并行程序设计实验

实验 2 LU 分解

联系方式：NJU-TJL@outlook.com

目录

实验 2 LU 分解---------- 3
一、 实验设计---------- 3
1. OpenMP 实现部分 ---------- 3
1) 串行程序思路 ---------- 3
2) 并行化思想 ---------- 3
2. MPI 实现部分---------- 4
1) 初版程序 ---------- 4
2) 重新设计 LU 分解计算思路---------- 5
3) 并行化思想 ---------- 5
二、 实验结果---------- 6
1. 运行环境 ---------- 6
2. OpenMP 实现运行结果 ---------- 6
3. MPI 实现运行结果---------- 7
4. 结果分析 ---------- 8
三、 实验中遇到的问题及解决办法---------- 8
1. MPI 运行报错---------- 8
四、 实验总结---------- 9
五、 参考文献及资料---------- 9

实验 2 LU 分解

一、 实验设计

1. OpenMP 实现部分

1）串行程序思路

LU 分解可将一个矩阵 A 分解为一个下三角矩阵 L 和上三角矩阵 U 的乘积，即：

$$A = LU$$

并非所有的矩阵都有 LU 分解，而对于非奇异矩阵（任 n 阶顺序主子式不全为 0）的方阵 A，可以采用 Doolittle 分解的方法来完成 LU 分解。

假设 A、L、U 矩阵表示如下：

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$$

$$A = LU = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \times \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

那么在 Doolittle 分解中，计算 L、U 矩阵的公式为：

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} \cdot u_{kj} \qquad (j \geq i)$$

$$l_{ji} = (a_{jl} - \sum_{k=1}^{i-1} l_{jk} \cdot u_{ki}) / u_{ii} \qquad (j > i)$$

从此公式中，我们可以发现：计算u_{ij}所依赖的数据都是下标值小于 i、j 的位置上的值。同理，l_{ji}也是。所以，只要按照下标 i、j 从小到大的顺序，使用二重循环，依次计算u_{ij}和l_{ji}，即可利用此公式算出 L、U 矩阵中的全部值。

2）并行化思想

要进行并行化，我们必须先理清楚计算过程中的数据依赖，即有哪些数据得先算，哪些必须后算。首先，观察u_{ij}的计算公式可以发现，计算u_{ij}依赖的数是$l_{i,1\sim i-1}$和$u_{1\sim i-1,j}$的数据。这里可以通过举一个例子来形象地说明：

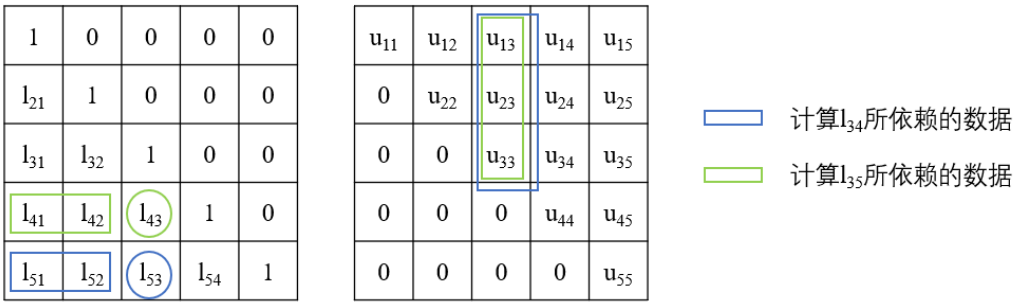
1	0	0	0	0
l_{21}	1	0	0	0
l_{31}	l_{32}	1	0	0
l_{41}	l_{42}	l_{43}	1	0
l_{51}	l_{52}	l_{53}	l_{54}	1

u_{11}	u_{12}	u_{13}	u_{14}	u_{15}
0	u_{22}	u_{23}	u_{24}	u_{25}
0	0	u_{33}	u_{34}	u_{35}
0	0	0	u_{44}	u_{45}
0	0	0	0	u_{55}

