南開大學
Nankai University

计算机学院

并行程序设计作业报告

MPI 编程

姓名：徐文斌

学号：2010234

专业：计算机科学与技术

2022 年 6 月 23 日

目录

1 普通高斯消去算法

1.1 实验介绍

高斯消元法（Gaussian Elimination）是数学上线性代数中的一个算法，可以把矩阵转化为行阶梯形矩阵。高斯消元法可用来为线性方程组求解，求出矩阵的秩，以及求出可逆方阵的逆矩阵。

本实验将在 x86 架构的金山云集群上利用 MPI 分布式多进程编程技术进行普通高斯算法的多进程优化实验，以加深对 MPI 编程的理解和运用。

1.2 程序设计思路

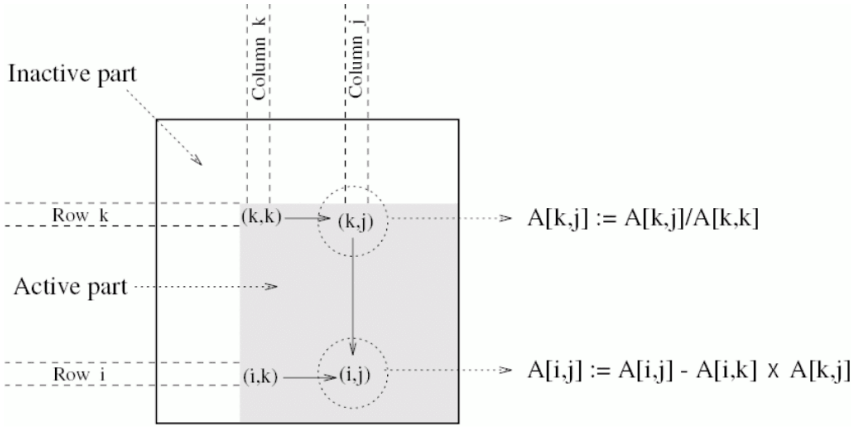

图 1.1: 高斯消去算法示意图

高斯消去的计算模式如图1.1所示，在第 k 步时，对第 k 行从 (k, k) 开始进行除法操作，并且将后续的 k + 1 至 N 行进行减去第 k 行的操作，串行算法如下面伪代码所示。

```
1  procedure LU (A)
2  begin
3      for k := 1 to n do
4          for j := k + 1 to n do
5              A[k, j ] := A[k, j] / A[k, k];
6          endfor;
7          A[k, k] := 1.0;
8          for i := k + 1 to n do
9              for j := k + 1 to n do
10                 A[i, j ] := A[i, j ] - A[i, k] * A[k, j];
11             endfor;
12             A[i, k] := 0;
13         endfor;
14     endfor;
15 end LU
```

由高斯消去伪代码容易推出该算法时间复杂度为 $O(n^3)$。观察高斯消去算法伪代码，我们可以将不同的行分配给不同的进程，每一轮消去过程中，每个进程只处理被分配到的行。关键是我们如何进行行的分配和进程间的通信。任务分配方面，我们可以使用块划分和循环划分；通信方面，我们可以使用广播通信和流水线通信；至于通信函数的选择，我们可以使用阻塞式通信函数和非阻塞式通信函数。具体代码请见文末 GitHub 链接。

1.3 实验环境说明

本实验使用教学提供的 x86 架构的金山云集群进行程序性能测试。

1.4 实验方案设计

1. 首先我们编写块划分算法，并对其进行 OpenMp 以及 SSE 优化，探究不同问题规模、不同节点数目、单个节点中不同线程数目对算法性能的影响。2. 然后我们探究不同的任务划分方式，即块划分和循环划分，对程序性能的影响。3. 之后我们将探究不同的通信方式，即广播通信和流水线通信对程序性能的影响（这里所说的广播通信指一个进程向其余所有进程发送数据的通信方式）。4. 以上均使用阻塞式通信函数，在最后一部分，我们将尝试使用非阻塞通信函数来优化程序的性能。

1.5 实验过程与结果

1.5.1 块划分算法结合 OpenMp 与 SSE

首先，我们编写块划分算法，每个进程负责整个矩阵中的若干连续的行的消去操作，同时，我们使用阻塞式广播通信的方式。考虑该算法的负载均衡，随着矩阵消去的进行，每轮消去过程中，编号较小的进程所作的工作逐渐小于编号较大的进行所作的工作。因此，总体上，编号较大的进程进行的运算要多于编号较小的进程进行的运算，负载并不是很均衡。

