多路处理器计算机教学实验系统

《并行算法实践》

实验指导书

国家高性能计算中心深圳分中心（深圳大学计算机与软件学院）

目　录

第一部分　教学实验环境介绍

1. 实验平台系统介绍

多路处理器计算机教学实验系统是由龙芯 3A 处理器构成的、内可配置外可扩展结构的实验硬件平台。该实验平台由多路处理器教学实验箱和一台文件系统服务器组成。多路处理器教学实验箱上有对称的 4 个处理单元，每个处理单元包含一颗龙芯 3A 四核处理器。4 个处理单元既可通过网络互连为多处理机集群架构，也可通过 HT 总线互连为 CC-NUMA 架构。并且多个实验平台可通过网络和 HT 实现更多的处理器互连。文件系统服务器用来统一存放 NFS 系统，为处理单元提供内核和文件系统。在教学中，教师办公用机即可配置为文件系统服务器。实验平台的基本结构和软硬件结构图如下所示。

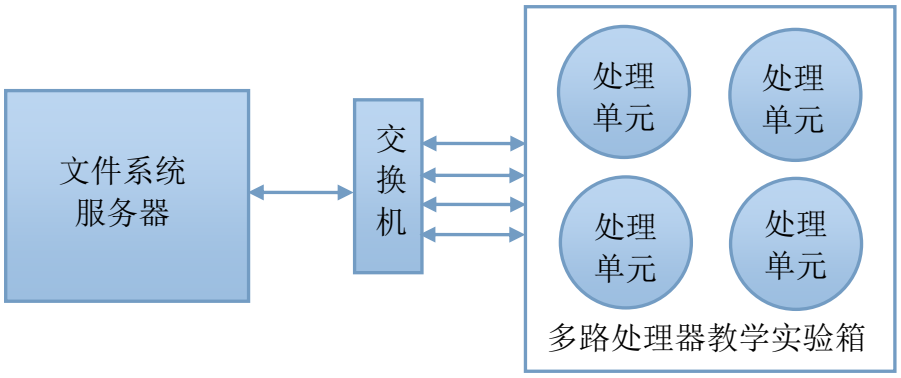


图 1-1　实验平台基本结构图

图 1-2　多路处理器教学实验箱实物图

多路处理器教学实验箱由处理板和控制板两部分组成，整体实物如图 1-2 所示。处理板是实验平台的主要部分，可运行多处理器程序，实现并行程序的设计与开发。控制板主要负责把处理板上的网络、串口等资源扩展成标准接口，同时负责给处理板供电和进行相应的控制。

1.1. 处理板

处理板承载 4 个处理单元，编号如图 1-3 所示，每个处理单元包括一颗龙芯 3A 四核处理器、DDR2 内存（内存数量和单条容量可根据需要进行配置）、RTL8110 千兆以太网卡芯片、BIOS Flash、串口收发芯片以及电源变换电路等。四个龙芯 3A 处理器在处理板上通过 HT 总线实现互连。另外，通过以太网四个处理单元还可以在板外进行互连。处理器板的结构示意图如图 1-3 所示。

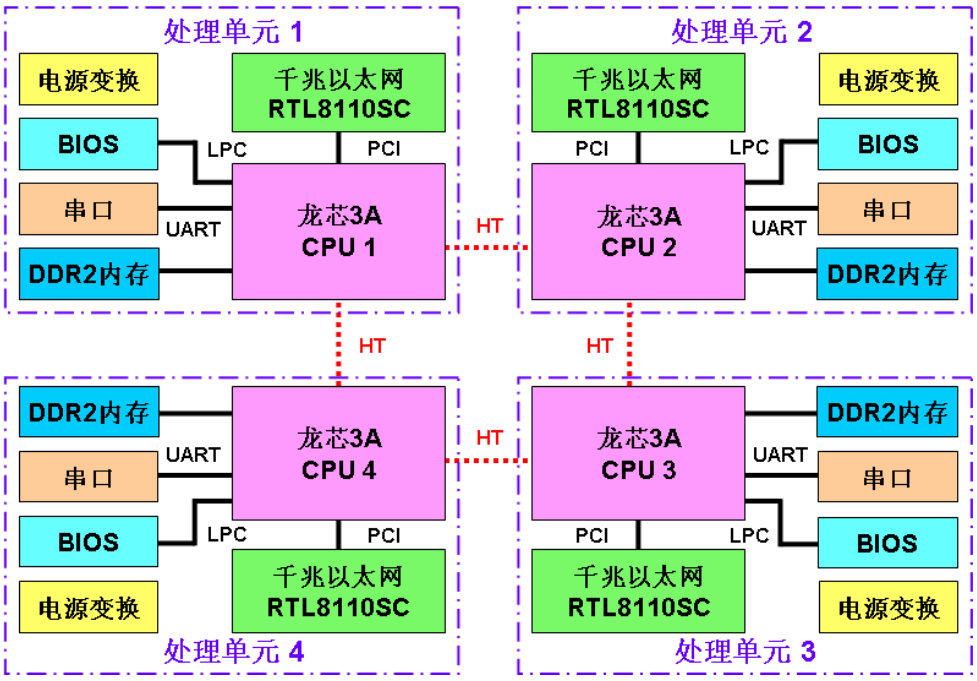


图 1-3　四路龙芯 3A 教学实验箱处理板框图

处理单元使用龙芯 3A 四核处理器，龙芯 3A 四核处理器工作在 825 MHz 时钟频率下，处理器单核的计算能力是 3.3 Gflops，单处理器的计算能力是 13.2 Gflops。PCI 接口工作模式为 32 位，时钟频率 33 MHz，I/O 带宽为 132 MB/s。DDR2 内存的工作频率为 200 MHz，带宽可达 3.2 GB/s。

处理板的设计可满足多种并行层次应用上的需求。在 HT 总线不使用时，实现单处理器结构、SMP 结构和 SMP 集群结构等；在处理器支持 Cache 一致性的

HT 通信并使用 HT 总线的情况下，可以实现 4 路 4 核的 CC-NUMA 结构和 CC-NUMA 集群结构。因此，利用该处理板可以构建实验平台，进行计算机体系结构教学在多种并行层次上的教学实验。

1.2. 控制板

控制板为用户提供了操作主板的接口，控制板主要包括与每个处理器对应的千兆 RJ45 网口、可 4 选 1 的 DB9 串口、一个切换按键功能的摇柄开关、4 个功能按键和一个控制 4 个处理器开关的 AT89S52 单片机，控制板结构如图 1-4 所示：

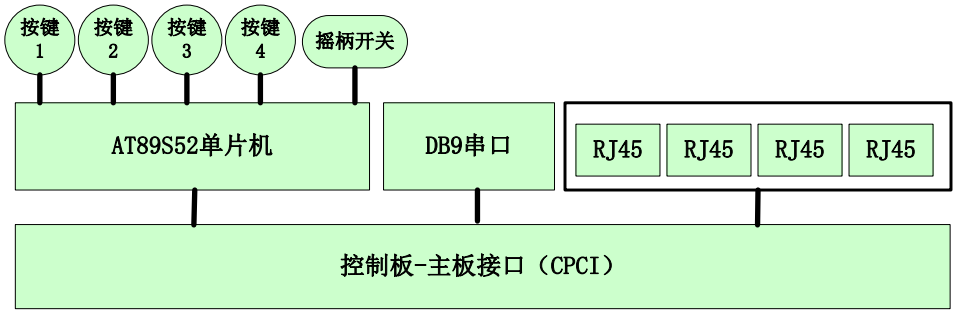


图 1-4　控制板结构

该多路处理器实验系统由处理板和控制板组成，处理板的设计可满足多种并行层次应用上的需求。在 HT 总线不使用时，实现单处理器结构、SMP 结构和 SMP 集群结构等；在处理器支持 Cache 一致性的 HT 通信并使用 HT 总线的情况下，可以实现 4 路 4 核的 CC-NUMA 结构和 CC-NUMA 集群结构。

控制板提供了四个按键开关和一个摇柄开关来操作不同处理器，按键说明见表 2-1。每个按键开关下配置一个 LED 来指示相应的状态，LED 灯指示说明见表 2-2。

注：关机动作需长按相应开关 4 秒。

表 1-1　按键功能说明

摇柄开关	按键	指示灯	按键功能描述
前	红	LED1	控制计算板 4 个处理器的同时开关机
	黑	LED2	控制处理器 3 的独立开关机
	蓝	LED3	控制处理器 2 的独立开关机
	黄	LED4	控制处理器 1 的独立开关机

后	红	LED1	切换处理器 4 串口 1 到控制板串口
	黑	LED2	切换处理器 3 串口 1 到控制板串口
	蓝	LED3	切换处理器 2 串口 1 到控制板串口
	黄	LED4	切换处理器 1 串口 1 到控制板串口

表 1-2　控制板 LED 指示说明

摇柄开关	指示灯	指示灯状态描述	
前	LED1、LED2、LED3、LED4	亮	所对应的处理器处于开机状态
		灭	所对应的处理器处于关机状态
后	LED1、LED2、LED3、LED4	亮	控制板串口连通于所对应的处理器
		灭	/

1.3. 实验平台运行流程

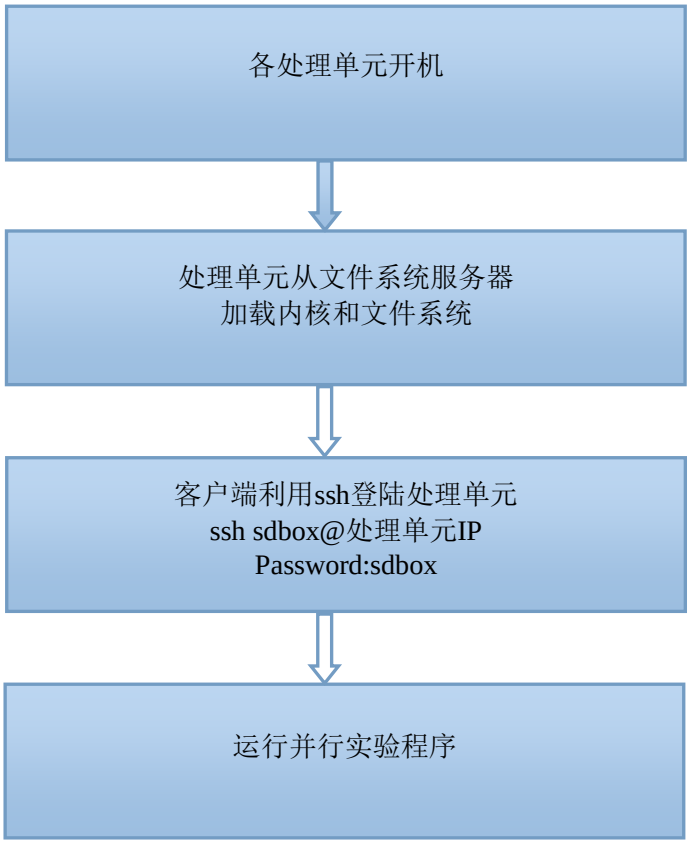


图 1-5　实验平台运行流程

文件系统服务器的配置和多路处理器实验箱的启动流程请参考用户手册。

第二部分 教学实验指导

课 程 名 称 ：	并行算法实践				
英 文 名 称 ：					
设 置 形 式 ：	独立设课			课 程 模 块 ：	专业核心课
实 验 课 性 质 ：	专业实验			课 程 编 号 ：	
一、学时、学分					
课程总学时：		实验学时：		课程学分：	

1. 教学实验要求及内容

一、 适用专业及年级

高等学校计算机及相关专业的本科高年级学生和研究生。

二、 课程目标与基本要求

通过本课程的学习，帮助学生了解并行体系结构特征是如何影响并行程序设计决定的，学会做硬件和软件之间的权衡；理解在并行程序设计过程中所做的决定是如何影响体系结构的运行时特征的；了解并行程序性能的关键因素在许多问题中是如何表现出来的，如何为他们要求解的问题选择并行机，如何在一个具体的并行机上开发一个高效能的并行程序。

通过在并行计算机上完成若干典型应用的并行编程实现，掌握并行程序的设计、实现、调试和性能优化技术。本课程全面系统地介绍现代并行计算机系统上并行程序设计模型、开发方法、语言、编译和调试技术、环境和工具，从大量实例入手，渐近地展开并行程序设计的各个技术层面。

三、 主要仪器设备

龙芯 4 核并行实验平台。

四、 实验项目及教学安排

序号	实验项目名称	实验基本方法和内容	项目学时	项目类型	每组人数	教学要求
1	LU 分解的 OpenMP 实现	编写 LU 分解的 OpenMP 程序				必修
2	KMP 算法的 OpenMP 实现	编写 KMP 算法的 OpenMP 程序				必修
3	高斯消元法解线性方程组的 OpenMP 实现	编写高斯消元法解线性方程组的 OpenMP 程序				必修

4	计算 pi 值的 OpenMP 实现	编写计算 pi 值的 OpenMP 程序				必修
5	高斯消元法解线性方程组的 MPI 实现	编写高斯消元法解线性方程组的 MPI 程序				必修
6	约当消元法解线性方程组的 MPI 实现	编写约当消元法解线性方程组的 MPI 程序				必修
7	雅可比迭代法解线性方程组的 MPI 实现	编写雅可比迭代法解线性方程组的 MPI 实现				必修
8	LU 分解的 MPI 实现	编写 LU 分解的 MPI 程序				必修
9	随机串匹配算法的 MPI 实现	编写随机串匹配算法的 MPI 程序				必修
10	顶点倒塌算法求连通分量的 MPI 实现	编写顶点倒塌算法的 MPI 程序				必修
11	快速排序算法的 MPI 实现	编写快速排序算法的 MPI 程序				必修
12	KVM 虚拟机计算性能测试	测试 KVM 虚拟机的计算性能				必修
13	KVM 虚拟机网络性能测试	测试 KVM 虚拟机用户网络模式下的性能				必修

五、 考核方式及成绩评定

- **考核方式**

1、实验平时成绩：

（1）实验考勤：每次考勤分出勤（2 分）；请假、迟到、早退（1 分）；旷课（0 分）记分。

（2）预习报告：要求写明实验目的、主要实验设备名称、实验原理和内容。分优秀（4 分）、良好（3 分）、中等（2 分）、及格（1 分）和不及格（0 分）记分。

（3）实验报告：要求写明实验设备名称和型号、实验步骤、实验分析及注意事项。分优秀（4 分）、良好（3 分）、中等（2 分）、及格（1 分）和不及格（0 分）记分。

2、实验考试：上机考试，满分为 100 分。

- **成绩评定**

总实验成绩占本课程成绩的 20%。

总实验成绩 = 实验平时成绩×50%+实验考试成绩×50%。

六、 实验教科书、参考书

1．实验教科书

2．实验参考书

实验一、LU 算法的 OpenMP 实现

一、 实验目的

在多核实验平台上使用 OpenMP 编程技术实现 LU 矩阵分解算法，从而达到理解多核编程和 LU 并行分解算法的目的。

二、 实验原理

在线性代数中， LU 分解是矩阵分解的一种，可以将一个矩阵分解为一个下三角矩阵和一个上三角矩阵的乘积（有时是它们和一个置换矩阵的乘积）。LU 分解主要应用在数值分析中，用来解线性方程、求反矩阵或计算行列式。

在 LU 分解的过程中，主要的计算是利用主行 i 对其余各行 $j\ (j>i)$作初等行变换，各行计算之间没有数据相关关系，因此可以对矩阵按行划分来实现并行计算。考虑到计算过程中处理器负载的均衡，对矩阵采用行交叉划分；假设处理器个数为 p，矩阵的阶数为 n，则每个处理器处理的行数为 $m=\lceil n/p \rceil$ 。

OpenMP 是基于线程的编程模型，设计基于多线程的 OpenMP 的 LU 分解算法，其主要思想为：外层设置一个列循环，在每次循环中开设 THREAD_NUMS 个线程，每个线程处理的矩阵 A 的行为上述的 m，一次循环过后则完成对应列的变换，这样在 n 次循环过后便可完成矩阵 A 的 LU 分解。即 L 为 $A[k][j]$ 中 $k>j$ 的元素，其对角线上元素为 1，其它为 0，U 为 $A[k][j]$ 中 $k\le j$ 的元素，其余为 0。