计算u_{34}所依赖的数据

计算u_{35}所依赖的数据

在上图中，如果u_{34}和u_{35}所依赖的数据都已计算完成，那么此时u_{34}和u_{35}可并行地进行计算。同理，再来看 L 矩阵部分的计算：



数据的依赖性是类似的，所以也可以并行地计算l_{43}和l_{53}。

综上所述，计算u_{ij}的时候每一行内的元素可以并行地计算，计算l_{ji}的时候每一列内的元素可以并行地计算。

理清楚了并行化思想后，非常容易从上述的串行程序中，改造得到 OpenMP 版本的并行程序。如下图所示，仅需要添加一行编译指示，让内循环进行并行计算即可。

```
//计算L、U矩阵
for (int i = 0; i < N; i++) {
    U[i][i] = A[i][i] - sum_i_j_K(i, i, i);
    L[i][i] = 1;
    #pragma omp parallel for
    for (int j = i+1; j < N; j++) {
        U[i][j] = A[i][j] - sum_i_j_K(i, j, i);
        L[j][i] = (A[j][i] - sum_i_j_K(j, i, i)) / U[i][i];
    }
}
```

2. MPI 实现部分

1）初版程序

写出了 OpenMP 并行程序后，理清楚了并行思想，MPI 也是类似的实现，只是其中的进程通信部分需要额外处理。按照上述的并行原理，也是内循环中各进程并行计算。按照这样的规定分配计算：u_{ij}和l_{ji}由第 j % n_threads 号进程计算（n_threads 为进程总数）。

另外，由于各进程负责计算的元素，所依赖的数据并不是都在本进程，所以还需要在每轮计算前，拿到自己所依赖的数据。比如，上图中，计算u_{34}所依赖的数据中，l_{31}和l_{32}都是由 3 号进程负责计算，而u_{34}是由 4 号进程负责计算（假设 n_threads=8）。

所以，这其中有着大量的进程通信过程。

最终，按此方式实现的 MPI 版本，计算结果正确（与 OpenMP 版本输出结果进行对比，结果一致）。MPI 单进程版本与 OpenMP 的单线程版本运行时间基本一致，但是加速效果很差，最高加速比只能到达 1.2 左右。且随着从 4 到 8 进程，进程数增加，运行时间增加很多，而 16 进程则无法在数分钟内运行结束。

仔细分析，发现这其中的进程通信开销很大，进程数增大带来并行化效果不足以抵消的通信额外开销。至于为什么开销很大，分析发现，其最根本原因，还是因为数据结构与并行划分方式不太契合，比如：本来同一行内的元素，在地址空间上是放在一个块一起的，整个通信则可以传输一个块的数据。但是按照这里采用的并行划分方式，导致了每次只能传送一个单位数据，比如说u_{34}和u_{35}是同一行内的数，且在地址空间处于相邻位置，本来是可以一

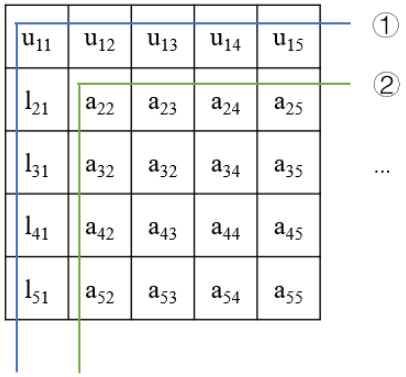
起传送。但是按照这里的计算方式，它们两个分别是 4 号和 5 号进程负责计算（假设进程总数为 8），那么这两个元素就分别需要由 4 号和 5 号进程来 MPI_Send。

而在这种 LU 分解计算方法下（Doolittle 分解，也即递推公式法），数据结构和计算划分方法都难以改变，所以这种计算方式在 MPI 实现下难以产生较好的并行效果。

