改进的并行高斯全主元消去法

孙济洲，樊莉亚，孙 敏，于 策，张绍敏
（天津大学计算机学院，天津 300072）

摘 要：为减少 Gauss 全主元消法的运行时间，用多进程与多线程混合的方式对其进行了并行化，同时对该算法进行了改进. 采用 MPI 并行 I/O 技术提高读取数据文件的速度，降低对内存的需求；采用标志数组避免了选主元后换行带来的通信开销；使用线程模型确定最优线程数，提高运行消去的速度；通过预先发布机制降低回代求解步骤的时间复杂度. 实际运行结果表明，随着方程组阶数增大，加速比也逐渐增大，对于 5 000 元的方程组，8 进程同时运行，加速比可达 6.68，并行效率稳定在 0.85 左右. 这表明该算法具有可扩展性和稳定的并行效率，适用于大规模并行计算.

关键词：Gauss 全主元消去法；MPI；Pthreads；并行算法
中图分类号：TP301.6 **文献标志码**：A **文章编号**：0493-2137(2006)09-01115-05

Improved Parallel Complete Gaussian Pivoting Elimination

SUN Ji-zhou, FAN Li-ya, SUN Min, YU Ce, ZHANG Shao-min
(School of Computer Science, Tianjin University, Tianjin 300072, China)

Abstract: To reduce the executing time of complete Gaussian pivoting elimination, it is parallelized in a manner combined multi-thread and multi-process, and improved at the same time. Message passing interface (MPI) parallel I/O technology is adopted to increase the speed of reading data files and reduce the requirement to memory. A flag array is used to avoid the communication cost caused by exchanging rows after finding pivot elements. A thread model is used to find the optimal number of threads, which speeds up the elimination. And the time complexity of backward substitution is reduced by preinforming strategy. Actual implementation shows that the speedup enhances as the number of equations increases, for a linear system of five thousand equations, the speedup of eight processes can be as high as 6.68, the parallel efficiency stabilizes at about 0.85. All these indicate that the algorithm has good scalability and stable parallel efficiency, and can be used in large-scaled parallel computation.

Keywords: complete Gaussian pivoting elimination; message passing interface; Pthreads; parallel algorithm

Gauss 全主无消去法是一种常用的解线性方程组的算法. 它具有较高的精度和稳定性，但时间复杂度也较高，并且随着方程组阶数增长，时间复杂度增长较快. 因此，将其并行化是非常有价值的.

国内外对 Gauss 消去法并行化进行的研究中，McGinn 等[1]实现了多进程和多线程 2 种方式的并行 Gauss 消去法，并指出多线程方式扩展性较差. Quinn 的研究[2]表明，在并行化过程中，数据分割有 2 种策略：面向行的算法并行算主元，但在消去时有较大的广播通信量；面向列的算法则刚好相反. Quinn 提出了一种流水化的、面向行的算法，将计算和通信并行化[2]，但它本质上是 Gauss-Jourdan 消去法的并行化. Murthy 等[3]提出了 SGE 算法，对矩阵采用分治策略，省去回代求解步骤，且回代求解在整个算法中只占用较少的时间[1]，但该算法只是部分选主元. Zhang 和 Maple[4]采用循环数据映射、流水化的通信模型，得到好的负载平衡，但它串行选择主元，在消去时需要较大通信开销. 本算法在一定程度上解决了上述问题.

1 Gauss 全主元消去法流程

Gauss 全主元消去法需要经过输入数据、选主元、

收稿日期：2005-09-26；**修回日期**：2006-03-21.
作者简介：孙济洲(1949—)，男，教授，jzsun@tju.edu.cn

消去、回代求解和写回文件. 图 1 是 Gauss 全主元消去法的流程图(其中 n 表示方程组的阶数).

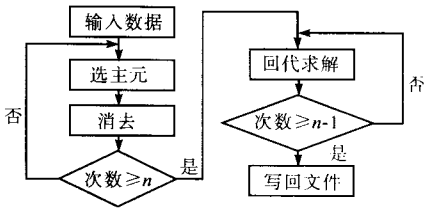

图 1 算法流程

Fig. 1 Flow chart of the algorithm

2 对算法的改进

在以下的讨论中,假设要解的方程组为

$$\boldsymbol{Ax}=\boldsymbol{b} \tag{1}$$

式中:$\boldsymbol{A}$ 是一个 $n\times n$ 的矩阵;$\boldsymbol{b}$ 是一个 $n\times 1$ 的列向量.

2.1 输入数据

输入数据一般较多,因此从文件输入. 对于传统串行算法,这一步需串行的将 $\boldsymbol{A}$ 与 $\boldsymbol{b}$ 中的数据依次读入内存,总时间为

$$t_{\mathrm{in}}=n(n+1)t_{\mathrm{r}} \tag{2}$$

式中:t_{r} 为程序从硬盘中读取单位数据的时间;n 为启动并行程序中的进程数.