这里如果使用的是一般的 pthread 多线程编程，则在开启 THREAD _NUMS 个线程后，在下次循环开始之前，需要手动配置等待线程同步，不然可能出现错误。但由于 OpenMP 使用 Fork-Join 并行执行模型，其会在线程队执行完以后才转到主线程执行，所以不需要等待线程同步。

三、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 OpenMP 并行程序。相关操作请参考实验环境配套手册。

四、 实验要求与内容

a) 根据实验原理描述的背景，编写基于 C 语言的 n 阶非奇异方阵做 LU 分解。

b) 在教学实验平台上编译、运行、调试上述程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 lu_bx.c：

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <omp.h>
#define MaxN 1010
#define infilename "LU.in"
#define outfilename "LU_bx.out"
#define NUM_OF_THREADS 4

FILE *fin,*fout;                                    //fin 为输入文件  fout 为输出文件
double A[MaxN][MaxN];                               //A 为原矩阵
double L[MaxN][MaxN],U[MaxN][MaxN];                 //L 和 U  为分解后的矩阵
int n;                                                    //n 为矩阵行数
int nthreads,tid;

int init()
{
      int i,j;
      fscanf(fin,"%d",&n);
      for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                  fscanf(fin,"%lf",&A[i][j]);
      omp_set_num_threads(NUM_OF_THREADS);
      for (i=0; i<n; i++)
            for (j=0; j<n; j++)
            {
                  L[i][j]=0;
                  U[i][j]=0;
            }
      for (i=0; i<n; i++) L[i][i]=1;
      return 0;
}

int factorize()
{
      int i,j,k;
#pragma omp parallel shared(A) private(tid,i,j,k)
      {
            for (k=0; k<n; k++)
            {
#pragma omp for
                  for (i=k+1; i<n; i++)
                        A[i][k]=A[i][k]/A[k][k];
#pragma omp for
                  for (i=k+1; i<n; i++)
                        for (j=k+1; j<n; j++)
                        {
                              A[i][j]=A[i][j]-A[i][k]*A[k][j];
                        }
            }
      }
      for (i=0; i<n; i++)
```

```
            for (j=0; j<n; j++)
            {
                if (i<=j) U[i][j]=A[i][j];
                else L[i][j]=A[i][j];
            }
    return 0;
}

int output()
{
    int i,j;
    //输出 L 矩阵
    fprintf(fout,"Matrix L:\n");
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            fprintf(fout,"%.10lf%c",L[i][j],(j==n-1)?'\n':' ');
    //输出 U 矩阵
    fprintf(fout,"Matrix U:\n");
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            fprintf(fout,"%.10lf%c",U[i][j],(j==n-1)?'\n':' ');
    return 0;
}

int main()
{
    double ts,te;
    fin=fopen(infilename,"r");
    fout=fopen(outfilename,"w");
    //初始化
    init();
    //矩阵分解
    ts = omp_get_wtime();
    factorize();
    te = omp_get_wtime();
    printf("time = %f s\n", te - ts);
    //输出结果
    output();
    fclose(fin);
    fclose(fout);
    return 0;
}
```

编译：`gcc -o lu_bx lu_bx.c -fopenmp`
运行：`./lu_bx`

实验结果：
对 2000 阶的非奇异方阵进行 LU 分解，运行时间如下表所示：

线程数	1	4	8	16
运行时间	648.71 s	147.47 s	77.339 s	59.397 s

前三组数据显示，运行时间随线程数的增加同比例减小。由于运算规模较小，运行 16 线程的数据没有同比例减少，是因为多个线程同时到主存处存取数据的时间占程序总运行时间的比例较高，当运算规模加大后，I/O 访问的时间比例降低，运算结果则会出现理想结果。

实验二　KMP 算法的 OpenMP 实现

1、 实验目的

在多核实验平台上使用 OpenMP 编程技术实现 KMP 字符串匹配算法，从而达到理解多核编程和 KMP 算法并行实现的目的。

2、 实验原理及背景

KMP 算法是一种改进的字符串匹配算法。假设文本是一个长度为 n 的数组 $T[1...n]$，模式串是一个长度为 $m \le n$ 的数组 $P[1...m]$，字符串匹配是指找出文本串 T 中与模式串 P 所有精确匹配的子串的起始位置。普通的字符串匹配算法是将主串 T 中某个位置 i 起始的子串和模式串 P 相比较。即从 $j=0$ 起比较 $T[i+j]$ 与 $P[j]$，若相等，则在主串 T 中存在以 i 为起始位置匹配成功的可能性，继续往后比较(j 逐步增 1)，直至与模式串 P 中最后一个字符相等为止，否则改从主串 T 的下一个字符起重新开始进行下一轮的"匹配"，即将模式串 P 向后滑动一位，即 i 增 1，而 j 退回至 0，重新开始新一轮的匹配。普通的字符串匹配算法的时间复杂度为 $O(m*n)$ 。

KMP 算法是一种线性时间复杂的字符串匹配算法，KMP 算法的时间复杂度 $O(strlen(T)+strlen(P))$。KMP 算法中，如果当前字符匹配成功，即 $T[i]==P[j]$，令 $i++$， $j++$，继续匹配下一个字符；如果匹配失败，即 $T[i]!=P[j]$，需要保持 i 不变，并且让 $j=next[j]$，这里 $next[j] \le j-1$，即模式串 P 相对于原始串 T 向右移动了至少 1 位(移动的实际位数 $j-next[j] \ge 1$)，同时移动之后，i 之前的部分（即 $T[i-j+1]$□ $T[i-1]$），和 $j=next[j]$ 之前的部分（即 $p[0]$□ $p[j-2]$）仍然相等。相对于普通的字符串匹配算法来说，KMP 算法移动更多的位数，起到加速的作用。事实上。KMP 匹配算法的关键在于模式串 P 的模式函数值 $next[m]$，它利用了因为模式串 P 本身有前后“部分匹配”的性质，减少回溯的次数。

OpenMP 是基于线程的编程模型，设计基于多线程的 OpenMP 的 KMP 字符串匹配算法，其主要思想为：在计算完模式串 P 的 $next[m]$ 后，开设 THREAD_NUMS 个线程开始字符串匹配算法。设置私有变量 tid，i，j，每个线程都有这个三个

变量的私有副本，并行完成 KMP 字符串匹配算法。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 OpenMP 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写基于 C 语言的 KMP 算法，查找模式串 $p[1,m]$ 在文本串 $T[1,n]$ 中的所有匹配位置。

b) 在教学实验平台上编译、运行、调试上述 OpenMP 并行程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 kmp_bx.c：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <omp.h>
#define MaxN 100000010
#define infilename "KMP.in"
#define outfilename "KMP_parallel.out"
#define NUM_OF_THREADS 4
FILE *fin,*fout;
int n,m;
char S[MaxN],P[MaxN];
int next[MaxN],newnext[MaxN],match[MaxN];
int init()
{
	int i;
	fscanf(fin,"%d%d\n",&n,&m);
	S[0]='.';
	P[0]='.';
	for (i=1;i<=n;i++) fscanf(fin,"%c",&S[i]);
	fscanf(fin,"\n");
	for (i=1;i<=m;i++) fscanf(fin,"%c",&P[i]);
	fscanf(fin,"\n");
	S[n+1]='\0';
	P[n+1]='\0';
	for (i=0;i<=n;i++) match[i]=0;
	return 0;
}
int get_next()
{
	int i,j;
	next[0]=0;
	next[1]=0;
	j=2;
	while (j<=m)
	{
		i=next[j-1];
		while (i!=0&&P[i]!=P[j-1])
			i=next[i];
		next[j]=i+1;
		j=j+1;
	}
	newnext[0]=0;
	newnext[1]=0;
	j=2;
	while (j<=m)
	{
		i=next[j];
		if (i==0||P[j]!=P[i])
			newnext[j]=i;
		else
			newnext[j]=newnext[i];
		j=j+1;
```

```
    }
    return 0;
}

int work()
{
    int size,tid;
    int i,x[128],j,y[128];
    size=(n-1)/NUM_OF_THREADS+1;
#pragma omp parallel private(tid, i, j)
    {

        tid=omp_get_thread_num();
        i=tid*size+1;
        j=1;
        while (i<=(tid+1)*size)
        {
            if (!S[i]) break;
            while (j!=0&&P[j]!=S[i])
                j=newnext[j];
            if (j==m)
            {
                match[i-(m-1)]=1;
                j=next[m+1];
                i++;
            }
            else
            {
                i++;
                j++;
            }
        }
        i=(tid+1)*size-m;
        j=1;
        while (i<=(tid+1)*size+m-1)
        {
            if (!S[i]) break;
            while (j!=0&&P[j]!=S[i])
                j=newnext[j];
            if (j==m)
            {
                match[i-(m-1)]=1;
                j=next[m+1];
                i++;
            }
            else
            {
                i++;
                j++;
            }
        }
    }
    return 0;
}

int output()
{
```

```
    int i;
    for (i=1;i<=n;i++)
    {
        if (match[i]==1)
            fprintf(fout,"match[%d] = %d\n",i,match[i]);
    }
    return 0;
}

int main()
{
    double ta,tb;
    fin=fopen(infilename,"r");
    fout=fopen(outfilename,"w");
    omp_set_num_threads(NUM_OF_THREADS);
    init();
    get_next();
    ta = omp_get_wtime();
    work();
    tb = omp_get_wtime();
    output();
    printf("time = %f s\n",tb-ta);
    fclose(fin);
    fclose(fout);
    return 0;
}
```

编译：`gcc -o kmp_bx kmp_bx.c -fopenmp`
运行：`./kmp_bx`

实验结果：
对长度为 100000000 的文本串，长度为 100 的模式串进行匹配。运行时间如下表所示：

线程数	1	4	8	16
运行时间	1.5693 s	0.36305 s	0.21547 s	0.16804 s

程序开启 1-4 线程时，运行时间随线程数的增加同比例减小。增大运算规模，降低 SMP 模式下各内核访问主存 I/O 资源的时间比例，采用 4-16 线程运行本程序，运行时间同样将随线程数的增加同比例减小。

实验三、高斯消元法解线性方程组的 OpenMP 实现

1、 实验目的

在多核实验平台上使用 OpenMP 编程技术实现高斯消元法解线性方程组，从而达到理解多核编程和解线性方程组并行求解的目的。

2、 实验原理

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \qquad\qquad\qquad \cdots\cdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases} \qquad (1)$$

数学上，高斯消元法是线性代数中的一个算法，可用求解形如（1）的线性方程组 $Ax = b$，其中 A 为 n 阶非奇异阵，其各阶主子行列式不为零， x， b 为 n 维向量。可按照以下三个步骤求出解集：① 建立增广矩阵，将方程组的系数和常数取出，得到如下矩阵：$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{bmatrix}$。

②消元，对增广矩阵反复进行初等变换，即将某一行乘上一个数加到另一行上，得到一个这样的一个三角矩阵：$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} & d_1 \\ 0 & c_{22} & \cdots & c_{2n} & d_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & c_{nn} & d_n \end{bmatrix}$。即可得到一个与原方程同解的方程组：

$$\begin{cases} c_{11}x_1 + c_{12}x_2 + \ldots + c_{1n}x_n = d_1 \\ \qquad c_{22}x_2 + \ldots + c_{2n}x_n = d_2 \\ \qquad\qquad \cdots\cdots \\ \qquad\qquad c_{nn}x_n = d_n \end{cases} \qquad (2)$$

③回代，由方程组（2）的最后一个方程，可以得到 x_n 的值，将 x_n 带入倒数

第二个方程可以得到x_{n-1}的值，以此类推，我们可以求出解集$\{x_1, x_2, \ldots, x_n\}$。

高斯消元法解线性方程组实质上分成两部分：一是利用列运算将增广矩阵代成上三角矩阵；二是回代求解。矩阵的基本列运算规则为：（1）任一列均可乘以一非零的常数；（2）将任一列乘以一常数后加到其他列；（3）可任意对调任意两列。OpenMP 是基于线程的编程模型，设计基于多线程的 OpenMP 的高斯消元法解线性方程组，其主要思想为：在对循环进行初等变换的过程中，开设 THREAD_NUMS 个线程，每个线程同时处理的增广矩阵的行变换。由于在运算过程中需要求矩阵行数据的最大值，采用多线程计算的数值不一定是真实的最大值，所以只能用单线程执行。故使用 OpenMP 实现高斯消元法解线性方程组，多线程和单线程交替进行，单线程完成只有唯一解的运算。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 OpenMP 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写基于 C 语言程序，实现对于给定的系数矩阵$A_{n\times n}$，常数向量$b_{n\times 1}$，求出解向量$x_{n\times 1}$。

b) 在教学实验平台上编译、运行、调试上述 OpenMP 并行程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 gauss_bx.c ：

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <omp.h>
#define MaxN 1010
#define infilename "Gauss.in"
#define outfilename "Gauss_bx.out"
#define NUM_OF_THREADS 2
#define FS 8

FILE *fin,*fout;                          //fin 为输入文件  fout 为输出文件
double A[MaxN][MaxN],b[MaxN];             //A 为系数矩阵，b 为常数向量
double x[MaxN];                           //x 为方程组的解
int n;                                    //n 为矩阵行数
int nthreads,tid;

int init()
{
    int i,j;
    fscanf(fin,"%d",&n);
    //读入系数矩阵 A
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            fscanf(fin,"%lf",&A[i][j]);
    //读入常数向量 b
    for (i=0; i<n; i++)
        fscanf(fin,"%lf",&b[i]);
    return 0;
}

int solve()
{
    int i,j,k,js,l,t,index;
    int shift[MaxN];
    double tmp[MaxN];
    int pos[MaxN];
    double td,d;
    double tmax[NUM_OF_THREADS*FS];
    int ti[NUM_OF_THREADS*FS],tj[NUM_OF_THREADS*FS];

    for (i=0; i<n; i++)
    {
        shift[i]=i;
        pos[i]=i;
    }

#pragma omp parallel shared(A,b,tmax,ti,tj) private(tid,index,i,j,td,k)
    {
        tid=omp_get_thread_num();
        index = tid*FS;
        for (k=0; k<n; k++)
        {
```

```
#pragma omp single
        {
            d=0;
            for (i=0; i<NUM_OF_THREADS*FS; i += FS)
            {
                tmax[i]=0;
                ti[i]=-1;
                tj[i]=-1;
            }
        }

#pragma omp for
        for (i=k; i<n; i++)
            for (j=k; j<n; j++)
            {
                td=fabs(A[i][j]);
                if (td>tmax[index])
                {
                    tmax[index]=td;
                    tj[index]=j;
                    ti[index]=i;
                }
            }
#pragma omp single
        {
            for (i=0; i<NUM_OF_THREADS*FS; i += FS)
                if (tmax[i]>d)
                {
                    d=tmax[i];
                    js=tj[i];
                    l=ti[i];
                }
            if (js!=k)
            {

                for (i=0; i<n; i++)
                {
                    d=A[i][k];
                    A[i][k]=A[i][js];
                    A[i][js]=d;
                }
                t=shift[k];
                shift[k]=shift[js];
                shift[js]=t;
            }
            if (l!=k)
            {
                for (j=k; j<n; j++)
                {
                    d=A[k][j];
                    A[k][j]=A[l][j];
                    A[l][j]=d;
                }
                d=b[k];
                b[k]=b[l];
                b[l]=d;
            }
```

