

高性能计算实验报告

|  |  |
| --- | --- |
| 实验题目 | 高性能计算 |
| 姓名 | 欧炜标 |
| 学号 | 3121005358 |
| 专业 | 人工智能 |
| 所在学院 | 计算机 |
| 指导教师 | 王卓薇 |
| 论文提交日期 |  |

目录

一、实验目的 3

二、实验内容 3

三、实验原理 4

3.1、矩阵LU分解算法的设计： 4

3.2、OpenMP编程 6

3.3、MPI编程 7

3.4、程序调优：OpenMP和MPI混合编程 9

四、实验步骤及实验结果 10

4.1、固定线程数时，不同矩阵大小下性能测试 11

4.2、固定矩阵大小时，不同线程数下的性能测试 12

五、实验遇到的问题及其解决方法 16

六、实验结论 16

目录

[目录 2](#_Toc151916873)

[一、实验目的 3](#_Toc151916874)

[二、实验内容 3](#_Toc151916875)

[三、实验原理 3](#_Toc151916876)

[3.1、矩阵LU分解算法的设计： 3](#_Toc151916877)

[3.2、OpenMP编程 5](#_Toc151916878)

[3.3、MPI编程 7](#_Toc151916879)

[3.4、程序调优：OpenMP和MPI混合编程 9](#_Toc151916880)

[四、实验步骤及实验结果 10](#_Toc151916881)

[4.1、固定线程数时，不同矩阵大小下性能测试 10](#_Toc151916882)

[4.2、固定矩阵大小时，不同线程数下的性能测试 12](#_Toc151916883)

[五、实验遇到的问题及其解决方法 16](#_Toc151916884)

[六、实验结论 16](#_Toc151916885)

# 一、实验目的

本实验的目的是通过练习掌握 OpenMP 和MPI 并行编程的知识和技巧。

1、熟悉 OpenMP 和MPI 编程环境和工具的使用；

2、掌握并行程序编写的基本步骤；

3、了解并行程序调试和调优的技巧。

# 二、实验内容

1、独立完成实验内容；

2、了解并行算法的设计基础；

3、熟悉OpenMP和MPI的编程环境以及运行环境；

4、理解不同线程数，进程数对于加速比的影响。

# 三、实验原理

3.1、矩阵LU分解算法的设计：

参考文档sy6.doc所使用的并行算法：

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，，对矩阵A行交叉划分后，编号为i(i=0,1,…,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,…,m-1行数据做行变换，其它处理器利用主行元素对其第i,…,m-1行数据做行变换。

根据上述算法原理用代码表示如下（关键代码）：

for(k = 0;k<N;k++)

{

for (i = 0; i < THREADS\_NUM; i++) {

thread\_data\_arrray[i].thread\_id = i;

thread\_data\_arrray[i].K\_number = k;

thread\_data\_arrray[i].chushu = a[k][k];

//创建线程

rc = pthread\_create(&pid[i], NULL, work, (void\*)&thread\_data\_arrray[i]);

…

}

for (i = 0; i < THREADS\_NUM; i++){

//等待线程同步

rc = pthread\_join(pid[i], &ret); …

}

}

void \*work(void \*arg)

{

struct thread\_data \*my\_data;

my\_data = (struct thread\_data\*)arg;

int myid = my\_data->thread\_id; //线程ID

int myk = my\_data->K\_number; //外层循环计数K

float mychushu = my\_data->chushu; //对角线的值

int s, e;

int i, j;

s = (N-myk-1) \* myid / THREADS\_NUM; //确定起始循环的行数的相对位置

e = (N-myk-1) \* (myid + 1) / THREADS\_NUM;//确定终止循环的行数的相对位置

for (i = s+myk+1; i < e+myk+1; i++) //由于矩阵规模在缩小，找到偏移位置

{

a[i][myk]=a[i][myk]/mychushu;

for (j = myk+1; j < N; j++)

a[i][j]=a[i][j]-a[i][myk]\*a[myk][j];

}

//printMatrix(a);

return NULL;

}

第一部分为入口函数，其创建指定的线程数，并根据不同的线程id按行划分矩阵，将矩阵的不同部分作为参数传递给线程，在多处理器电脑上，不同的线程并行执行，实现并行计算LU分解。

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）做初等行变换，由于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算法。

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。

由于在OpenMP和MPI中并行算法的实现不太一样，所以接下来的两小节中我将分别针对两个编程环境设计LU分解的并行实现。

3.2、OpenMP编程

因为OpenMP是基于线程的编程模型，所以设计了一个基于多线程的OpenMP的LU分解算法，关键代码如下：

for(k = 0;k<N;k++){ omp\_set\_num\_threads(THREADS\_NUM); #pragma omp parallel private(tid) { tid=omp\_get\_thread\_num(); //当前线程ID int myid = tid; printf("hello world from OMP thread %d \n",tid); int myk = k; float mychushu = A[k][k]; int s, e; int i, j; s = (N-myk-1) \* myid / THREADS\_NUM;//确定起始循环的行数的相对位置 e = (N-myk-1) \* (myid + 1) / THREADS\_NUM;//确定终止循环的行数的相对位置 for (i = s+myk+1; i < e+myk+1; i++) //由于矩阵规模在缩小，找到偏移位置 { A[i][myk]=A[i][myk]/mychushu; for (j = myk+1; j < N; j++) A[i][j]=A[i][j]-A[i][myk]\*A[myk][j];　//对行进行初等行变换 } } }