本文采用 MPI 并行 I/O 输入数据. 这种技术在 MPI-2 标准中实现,由许多快速读写文件的 API 组成,是一种比较新的技术. MPI 并行 I/O 的特点是,各进程并发的从文件的不同位置读入数据[6]. 所以,MPI 并行 I/O 与传统方法相比,无论是运行时间还是对内存的需求量都相应减小. 改进后这一步的执行时间为

$$t_{\mathrm{in}}=\frac{1}{N}(n+1)nt_{\mathrm{r}} \tag{3}$$

对于没有采用 MPI 并行 I/O 技术的并行实现,一般先由主进程将所有数据读入自己的内存,再将数据发送到相应的其他进程. 这一步的总时间为

$$t_{\mathrm{in}}=(n+1)nt_{\mathrm{r}}+\frac{N-1}{N}(n+1)nt_{\mathrm{t}} \tag{4}$$

式中 t_{t} 为进程之间传送单位数据的时间.

由此可见,采用 MPI 并行 I/O 技术,这一步节省了时间. 在内存需求方面,如果未采用 MPI 并行 I/O 技术,需要的内存是 $(n+1)ns$,而对算法改进后,单个节点内存的需求只有 $\frac{(n+1)ns}{N}$. 式中 s 为程序中所用的类型单位数据占用的内存空间.

2.2 选主元

假设选主元执行到第 i 步,对于传统串行算法,这一步需要在系数矩阵右下角的 $(n-i+1)\times(n-i+1)$ 子阵中寻找主元,因此一步执行时间大约是 $t_{\mathrm{pi}}(i)=(n-i+1)^2t_{\mathrm{a,s}}$,总的执行时间是(进程内部内存中交换行的时间由于较短,在此可以忽略不计).

$$t_{\mathrm{pi}}=\sum_{i=1}^{n}(n-i+1)^2t_{\mathrm{a,s}}=\sum_{i=1}^{n}i^2t_{\mathrm{a,s}}=\frac{n(n+1)(2n+1)}{6}t_{\mathrm{a,s}} \tag{5}$$

式中 $t_{\mathrm{a,s}}$ 为机器执行一次加减法的时间.

并行 Gauss 全选主元消去法中各进程先独立的选出本地主元,然后通过全局规约的方法选出全局的主元. 在选定全局主元之后,按照算法,要将主元所在行与第 i 行相互交换(假设当时是进行第 i 步选主元),将主元所在列与第 i 列交换. 在并行算法中,由于采用了面向行的分割法,交换列还是在同一个进程中进行,而交换行将涉及到进程之间的通信. 为了避免进程间通信所带来的开销,对算法进行了改进,在程序中引入了标志数组 num[ ],记录了每一行在矩阵中对应的行号,这样当需要交换第 i 与第 j 行的时候,只需交换 num[i] 与 num[j] 的值,而不需要将整行数据全部进行交换,省去了大量的数据通信[7]. 在对算法进行改进后,这一步的执行时间减少为

$$t_{\mathrm{pi}}=\frac{n(n+1)(2n+1)}{6N}t_{\mathrm{a,s}} \tag{6}$$

如果不进行这一步优化,在最优的情况下(每次消元都不需要交换行),时间与该优化的算法相当,但在最坏情况下(每次消元都需要换行),这一步的时间将变为

$$t_{\mathrm{pi}}=\frac{n(n+1)(2n+1)}{6N}t_{\mathrm{a,s}}+2(n+1)nt_{\mathrm{t}} \tag{7}$$

假设数据是随机分布的,那么这一步的执行时间的数学期望将是

$$t_{\mathrm{pi}}=\frac{n(n+1)(2n+1)}{6N}t_{\mathrm{a,s}}+\frac{2(N-1)}{N}(n+1)nt_{\mathrm{t}} \tag{8}$$

可见,算法的改进确实缩短了程序运行时间.

2.3 消 去

这一步采用多线程消去. 目前使用多线程通常有 2 种选择:OpenMP 和 Pthreads. OpenMP 容易使用,但灵活性较差,不能控制线程的创建与销毁,不能控制具体任务的划分等,在性能上也有一定的损失;Pthreads 虽然使用较为复杂,但可以完全控制线程的生命周期,性能上的损失也较小. 本算法采用 Pthreads 多线程.

下面讨论只考虑线程数小于等于 CPU 数的情形.