```
                    for (j=k+1; j<n; j++)
                    {
                        A[k][j]=A[k][j]/A[k][k];
                    }

                    b[k]=b[k]/A[k][k];
                    A[k][k]=1;
                }
#pragma omp for
                for (i=k+1; i<n; i++)
                {
                    for (j=k+1; j<n; j++)
                        A[i][j]=A[i][j]-A[i][k]*A[k][j];
                    b[i]=b[i]-A[i][k]*b[k];
                    A[i][k]=0;
                }
            }
        }
        for (i=n-1; i>=0; i--)
        {
            tmp[i]=b[i];
            for (j=i+1; j<n; j++)
                tmp[i]-=A[i][j]*tmp[j];
        }
        for (i=0; i<n; i++)
            pos[shift[i]]=i;
        for (i=0; i<n; i++)
            x[i]=tmp[pos[i]];
        return 0;
}

int output()
{
        int i;
        for (i=0; i<n; i++)
            fprintf(fout,"%.10lf%c",x[i],(i==n-1)?'\n':' ');
        return 0;
}
int main()
{
        int ta,tb,tc;
        double ts, te;
        fin=fopen(infilename,"r");
        fout=fopen(outfilename,"w");
        omp_set_num_threads(NUM_OF_THREADS);
        init();
        ts = omp_get_wtime();
        solve();
        te = omp_get_wtime();
        printf("time = %f s\n", te-ts);
        output();
        fclose(fin);
        fclose(fout);
        return 0;
}
```

编译：`gcc -o gauss_bx gauss_bx.c -fopenmp`
运行：`./gauss_bx`

实验结果：
解 2000 元的线性方程组，运行时间如下表所示：

线程数	1	4	8	16
运行时间	877.67 s	228.79 s	165.18 s	130.99 s

程序开启 1-4 线程时，运行时间随线程数的增加同比例减小。增大运算规模，降低 SMP 模式下各内核访问主存 I/O 资源的时间比例，采用 4-16 线程运行本程序，运行时间同样将随线程数的增加同比例减小。

实验四、计算π值的 OpenMP 实现

1、 实验目的

在多核实验平台上使用 OpenMP 编程技术实现π值的计算，从而达到理解多核编程和并行分解法计算π值的目的。

2、 实验原理

π，又称圆周率，是指平面上的圆的周长与直径的比值，古今中外，有许多数学家致力于圆周率的研究与计算。有割圆法，用圆的内接或外切正多边形来逼近圆的周长，从而计算π值。但这种计算圆周率的方法精确度较低，只能计算到 3.15926 与 3.15927 之间。1706 年，英国天文学教授 John Machin 发现计算π的公式：

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots + (-1)^{n-1}\frac{x^{2n-1}}{2n-1}$$

利用这个公式可以计算出 100 位的圆周率。Machin 公式每计算一项可以得到 1.4 位的十进制精度。因为它的计算过程中被乘数和被除数都不大于长整数，所以可以很容易地在计算机上编程实现。本实验计算π值，就是利用 Machin 公式，计算下式：

π≈1 - 1/3 + 1/5 - 1/7 + 1/9 + 1/11 + 1/13 ...

使用 OpenMP 编程技术实现π值的计算，在循环叠加的运算过程中，开设 THREAD_NUMS 个线程，以线程数为间隔，每个线程计算对应分量，最后累加所有分量，得到π值，并使得π值满足设定的精确度。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 OpenMP 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写基于 C 语言程序，计算π值。
b) 在教学实验平台上编译、运行、调试上述程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 pi_bx.c：

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 1e9;
double step;
#define NUM_THREADS 4
#define FS 8
int main()
{
	int i;
	double x, pi, sum[NUM_THREADS*FS], ts, te;
	step = 1.0/(double)num_steps;
	omp_set_num_threads(NUM_THREADS);
	ts = omp_get_wtime();
#pragma omp parallel private(i)
	{
		double x;
		int id;
		id = omp_get_thread_num();
		for ( i = id, sum[id*FS] = 0.0; i < num_steps; i += NUM_THREADS )
		{
			x = ( i+0.5 )*step;
			sum[id*FS] += 4.0/(1.0+x*x);
		}
	}
	for ( i = 0, pi = 0.0; i < NUM_THREADS; i++ )
	{
		pi += sum[i*FS]*step;
	}
	te = omp_get_wtime();
		printf("pi = %.15f\ntime = %f s\n", pi, te - ts );
	return 1;
}
```

编译：`gcc -o pi_bx pi_bx.c -fopenmp`
运行：`./pi_bx`

实验结果：
运行时间如下表所示：

线程数	1	4	8	16
运行时间	159.81 s	39.806 s	17.127 s	8.6713 s

程序开启 1-16 线程时，运行时间随线程数的增加同比例减小。

实验五、高斯消元法解线性方程组的 MPI 实现

1、 实验目的

在多核实验平台上使用 MPI 编程技术实现高斯消元法解线性方程组，理解基于消息传递模型的并行程序设计思想。

2、 实验原理

MPI（Messge Passing Interface）是一个基于消息传递的并行编程工具，用于开发基于消息传递的并行程序，提供实际可用的、可移植的、高效的消息传递接口库，支持 C 语言。MPI 适用于分布式存储系统的并行编程，系统通过互连网络将多个处理器连接起来，每个处理器均有自己的局部存储器，所有的局部存储器构成整个地址空间，系统中各局部存储器独立编址，用户程序空间则是多地址空间，远程存储器的访问通过消息传递库程序，即 MPI 实现。

本实验采用 MPI 编程技术实现高斯消元法解线性方程组。实验原理参见实验三的详细讲解。高斯消去法求解向量 $x_{n\times 1}$ 是利用主行 i 对其余各行 j（$j>i$）作初等行变换，各行计算之间没有数据相关关系，可对矩阵 A 按行划分。考虑在计算过程中处理器件的负载均衡，对 A 采用行交叉划分。设处理器个数为 p，矩阵 A 的阶数为 n，$m=\lceil n/p \rceil$，对矩阵 A 行交叉划分后，编号为 i 的处理器含有 A 的第 i，$i+p$，…，$i+(m-1)p$ 行和向量 B 的第 i，$i+p$，…，$i+(m-1)p$ 一共 m 个元素。

消去过程的并行是依次以第 0，1，…，$n-1$ 行作为主行进行消去计算。在每次消去计算前，各处理器并行求其局部存储器中右下角阶子阵的最大元。然后通过扩展收集操作将局部存储器中的最大元按处理器编号连接起来并广播给所有处理器，各处理器以此求得整个矩阵右下角阶子阵的最大元及其所在行号、列号和处理器编号。在消去计算中，首先对主行元素作归一化操作，然后将主行广播给所有处理器，各处理器利用接收到的主行元素对其部分行向量做行变换。

回代过程的并行是按 $x_n, x_{n-1}, \cdots, x_1$ 的顺序由各处理器依次计算，一旦 x_k 被计算出来就立即广播给所有处理器，用于与对应项相乘并做求和计算。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 MPI 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写基于 C 语言程序，实现对于给定的系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$，求出解向量 $x_{n\times 1}$。

b) 在教学实验平台上编译、运行、调试上述 MPI 并行程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 gauss.c：

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#include "math.h"
#define a(x,y) a[x*M+y]
#define b(x) b[x]
#define A(x,y) A[x*M+y]
#define B(x) B[x]
#define floatsize sizeof(float)
#define intsize sizeof(int)
int M;
int N;
int m;
float *A;
float *B;
double starttime;
double time1;
double time2;
int my_rank;
int p;
int l;
MPI_Status status;

void fatal(char *message)
{
	printf("%s\n",message);
	exit(1);
}

void Environment_Finalize(float *a,float *b,float *x,float *f)
{
	free(a);
	free(b);
	free(x);
	free(f);
}

int main(int argc, char **argv)
{
	int i,j,t,k,my_rank,group_size;
	int i1,i2;
	int v,w;
	float temp;
	int tem;
	float *sum;
	float *f;
	float lmax;
	float *a;
	float *b;
	float *x;
	int *shift;
	FILE *fdA,*fdB;
```

```
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&group_size);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    p=group_size;

    if (my_rank==0)
    {
        starttime=MPI_Wtime();

        fdA=fopen("dataIn.txt","r");
        fscanf(fdA,"%d %d", &M, &N);
        if (M != N-1)
        {
            printf("the input is wrong\n");
            exit(1);
        }

        A=(float *)malloc(floatsize*M*M);
        B=(float *)malloc(floatsize*M);

        for(i = 0; i < M; i++)
        {
            for(j = 0; j < M; j++)
            {
                fscanf(fdA,"%f", A+i*M+j);
            }
            fscanf(fdA,"%f", B+i);
        }
        fclose(fdA);
    }

    MPI_Bcast(&M,1,MPI_INT,0,MPI_COMM_WORLD);        /* 0 号处理机将 M 广播给所有处理机 */
    m=M/p;
    if (M%p!=0) m++;

    f=(float*)malloc(sizeof(float)*(M+1));           /* 各处理机为主行元素建立发送和接收缓冲区(M+1) */
    a=(float*)malloc(sizeof(float)*m*M);             /* 分配至各处理机的子矩阵大小为 m*M */
    b=(float*)malloc(sizeof(float)*m);               /* 分配至各处理机的子向量大小为 m */
    sum=(float*)malloc(sizeof(float)*m);
    x=(float*)malloc(sizeof(float)*M);
    shift=(int*)malloc(sizeof(int)*M);

    if (a==NULL||b==NULL||f==NULL||sum==NULL||x==NULL||shift==NULL)
        fatal("allocate error\n");

    for(i=0;i<M;i++)
        shift[i]=i;
    /*
     0 号处理机采用行交叉划分将矩阵 A 划分为大小为 m*M 的 p 块子矩阵,将 B 划分为大小
     为 m 的 p 块子向量，依次发送给 1 至 p-1 号处理机
    */
    if (my_rank==0)
```

```
    {
        for(i=0;i<m;i++)
            for(j=0;j<M;j++)
                a(i,j)=A(i*p,j);

        for(i=0;i<m;i++)
            b(i)=B(i*p);
    }

    if (my_rank==0)
    {
        for(i=0;i<M;i++)
            if ((i%p)!=0)
        {
            i1=i%p;
            i2=i/p+1;

            MPI_Send(&A(i,0),M,MPI_FLOAT,i1,i2,MPI_COMM_WORLD);
            MPI_Send(&B(i),1,MPI_FLOAT,i1,i2,MPI_COMM_WORLD);
        }
    }                                                    /*    my_rank==0 */
    else                                                 /*    my_rank !=0 */
    {
        for(i=0;i<m;i++)
        {
            MPI_Recv(&a(i,0),M,MPI_FLOAT,0,i+1,MPI_COMM_WORLD,&status);
            MPI_Recv(&b(i),1,MPI_FLOAT,0,i+1,MPI_COMM_WORLD,&status);
        }
    }

    time1=MPI_Wtime();                                   /* 开始计时 */

    for(i=0;i<m;i++)                                     /* 消去 */
        for(j=0;j<p;j++)
    {
        if (my_rank==j)                                  /* j 号处理机负责广播主行元素 */
        {
            v=i*p+j;                                     /* 主元素在原系数矩阵 A 中的行
号和列号为 v */
            lmax=a(i,v);
            l=v;

            for(k=v+1;k<M;k++)                           /* 在同行的元素中找最大元，并确
定最大元所在的列 l */
                if (fabs(a(i,k))>lmax)
            {
                lmax=a(i,k);
                l=k;
            }

            if (l!=v)                                    /* 列交换 */
            {
                for(t=0;t<m;t++)
                {
                    temp=a(t,v);
                    a(t,v)=a(t,l);
```

```
                    a(t,l)=temp;
                }

                tem=shift[v];
                shift[v]=shift[l];
                shift[l]=tem;
            }

            for(k=v+1;k<M;k++)                          /* 归一化 */
                a(i,k)=a(i,k)/a(i,v);

            b(i)=b(i)/a(i,v);
            a(i,v)=1;

            for(k=v+1;k<M;k++)
                f[k]=a(i,k);
            f[M]=b(i);

            /* 发送归一化后的主行 */
            MPI_Bcast(&f[0],M+1,MPI_FLOAT,my_rank,MPI_COMM_WORLD);
            /* 发送主行中主元素所在的列号 */
            MPI_Bcast(&l,1,MPI_INT,my_rank,MPI_COMM_WORLD);
        }
        else
        {
            v=i*p+j;
            MPI_Bcast(&f[0],M+1,MPI_FLOAT,j,MPI_COMM_WORLD);
            MPI_Bcast(&l,1,MPI_INT,j,MPI_COMM_WORLD);

            if (l!=v)
            {
                for(t=0;t<m;t++)
                {
                    temp=a(t,v);
                    a(t,v)=a(t,l);
                    a(t,l)=temp;
                }

                tem=shift[v];
                shift[v]=shift[l];
                shift[l]=tem;
            }
        }

        if (my_rank<=j)
            for(k=i+1;k<m;k++)
        {
            for(w=v+1;w<M;w++)
                a(k,w)=a(k,w)-f[w]*a(k,v);
            b(k)=b(k)-f[M]*a(k,v);
        }

        if (my_rank>j)
            for(k=i;k<m;k++)
        {
            for(w=v+1;w<M;w++)
                a(k,w)=a(k,w)-f[w]*a(k,v);
```

```
            b(k)=b(k)-f[M]*a(k,v);
        }
    }                                                   /* for i j */

    for(i=0;i<m;i++)
        sum[i]=0.0;

    for(i=m-1;i>=0;i--)                                 /*  回代  */
        for(j=p-1;j>=0;j--)
            if (my_rank==j)
            {
                x[i*p+j]=(b(i)-sum[i])/a(i,i*p+j);

                MPI_Bcast(&x[i*p+j],1,MPI_FLOAT,my_rank,MPI_COMM_WORLD);

                for(k=0;k<i;k++)
                    sum[k]=sum[k]+a(k,i*p+j)*x[i*p+j];
            }
            else
            {
        MPI_Bcast(&x[i*p+j],1,MPI_FLOAT,j,MPI_COMM_WORLD);

        if (my_rank>j)
            for(k=0;k<i;k++)
                sum[k]=sum[k]+a(k,i*p+j)*x[i*p+j];

        if (my_rank<j)
            for(k=0;k<=i;k++)
                sum[k]=sum[k]+a(k,i*p+j)*x[i*p+j];
    }

    if (my_rank!=0)
        for(i=0;i<m;i++)
            MPI_Send(&x[i*p+my_rank],1,MPI_FLOAT,0,i,MPI_COMM_WORLD);
    else
        for(i=1;i<p;i++)
            for(j=0;j<m;j++)
                MPI_Recv(&x[j*p+i],1,MPI_FLOAT,i,j,MPI_COMM_WORLD,&status);

    if (my_rank==0)
    {
        printf("Input of file \"dataIn.txt\"\n");
        printf("%d\t%d\n", M, N);
        for(i=0;i<M;i++)
        {
            for(j=0;j<M;j++) printf("%f\t",A(i,j));
            printf("%f\n",B(i));
        }
        printf("\nOutput of solution\n");
        for(k=0;k<M;k++)
        {
            for(i=0;i<M;i++)
            {
                if (shift[i]==k) printf("x[%d]=%f\n",k,x[i]);
            }
        }
    }
```

```
    time2=MPI_Wtime();

    if (my_rank==0)
    {
        printf("\n");
        printf("Whole running time       = %f seconds\n",time2-starttime);
        printf("Distribute data time    = %f seconds\n",time1-starttime);
        printf("Parallel compute time = %f seconds\n",time2-time1);
    }