首先，我们分别固定块划分算法进程数为 2、4、8，并分别在 64、128、256、512、1024、2048 问题规模下对程序运行时间进行测试，结果如表1所示。可以看到，在数据规模较小时，块划分算法运行时间要长于平凡算法的运行时间，且进程数目越大 MPI 算法的运行时间越大，这是由于数据规模较小时，通信开销占比较大造成的，且通信开销随着进程数目的增多而增大。而当数据规模较大时，块划分算法得到了较好的性能提升，且相同规模下进程数越多程序运行时间越短。块划分算法加速比如表2所示，当数据规模较大时，各个进程数目下的块划分算法均得到了较好的加速比。

算法\规模		64	128	256	512	1024	2048
平凡算法		0.376606	2.93622	23.113	183.931	1481.21	12026.8
块划分算法	2	0.441895	2.29004	16.2961	125.938	1004.86	8020.89
	4	2.35596	3.14111	11.812	74.1189	621.317	4366.89
	8	17.21	9.26001	15.3059	48.9399	302.126	2287.27

表 1: 不同进程数下块划分算法程序运行时间（ms）

算法\规模		64	128	256	512	1024	2048
块划分算法	2	0.852252	1.28217	1.418315	1.460488	1.474046	1.499435
	4	0.159852	0.934771	1.956739	2.481567	2.383984	2.754088
	8	0.021883	0.317086	1.510071	3.758304	4.902623	5.258146

表 2: 不同进程数下块划分算法程序加速比

算法\规模		64	128	256	512	1024	2048
块划分 SSE	2	0.25708	1.26904	8.95483	68.2629	548.652	4462.84
	4	1.12793	2.33594	7.5708	45.9631	306.736	2391.39
	8	2.06787	2.17603	6.30493	30.1509	169.284	1370.52

表 3: 不同进程数下块划分算法结合 SSE 程序运行时间（ms）

算法\规模		64	128	256	512	1024	2048
块划分 SSE	2	1.464937	2.313733	2.581065	2.69445	2.699726	2.694876
	4	0.333891	1.256976	3.052914	4.00171	4.828941	5.029209
	8	0.182123	1.349347	3.665861	6.100349	8.749852	8.775355

表 4: 不同进程数下块划分算法结合 SSE 程序加速比

接下来，我们对块划分算法进行 SSE 优化，优化后的算法运行时间如表3所示，加速比如表4所示。可以看到优化后的算法和原算法的性能变化趋势是相似的，相比于普通的块划分算法，结合 SSE 优化的块划分算法具有更好的性能，问题规模较大时，可以达到超过进程数目的加速比。

最后，我们在 SSE 算法的基础上再对 MPI 算法进行 OpenMp 多线程优化。首先考虑程序可能体现出来的性能，通过查阅金山云 CPU 信息，我们发现，每个核心只支持 1 个线程，并未开启超线程机制，如图1.2所示，且 OpenMp 多线程无法跨节点并行，由于资源较少，可能导致多线程加速效果不是十分明显。