采用多线程会带来一些额外的开销,例如线程的创建、销毁和等待线程等. 在算法的消去阶段,每一步

的计算量是不同的,第 1 步消去的计算量最大,最后一步消去计算量最小,总之,第 i 步消去需要处理一个 $(n-i+1)\times(n-i+2)$ 的矩阵. 由此可知,当消元刚开始进行,计算量较大,线程数应尽量大一些,以便提高性能;但随着消去步数的增加,计算量逐渐减小,线程带来的开销所占的比重会增大,线程数应当小一些. 因此应根据消元步数的不同选择最适合的线程个数.

为解决上述问题,提出了以下模型. 假设消元执行到第 i 步,线程个数为 m. 那么如果创建一个线程的时间是 t_c,消灭一个线程的时间是 t_k,线程之间等待的平均时间是 t_w(如果负载平衡较好,t_w 可以接近零),m 个线程带来的的总开销是 $(t_c+t_k+t_w)(m-1)$,执行一步消去需要一次乘法和一次加法,如果 m 个线程的任务量相同,并且系数矩阵的数据完全随机分步,消去计算所用的时间是 $\frac{(n-i)(n-i+1)}{Nm}(t_{a,s}+t_{m,d})$. 那么在 m 个线程的情况下,第 i 步 $(0\leqslant i\leqslant n-1)$ 消去的总时间

$$t_e(i,m)=(t_c+t_k+t_w)(m-1)+\frac{(n-i)(n-i+1)}{Nm}(t_{a,s}+t_{m,d}) \tag{9}$$

式中 $t_{m,d}$ 为执行一次乘除法运算的时间.

这样,要解决的问题可归纳为选择合适的 m,使得 $t_e(i,m)$ 的值最小. 为此,对 $t_e(i,m)$ 求 m 的偏导数得

$$\frac{\partial t_e(i,m)}{\partial m}=(t_c+t_k+t_w) \tag{10}$$

令偏导数的值等于 0,解得当 $t_e(i,m)$ 值最小时,m 的取值为

$$m_o=\sqrt{\frac{(n-i)(n-i+1)}{N}\,\frac{t_{a,s}+t_{m,d}}{t_c+t_k+t_w}} \tag{11}$$

要计算上式的值需要开方运算和较多的浮点运算,并且消元的每一步都需要重新计算该值,这会带来较大的开销,为此需对其进行适当的简化.

首先利用 $\sqrt{(n-i)(n-i+1)}\approx(n-i)$,其次令 $\alpha=\frac{t_{a,s}+t_{m,d}}{t_c+t_k+t_w}$,式(11)可化简为

$$m_o=(n-i)\sqrt{\frac{\alpha}{N}} \tag{12}$$

α 可以看作是计算时间与创建线程时间的比值,这个值与 CPU 速度以及所用的操作系统等有关. 用实验的方法测得在并行机上(IBM Cluster 1350, Linux)的值为 $5.7\times10^{-6}\sim6.4\times10^{-6}$. 进程个数 N 在程序一开始就可以确定并且不会变化,因此 $\sqrt{\frac{\alpha}{N}}$ 的值在程序一开始就可以得到,每一步不用重新计算,这样最优线程数 m_o 变成了消去步数 i 的线性函数,从而减少了性能上的开销.

得到最优线程数 m_o 之后还有 2 个问题:一是计算得到的 m_o 是一个浮点数,而线程数 m 应当是一个整数;二是为了性能上的考虑,线程数应当小于等于并行机单节点 CPU 数. 因此,如果计算得到的 m_o 小于等于 CPU 个数,m 取 m_o 的四舍五入值;如果 m_o 大于 CPU 个数,m 就取 CPU 的个数. 假设所使用的并行机每个节点有 M 个 CPU,伪代码为

$$m_o=(n-i)\sqrt{\frac{\alpha}{N}}$$

if $m_o>M$

　　$m=M$

else

　　$m=[m_o+\frac{1}{2}]$

采用这个模型,在理想的情况 $m_o\leqslant M$ 时,第 i 步消元所用的时间

$$t_{min}(i)=2(n-i)\sqrt{\frac{(t_c+t_k+t_w)(t_{a,s}+t_{m,d})}{N}}-(t_c+t_k+t_w) \tag{13}$$

如果 $m_o>M$,第 i 步消元所用的时间

$$t_e(i,M)=(t_c+t_k+t_w)(M-1)+\frac{(n-i)(n-i+1)}{NM}(t_{a,s}+t_{m,d}) \tag{14}$$

对于串行算法,这一步的执行时间是 $(n-i)(n-i+1)(t_{a,s}+t_{m,d})$,明显要大于改进后的运行时间(无论是对于 $m_o\leqslant M$ 的情形还是 $m_o>M$ 的情形).