    MPI_Finalize();
    Environment_Finalize(a,b,x,f);
    return(0);
}
```

编译：`mpicc gauss.c -o gauss`
运行：`mpirun -np 4 ./gauss`

实验结果：
解 2000 元的线性方程组，运行时间如下表所示：

线程数	1	4	8	16
运行时间	174.92 s	42.871 s	28.931 s	22.292 s
数据分配时间	10.525 s	11.436 s	11.408 s	11.442 s
并行计算时间	164.39 s	31.434 s	17.521 s	10.849 s

程序总的运行时间等于数据分配时间与并行计算时间之和，相同的实验数据占用的分配时间基本一致。分析并行计算时间，程序开启 1-16 线程时，并行计算时间基本随线程数的增加线性减小。部分偏差是由于网络交互时间造成，加大运算规模，可以得到理想数据，也就是并行计算时间随线程数的增加同比例减小。

实验六、约当消元法解线性方程组的 MPI 实现

1、 实验目的

在多核实验平台上使用 MPI 编程技术实现约当消元法解线性方程组，理解基于消息传递模型的并行程序设计思想。

2、 实验原理

约当消去法(Jordan Elimination)是一种无回代过程的直接解法，它直接将系数矩阵 A 变换为单位矩阵，经变换后的常数向量 b 即是方程组的解。这种消去法的基本过程与高斯消去法相同，只是在消去过程中，不但将主对角线以下的元素消成 0，而且将主对角线以上的元素也同时消成 0。一般约当消去法的计算过程是按 $i=1,2,...,n$ 的顺序，逐次以第 i 行作为主行进行消去，以消去第 i 列除主元素以外的所有元素。

实际上，约当消去法解线性方程组是把一个非奇异矩阵 A 变成了单位矩阵 I，也就是相当于在 A 的左边乘上了 A^{-1}，即：$(A|b)\xrightarrow{\text{左乘}A^{-1}}(A^{-1}A|A^{-1}b)=(I|A^{-1}b)$

$A^{-1}b=x$ 为线性方程组 $Ax=b$ 的解。

本实验采用 MPI 编程技术实现约当消元法解线性方程组。约当消去法采用与高斯消去法相同的数据划分和选主元的方法。在并行消去过程中，首先对主行元素作归一化操作，然后将主行广播给所有处理器，各处理器利用接收到的主行元素对其部分行向量做行变换。若以编号为 my_rank 的处理器的第 i 行作为主行，在归一化操作之后，将它广播给所有处理器，则编号不为 my_rank 的处理器利用主行对其第 $1,2,...,m-1$ 行数据和子向量做变换，编号为 my_rank 的处理器利用主行对其除 i 行以外的数据和子向量做变换（第 i 个子向量除外）。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 MPI 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写基于 C 语言程序，实现对于给定的系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$，采用约当消元法求解解向量 $x_{n\times 1}$。

b) 在教学实验平台上编译、运行、调试上述 MPI 并行程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 jordan.c：

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#include "math.h"

/* somethings convinient for programming use */
#define a(x,y) a[x*M+y]
#define b(x) b[x]
#define A(x,y) A[x*M+y]
#define B(x) B[x]
#define floatsize sizeof(float)
#define intsize sizeof(int)

/* M*M matrix */
int M;

/* dimision for per node */
int m;

/* A, B array pointer */
float *A;
float *B;

/* time starting to compute, and other virable for time use */
double starttime;
double time1;
double time2;

/* node rank */
int my_rank;

/* size of nodes */
int p;

int l;
MPI_Status status;

/*
  *Function: fatal()
  *Desc:         output some message for user to know, and exit
  *Param:        message-char pointer to msg going to print on the screen
  */
void
fatal(char *message)
{
        printf("%s\n",message);
        exit(1);
}

/*
  *Function: Environment_Finalize()
  *Desc:         release some pointer before all be finished
  *Param:        a,b,x,f - float pointer
```

```
 *
 */
void
Environment_Finalize(float *a,float *b,float *x,float *f)
{
	free(a);
	free(b);
	free(x);
	free(f);
}

/*
 *Function:    main()
 *Desc:          program entry
 *Param:
 */
int
main(int argc, char **argv)
{
	int i,j,t,k,my_rank,group_size;
	int i1,i2;
	int v,w;
	float s;
	float temp;
	int tem;
	float *f;
	float lmax;
	float *a;
	float *b;
	float *x;
	int *shift;
	FILE *fdA,*fdB;

	/* initialize MPI , group_size, my_rank */
	MPI_Init(&argc,&argv);
	MPI_Comm_size(MPI_COMM_WORLD,&group_size);
	MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
	if(my_rank==0)
		printf("group size: %d\n",group_size);

	/* node number */
	p=group_size;

	/*
	  for the node which my_rank==0, set starttime
	  and read A,B elements from file"dataIn.txt",
	  p.s. M=N-1
	*/
	if (my_rank==0)
	{
		starttime=MPI_Wtime();

		fdA=fopen("dataIn.txt","r");

		fscanf(fdA,"%d",&M);

		/* allocate memory for matrix A an B */
```

```
        A=(float *)malloc(floatsize*M*M);
        B=(float *)malloc(floatsize*M);

        /* read A&B elements(M*(M+1)) from the file */
        for(i = 0; i < M; i++)
        {
            /* the 0 to M-1 th element of each row is the element of A */
            for(j = 0; j < M; j++)
            {
                fscanf(fdA,"%f", A+i*M+j);
            }
            /* the last one of each row    is the element of B */
            fscanf(fdA,"%f", B+i);
        }
        fclose(fdA);
    }

    /* broadcast the value of M */
    MPI_Bcast(&M,1,MPI_INT,0,MPI_COMM_WORLD);
    /* allocate area for each node */
    m=M/p;
    if (M%p!=0) m++;

    f=(float*)malloc(sizeof(float)*(M+1));
    a=(float*)malloc(sizeof(float)*m*M);
    b=(float*)malloc(sizeof(float)*m);
    x=(float*)malloc(sizeof(float)*M);
    shift=(int*)malloc(sizeof(int)*M);

    /* initialize shift value for each column */
    for(i=0;i<M;i++)
        shift[i]=i;

    /* node which my_rank==0 appoints the a,b matrix array */
    if (my_rank==0)
    {
        for(i=0;i<m;i++)
            for(j=0;j<M;j++)
                a(i,j)=A(i*p,j);

        for(i=0;i<m;i++)
            b(i)=B(i*p);
    }

    if (my_rank==0)
    {
        for(i=0;i<M;i++)
            if ((i%p)!=0)
        {
            i1=i%p;
            i2=i/p+1;
            /* send A, B elements to node i1(i1!=0) */
            MPI_Send(&A(i,0),M,MPI_FLOAT,i1,i2,MPI_COMM_WORLD);
            MPI_Send(&B(i),1,MPI_FLOAT,i1,i2,MPI_COMM_WORLD);
        }
    }                                                    /*   my_rank==0 */
    else                                                 /*   my_rank !=0 */
```

```
{
    /* receive A,B elements from node 0 */
    for(i=0;i<m;i++)
    {
        MPI_Recv(&a(i,0),M,MPI_FLOAT,0,i+1,MPI_COMM_WORLD,&status);
        MPI_Recv(&b(i),1,MPI_FLOAT,0,i+1,MPI_COMM_WORLD,&status);
    }
}

time1=MPI_Wtime();                                  /* computing start */

for(i=0;i<m;i++)
    for(j=0;j<p;j++)
{
    if (my_rank==j)                                 /* node rank= j*/
    {
        /* find lmax of matrix a on this node */
        v=i*p+j;
        lmax=a(i,v);
        l=v;

        for(k=v+1;k<M;k++)
            if (fabs(a(i,k))>lmax)
        {
            lmax=a(i,k);
            l=k;
        }

        /* exchange the with the max */
        if (l!=v)
        {
            for(t=0;t<m;t++)
            {
                temp=a(t,v);
                a(t,v)=a(t,l);
                a(t,l)=temp;
            }
            /* record the shift in order to output the right x */
            tem=shift[v];
            shift[v]=shift[l];
            shift[l]=tem;
        }

        for(k=v+1;k<M;k++)
            a(i,k)=a(i,k)/a(i,v);

        b(i)=b(i)/a(i,v);
        a(i,v)=1;

        /* record f array for future use */
        for(k=v+1;k<M;k++)
            f[k]=a(i,k);
        f[M]=b(i);

        /* broadcast f & l value to node which rank= my_rank */
        MPI_Bcast(&f[0],M+1,MPI_FLOAT,my_rank,MPI_COMM_WORLD);
        MPI_Bcast(&l,1,MPI_INT,my_rank,MPI_COMM_WORLD);
```

```
        }
        else /*rank!=j*/
        {
            v=i*p+j;
            MPI_Bcast(&f[0],M+1,MPI_FLOAT,j,MPI_COMM_WORLD);
            MPI_Bcast(&l,1,MPI_INT,j,MPI_COMM_WORLD);
        }

        if(my_rank!=j)
            if (l!=v)
        {
            for(t=0;t<m;t++)
            {
                temp=a(t,v);
                a(t,v)=a(t,l);
                a(t,l)=temp;
            }

            tem=shift[v];
            shift[v]=shift[l];
            shift[l]=tem;
        }

        if (my_rank!=j)
            for(k=0;k<m;k++)
        {
            for(w=v+1;w<M;w++)
                a(k,w)=a(k,w)-f[w]*a(k,v);
            b(k)=b(k)-f[M]*a(k,v);
        }

        if (my_rank==j)
            for(k=0;k<m;k++)
                if (k!=i)
                {
                    for(w=v+1;w<M;w++)
                        a(k,w)=a(k,w)-f[w]*a(k,v);
                    b(k)=b(k)-f[M]*a(k,v);
                }
    } /* for i j */

    /* caculate x value at node which rank is zero */
    if (my_rank==0)
        for(i=0;i<m;i++)
            x[i*p]=b(i);

    /* node which rank is not zero send b() value to node zero */
    if (my_rank!=0)
        for(i=0;i<m;i++)
            MPI_Send(&b(i),1,MPI_FLOAT,0,my_rank,MPI_COMM_WORLD);
    else
    {
        /*
          node which rank is    zero recieve b() value from node i
          and caculate x value
        */
        for(i=1;i<p;i++)
```

```
            for(j=0;j<m;j++)
        {
            MPI_Recv(&b(j),1,MPI_FLOAT,i,i,MPI_COMM_WORLD,&status);
            x[j*p+i]=b(j);
        }
    }

    /* write the result to the screen and to file "dataOut.txt" */
    fdA=fopen("dataOut.txt","w");

    if (my_rank==0)
    {
        printf("Input of file \"dataIn.txt\"\n");
        printf("%d\n", M);
        for(i=0;i<M;i++)
        {
            for(j=0;j<M;j++) printf("%f\t",A(i,j));
            printf("%f\n",B(i));
        }
        printf("\nOutput of solution\n");

        fprintf(fdA,"Output of solution\n");

        for(k=0;k<M;k++)
        {
            for(i=0;i<M;i++)
            {
                if (shift[i]==k)
                {
                    printf("x[%d]=%f\n",k,x[i]);
                    fprintf(fdA,"x[%d]=%f\n",k,x[i]);
                }
            }
        }
    }

    time2=MPI_Wtime();

    if (my_rank==0)
    {
        printf("\n");
        printf("Whole running time       = %f seconds\n",time2-starttime);
        printf("Distribute data time    = %f seconds\n",time1-starttime);
        printf("Parallel compute time = %f seconds\n",time2-time1);

        fprintf(fdA,"Whole running time       = %f seconds\n",time2-starttime);
        fprintf(fdA,"Distribute data time    = %f seconds\n",time1-starttime);
        fprintf(fdA,"Parallel compute time = %f seconds\n",time2-time1);
    }
    fclose(fdA);

    /* finalize and release pointer */
    MPI_Finalize();
    Environment_Finalize(a,b,x,f);
    return(0);
}
```

编译：`mpicc jordan.c -o jordan`
运行：`mpirun -np 4 ./jordan`

实验结果：
解 2000 元的线性方程组，运行时间如下表所示：

线程数	1	4	8	16
运行时间	262.86 s	58.115 s	36.617 s	26.269 s
数据分配时间	10.398 s	11.282 s	11.206 s	11.984 s
并行计算时间	252.46 s	46.833 s	25.411 s	14.285 s

程序总的运行时间等于数据分配时间与并行计算时间之和，相同的实验数据占用的分配时间基本一致。分析并行计算时间，程序开启 1-16 线程时，运行计算时间基本随线程数的增加线性减小。偏差是由于网络交互时间造成，加大运算规模，可以得到理想数据，也就是并行计算时间随线程数的增加同比例减小。

实验七、雅可比迭代法解线性方程组的 MPI 实现

1、 实验目的

在多核实验平台上使用 MPI 编程技术实现雅可比迭代法解线性方程组，理解基于消息传递模型的并行程序设计思想。

2、 实验原理

在阶数较大、系数阵为稀疏阵的情况下，可以采用迭代法求解线性方程组。用迭代法求解线性方程组的优点是方法简单，便于编制计算机程序，但必须选取合适的迭代格式及初始向量，以使迭代过程尽快地收敛。雅克比迭代法是其中一种迭代格式，考虑计算机内存和运算这两方面，雅克比迭代法可充分利用稀疏矩阵中大量零元素的特点。

雅可比迭代的原理是：求解 n 阶线性方程组 $Ax=b$，假设系数矩阵 A 的主对角线元素 $a_{ii}\neq 0$，且按行严格对角占优，即：

$$|a_{ii}|>\sum_{j\neq i}^{n}|a_{ii}|\ (i=1,2,...,n)$$

将原方程组的每一个方程 $a_{i1}x_1+a_{i2}x_2+\cdots+a_{in}x_n=b_i$ 改写为未知向量 x 的分量的形式：

$$x_i=(b_i-\sum_{j\neq i}^{n}a_{ij}x_j)/a_{ii}\quad (1\le i\le n)$$

然后使用第 $k-1$ 步所计算的变量 $x_i^{(k-1)}$ 来计算第 k 步的 $x_i^{(k)}$ 的值：

$$x_i^{(k)}=(b_i-\sum_{j\neq i}^{n}a_{ij}x_j^{(k-1)})/a_{ii}\quad (1\le i,\ k\le n)$$

这里，$x_i^{(k)}$ 为第 k 次迭代得到的近似解向量 $x^{(k)}=(x_1^{(k)},x_2^{(k)},\cdots,x_n^{(k)})^T$ 的第 i 个分量。取适当初始解向量 $x^{(0)}$ 代入上述迭代格式中，可得到 $x^{(1)}$，再由 $x^{(1)}$ 得到 $x^{(2)}$，依次迭代下去得到近似解向量序列 $\{x^{(k)}\}$。若原方程组的系数矩阵按行严格对角占优，则 $\{x^{(k)}\}$ 收敛于原方程组的解 x。

本实验采用 MPI 编程技术实现雅可比迭代法解线性方程组。若处理器个数为 p，则对矩阵 A 按行划分。设矩阵被划分为 p 块，每块含有连续的 m 行向量，这

里 $m=\lceil n/p \rceil$，编号为 i 的处理器含有 A 的第 im 至第 $(i+1)m-1$ 行数据，同时向量 b 中下标为 im 至 $(i+1)m-1$ 的元素也被分配至编号为 i 的处理器($i=0,1,\cdots,p-1$)，初始解向量 x 被广播给所有处理器。

迭代计算中，各处理器并行计算解向量 x 的各分量值，编号为 i 的处理器计算分量 x_{im} 至 $x_{(i+1)m-1}$ 的新值。并求其分量中前后两次值的最大差 localmax，然后通过归约操作的求最大值运算求得整个 n 维解向量中前后两次值的最大差 max 并广播给所有处理器。最后通过扩展收集操作将各处理器中的解向量按处理器编号连接起来并广播给所有处理器，以进行下一次迭代计算，直至收敛。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 MPI 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写基于 C 语言程序，实现对于给定的系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$，采用雅可比迭代法求解解向量 $x_{n\times 1}$。

b) 在教学实验平台上编译、运行、调试上述 MPI 并行程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 jacobi.c：

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#include "math.h"
#define E 0.0001
#define a(x,y) a[x*size+y]
#define b(x) b[x]
#define v(x) v[x]
#define v1(x) v1[x]
#define A(x,y) A[x*size+y]
#define B(x) B[x]
#define V(x) V[x]
#define intsize sizeof(int)
#define floatsize sizeof(float)
#define charsize sizeof(char)

int size,N;
int m;                                  /* the size of work on eath processor
*/
float *B;                               /* store matrix B */
float *A;                               /* store matrix A */
float *V;                               /* store the value x */
double starttime;
double time1;
double time2;
int my_rank;                            /* store the identifier of each
processor */
int p;                                  /* store the number of processor */
MPI_Status status;
FILE *fdA,*fdB;

int i,j,group_size;
float sum;
float *b;
float *v;
float *a;
float *v1;
float lmax;                             /* the differnce between v1 and v */
float max=1.0;                          /* max of lmax */
int loop=0;                             /* store the number of iteration */

