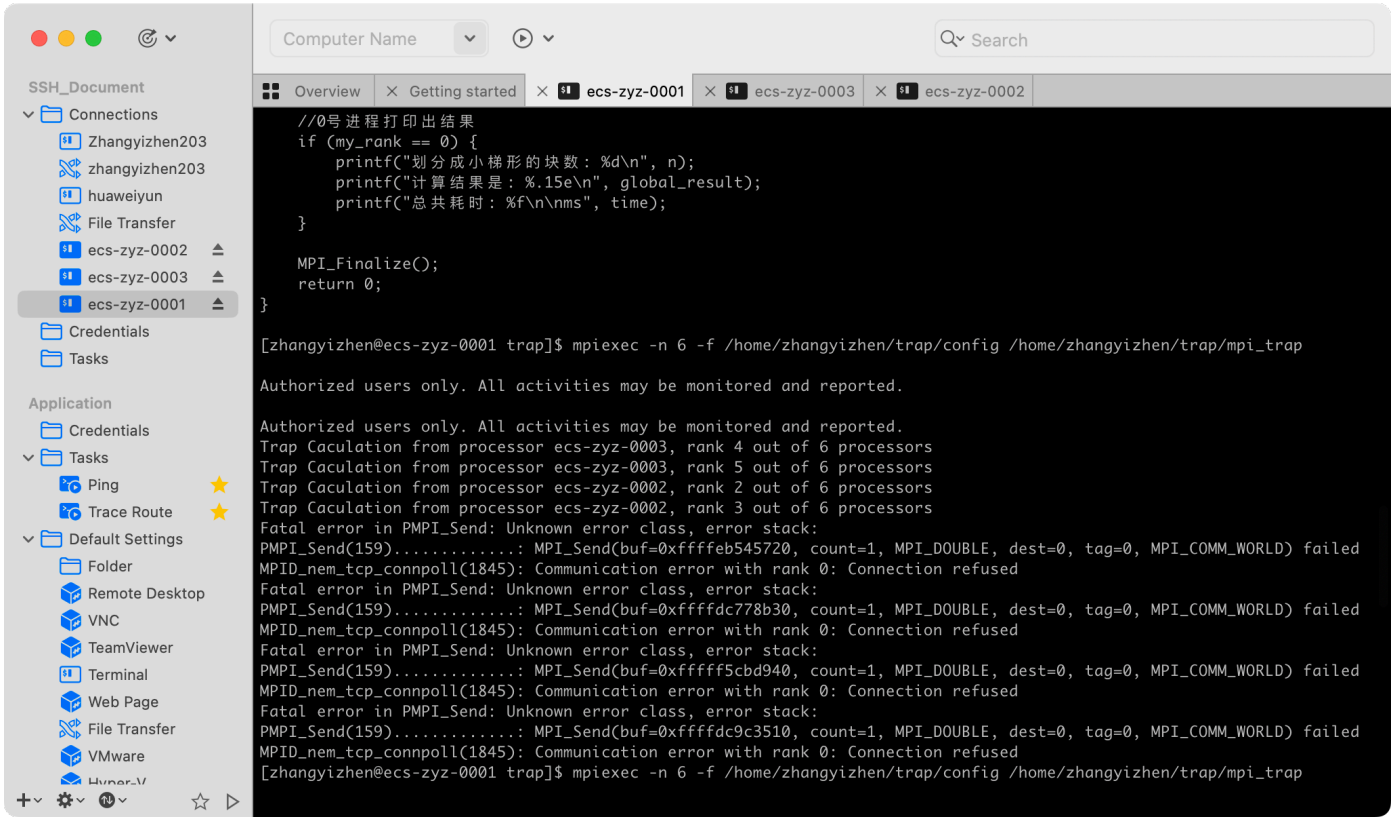
5.4 实验总结

size从512到1024，再到2048，随着矩阵规模的增大，高斯消去的用时也随着增加，符合正常的实验预期。

通过这次实验，我对于高斯消去以及MPI的用法更加了解熟悉了。

6 注意

每次重启华为云服务器后，在配置文件中均会出现一行新的主机配置， 需要将其注释掉，不然会报如下错。



解决方法

```
vim /etc/hosts
```