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel Xeon Processor (Cascadelake)
Stepping:              6
CPU MHz:               2593.906
```

图 1.2: 金山云 CPU 信息

这里为了简化工作，我们固定进程数为 2，测试程序在每个进程创建 2、4、6 线程情况下的运行时间及加速比如表5和表6所示。可以看到，各个线程数下，当数据规模较小时，程序的加速比小于 1，这应该是由于线程同步等开销造成的，随着问题规模逐渐增大，程序的加速比也逐渐增大，最终当数据规模为 2048 时，三个线程数下程序均达到了大于 4 的加速比。这与我一开始的思考不相符，考虑可能的原因，我认为可能是使用多线程可以更加充分地使用硬件资源，当一个线程由于通信阻塞时，可以交由另一个线程进行运算，这就充分地利用了硬件资源，从而带来了性能上的提升。与单纯进行 SSE 优化的表4中进程数为 2 时的程序运行时间做对比，数据规模较大时，多线程优化带来的加速效果还是

较为明显的。

线程数\规模	64	128	256	512	1024	2048
2	194.603	381.755	92.4099	305.279	1922.28	2480.2
4	3.62207	3.2959	13.6331	55.51	339.903	2609.18
6	2.05493	3.85205	11.6179	56.699	582.781	2463.78

表 5: 进程数为 2 时不同线程数下块划分算法程序运行时间（ms）

线程数\规模	64	128	256	512	1024	2048
2	0.001935	0.007691	0.250114	0.602501	0.770549	4.849125
4	0.103975	0.89087	1.695359	3.313475	4.357743	4.609418
6	0.18327	0.762249	1.98943	3.24399	2.541624	4.881442

表 6: 进程数为 2 时不同线程数下块划分算法程序加速比

1.5.2 任务循环划分算法–负载更加均衡

根据上一部分的分析我们得出，块划分 MPI 算法的负载均衡性能较差，接下来我们将探索更好的负载均衡算法–循环划分算法。

与块划分不同的是，循环划分算法将矩阵的每一行循环划分给每个进程，每个进程负责对满足 row%num_proc==rank（row 为行号，num_proc 为总的进程数，rank 为当前进程的编号）的行进行消去操作。思考这种任务划分方式的负载均衡性能，不难想到，随着矩阵消去的进行，每轮消去过程中每个进程要处理的行数是大致相等的。总体上，各个进程所执行的操作是相近的。因此，我们有理由认为循环划分算法具有更好的性能。在这一部分中，我们仍然实用阻塞式广播通信的方式实现进程间的通信。

接下来，我们对编写的循环划分算法程序进行性能测试。首先，我们分别固定程序执行进程数为 2、4、8，并分别在 64、128、256、512、1024、2048 问题规模下对程序运行时间进行测试，结果如表7所示。其表现出来的性能和块划分算法是相似的，数据规模较小时，由于通信开销占比较大，未能取得较好的加速效果；当数据规模较大时，可以看到明显的性能提升。循环划分算法的加速比如表8所示，算法取得了较好的加速比。

算法\规模		64	128	256	512	1024	2048
平凡算法		0.376606	2.93622	23.113	183.931	1481.21	12026.8
循环划分	2	0.258057	1.68311	12.4951	93.325	760.053	5870.75
	4	2.3999	5.1499	15.813	70.4351	426.235	3124.75
	8	2.77905	5.62695	14.543	49.8931	246.043	1664.73

表 7: 不同进程数下循环划分分算法程序运行时间（ms）

算法\规模		64	128	256	512	1024	2048
循环划分	2	1.459391	1.744521	1.849765	1.970865	1.948825	2.048597
	4	0.156926	0.570151	1.461645	2.611354	3.475102	3.848884
	8	0.135516	0.521814	1.589287	3.686502	6.020127	7.224475

表 8: 不同进程数下循环划分算法程序加速比

下面，我们将块划分算法和循环划分算法进行对比，以进程数 8 为例，如表9所示。可以看到，除了数据规模为 512 时，循环划分算法的加速比略逊于块划分算法的加速比。在其他的数据规模下，循

环划分算法的加速比均要大于块划分算法的加速比，甚至当数据规模为 2048 时，循环划分算法的加速比为块划分算法加速比的 1.374 倍，达到了不错的优化效果。这也证明了循环划分算法相比于块划分算法具有更好的负载均衡效果。

算法\规模	64	128	256	512	1024	2048
块划分	0.021883	0.317086	1.510071	3.758304	4.902623	5.258146
循环划分	0.135516	0.521814	1.589287	3.686502	6.020127	7.224475

表 9: 进程数为 8 时块划分算法和循环划分算法程序加速比

此外，我还对循环划分算法进行了 SSE 优化，对程序运行时间进行测试，结果如表10和11所示，结合 SSE 优化再次提升了程序性能。

算法\规模		64	128	256	512	1024	2048