/*name      Environment_Finalize
  *input    a,b,v,v1
  *usage    free a,b,v,v1
  */
void Environment_Finalize(float *a,float *b,float *v,float *v1)
{
      free(a);
      free(b);
      free(v);
      free(v1);
      if(my_rank==0)
      {
```

```
            free(A);
            free(B);
            free(V);
        }
}

/*name      Input
  *usage    read size,N,A,B,V from input file on the root
  */
void Input()
{
        if(my_rank==0)
        {
            starttime=MPI_Wtime();

            fdA=fopen("dataIn.txt","r");                    /* Read from file "dataIn.txt" */
            fscanf(fdA,"%d %d", &size, &N);
            if (size != N-1)
            {
                printf("the input is wrong\n");
                exit(1);
            }

            A=(float *)malloc(floatsize*size*size);
            B=(float *)malloc(floatsize*size);
            V=(float *)malloc(floatsize*size);

            for(i = 0; i < size; i++)
            {
                for(j = 0; j < size; j++)
                {
                    fscanf(fdA,"%f", A+i*size+j);
                }
                fscanf(fdA,"%f", B+i);
            }
            for(i = 0; i < size; i ++)
            {
                fscanf(fdA,"%f", V+i);
            }

            fclose(fdA);
        }
}

/*name      nodInit
  *usage    allocate spaces for v,a,b,v1 on each processor
  *          and store v value on the root
  */
void nodInit()
{
        m=size/p;if (size%p!=0) m++;
        v=(float *)malloc(floatsize*size);
        a=(float *)malloc(floatsize*m*size);
        b=(float *)malloc(floatsize*m);
        v1=(float *)malloc(floatsize*m);                    /* store v value */

        if (a==NULL||b==NULL||v==NULL||v1==NULL)
```

```
            printf("allocate space fail!");

        if (my_rank==0)
        {
            for(i=0;i<size;i++)
                v(i)=V(i);
        }
}

/*name      Data_send
  *usage    initialize the value of a,b on root and broadcast
  */
void Data_send()
{
        if (my_rank==0)
        {
            for(i=0;i<m;i++)
                for(j=0;j<size;j++)
                    a(i,j)=A(i,j);

            for(i=0;i<m;i++)
                b(i)=B(i);
        }

        /* root sends all other nodes the corresponding data */
        if (my_rank==0)
        {
            for(i=1;i<p;i++)
            {
                MPI_Send(&(A(m*i,0)),m*size,MPI_FLOAT,i,i,MPI_COMM_WORLD);
                MPI_Send(&(B(m*i)),m,MPI_FLOAT,i,i,MPI_COMM_WORLD);
            }
            /* free(A); free(B); free(V); */
        }
        else
        {
            MPI_Recv(a,m*size,MPI_FLOAT,0,my_rank,MPI_COMM_WORLD,&status);
            MPI_Recv(b,m,MPI_FLOAT,0,my_rank,MPI_COMM_WORLD,&status);
        }
}

/*name      Compute
  *usage    compute v1(i) =( b(i)- ¡Æa(i,j) * v(j) ) / a(i,my_rank * m + i)
  *          lmax = abs(v1(i)-v(my_rank * m + i))
  *          collect the max lmax on each processor,if it is greater than E
  *          then continue else end while
  */
void Compute()
{
        time1=MPI_Wtime();

        /* computing start */
        while (max>E)                                   /* The precision requirement */
        {
            lmax=0.0;

            for(i=0;i<m;i++)
```

```
            {
                if(my_rank*m+i<size)
                {
                    sum=0.0;
                    for(j=0;j<size;j++)
                        if (j!=(my_rank*m+i))
                            sum=sum+a(i,j)*v(j);

                                                        /* computes the new elements */
                    v1(i)=(b(i) - sum)/a(i,my_rank * m + i);

                    if (fabs(v1(i)-v(i))>lmax)
                        lmax=fabs(v1(i)-v(my_rank * m + i));
                }
            }

            /*Find the max element in the vector*/
            MPI_Allreduce(&lmax,&max,1,MPI_FLOAT,MPI_MAX,MPI_COMM_WORLD);

            /*Gather all the elements of the vector from all nodes*/
            MPI_Allgather(v1,m,MPI_FLOAT,v,m,MPI_FLOAT,MPI_COMM_WORLD);
            loop++;
        }                                               /* while */
        time2=MPI_Wtime();
}

/*name      Output
  *usage    print messages
  */
void Output()
{
        if (my_rank==0)
        {
            printf("Input of file \"dataIn.txt\"\n");
            printf("%d\t%d\n", size, N);
            for(i = 0; i < size; i ++)
            {
                for(j = 0; j < size; j ++) printf("%f\t",A(i,j));
                printf("%f\n",B(i));
            }
            printf("\n");
            for(i = 0; i < size; i ++)
            {
                printf("%f\t", V(i));
            }
            printf("\n\n");
            printf("\nOutput of solution\n");
            for(i = 0; i < size; i ++) printf("x[%d] = %f\n",i,v(i));
            printf("\n");
            printf("Iteration num = %d\n",loop);
            printf("Whole running time       = %f seconds\n",time2-starttime);
            printf("Distribute data time    = %f seconds\n",time1-starttime);
            printf("Parallel compute time = %f seconds\n",time2-time1);

            fdB=fopen("dataOut.txt","w");
            fprintf(fdB,"Input of file \"dataIn.txt\"\n");
            fprintf(fdB,"%d\t%d\n", size, N);
```

```
            for(i = 0; i < size; i ++)
            {
                for(j = 0; j < size; j ++) fprintf(fdB,"%f\t",A(i,j));
                fprintf(fdB,"%f\n",B(i));
            }
            fprintf(fdB,"\n");
            for(i = 0; i < size; i ++)
            {
                fprintf(fdB,"%f\t", V(i));
            }
            fprintf(fdB,"\n\n");
            fprintf(fdB,"\nOutput of solution\n");
            for(i = 0; i < size; i ++) fprintf(fdB,"x[%d] = %f\n",i,v(i));
            fprintf(fdB,"\n");
            fprintf(fdB,"Iteration num = %d\n",loop);
            fprintf(fdB,"Whole running time       = %f seconds\n",time2-starttime);
            fprintf(fdB,"Distribute data time    = %f seconds\n",time1-starttime);
            fprintf(fdB,"Parallel compute time = %f seconds\n",time2-time1);
        }
}

/*name     main
 *input    argc:parameter number of command lines
 *          argv:array of each command lines' parameter
 *output return 0 if program is running correctly
 */
int main(int argc, char **argv)
{
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&group_size);
        MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
        p=group_size;
        Input();

        /*Broadcast the size of    factor Matrix*/
        MPI_Bcast(&size,1,MPI_INT,0,MPI_COMM_WORLD);
        nodInit();

        /*Broadcast the initial vector*/
        MPI_Bcast(v,size,MPI_FLOAT,0,MPI_COMM_WORLD);
        Data_send();
        Compute();
        Output();
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
        Environment_Finalize(a,b,v,v1);
        return (0);
}
```

编译：`mpicc jacobi.c -o jacobi`
运行：`mpirun -np 4 ./jacobi`

实验结果
解 4000 元的线性方程组，运行时间如下表所示：

线程数	1	4	8	16
运行时间	106.41 s	61.975 s	53.988 s	50.241 s
数据分配时间	42.361 s	45.802 s	45.808 s	45.985 s
并行计算时间	64.041 s	16.173 s	8.1795 s	4.2549 s

程序总的运行时间等于数据分配时间与并行计算时间之和，相同的实验数据占用的分配时间基本一致。本实验运算规模较大，分析并行计算时间，程序开启 1-16 线程时，并行计算时间基本随线程数的增加同比例减小。

实验八、LU 分解的 MPI 实现

1、 实验目的

在多核实验平台上使用 MPI 编程技术实现 LU 矩阵分解算法，理解基于消息传递模型的并行程序设计思想。

2、 实验原理

LU 矩阵分解算法的实验原理参见实验一的详细讲解。本实验采用 MPI 编程技术实现 LU 分解算法，主要的计算是利用主行 i 对其余各行 j $(j>i)$作初等行变换，各行计算之间没有数据相关关系，因此可以对矩阵 A 按行划分来实现并行计算。考虑到在计算过程中处理器之间的负载均衡，对 A 采用行交叉划分：设处理器个数为 p，矩阵 A 的阶数为 n，$m=\lceil n/p \rceil$，对矩阵 A 行交叉划分后，编号为 $i\,(i=0,1,\cdots,p-1)$的处理器存有 A 的第 i，$i+p$,…，$i+(m-1)p$ 行。然后依次以第 0,1,…，$n-1$行作为主行，将其广播给所有处理器，各处理器利用主行对其部分行向量做行变换，这实际上是各处理器轮流选出主行并广播。若以编号为 my_rank 的处理器的第 i 行元素作为主行，并将它广播给所有处理器，则编号大于等于 my_rank 的处理器利用主行元素对其第 $i+1,\cdots,m-1$ 行数据做行变换，其它处理器利用主行元素对其第 $i,\cdots,m-1$ 行数据做行变换。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 MPI 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写基于 C 语言的 n 阶非奇异方阵做 LU 分解。
b) 在教学实验平台上编译、运行、调试上述程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 lu.c：

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#define a(x,y) a[x*M+y]
/*A 为 M*M 矩阵*/
#define A(x,y) A[x*M+y]
#define l(x,y) l[x*M+y]
#define u(x,y) u[x*M+y]
#define floatsize sizeof(float)
#define intsize sizeof(int)

int M,N;
int m;
float *A;
int my_rank;
int p;
MPI_Status status;

void fatal(char *message)
{
	printf("%s\n",message);
	exit(1);
}

void Environment_Finalize(float *a,float *f)
{
	free(a);
	free(f);
}

int main(int argc, char **argv)
{
	int i,j,k,my_rank,group_size;
	int i1,i2;
	int v,w;
	float *a,*f,*l,*u;
	FILE *fdA;

	MPI_Init(&argc,&argv);
	MPI_Comm_size(MPI_COMM_WORLD,&group_size);
	MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

	p=group_size;

	if (my_rank==0)
	{
		fdA=fopen("dataIn.txt","r");
		fscanf(fdA,"%d %d", &M, &N);
		if(M != N)
		{
			puts("The input is error!");
			exit(0);
		}
```

```
        A=(float *)malloc(floatsize*M*M);
        for(i = 0; i < M; i ++)
            for(j = 0; j < M; j ++)
                fscanf(fdA, "%f", A+i*M+j);
        fclose(fdA);
    }

    /*0 号进程将 M 广播给所有进程*/
    MPI_Bcast(&M,1,MPI_INT,0,MPI_COMM_WORLD);
    m=M/p;
    if (M%p!=0) m++;

    /*分配至各进程的子矩阵大小为 m*M*/
    a=(float*)malloc(floatsize*m*M);

    /*各进程为主行元素建立发送和接收缓冲区*/
    f=(float*)malloc(floatsize*M);

    /*0 号进程为 l 和 u 矩阵分配内存，以分离出经过变换后的 A 矩阵中的 l 和 u 矩阵*/
    if (my_rank==0)
    {
        l=(float*)malloc(floatsize*M*M);
        u=(float*)malloc(floatsize*M*M);
    }

    /*0 号进程采用行交叉划分将矩阵 A 划分为大小 m*M 的 p 块子矩阵，依次发送给 1 至
p-1 号进程*/
    if (a==NULL) fatal("allocate error\n");

    if (my_rank==0)
    {
        for(i=0;i<m;i++)
            for(j=0;j<M;j++)
                a(i,j)=A((i*p),j);
        for(i=0;i<M;i++)
            if ((i%p)!=0)
        {
            i1=i%p;
            i2=i/p+1;
            MPI_Send(&A(i,0),M,MPI_FLOAT,i1,i2,MPI_COMM_WORLD);
        }
    }
    else
    {
        for(i=0;i<m;i++)
            MPI_Recv(&a(i,0),M,MPI_FLOAT,0,i+1,MPI_COMM_WORLD,&status);
    }

    for(i=0;i<m;i++)
        for(j=0;j<p;j++)
    {
        /*j 号进程负责广播主行元素*/
        if (my_rank==j)
        {
            v=i*p+j;
            for (k=v;k<M;k++)
```

```
                f[k]=a(i,k);

            MPI_Bcast(f,M,MPI_FLOAT,my_rank,MPI_COMM_WORLD);
        }
        else
        {
            v=i*p+j;
            MPI_Bcast(f,M,MPI_FLOAT,j,MPI_COMM_WORLD);
        }

        /*编号小于 my_rank 的进程（包括 my_rank 本身）利用主行对其第 i+1,…,m-1 行数
据做行变换*/
        if (my_rank<=j)
            for(k=i+1;k<m;k++)
        {
            a(k,v)=a(k,v)/f[v];
            for(w=v+1;w<M;w++)
                a(k,w)=a(k,w)-f[w]*a(k,v);
        }

        /*编号大于 my_rank 的进程利用主行对其第 i,…,m-1 行数据做行变换*/
        if (my_rank>j)
            for(k=i;k<m;k++)
        {
            a(k,v)=a(k,v)/f[v];
            for(w=v+1;w<M;w++)
                a(k,w)=a(k,w)-f[w]*a(k,v);
        }
    }

    /*0 号进程从其余各进程中接收子矩阵 a，得到经过变换的矩阵 A*/
    if (my_rank==0)
    {
        for(i=0;i<m;i++)
            for(j=0;j<M;j++)
                A(i*p,j)=a(i,j);
    }
    if (my_rank!=0)
    {
        for(i=0;i<m;i++)
            MPI_Send(&a(i,0),M,MPI_FLOAT,0,i,MPI_COMM_WORLD);
    }
    else
    {
        for(i=1;i<p;i++)
            for(j=0;j<m;j++)
        {
            MPI_Recv(&a(j,0),M,MPI_FLOAT,i,j,MPI_COMM_WORLD,&status);
            for(k=0;k<M;k++)
                A((j*p+i),k)=a(j,k);
        }
    }
    if (my_rank==0)
    {
        for(i=0;i<M;i++)
            for(j=0;j<M;j++)
                u(i,j)=0.0;
```

```
            for(i=0;i<M;i++)
                for(j=0;j<M;j++)
                    if (i==j)
                        l(i,j)=1.0;
            else
                l(i,j)=0.0;
            for(i=0;i<M;i++)
                for(j=0;j<M;j++)
                    if (i>j)
                        l(i,j)=A(i,j);
            else
                u(i,j)=A(i,j);
            printf("Input of file \"dataIn.txt\"\n");
            printf("%d\t %d\n",M, N);
            for(i=0;i<M;i++)
            {
                for(j=0;j<N;j++)
                    printf("%f\t",A(i,j));
                printf("\n");
            }
            printf("\nOutput of LU operation\n");
            printf("Matrix L:\n");
            for(i=0;i<M;i++)
            {
                for(j=0;j<M;j++)
                    printf("%f\t",l(i,j));
                printf("\n");
            }
            printf("Matrix U:\n");
            for(i=0;i<M;i++)
            {
                for(j=0;j<M;j++)
                    printf("%f\t",u(i,j));
                printf("\n");
            }
        }
        MPI_Finalize();
        Environment_Finalize(a,f);
        return(0);
}
```

编译：`mpicc lu.c -o lu`
运行：`mpirun -np 4 ./lu`

实验结果

对 2000 阶非奇异方阵做 LU 分解，运行时间如下表所示：

线程数	1	4	8	16
运行时间	134.055 s	32.256 s	19.041 s	12.818 s

程序开启 1-4 线程时，运行时间随线程数的增加同比例减小。开启 4-16 线程时，运行时间基本随线程数的增加线性减小。偏差是由于网络交互时间造成，加大运算规模，可以得到理想数据，也就是并行计算时间随线程数的增加同比例减小。