2）重新设计 LU 分解计算思路

在网上查阅许多资料后，我发现了一篇名为《方阵 A 的 LU 分解的初等行变换法》的论文。其中提到使用矩阵的初等行变换来实现 LU 分解的方法，非常适合于 MPI 并行化。

在线性代数课程中，我们十分熟悉用高斯消元法产生行阶梯型矩阵的过程：从上往下，依次将下三角区域的第 1 列元素变为 0，然后是第 2 列、第 3 列.....这样就可以得到一个上三角矩阵（也是行阶梯型矩阵）。文章中指出，对矩阵 A 进行高斯消元法中的行初等变换，而得到那个上三角矩阵就是 LU 分解的 U。而矩阵 L 的计算则可以在初等行变换过程中一起计算，非常简便：只需在把下三角区域的一列列元素变为 0 的时候，除以主行元素对应列的那个元素，即可算出。这里通过举例来进行说明：



在如图这样的矩阵中，计算过程依次按照①、② ... 的顺序进行初等行变换。但是与行变换不同的地方在于，原来该填 0 的地方，现在要填入l_{ij}。而l_{ij}的计算遵循：在对本列进行初等行变换消元时，比如在如图这个时刻，计算②部分，可以算出$l_{32} = a_{32}/a_{22}$、$l_{42} = a_{42}/a_{22}$、$l_{52} = a_{52}/a_{22}$。其他部分进行正常的初等行变换即可，和高斯消元法产生行梯型矩阵的过程一致。

最终，按照这样的顺序计算，从左上角算到右下角，一层层进行行初等变换过程，最终 a 矩阵的上三角区域就是 U 的上三角的值，下三角就是 L 的下三角的值。当然，L 中，l_{ii}的值为 1。

3）并行化思想

从以上分析过程中可见，最外层的计算是一个循环，从上往下，依次选取第 i 行作为初等行变换中的主行元素，然后对第 i+1 行到第 N 行做初等行变换。这里我们按行来划分矩阵，即第 j 行由第 j % n_threads 号进程来进行计算（做初等行变换）。而每次进行初等行变换前，都先使用 MPI_Bcast 将主行元素广播到所有进程，这样每个进程即可各自独立地完成初等行变换过程，这也就是并行化的核心所在。

二、 实验结果

注：关于程序正确性的验证。测试手工构造的一个 3×3 小矩阵 LU 分解，二者的结果均为正确。之后，将给定的 LU.in 文件作为输入，所实现 OpenMP 和 MPI 程序都会输出 L.out 和 U.out 两个文件作为输出结果。再将 OpenMP 和 MPI 程序输出的结果文件使用 Linux 终端下的 diff 命令进行比对，发现完全一致，所以基本可以确定 OpenMP 和 MPI 所实现的 LU 分解程序是正确的。

1. 运行环境

物理机 CPU：Intel i5-7200U （双核），内存 16GB
虚拟机：CPU 分配 2 个核，内存 4GB
虚拟机系统环境：Ubuntu 16.04（64 位版本）、gcc version 5.4.0 20160609、mpicc for MPICH version 3.3.2