循环划分 SSE	2	0.203857	1.04907	6.91382	50.8792	397.394	3324.77
	4	2.17285	4.80493	11.8052	47.6069	255.809	1966.04
	8	2.84399	5.55005	13.0139	39.335	169.382	1027.86

表 10: 不同进程数下循环划分算法结合 SSE 程序运行时间（ms）

算法\规模		64	128	256	512	1024	2048
循环划分 SSE	2	1.847403	2.798879	3.343014	3.615053	3.727308	3.617333
	4	0.173324	0.611085	1.957866	3.863537	5.790297	6.117271
	8	0.132422	0.529044	1.776024	4.676014	8.74479	11.70082

表 11: 不同进程数下循环划分算法结合 SSE 程序加速比

1.5.3 流水线通信算法

接下来，我们尝试更改进程间的通信方式，使用阻塞式流水线通信方式来实现进程间的通信。具体为将上一部分编写的循环划分算法中的广播通信改为流水线通信方式，并测试程序性能。我们思考程序可能的性能体现。首先，与广播通信不同的是，流水线通信方式分摊了通信开销，由原来的一对多变为一对一通信，这样可以缓解做第 k 行除法运算的进程的通信压力。接下来，我们思考程序运行时间方面可能的变化，在广播通信方式中，做第 k 行除法运算的进程向其余 n-1 个进程发送数据后才开始自己负责的矩阵的消去工作，每轮消去过程中程序的瓶颈大概率为该进程。而在流水线算法中，每轮消去过程中流水线上的最后一个进程为程序运行的瓶颈，在该进程进行消去之前，流水线上也串行执行了 n-1 次通信。由此可见，广播通信和流水线通信每轮消去过程中关键路径所花费的时间应该是相近的，均为 n-1 次通信时间加上单个进程的消去时间，我们编写的流水线通信算法性能可能得不到好的提升。

下面，我们对编写的循环划分结合流水线通信算法程序进行性能测试。首先，我们分别固定程序执行进程数为 2、4、8，并分别在 64、128、256、512、1024、2048 问题规模下对程序运行时间进行测试，结果如表12所示。将该表与表7做对比，我们可以看到，广播通信方式和流水线通信方式不同进程数、不同问题规模下程序的运行时间均是相近的，这就验证了我们的上述分析。结合 SSE 优化算法的性能结果也是类似的，在此不做赘述。

算法\规模		64	128	256	512	1024	2048
流水线算法	2	0.250977	1.64087	12.0139	96.4021	778.384	5824.1
	4	2.75195	6.07397	15.979	67.4229	422.048	3127.47
	8	3.37402	6.66699	15.77	49.345	247.485	1626.94

表 12: 循环划分结合流水线通信算法程序运行时间（ms）

1.5.4 非阻塞式通信

经过我们上述一系列的工作，我们首先将块划分任务分配算法改进为行循环划分任务分配算法以得到更好的负载均衡，使程序性能得到了一定的提升。之后我们尝试使用流水线通信方式替代广播通信方式，但程序并没有取得好的性能提升。

在我们上一部分的分析中，不难发现，线程间的通信等待时间是阻碍程序性能提升的另一个重要原因。通过对算法的进一步思考，我发现，由于之前的算法均使用阻塞式通信函数进行通信，这就导致在广播通信中，执行第 k 行除法的进程必须等待其余所有进程都接收到数据后才可以执行它自己后续的消去工作，而这些等待并不是必须的；同时，在流水线通信中也有类似现象，每个进程必须等待下一个进程接收到数据后才开始自己的消去工作。我们可以尝试使用非阻塞式通信函数来减少进程间的通信等待时间。接下来，我对算法进行了进一步的改进，使用循环划分的任务方式、非阻塞式广播通信。这里我们只尝试对广播通信进行非阻塞式优化而不尝试对流水线通信方式进行非阻塞式优化，原因为即使对流水线通信方式进行非阻塞式优化，每轮消去过程中，流水线中的一个进程也必须等待成功接收到前一个进程传递的数据后才可以进行自己的消去工作并向下一个进程传递信息。也就是说，无论通信是否阻塞，当流水线中的最后一个进程接收到数据并开始进行矩阵消去时，在这之前也已经串行进行了 n-1 次数据传输操作，这与阻塞式通信耗费的时间是一致的，使用非阻塞式通信函数大概率不会得到性能提升。因此，这里我们只尝试对广播通信算法进行非阻塞式通信优化。

下面，我们对编写的循环划分结合非阻塞式广播通信算法程序进行性能测试。首先，我们固定程序执行进程数为 4，并分别在 64、128、256、512、1024、2048 问题规模下对程序运行时间进行测试，将其与阻塞式广播通信程序进行对比，结果如表13所示（在其他进程数下结果是类似的，在此不做展示）。

算法\规模	64	128	256	512	1024	2048
阻塞式通信	2.3999	5.1499	15.813	70.4351	426.235	3124.75
非阻塞式通信	1.9892	4.3473	11.9973	58.7036	389.991	2910.23

表 13: 进程数为 4 时阻塞式广播通信与非阻塞式广播通信程序运行时间对比（ms）

由表13我们可以很明显地看到，循环划分结合非阻塞式广播通信算法性能要优于阻塞式广播通信算法。这是因为，与阻塞式通信不同的是，每轮消去过程中，执行时间最长的进程，即对第 k 行进程除法操作的进程在进行完除法操作后，可以立刻进行自己所负责的其余行的消去操作，而不需要再等待所有的进程都接收到数据后再进行自己所负责的消去操作，这就在一定程度上提升了程序的性能。

2 实验源码项目 GitHub 链接

实验源码项目 GitHub 链接