实验九、随机串匹配算法的 MPI 实现

1、 实验目的

在多核实验平台上使用 MPI 编程技术实现随机串匹配算法，理解基于消息传递模型的并行程序设计思想。

2、 实验原理

假设文本是一个长度为 n 的数组 $T[1..n]$，模式串是一个长度为 $m \le n$ 的数组 $P[1..m]$，字符串匹配是指找出文本串 T 中与模式串 P 所有精确匹配的子串的起始位置。实验二已经介绍了 KMP 算法，该算法虽然能够找到所有的匹配位置，但是算法的复杂度十分高，在某些领域并不实用。本实验将介绍随机串匹配算法，该算法的主要采用了散列（Hash）技术的思想，它能提供对数的时间复杂度。其基本思想是：为了处理模式长度为 m 的串匹配问题，可以将任意长为 m 的串映射到 O(logm)整数位上，映射方法须得保证两个不同的串映射到同一整数的概率非常小。所得到的整数之被视为该串的指纹（Fingerprint），如果两个串的指纹相同则可以判断两个串相匹配。

本实验采用 MPI 编程技术实现随机串匹配算法，利用指纹函数构造一个随机串匹配算法。并行化的关键是设计合适的映射函数 f_p，映射函数 f_p 将长度为 m 的串映射到域 D，且要求映射函数 f_p 满足下述三个性质：①模式串 T 的所有子串集为集合 B。对于任意串 $X \in B$ 以及每一个 $p \in S$（S 为模式串的取值域）， 由 $O(\log m)$ 位组成；②随机选择，它将两个不等的串 X 和 Y 映射到 D 中同一元素的概率是很小的；③对于每个 $p \in S$ 和 B 中所有串，应该能够很容易的并行计算 。

然后计算模式串 $P[1...m]$ 和子串 $T[i,i+m-1]$ 的指纹函数（其中 $1 \le i \le n-m+1$，$m \le n$），每当 P 的指纹和 $T[i,i+m-1]$ 的指纹相等时，则标记在文本 T 的位置 i 与 P 出现匹配。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 MPI 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写 C 语言程序，查找模式串 $P[1...m]$ 在文本串 $T[1,n]$ 中的所有匹配位置。

b) 在教学实验平台上编译、运行、调试上述程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 rand_match.c：

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

#define PiM 21

/*初始化 Fp 矩阵*/
int f[2][2][2]={{{1,0},{1,1}},{{1,1},{0,1}}};
int g[2][2][2];
int pdata[]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,67,71,79,83};

/*生成字符串*/
void gen_string(int stringlen,char *String,int seed)
{
	int i,num;

	srand(seed*100);
	for(i=0;i<stringlen;i++){
		num=rand()%2;
		String[i]='0'+num;
	}
	String[stringlen]='\0';
}

/*从素数存放数组中随机选取一个素数*/
int drawp(int seed){
	srand(seed*100);
	return pdata[rand()%PiM];
}

/*该算法中第一个参数 n 是正文串的长度，第二个参数 m 是模式串的长度。(约束
0<m,n<=128)*/
main(int argc,char *argv[]){
	int groupsize,myrank,n,m,p,i,j,h,tmp;
	int textlen;
	char *Text,*Pattern;
	int *MATCH;
	int B[128][2][2],C[128][2][2],D[128][2][2],L[128][2][2];
	int fp_pattern[2][2];
	MPI_Status status;


	MPI_Init(&argc,&argv);
	MPI_Comm_size(MPI_COMM_WORLD,&groupsize);
	MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

	n=atoi(argv[1]);
	m=atoi(argv[2]);

	textlen=n/groupsize;
```

```
if(myrank==groupsize-1)
    textlen=n-textlen*(groupsize-1);

if((Text=(char *)malloc(textlen*sizeof(char)+1))==NULL){
    printf("no enough memory\n");
    exit(1);
}

if((Pattern=(char *)malloc(m*sizeof(char)+1))==NULL){
    printf("no enough memory\n");
    exit(1);
}

if((MATCH=(int *)malloc(textlen*sizeof(int)+1))==NULL){
    printf("no enough memory\n");
    exit(1);
}

/*初始化 Fp 的逆矩阵 Gp*/
g[0][0][0]=1;g[0][0][1]=0;g[0][1][0]=p-1;g[0][1][1]=1;
g[1][0][0]=1;g[1][0][1]=p-1;g[1][1][0]=0;g[1][1][1]=1;

/*产生正文串和模式串*/
gen_string(textlen,Text,m*n*myrank);

/*随机选择一个素数，对应算法 14.9 步骤（2）*/
if(myrank==0) {
    p=drawp(n/m);
    gen_string(m,Pattern,p*m*n);
}

/*播送选中的素数与 m 给其余每个节点*/
MPI_Bcast(&p,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(Pattern,m,MPI_CHAR,0,MPI_COMM_WORLD);

printf("one node %d n=%d,m=%d,p=%d\n",myrank,n,m,p);
printf("one node %d Text=%s\n",myrank,Text);
printf("one node %d Pattern=%s\n",myrank,Pattern);

MPI_Barrier(MPI_COMM_WORLD);

/*计算模式串的指纹函数值，对应算法 14.9 步骤（3）*/
if(myrank==0) {
    for(i=1;i<=m;i++) {
        B[i][0][0]=f[Pattern[i-1]-'0'][0][0];
        B[i][0][1]=f[Pattern[i-1]-'0'][0][1];
        B[i][1][0]=f[Pattern[i-1]-'0'][1][0];
        B[i][1][1]=f[Pattern[i-1]-'0'][1][1];
    }

    B[0][0][0]=(B[1][0][0]*B[2][0][0]+B[1][0][1]*B[2][1][0])%p;
    B[0][0][1]=(B[1][0][0]*B[2][0][1]+B[1][0][1]*B[2][1][1])%p;
    B[0][1][0]=(B[1][1][0]*B[2][0][0]+B[1][1][1]*B[2][1][0])%p;
    B[0][1][1]=(B[1][1][0]*B[2][0][1]+B[1][1][1]*B[2][1][1])%p;

    for(j=1;j<=m-2;j++){
```

```
            B[j][0][0]=(B[j-1][0][0]*B[j+2][0][0]+B[j-1][0][1]*B[j+2][1][0])%p;
            B[j][0][1]=(B[j-1][0][0]*B[j+2][0][1]+B[j-1][0][1]*B[j+2][1][1])%p;
            B[j][1][0]=(B[j-1][1][0]*B[j+2][0][0]+B[j-1][1][1]*B[j+2][1][0])%p;
            B[j][1][1]=(B[j-1][1][0]*B[j+2][0][1]+B[j-1][1][1]*B[j+2][1][1])%p;
        }

        fp_pattern[0][0]=B[m-2][0][0];
        fp_pattern[0][1]=B[m-2][0][1];
        fp_pattern[1][0]=B[m-2][1][0];
        fp_pattern[1][1]=B[m-2][1][1];
    }

    MPI_Bcast(fp_pattern,4,MPI_INT,0,MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    /*计算 Ni，存放在数组 C 中*/
    for(i=1;i<=textlen;i++) {
        B[i][0][0]=f[Text[i-1]-'0'][0][0];
        B[i][0][1]=f[Text[i-1]-'0'][0][1];
        B[i][1][0]=f[Text[i-1]-'0'][1][0];
        B[i][1][1]=f[Text[i-1]-'0'][1][1];
    }

    C[1][0][0]=B[1][0][0];
    C[1][0][1]=B[1][0][1];
    C[1][1][0]=B[1][1][0];
    C[1][1][1]=B[1][1][1];

    for(i=2;i<=textlen;i++) {
        C[i][0][0]=(C[i-1][0][0]*B[i][0][0]+C[i-1][0][1]*B[i][1][0])%p;
        C[i][0][1]=(C[i-1][0][0]*B[i][0][1]+C[i-1][0][1]*B[i][1][1])%p;
        C[i][1][0]=(C[i-1][1][0]*B[i][0][0]+C[i-1][1][1]*B[i][1][0])%p;
        C[i][1][1]=(C[i-1][1][0]*B[i][0][1]+C[i-1][1][1]*B[i][1][1])%p;
    }

    if(groupsize>1) {
        if(myrank==0) {
            MPI_Send(&C[textlen],4,MPI_INT,myrank+1,myrank,MPI_COMM_WORLD);
        }
        else if(myrank==groupsize-1) {

    MPI_Recv(&C[0],4,MPI_INT,myrank-1,myrank-1,MPI_COMM_WORLD,&status);
        }
        else {

    MPI_Recv(&C[0],4,MPI_INT,myrank-1,myrank-1,MPI_COMM_WORLD,&status);

            i=textlen;

            C[textlen+1][0][0]=(C[0][0][0]*C[i][0][0]+C[0][0][1]*C[i][1][0])%p;
            C[textlen+1][0][1]=(C[0][0][0]*C[i][0][1]+C[0][0][1]*C[i][1][1])%p;
            C[textlen+1][1][0]=(C[0][1][0]*C[i][0][0]+C[0][1][1]*C[i][1][0])%p;
            C[textlen+1][1][1]=(C[0][1][0]*C[i][0][1]+C[0][1][1]*C[i][1][1])%p;

            C[i][0][0]=C[textlen+1][0][0];
            C[i][0][1]=C[textlen+1][0][1];
            C[i][1][0]=C[textlen+1][1][0];
```

```
                C[i][1][1]=C[textlen+1][1][1];

                MPI_Send(&C[textlen],4,MPI_INT,myrank+1,myrank,MPI_COMM_WORLD);
            }
        }

        MPI_Barrier(MPI_COMM_WORLD);

        if(myrank!=0) {
            for(i=1;i<textlen;i++) {
                C[textlen+1][0][0]=(C[0][0][0]*C[i][0][0]+C[0][0][1]*C[i][1][0])%p;
                C[textlen+1][0][1]=(C[0][0][0]*C[i][0][1]+C[0][0][1]*C[i][1][1])%p;
                C[textlen+1][1][0]=(C[0][1][0]*C[i][0][0]+C[0][1][1]*C[i][1][0])%p;
                C[textlen+1][1][1]=(C[0][1][0]*C[i][0][1]+C[0][1][1]*C[i][1][1])%p;
                C[i][0][0]=C[textlen+1][0][0];
                C[i][0][1]=C[textlen+1][0][1];
                C[i][1][0]=C[textlen+1][1][0];
                C[i][1][1]=C[textlen+1][1][1];
            }
        }

        MPI_Barrier(MPI_COMM_WORLD);

        if(groupsize>1) {
            if(myrank==0) {

        MPI_Recv(&C[textlen+1],4*(m-1),MPI_INT,myrank+1,myrank,MPI_COMM_WORLD,&st
atus);
            }
            else if(myrank==groupsize-1) {

        MPI_Send(&C[1],4*(m-1),MPI_INT,myrank-1,myrank-1,MPI_COMM_WORLD);
            }
            else {

        MPI_Recv(&C[textlen+1],4*(m-1),MPI_INT,myrank+1,myrank,MPI_COMM_WORLD,&st
atus);

        MPI_Send(&C[1],4*(m-1),MPI_INT,myrank-1,myrank-1,MPI_COMM_WORLD);
            }
        }

        /*计算 Ri，存放在数组 D 中*/
        for(i=1;i<=textlen;i++) {
            B[i][0][0]=g[Text[i-1]-'0'][0][0];
            B[i][0][1]=g[Text[i-1]-'0'][0][1];
            B[i][1][0]=g[Text[i-1]-'0'][1][0];
            B[i][1][1]=g[Text[i-1]-'0'][1][1];
        }

        D[1][0][0]=B[1][0][0];
        D[1][0][1]=B[1][0][1];
        D[1][1][0]=B[1][1][0];
        D[1][1][1]=B[1][1][1];

        for(i=2;i<=textlen;i++) {
            D[i][0][0]=(B[i][0][0]*D[i-1][0][0]+B[i][0][1]*D[i-1][1][0])%p;
```

```
        D[i][0][1]=(B[i][0][0]*D[i-1][0][1]+B[i][0][1]*D[i-1][1][1])%p;
        D[i][1][0]=(B[i][1][0]*D[i-1][0][0]+B[i][1][1]*D[i-1][1][0])%p;
        D[i][1][1]=(B[i][1][0]*D[i-1][0][1]+B[i][1][1]*D[i-1][1][1])%p;
    }

    if(groupsize>1) {
        if(myrank==0) {
            MPI_Send(&D[textlen],4,MPI_INT,myrank+1,myrank,MPI_COMM_WORLD);
        }
        else if(myrank==groupsize-1) {

    MPI_Recv(&D[0],4,MPI_INT,myrank-1,myrank-1,MPI_COMM_WORLD,&status);
        }
        else {

    MPI_Recv(&D[0],4,MPI_INT,myrank-1,myrank-1,MPI_COMM_WORLD,&status);

            i=textlen;
            D[textlen+1][0][0]=(D[i][0][0]*D[0][0][0]+D[i][0][1]*D[0][1][0])%p;
            D[textlen+1][0][1]=(D[i][0][0]*D[0][0][1]+D[i][0][1]*D[0][1][1])%p;
            D[textlen+1][1][0]=(D[i][1][0]*D[0][0][0]+D[i][1][1]*D[0][1][0])%p;
            D[textlen+1][1][1]=(D[i][1][0]*D[0][0][1]+D[i][1][1]*D[0][1][1])%p;

            D[i][0][0]=D[textlen+1][0][0];
            D[i][0][1]=D[textlen+1][0][1];
            D[i][1][0]=D[textlen+1][1][0];
            D[i][1][1]=D[textlen+1][1][1];

            MPI_Send(&D[textlen],4,MPI_INT,myrank+1,myrank,MPI_COMM_WORLD);
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if(myrank==0) {
        D[0][0][0]=1;
        D[0][0][1]=0;
        D[0][1][0]=0;
        D[0][1][1]=1;
    }

    if(myrank!=0) {
        for(i=1;i<textlen;i++) {
            D[textlen+1][0][0]=(D[i][0][0]*D[0][0][0]+D[i][0][1]*D[0][1][0])%p;
            D[textlen+1][0][1]=(D[i][0][0]*D[0][0][1]+D[i][0][1]*D[0][1][1])%p;
            D[textlen+1][1][0]=(D[i][1][0]*D[0][0][0]+D[i][1][1]*D[0][1][0])%p;
            D[textlen+1][1][1]=(D[i][1][0]*D[0][0][1]+D[i][1][1]*D[0][1][1])%p;

            D[i][0][0]=D[textlen+1][0][0];
            D[i][0][1]=D[textlen+1][0][1];
            D[i][1][0]=D[textlen+1][1][0];
            D[i][1][1]=D[textlen+1][1][1];
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
```

```
    for(i=1;i<=textlen;i++)
        MATCH[i]=0;

    /*各个处理器分别计算所分配的文本串的 Li，并判别是否存在匹配位置，对应算法 14.9
的步骤（4）*/
    if(myrank==groupsize-1) {
        for(i=1;i<=textlen-m+1;i++){
            L[i][0][0]=(D[i-1][0][0]*C[i+m-1][0][0]+D[i-1][0][1]*C[i+m-1][1][0])%p;
            L[i][0][1]=(D[i-1][0][0]*C[i+m-1][0][1]+D[i-1][0][1]*C[i+m-1][1][1])%p;
            L[i][1][0]=(D[i-1][1][0]*C[i+m-1][0][0]+D[i-1][1][1]*C[i+m-1][1][0])%p;
            L[i][1][1]=(D[i-1][1][0]*C[i+m-1][0][1]+D[i-1][1][1]*C[i+m-1][1][1])%p;
            if(((L[i][0][0]+p)%p==fp_pattern[0][0])
                && ((L[i][0][1]+p)%p==fp_pattern[0][1])
                && ((L[i][1][0]+p)%p==fp_pattern[1][0])
                && ((L[i][1][1]+p)%p==fp_pattern[1][1])) {
                MATCH[i]=1;
                printf("on node %d match pos is %d\n",myrank,i);
            }
        }
    }
    else {
        for(i=1;i<=textlen;i++){
            L[i][0][0]=(D[i-1][0][0]*C[i+m-1][0][0]+D[i-1][0][1]*C[i+m-1][1][0])%p;
            L[i][0][1]=(D[i-1][0][0]*C[i+m-1][0][1]+D[i-1][0][1]*C[i+m-1][1][1])%p;
            L[i][1][0]=(D[i-1][1][0]*C[i+m-1][0][0]+D[i-1][1][1]*C[i+m-1][1][0])%p;
            L[i][1][1]=(D[i-1][1][0]*C[i+m-1][0][1]+D[i-1][1][1]*C[i+m-1][1][1])%p;
            if(((L[i][0][0]+p)%p==fp_pattern[0][0])
                && ((L[i][0][1]+p)%p==fp_pattern[0][1])
                && ((L[i][1][0]+p)%p==fp_pattern[1][0])
                && ((L[i][1][1]+p)%p==fp_pattern[1][1])) {
                MATCH[i]=1;
                printf("on node %d match pos is %d\n",myrank,i);
            }
        }
    }