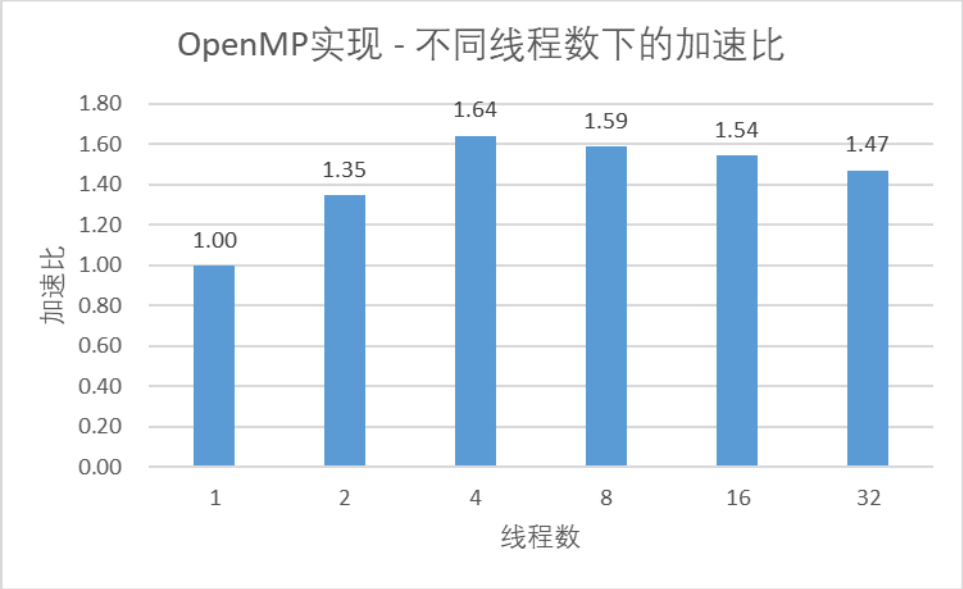
2. OpenMP 实现运行结果

使用如下命令编译源代码：
gcc -fopenmp -o LU_OpenMP LU_OpenMP.c
再按如下格式运行可执行文件：
./LU_OpenMP　4　LU.in
（第 1 个参数表示线程个数、第 2 个代表输入文件；缺省值分别为 1 和 LU.in）
程序会在终端打印输出运行时间，且在同目录下生成 L.out 和 U.out 两个结果文件。

为减少误差，这里使用了 shell 脚本进行 10 轮实验。每一轮中，线程数分别取 1、2、4、8、16 和 32。最终，可得到数据如下表：

线程数		1	2	4	8	16	32
执行时间（单位：s）	Round 1	19.30	14.34	11.94	12.07	12.31	12.93
	Round 2	19.78	14.31	11.58	11.95	12.68	13.42
	Round 3	19.12	14.06	11.87	12.26	13.19	12.98
	Round 4	19.45	14.13	11.49	12.10	12.32	13.19
	Round 5	19.08	14.33	11.59	12.58	12.42	13.31
	Round 6	19.09	14.73	12.12	12.10	12.26	13.28
	Round 7	19.42	13.93	11.21	12.15	12.45	12.83
	Round 8	18.99	14.20	11.71	12.14	12.47	12.87
	Round 9	19.33	14.80	11.98	12.00	12.66	13.29
	Round 10	19.32	14.12	11.93	12.26	12.29	13.04
平均时间		19.29	14.30	11.74	12.16	12.50	13.11
加速比		**1.00**	**1.35**	**1.64**	**1.59**	**1.54**	**1.47**