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此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1.0，其它为0，U为A[k][j]中k<=j的元素，其余为0。

这里如果我们使用的是一般的多线程编程，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。详细的代码请参看附带源程序。

3.3、MPI编程

设处理器个数为p，矩阵A的阶数为n，，对矩阵A行交叉划分后，编号为i(i=0,1,…,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,…,m-1行数据做行变换，其它处理器利用主行元素对其第i,…,m-1行数据做行变换，计算完成后，编号为0的处理器收集各处理器中的计算结果，并从经过初等行变换的矩阵A中分离出下三角矩阵L和上三角矩阵U。

关键代码如下：

/\*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 (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);

}

}

3.4、程序调优：OpenMP和MPI混合编程

我们知道OpenMP是基于线程的并行编程模型，一个共享存储的进程由多个线程组成，OpenMP就是基于已有线程的共享编程模型；而MPI属于消息传递的并行编程模型，这个从前两小节中可以看到，因为在LU的MPI实现中，我们对矩阵采用交叉划分，根据p（处理器号）划分行，因此可以对每个划分出来的矩阵采用多线程并行算法，也即可以采用OpenMP计算。

在MPI的编号大于等于my\_rank的处理器利用主行元素对其第i+1,…,m-1行数据做行变换，其它处理器利用主行元素对其第i,…,m-1行数据做行变换部分采用OpenMP计算，所以混合编程的核心代码如下：

/\*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){

int tid;

omp\_set\_num\_threads(THREADS\_NUM);

#pragma omp parallel private(tid)

{

tid = omp\_get\_thread\_num();

int myid = tid;

int myk = i+1;

float mychushu = f[v];

int s,e;

int c,d;

s = (m-myk-1)\*myid/THREADS\_NUM;

e = (m-myk-1)\*(myid+1)/THREADS\_NUM;

for(c=s+myk+1;c<e+myk+1;c++){

a(c,v)=a(c,v)/mychushu;

for(d=v+1;d<M;d++){

a(c,d)=a(c,d)-f[d]\*a(c,v);

}

}

}

}

# 四、实验步骤及实验结果

这里将通过测试在确定的线程下，对于随机生成的不同大小的矩阵，串行及OpenMP&MPI的运行时间来分析程序性能。并通过改变线程的数量，在不同情况下多次测量，测试编译参数对程序性能的影响。

4.1、固定线程数时，不同矩阵大小下性能测试

1. 当线程数为5时：

测试得到的不同矩阵大小下的加速比数据如图1所示：

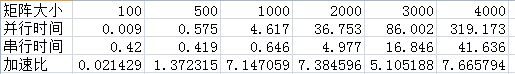
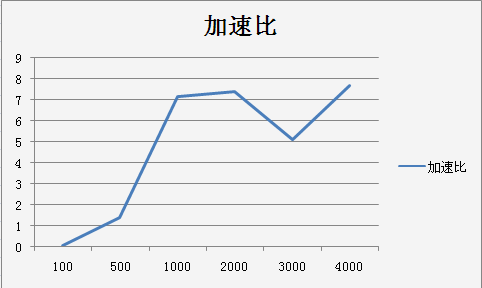


图1

根据图1中所示测得的数据，生成折线图如下：



b．当线程数为100时，测得的实验数据如图2所示：

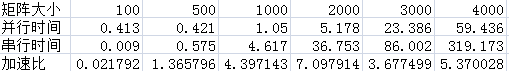
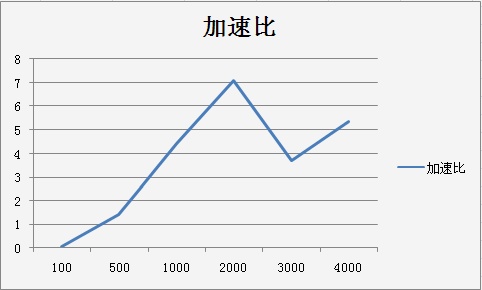


图2

根据图2中所示测得实验数据生成的折线图如下：



结果分析：

从以上的结果可以看出在不同的线程下，OpenMP的性能都会随着矩阵规模的增大而变好，也就是在矩阵规模变大时，OpenMP的加速比变大，虽然在实验中有出现下降，但总体来说加速比在增大。同时，可以看到在矩阵很小时，加速比几乎为零，也即此时的OpenMP运行时间比串行时间久，这主要是在矩阵规模很小时，OpenMP多线程减少的计算时间相比为维护这么多线程花费的时间要少很多，所以在数据规模很小时，不应该采用OpenMP编程并行编程。相反，在数据规模很大时，采用OpenMP并行编程模型，将带来巨大的性能提升。