    MPI_Finalize();
}
```

编译命令：`mpicc rand_match.c -o rand_match`
运行命令：`mpirun -np 4 ./rand_match 7 2`

说明：可以使用命令 `mpirun -np SIZE rand_match m n` 来运行该串匹配程序，其中 SIZE 是所使用的处理器个数，m 表示文本串长度，n 为模式串长度。本实例中使用了 SIZE=4 个处理器，m=7，n=2。

实验结果：

```
one node 0 n=7,m=2,p=47
one node 2 n=7,m=2,p=47
one node 2 Text=0
one node 2 Pattern=01W+,)W+
one node 1 n=7,m=2,p=47
one node 1 Text=0
one node 1 Pattern=010+,)0+
```

```
one node 3 n=7,m=2,p=47
one node 3 Text=0011
one node 3 Pattern=01 +,i +
one node 0 Text=1
one node 0 Pattern=01
on node 3 match pos is 2
```

说明：该运行实例中，令文本串长度为 7，随机产生的文本串为 1000011，分布在 4 个节点上；模式串长度为 2，随机产生的模式串为 01。最后，节点 3 上得到一个匹配位置。本实验准确找到模式串在文本串中的位置。

实验十、顶点倒塌算法求连通分量的 MPI 实现

1、 实验目的

在多核实验平台上使用 MPI 编程技术实现顶点倒塌算法求连通分量，理解基于消息传递模型的并行程序设计思想。

2、 实验原理

图 G 的连通分量是 G 的最大连通子图，该子图中每对顶点间均有一条路径。顶点倒塌算法可以找出图 G 所有的连通分量。该算法将图中的 N 个顶点看作 N 个孤立的超顶点，算法运行中，有边连通的超顶点相继合并，直到形成最后的整个连通分量。每个顶点属于且仅属于一个超顶点，超顶点中标号最小者称为该超顶点的根。

该算法的流程由一系列循环组成。每次循环分为三步：①发现每个顶点的最小标号邻接超顶点；②把每个超顶点的根连到最小标号邻接超顶点的根上；③所有在第②步连接在一起的超顶点倒塌合并成为一个较大的超顶点。

图 G 的顶点总数为 N，因为超顶点的个数每次循环后至少减少一半，所以把每个连通分量倒塌成单个超顶点至多 $\log N$ 次循环即可。顶点 i 所属的超顶点的根记为 $D(i)$，则一开始时有 $D(i)=i$，算法结束后，所有处于同一连通分量中的顶点具有相同的 $D(i)$。

顶点倒塌算法是专为并行程序设计的，本实验采用 MPI 编程技术实现该算法。多个顶点的处理具有很强的独立性，适合分配给多个处理器并行处理，让 p 个处理器分管 N 个顶点。算法中为顶点设置数组变量 D 和 C，其中 $D(i)$ 为顶点 i 所在的超顶点号，$C(i)$ 为和顶点 i 或超顶点 i 相连的最小超顶点号等，根据程序运行的阶段不同，意义也有变化。算法的主循环由 5 个步骤组成：①各处理器并行为每个顶点找出对应的 $C(i)$；②各处理器并行为每个超顶点找出最小邻接超顶点，编号放入 $C(i)$ 中；③修改所有 $D(i)=C(i)$；④修改所有 $C(i)=C(C(i))$，运行 $\log N$ 次；⑤修改所有 $D(i)$ 为 $C(i)$ 和 $D(C(i))$ 中较小者。其中最后 3 步对应意义为超顶点的合并。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 MPI 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写 C 语言程序，找出图G 的所有连通分量。

b) 在教学实验平台上编译、运行、调试上述程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 connect.c：

```
/**本算法中处理器数目须小于图的顶点数**/
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include <mpi.h>

/**使用动态分配的内存存储邻接矩阵，以下为宏定义**/
#define A(i,j) A[i*N+j]
/**以下为 N:顶点数   n:各处理器分配的顶点数   p:处理器数**/
int N;
int n;
int p;

int *D,*C;
int *A;
int temp;
int myid;
MPI_Status status;

/**输出必要信息**/
void print(int *P)
{
    int i;
    if(myid==0)
    {
        for(i=0;i<N;i++)
            printf("%d ",P[i]);
        printf("\n");
    }
}

/**读入邻接矩阵**/
void readA()
{
    char *filename;
    int i,j;
    printf("\n");
    printf("Input the vertex num:\n");
    scanf("%d",&N);
    n=N/p;
    if(N%p!=0) n++;
    A=(int*)malloc(sizeof(int)*(n*p)*N);
    if(A==NULL)
    {
        printf("Error when allocating memory\n");
        exit(0);
    }
    printf("Input the adjacent matrix:\n");
    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
            scanf("%d",&A(i,j));
    for(i=N;i<n*p;i++)
```

```
            for(j=0;j<N;j++)
                A(i,j)=0;
}

/**处理器 0 广播特定数据**/
void bcast(int *P)
{
    MPI_Bcast(P,N,MPI_INT,0,MPI_COMM_WORLD);
}

/**两者中取最小的数学函数**/
int min(int a,int b)
{
    return(a<b?a:b);
}

/**为各顶点找最小的邻接超顶点，对应算法步骤(2.1)**/
void D_to_C()
{
    int i,j;
    for(i=0;i<n;i++)
    {
        C[n*myid+i]=N+1;
        for(j=0;j<N;j++)
            if((A(i,j)==1)&&(D[j]!=D[n*myid+i])&&(D[j]<C[n*myid+i]))
        {
            C[n*myid+i]=D[j];
        }
        if(C[n*myid+i]==N+1)
            C[n*myid+i]=D[n*myid+i];
    }
}

/**为各超顶点找最小邻接超顶点，对应算法步骤(2.2)**/
void C_to_C()
{
    int i,j;
    for(i=0;i<n;i++)
    {
        temp=N+1;
        for(j=0;j<N;j++)
            if((D[j]==n*myid+i)&&(C[j]!=n*myid+i)&&(C[j]<temp))
        {
            temp=C[j];
        }
        if(temp==N+1) temp=D[n*myid+i];
        C[myid*n+i]=temp;
    }
}

/**调整超顶点标识**/
void CC_to_C()
{
    int i;
    for(i=0;i<n;i++)
        C[myid*n+i]=C[C[myid*n+i]];
}
```

```
/**产生新的超顶点，对应算法步骤(2.5)**/
void CD_to_D()
{
    int i;
    for(i=0;i<n;i++)
        D[myid*n+i]=min(C[myid*n+i],D[C[myid*n+i]]);
}

/**释放动态内存**/
void freeall()
{
    free(A);
    free(D);
    free(C);
}

/**主函数**/
void main(int argc,char **argv)
{
    int i,j,k;
    double l;
    int group_size;

    /**以下变量用来记录运行时间**/
    double starttime,endtime;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&group_size);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    p=group_size;
    MPI_Barrier(MPI_COMM_WORLD);
    if(myid==0)
        starttime=MPI_Wtime();

    /**处理器 0 读邻接矩阵**/
    if(myid==0)
        readA();
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);
    if(myid!=0)
    {
        n=N/p;
        if(N%p!=0) n++;
    }

    D=(int*)malloc(sizeof(int)*(n*p));
    C=(int*)malloc(sizeof(int)*(n*p));
    if(myid!=0)
        A=(int*)malloc(sizeof(int)*n*N);

    /**初始化数组 D，步骤(1)**/
    for(i=0;i<n;i++)
        D[myid*n+i]=myid*n+i;
```

```
    MPI_Barrier(MPI_COMM_WORLD);

MPI_Gather(myid?&D[n*myid]:MPI_IN_PLACE),n,MPI_INT,D,n,MPI_INT,0,MPI_COMM_W
ORLD);
    bcast(D);

    MPI_Barrier(MPI_COMM_WORLD);

    /**处理器 0 向其它处理器发送必要数据**/
    if(myid==0)
        for(i=1;i<p;i++)
            MPI_Send(&A(i*n,0),n*N,MPI_INT,i,i,MPI_COMM_WORLD);
    else
        MPI_Recv(A,n*N,MPI_INT,0,myid,MPI_COMM_WORLD,&status);
    MPI_Barrier(MPI_COMM_WORLD);

    l=log(N)/log(2);

    /**主循环体：算法步骤(2)**/
    for(i=0;i<l;i++)
    {
        if(myid==0) printf("Stage %d:\n",i+1);

        /**算法步骤(2.1)**/
        D_to_C();
        MPI_Barrier(MPI_COMM_WORLD);

MPI_Gather(myid?&C[n*myid]:MPI_IN_PLACE),n,MPI_INT,C,n,MPI_INT,0,MPI_COMM_W
ORLD);
        print(C);
        bcast(C);
        MPI_Barrier(MPI_COMM_WORLD);

        /**算法步骤(2.2)**/
        C_to_C();
        print(C);
        MPI_Barrier(MPI_COMM_WORLD);

MPI_Gather(myid?&C[n*myid]:MPI_IN_PLACE),n,MPI_INT,C,n,MPI_INT,0,MPI_COMM_W
ORLD);

MPI_Gather(myid?&C[n*myid]:MPI_IN_PLACE),n,MPI_INT,D,n,MPI_INT,0,MPI_COMM_W
ORLD);
        MPI_Barrier(MPI_COMM_WORLD);

        /**算法步骤(2.3)**/
        if(myid==0)
            for(j=0;j<n;j++)
                D[j]=C[j];

        /**算法步骤(2.4)**/
        for(k=0;k<l;k++)
        {
            bcast(C);
            CC_to_C();
```

```
MPI_Gather(myid?&C[n*myid]:MPI_IN_PLACE),n,MPI_INT,C,n,MPI_INT,0,MPI_COMM_W
ORLD);
            }
            bcast(C);
            bcast(D);

            /**算法步骤(2.5)**/
            CD_to_D();

MPI_Gather(myid?&D[n*myid]:MPI_IN_PLACE),n,MPI_INT,D,n,MPI_INT,0,MPI_COMM_W
ORLD);
            print(D);
            bcast(D);
      }

      if(myid==0) printf("Result: \n");
      print(D);
      if(myid==0)
      {
            endtime=MPI_Wtime();
            printf("The running time is %d\n",endtime-starttime);
      }

      freeall();
      MPI_Finalize();

}
```

编译： `mpicc connect.c -o connect -lm`
运行： `mpirun -np 3 ./connect`

运行结果：

```
Input the vertex num:
8
Input the adjacent matrix:
0 0 0 0 0 0 0 1
0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 1
0 0 0 0 0 0 1 1
0 1 0 1 0 0 0 0
0 1 0 0 1 0 0 0
1 0 0 1 1 0 0 0
Stage 1:
7 5 2 5 6 1 1 0
7 5 2 5 6 1 1 0
0 1 2 1 1 1 1 0
Stage 2:
0 1 2 0 0 1 1 1
1 0 2 0 0 1 1 1
0 0 2 0 0 0 0 0
Stage 3:
0 0 2 0 0 0 0 0
0 0 2 0 0 0 0 0
0 0 2 0 0 0 0 0
Result:
```