将加速比绘制成柱状图如下：



3. MPI 实现运行结果

使用如下命令编译源代码：

mpicc -o LU_MPI LU_MPI.c

再按如下格式运行可执行文件：

mpirun -np 2 ./LU_MPI　LU.in

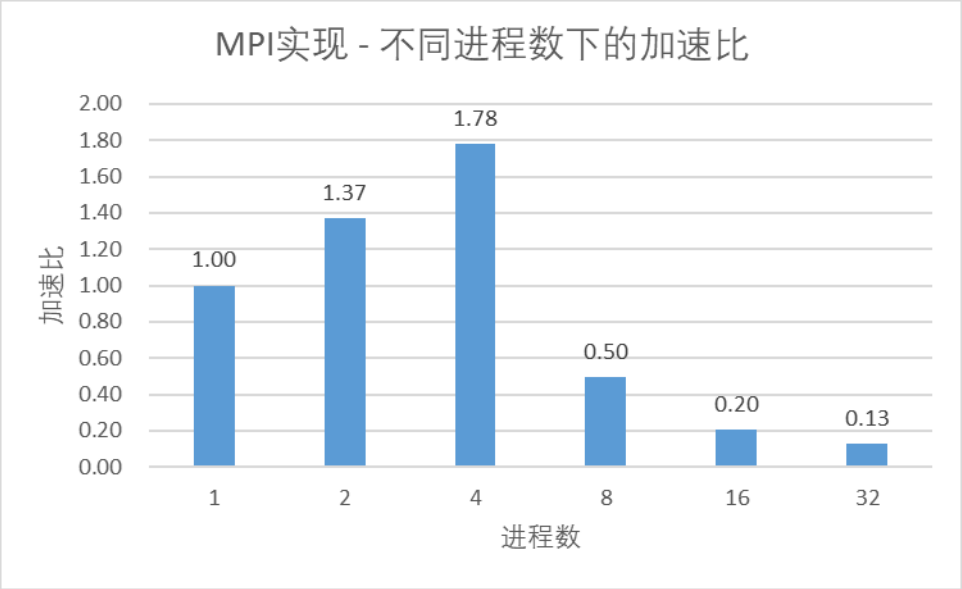
（“2”表示进程个数；“LU.in”代表输入文件路径，缺省值为 LU.in）

程序会在终端打印输出运行时间，且在同目录下生成 L.out 和 U.out 两个结果文件。

为减少误差，这里使用了 shell 脚本进行 10 轮实验。每一轮中，进程数分别取 1、2、4、8、16 和 32。最终，可得到数据如下表：

进程数		1	2	4	8	16	32
执行时间（单位：s）	Round 1	15.59	10.69	8.15	30.45	69.49	134.36
	Round 2	13.96	10.53	8.15	28.17	71.14	114.97
	Round 3	13.80	10.45	8.67	31.06	77.61	114.50
	Round 4	14.10	10.36	8.33	28.35	73.18	111.39
	Round 5	14.44	10.52	8.04	26.87	77.54	115.90
	Round 6	13.91	10.75	8.15	28.20	70.82	113.32
	Round 7	19.60	12.91	8.57	27.64	70.86	115.43
	Round 8	14.29	10.59	8.24	30.37	72.00	114.82
	Round 9	14.11	10.39	8.25	36.15	74.82	115.98
	Round 10	13.94	10.71	8.46	30.07	67.76	118.34
平均时间		14.77	10.79	8.30	29.73	72.52	116.90
加速比		**1.00**	**1.37**	**1.78**	**0.50**	**0.20**	**0.13**

将加速比绘制成柱状图如下：



4. 结果分析

这次实验中，OpenMP 的结果与上次实验的基本一致，即：随着线程数增加，加速比逐渐增大到最大值，再随着线程数的增大，加速比逐渐有所下降。当线程数增加，越来越充分利用 2 个物理 CPU 核的并行资源，加速比也就接近于 2。而线程数增加到一定程度，物理 CPU 核有限的，再增加线程数也无法从物理上增加程序运行的并行程度，反而因为线程数的增加，带来切换线程、线程通信和管理等的额外开销，所以使得运行时间增加，加速比下降。

而本次实验中，MPI 实现中的结果总体趋势上与 OpenMP 一致，加速比都是先上升再下降。但是，后期下降的幅度很大，显示出增加进程带来的额外开销增长较为明显。

经过分析，我认为其主要原因是本次实验 MPI 实现的 LU 分解程序中进程通信带来的额外开销较大。在 MPI 程序中，进程通信体现在每次行变换前，需要将主行元素 Bcast 到所有进程，这个 Bcast 的复杂度可以粗略计算为 O(p)（p 为线程数），总共需要 N 次 Bcast，所以总体进程通信的开销约为 O(N*p)（N 为方阵阶数）。而在结果数据表中，从进程 8、16、32 的平均运行时间来看（此时大部分耗时都在进程通信上），进程数为 2 倍时，时间也正好约为 2 倍，故可以验证此想法。