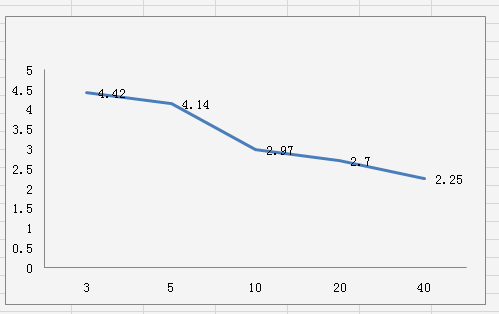
4.2、固定矩阵大小时，不同线程数下的性能测试

a．当矩阵大小为2000时，测得的数据如图3所示：



图3

根据图3测得实验数据，生成的折线图如下：



b．当矩阵大小为4000时，测得的数据如图4所示：

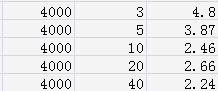
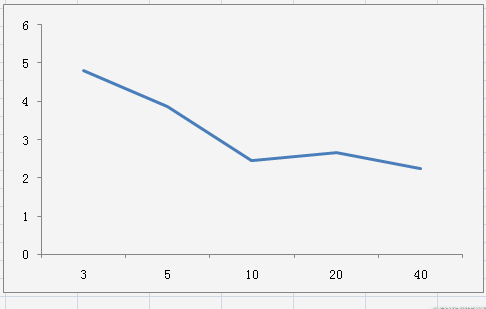


图4

根据图4测得实验数据，生成的折线图如下：



结果分析：

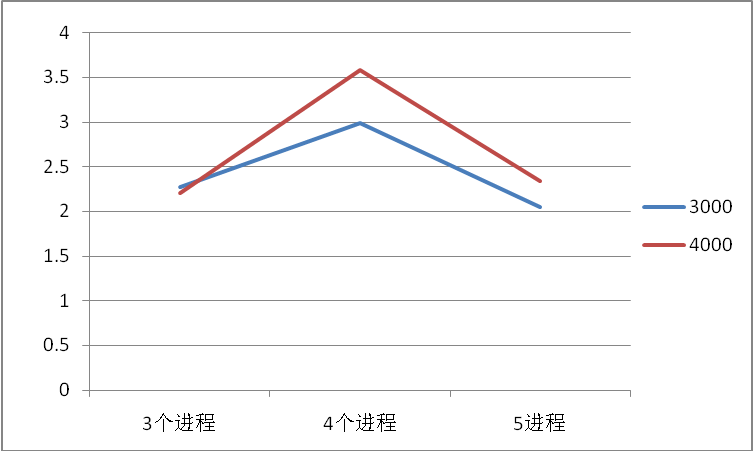
从上面的实验测试可以看出，在不同的线程数量下，OpenMP运行的加速比不同，同线程数量变大时，OpenMP的加速比并没有如所期望的一样变大，相反随线程数量的增加，OpenMP的加速比变小了。因为OpenMP是基于多线程的编程模型，而我们知道多线程程序性能的提高是基于多CPU同时运行线程，而本次程序测试的主机为4核的，也即可以同时运行4个线程，因此当线程数量增多到超过CPU数量时，多余的线程实际上并没有得到执行，相反这个时候还需要额外维护这大量的线程，因此性能降低，所以加速比随线程增多而有所降低。因此，在进行OpenMP并行编程时，我们需要根据所运行的主机的CPU数量合理的设置线程的数量，以获得最大的加速比。

以上是针对OpenMP并行算法与串行算法的性能的比较，下面再简单对不同进程情况下MPI的LU并行算法与串行算法的性能比较。

c. 分别在矩阵大小为3000和4000时测得的实验数据如下：



根据测得的实验数据，生成的折线图如下：



结果分析：

从上面的实验测试结果，可以看出，在进程为4时，MPI的加速比为最大，而在两边变化时，加速比有所下降。也即在进行MPI并行编程时，合理设置进程也是非常重要的，一般也是根据运行主机的CPU的数量来设置，如本次主机的CPU数量为4，所以设置进程数为4，每个CPU分别负责执行一个进程，在进程数量过多时，进程没有被执行，反而还增加了维护进程的开销，我们知道这个开销是很大的。

结论：

无论是MPI并行编程还是OpenMP并行编程，都要在大数据量时才能体现其优点，在运行大数据量时，MPI与OpenMP能极大的提高程序运行性能，提供较高的加速比。因为MPI要维护进程，进行消息传递等，而OpenMP要维护线程等，而这都是要花费时间的，特别是在进程或线程数量很大时，维护、通信、同步等的开销很大，所以在数据量很小时，不能采用MPI或OpenMP编程，因为其开销比其带来的性能提升要大。另外，在采用MPI及OpenMP编程时，要根据运行主机合理的设置进程或线程的数量。否则将不能最大化其性能。具体过程可参照上述分析。

最后感谢xx老师对我们的辛苦教学以及xx助教对我们的悉心指导。

# 五、实验遇到的问题及其解决方法

./configure

出现configure: error: Aborting because C++ compiler does not work. If you do not need a C++ compiler, configure with --disable-cxx

改为：

./configure --disable-cxx

即可正常进行编译。

# 六、实验结论