```
0 0 2 0 0 0 0 0
The running time is 100.865168
```

本实验准确的计算出例图的连通分量。

实验十一、快速排序算法的 MPI 实现

1、 实验目的

在多核实验平台上使用 MPI 编程技术实现并行快速排序算法，理解基于消息传递模型的并行程序设计思想。

2、 实验原理

快速排序是一种最基本的排序算法，它的基本思想是：在当前无序区 $R[1,n]$ 中取一个记录作为比较的“基准”（一般取第一个、最后一个或中间位置的元素），用此基准将当前的无序区 $R[1,n]$ 划分成左右两个无序的子区 $R[1,i-1]$ 和 $R[1,n]$ $(1 \le i \le n)$，且左边的无序子区中记录的所有关键字均小于等于基准的关键字，右边的无序子区中记录的所有关键字均大于等于基准的关键字；当 $R[1,i-1]$ 和 $R[1,n]$ 非空时，分别对它们重复上述的划分过程，直到所有的无序子区中的记录均排好序为止。

快速排序算法并行化的一个简单思想是，对每次划分过后所得到的两个序列分别使用两个处理器完成递归排序。例如对一个长为 n 的序列，首先划分得到两个长为 $n/2$ 的序列，将其交给两个处理器分别处理；而后进一步划分得到四个长为 $n/4$ 的序列，再分别交给四个处理器处理；如此递归下去最终得到排序好的序列。当然这里举的是理想的划分情况，如果划分步骤不能达到平均分配的目的，那么排序的效率会相对较差。最优的情况下该并行算法形成一个高度为 $\log n$ 的排序树，其计算复杂度为 $O(n)$。

3、 实验环境

使用多核并行实验环境，在 Linux 环境下，可以方便的编写、编译、运行和调试 MPI 并行程序。相关操作请参考实验环境配套手册。

4、 实验要求与内容

a) 根据实验原理描述的背景，编写 C 语言程序，实现并行快速排序算法。
b) 在教学实验平台上编译、运行、调试上述程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

源程序 quick_sort.c：

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define   TRUE 1
/*
     * 函数名: main
     * 功能：实现快速排序的主程序
     * 输入：argc 为命令行参数个数；
     *          argv 为每个命令行参数组成的字符串数组。
     * 输出：返回 0 代表程序正常结束
*/
main(int argc,char *argv[])
{
     int DataSize;
     int *data;
     /*MyID 表示进程标志符；SumID 表示组内进程数*/
     int   MyID, SumID;
     int i, j;
     int m, r;

     MPI_Status status;
     /*启动 MPI 计算*/
     MPI_Init(&argc,&argv);

     /*MPI_COMM_WORLD 是通信子*/
     /*确定自己的进程标志符 MyID*/
     MPI_Comm_rank(MPI_COMM_WORLD,&MyID);

     /*组内进程数是 SumID*/
     MPI_Comm_size(MPI_COMM_WORLD,&SumID);

     /*根处理机(MyID=0)获取必要信息，并分配各处理机进行工作*/
     if(MyID==0)
     {
          /*获取待排序数组的长度*/
          DataSize=GetDataSize();
          data=(int *)malloc(DataSize*sizeof(int));

          /*内存分配错误*/
          if(data==0)
               ErrMsg("Malloc memory error!");

          /*动态生成待排序序列*/
          srand(396);
          for(i=0;i<DataSize;i++)
          {
                    data[i]=1 + (int) (32768.0*rand() / (RAND_MAX + 1.0));
                    printf("%10d",data[i]);
          }
          printf("\n\n");
     }
```

```
    m=log2(SumID);

    /* 从根处理器将数据序列广播到其他处理器*/
    /*{"1"表示传送的输入缓冲中的元素的个数,        */
    /* "MPI_INT"表示输入元素的类型,                */
    /* "0"表示 root processor 的 ID   }            */
    MPI_Bcast(&DataSize,1,MPI_INT,0,MPI_COMM_WORLD);

    /*ID 号为 0 的处理器调度执行排序*/
    para_QuickSort(data,0,DataSize-1,m,0,MyID);

    /*ID 号为 0 的处理器打印排序完的有序序列*/
    if(MyID==0)
    {
        for(i=0;i<DataSize;i++)
        {
            printf("%10d",data[i]);
        }
        printf("\n");
    }

    MPI_Finalize();          //结束计算
}

/*
    * 函数名: para_QuickSort
    * 功能：并行快速排序，对起止位置为 start 和 end 的序列，使用 2 的 m 次幂个处理器进行排序
    * 输入：无序数组 data[1,n]，使用的处理器个数 2^m
    * 输出：有序数组 data[1,n]
*/
para_QuickSort(int *data,int start,int end,int m,int id,int MyID)
{
    int i, j;
    int r;
    int MyLength;
    int *tmp;
    MPI_Status status;
    MyLength=-1;

    /*如果可供选择的处理器只有一个，那么由处理器 id 调用串行排序，对应于算法 13.4 步骤(1.1)*/
    /*(1.1)    Pid call quicksort(data,i,j) */
    if(m==0)
    {
        if(MyID==id)
            QuickSort(data,start,end);
        return;
    }

    /*由第 id 号处理器划分数据，并将后一部分数据发送到处理器 id+exp2(m-1)，对应于算法 13.4 步骤(1.2,1.3)*/
    /*(1.2) Pid: r=patrition(data,i,j)*/
    if(MyID==id)
```

```
    {
        /*将当前的无序区 R[1，n]划分成左右两个无序的子区 R[1，i-1]和 R[i，n](1≤i≤n)*/
        r=Partition(data,start,end);
        MyLength=end-r;
        /*(1.3)    Pid send data[r+1,m-1] to P(id+2m-1) */
        /* {MyLength 表示发送缓冲区地址；*/
        /*   发送元素数目为 1;          */
        /*   MyID 是消息标签 }              */
        MPI_Send(&MyLength,1,MPI_INT,id+exp2(m-1),MyID,MPI_COMM_WORLD);
        /*若缓冲区不空，则第 id+2m-1 号处理器取数据的首址是 data[r+1]*/
        if(MyLength!=0)

    MPI_Send(data+r+1,MyLength,MPI_INT,id+exp2(m-1),MyID,MPI_COMM_WORLD);
    }

    /*处理器 id+exp2(m-1)接受处理器 id 发送的消息*/
    if(MyID==id+exp2(m-1))
    {
        MPI_Recv(&MyLength,1,MPI_INT,id,id,MPI_COMM_WORLD,&status);
        if(MyLength!=0)
        {
            tmp=(int *)malloc(MyLength*sizeof(int));
            if(tmp==0) ErrMsg("Malloc memory error!");
            MPI_Recv(tmp,MyLength,MPI_INT,id,id,MPI_COMM_WORLD,&status);
        }
    }

    /*递归调用并行排序，对应于算法 13.4 步骤(1.4，1.5)*/

    /*用 2^m-1 个处理器对 start--(r-1)的数据进行递归排序*/
    j=r-1-start;
    MPI_Bcast(&j,1,MPI_INT,id,MPI_COMM_WORLD);
    /*(1.4)    para_quicksort(data,i,r-1,m-1,id)*/
    if(j>0)
        para_QuickSort(data,start,r-1,m-1,id,MyID);

    /*用 2^m-1 个处理器对(r+1)--end 的数据进行递归排序*/
    j=MyLength;
    MPI_Bcast(&j,1,MPI_INT,id,MPI_COMM_WORLD);
    /*(1.5)    para_quicksort(data,r+1,j,m-1,id+2m-1)*/
    if(j>0)
        para_QuickSort(tmp,0,MyLength-1,m-1,id+exp2(m-1),MyID);

    /*将排序好的数据由处理器 id+exp2(m-1)发回 id 号处理器，对应于算法 13.4 步骤(1.6)*/
    /*(1.6)    P(id+2m-1) send data[r+1,m-1] back to Pid */
    if((MyID==id+exp2(m-1)) && (MyLength!=0))
        MPI_Send(tmp,MyLength,MPI_INT,id,id+exp2(m-1),MPI_COMM_WORLD);

    if((MyID==id) && (MyLength!=0))

    MPI_Recv(data+r+1,MyLength,MPI_INT,id+exp2(m-1),id+exp2(m-1),MPI_COMM_WOR
LD,&status);
}

/*
    * 函数名: QuickSort
```

```
    * 功能：对起止位置为 start 和 end 的数组序列，进行串行快速排序。
    * 输入：无序数组 data[1,n]
    * 返回：有序数组 data[1,n]
*/
QuickSort(int *data,int start,int end)
{
    int r;
    int i;
    if(start<end)
    {
        r=Partition(data,start,end);
        QuickSort(data,start,r-1);
        QuickSort(data,r+1,end);
    }
}

/*
    * 函数名: Partition
    * 功能：对起止位置为 start 和 end 的数组序列，将其分成两个非空子序列，
    *       其中前一个子序列中的任意元素小于后个子序列的元素。
    * 输入：无序数组 data[1,n]
    * 返回: 两个非空子序列的分界下标
*/
int Partition(int *data,int start,int end)
{
    int pivo;
    int i, j;
    int tmp;

    pivo=data[end];
    i=start-1;                  /*i(活动指针)*/

    for(j=start;j<end;j++)
        if(data[j]<=pivo)
        {
            i++;            /*i 表示比 pivo 小的元素的个数*/
            tmp=data[i];
            data[i]=data[j];
            data[j]=tmp;
        }

    tmp=data[i+1];
    data[i+1]=data[end];
    data[end]=tmp;              /*以 pivo 为分界，data[i+1]=pivo*/

    return i+1;
}

/*
    * 函数名: exp2
    * 功能：求 2 的 num 次幂
    * 输入：int 型数据 num
    * 返回: 2 的 num 次幂
*/
int exp2(int num)
```

```
{
	int i;
	i=1;
	while(num>0)
	{
		num--;
		i=i*2;
	}
	return i;
}

/*
	* 函数名: log2
	* 功能：求以 2 为底的 num 的对数
	* 输入：int 型数据 num
	* 返回: 以 2 为底的 num 的对数
*/
int log2(int num)
{
	int i, j;
	i=1;
	j=2;
	while(j<num)
	{
		j=j*2;
		i++;
	}
	if(j>num)
		i--;
	return i;
}

/*
	* 函数名: GetDataSize
	* 功能：读入待排序序列长度
*/
int GetDataSize()
{
	int i;
	while(TRUE)
	{
		printf("Input the Data Size :");
		scanf("%d",&i);
		/*读出正确的 i，返回；否则，继续要求输入*/
		if((i>0) && (i<=65535))
			break;
		ErrMsg("Wrong Data Size, must between [1..65535]");
	}
	return i;
}
/*输出错误信息*/
ErrMsg(char *msg)
{
	printf("Error: %s \n",msg);
}
```

编译：`mpicc quick_sort.c -o quick`
运行：`mpirun -np 5 ./quick`

使用命令 mpirun -np SIZE ./quick 来运行该程序，其中 SIZE 是所使用的处理器个数。本实例中使用了 SIZE=5 个处理器。

运行结果：

Input the Data Size :60

13102	18239	10699	24273	11470	13765	30735
26200	14575	16372	23404	18929	28967	15176
4495	15626	12270	19845	15944	17887	3382
14904	8867	13172	13240	24871	32537	29542
19308	22601	13816	32409	8072	24514	23913
19541	5511	21879	12973	20086	5482	3609
6246	1681	18784	10741	17306	31053	30585
481	16171	1199	15385	25038	14371	28624
17140	14139	25398	3679			

481	1199	1681	3382	3609	3679	4495
5482	5511	6246	8072	8867	10699	10741
11470	12270	12973	13102	13172	13240	13765
13816	14139	14371	14575	14904	15176	15385
15626	15944	16171	16372	17140	17306	17887
18239	18784	18929	19308	19541	19845	20086
21879	22601	23404	23913	24273	24514	24871
25038	25398	26200	28624	28967	29542	30585
30735	31053	32409	32537			

说明：该运行实例中可以动态的输入待排序序列的长度，这里选择的是 60。从实验结果可以看出，实验数据排序正确。

实验十二、KVM 虚拟机计算性能测试

1、 实验目的

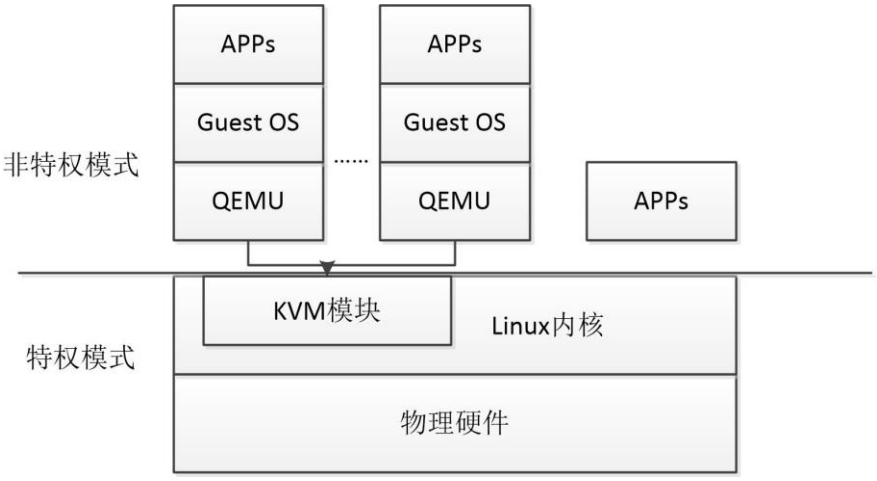
在多核平台上运行多个虚拟机，测试不同环境下多个虚拟机同时运行时的虚拟机性能，从而达到掌握虚拟机性能测试方法，同时理解 KVM 系统虚拟机概念。

2、 实验原理

x86 硬件辅助虚拟化规范提出，虚拟化方案的设计进入了一个新的时代。硬件辅助系统虚拟化规范使得系统虚拟机实现变得相对简单，完全基于此的虚拟机相比于 VMware、Xen 等方案要简单很多。基于这种思想，Qumranet 公司在 2007 年开发出了 KVM（Kernel-based Virtual Machine）虚拟化平台。

KVM 虚拟化平台充分利用了 x86 架构的硬件特性，并在 Linux 内核 2.6.20 版本正式被 Linux 内核社区接受。Qumranet 公司在 2008 年被 Redhat 公司收购，KVM 方案也成为 Redhat 公司主推的系统虚拟化方案。目前 KVM 方案是开源社区最活跃的方案之一，支持的架构也拓展到了 PowerPC、ARM 等其它架构上，未来前景广阔。

KVM 虚拟机结构如下图所示，KVM 本身作为内核的一个模块存在，用来模拟虚拟机的处理器和内存，配合 QEMU 模拟机器的外设便构成了整套虚拟化系统。每个虚拟机在宿主机看来都可以视为 QEMU 的一个运行进程。



龙芯系统虚拟化小组基于龙芯 3 号系列处理器，依托 KVM 平台实现了一套类虚拟化方案 KVM-Loongson，可以稳定运行于龙芯 3 号多核处理器上，支持多

核多虚拟机同时运行，依托 KVM-Loongson 可以更加具体深入的了解 KVM 系统虚拟化平台，并同时理解云计算基础技术之一的系统虚拟化技术。

虚拟机性能测试工具为 SPEC2000，SPEC 基准测试程序是由标准性能评估公司（StandardPerformance Evaluation Corporation，SPEC）提供的测试程序，是目前应用最广泛，最通用的微处理器性能评测工具之一。SPEC 的基准程序都来自实际使用的程序，可以比较可靠的反映出微处理器的处理能力和访存能力。

3、 实验环境

使用多核并行实验环境，使用部署好的支持 KVM-Loongson 的文件系统和内核，利用 libvirt 工具启动虚拟机，获取实验结果。具体操作见实验环境配套手册。

4、 实验要求与内容

a) 探索单个虚拟机的性能数据，列出单虚拟机性能表。
b) 在教学实验平台上编译、运行、调试上述程序，并观察不同处理器核数对性能的影响。

附录：参考实现：

启动测试板，通过ssh网络连接到测试板系统中：

1. 启动虚拟机并运行spec步骤

a) 进入启动虚拟机目录：cd <前置目录>/kvm/qemu-0.14.0/
b) 启动虚拟机：./g 1
c) 进入root用户。
d) 进入spec目录： cd /home/benchmark/spec2000-new
e) 运行./myrun.sh

2. 启动多个虚拟机

a) 进入启动虚拟机目录：cd <前置目录>/kvm/qemu-0.14.0/
b) 启动虚拟机：./g 1
c) 启动虚拟机：./g 2
d) ****
e) 在不同虚拟机上，进入root用户。
f) 进入spec目录： cd /home/benchmark/spec2000-new
g) 运行./myrun.sh

3. 测试结果

Myrun.sh会输出一系列测试程序运行时间，统计这些时间并使用合适方式对比实验结果

实验十三、KVM 虚拟机网络性能测试

1、 实验目的

在多核平台上运行虚拟机，测试 KVM 虚拟机的网络性能，同时更加深入了解 KVM 系统虚拟机相关知识。

2、 实验原理

网络作为虚拟机与外界交互的主要手段，性能影响整个云计算方案的整体使用效果，因此系统虚拟化方案需要较好的网络支持，KVM 虚拟机的网络组织分为两种类型：

1.用户网络（User Networking）：通过在本地配置虚拟子网让虚拟机能够访问主机、互联网或本地网络上的资源，由于虚拟机处在宿主机虚拟子网中，不能从网络上其他机器直接访问虚拟机。

2.虚拟网桥（Virtual Bridge）：通过配置虚拟网桥让虚拟机和宿主机同时通过虚拟网桥访问外网，虚拟机可以通过 dhcp 服务获得外部 IP，IP 地址与宿主机属于同一个网段。

一般地，测试网络性能最简单的方法是网络拷贝，即通过拷贝不同大小的文件估计网络效率。

3、 实验环境

使用多核并行实验环境，连接网络，使用部署好的支持 KVM-Loongson 的文件系统和内核，利用命令行启动虚拟机，将虚拟机配置为用户网络模式，通过拷贝不同大小的文件了解虚拟机网络效率。

4、 实验要求与内容

a) 在用户网络模式下，测试宿主机与虚拟机网络连接效率，通过网络拷贝 5M、50M 和 100M 包大小，获得网络拷贝效率。（选做：测试虚拟机拷贝至网络上其他机器的网络效率）。

附录：参考实现：

启动测试板，通过ssh网络连接到测试板系统中：

1. TAP方式网络性能测量

a) 配置环境：cp<前置目录>/kvm/network/tap/qemu-ifup /etc
b) 进入启动虚拟机目录：cd <前置目录>/kvm/qemu-0.14.0/
c) 启动虚拟机：./g 1
d) 进入root用户。
e) 创建不同大小文件：命令如下：
   i. dd if =/dev/zero of=50M count=50 bs=1M
   ii. 注：如果已经创建好了就不用去修改(文件系统大小有限不要超过300M)
f) 配置自己ip为192.168.0.*网段
   i. Ifconfig eth0 192.168.0.110
g) 查看宿主机网络ip，即tap0或tap*，与启动次序有关
   i. 一般为192.168.0.1*
h) 网络拷贝：
   i. scp 50M root@192.168.0.10:~输入用户名密码即可。
   ii. 拷贝不同大小块获得其数据。

2. 数据实例

i. 单个虚拟机网络数据

方式	**5M**	**50M**	**100M**
TAP方式	**2s**	**26s**	**51s**

附录 1　实验报告格式

实　验　报　告

课程名称					
学生姓名		学号		指导老师	
实验地点			实验时间		

一、 实验室名称：

二、 实验名称：

三、 实验目的：

四、 必修或选修：

五、 实验平台：

六、 实验内容及步骤:

根据不同的实验内容，写出具体的实验步骤。

七、 实验数据及结果分析：

根据不同的实验内容，记录具体的实验数据或程序运行结果。实验数据量较大时，最好制成表格形式。程序设计实验，要以电子文档形式附上程序功能、模块说明、完整源代码，源代码中尽量多带注释；阐述对程序性能改进的具体方法及主要调试过程。

八、 实验结论及总结：