OpenMP 利用多线程来加速程序执行，只能在共享内存系统上运行。MPI 则是基于进程，所以可以适用于非共享内存系统（比如计算机集群）。而线程与进程对比下，线程是更轻量级的，可以提供更细粒度的并行，且通信等的额外开销更小（线程不用完全复制出一整个独立的内存空间，多个线程可以共用一个；进程则是每个进程有自己独立的内存空间）。所以，此分析也符合本次实验中所呈现的结果（MPI 程序随进程数增大，额外开销增大比较多）。

三、 实验中遇到的问题及解决办法

1. MPI 运行报错

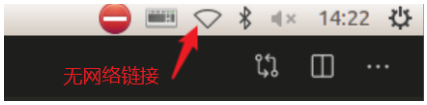
之前可以运行的 MPI 程序，忽然出现报错，无法运行。并且给出如下的报错提示：

```
Invalid error code (-2) (error ring index 127 invalid)
INTERNAL ERROR: invalid error code fffffffe (Ring Index out of range) in MPID_nem_tcp_init:373
Fatal error in PMPI_Init: Other MPI error, error stack:
MPIR_Init_thread(586).............:
MPID_Init(224)....................: channel initialization failed
MPIDI_CH3_Init(105)...............:
MPID_nem_init(324)................:
MPID_nem_tcp_init(175)............:
MPID_nem_tcp_get_business_card(401):
MPID_nem_tcp_init(373)............: gethostbyname failed, tjl-virtual-machine (errno 0)
```

解决：

根据提示信息的最后一行，"gethostbyname failed"，说明这是个与网络通信有关的错误，而这里的 MPI 都运行在本机这一个节点上，所以猜测可能是网络连接出现问题。

果然，检查 Ubuntu 的网络链接，发现没有启用联网。启用后即可以正常运行程序。



四、 实验总结

- 使用 OpenMP 可以较为方便地从串行代码中修改得到并行代码，但是操作的灵活性不如 MPI，会有更多的局限性，适用面不如 MPI 广，比如无法在计算机集群上使用。

- MPI 程序设计需要仔细考量进程间通信的设计，以免发送阻塞等问题。比如，进程不能尝试 Recv 接收源为自己进程的数据，否则就会处于一直阻塞。所以相比 OpenMP，MPI 的程序设计更为复杂，需要考虑更多的细节。

- 在本次实验中，OpenMP 与 MPI 的并行加速效果都还不错（最高加速比都比较接近 2，即 CPU 核心数），但是它们各自适用的地方有不同。OpenMP 非常适合改造计算主体为 for 循环的串行程序，所以这里使用 Doolittle 分解（即递推公式法）来完成 LU 分解比较适合；MPI 则非常灵活，可以完成一些复杂进程通信操作，所以这里适合使用初等行变换的方法来完成 LU 分解则非常不错，可以得到更为高效的程序。同时，MPI 程序中，在进程数增大到一定程度时，我们需要考虑到进程通信带来额外开销的影响。

五、 参考文献及资料

[1] 徐晓飞，曹祥玉，姚旭,等. 一种基于 Doolittle LU 分解的线性方程组并行求解方法[J]. 电子与信息学报, 2010, 32(008):2019-2022.

[2] 周涛，满迪，高文鹏,等. 基于矩阵 LU 分解的并行处理[J]. 电脑知识与技术, 2016, 12(021):219-221.

[3] 杨成，陈进之，魏吉朝. 方阵 A 的 LU 分解的初等行变换法[J]. 西北轻工业学院学报, 1998(2):136-138.

[4] 赵祥宇. 基于 Spark 平台的大矩阵 LU 分解及求逆算法的研究与实现[D]. 2016.

[5] LU 分解 - 百度文库. https://wenku.baidu.com/view/cd431e02de80d4d8d15a4f16.html#

[6] LU 分解（图解）_qq_40688707 的博客-CSDN 博客. https://blog.csdn.net/qq_40688707/article/details/89256737

[7] 线性代数笔记 10——矩阵的 LU 分解 - 我是 8 位的 - 博客园. https://www.cnblogs.com/bigmonkey/archive/2018/08/29/9555710.html