对于并行算法,如果不用多线程,第 i 步消元所用的时间($m=1$ 时)为

$$t_e(i,1)=\frac{(n-i)(n-i+1)}{N}(t_{a,s}+t_{m,d}) \tag{15}$$

对于 $m_o\leqslant M$ 的情形,式(13)的值一定小于式(15),因为式(13)是全局最优解;如果是 $m_o>M$ 的情形,由于式(9)中 m 在 $(0,m_o)$ 区间上单调递减,因此式(14)的值小于式(15)的值. 无论哪种情形,采用该模型都可以减少运行时间.

如果使用多线程,但不用该模型进行优化,第 i 步消元所用的时间是 $t_e(i,M)$ 的值,即式(14). 使用该模型,在 $m_o\leqslant M$ 的理想情况,由于式(13)小于式(14),可以减少运行时间;如果是 $m_o>M$ 的情况,采用模型的时间与直接使用多线程的时间相当. 因此,相对于直接使用多线程,使用该模型同样可以减少运行时间.

2.4　回代求解

这一步用迭代公式

$$\begin{cases} x_n = \dfrac{b_n}{a_{nn}} \\ x_i = (b_j - \sum_{j=i+1}^{n} a_{ij}x_j)/a_{ii} \end{cases} \tag{16}$$

得到方程组的最终解[8]. 后一步迭代要用到前一步计算得到的结果,因而不利于并行化. 对于串行算法和未加改进的并行 Gauss 全主元消去法,这一阶段需串行执行. 对于串行算法,第 i 步回代所用的时间

$$t_i(i) = (i-1)T_{a,s} + it_{m,d} \approx i(t_{a,s} + t_{m,d}) \tag{17}$$

所用的总时间是

$$t_i = \sum_{i=1}^{n} t_i(i) = \frac{n(n+1)}{2}(t_{a,s} + t_{m,d}) \tag{18}$$

对于未加改进的并行 Gauss 全主元消去法,还要加入通信时间. 总时间变为

$$t_i = \frac{n(n+1)}{2}(t_{a,s} + t_{m,d}) + (N-1)t_t \tag{19}$$

该算法采用了一种预先发布的机制,每算出一个未知数的值,先通过广播的方式发给其他节点,其他节点收到这些值之后,先乘以对应的系数 a_{ij},并从 b_i 中减去,这样当算到 x_i 时,只需要除以相应的系数 a_{ii} 即可(乘法和减法工作已完成). 这种方法将前面完全串行的算法变成了有一定并行性的算法,改进后回代求解的总时间

$$t_i = t_{m,d} + (n-1)(2t_{m,d} + t_{a,s} + t_{mc}) \tag{20}$$

式中 t_{mc} 为广播单位数据的时间.

时间复杂度从 $O(n^2)$ 下降到 $O(n)$.

3 运行效果

用上面讨论的所有改进将算法并行化实现,对程序实际的运行效果进行了测试,程序的运行环境是 IBM 1350 机群(Cluster 1350 机群,采用 Linux 操作系统,共 8 个节点,每个节点 2 个 Intel XEON CPU,主频 2.4 GHz).

算法执行时间是包括读写文件在内的所有步骤的总执行时间. 通过执行时间进而可以算出加速比和并行效率[9]. 另外,笔者编写了一个生成随机方程组的辅助程序(generatedata),生成方程组的系数矩阵(由几乎全都非零的随机数构成)和方程组的精确解,分别放在不同的文件中,比较该程序计算出的解与辅助程序生成的精确解,可看出计算结果的精度.

采用双精度(double)浮点数进行计算,通过对比可以看出,该算法计算出的解精确到小数点 6 位以后.

在进程个数不同、方程组阶数不同的各种情况下,并行程序的运行结果如表 1 所示.

由表 1 可知,表内的时间快于文献[5]的实验数据,并且文献[5]仅仅是对消元这一个步骤进行计时,而本文测量到的数据是对整个过程的计时. 仅从运行时间的比较不足以判断改进后算法的好坏,因为运行时间很大程度上取决于并行机硬件配置和节点之间的网络带宽. 更有说服力的参数是程序运行的加速比和并行效率. 为了计算这些值,必须测量出串行程序的运行时间[9],其结果如表 2 所示.

表 1 不同情况下并行程序的运行时间

Tab. 1 Executing time of parallel programs in different situations

s

进程数	阶数				
	1 000	2 000	3 000	4 000	5 000
2	8.013	61.392	216.258	504.198	1 020.674
3	5.557	42.029	144.553	336.485	667.432
4	5.417	33.412	108.667	246.386	484.107
5	4.647	26.831	91.031	208.181	408.689
6	4.891	24.471	77.012	172.623	331.134
7	4.336	22.009	67.166	150.031	284.543
8	4.471	20.082	60.676	134.313	255.788

表 2 不同情况下串行程序的运行时间

Tab. 2 Executing time of serial programs in different situations

s

方程组阶数	运行时间
1 000	14.056 76
2 000	110.761 80
3 000	370.881 80
4 000	868.371 60
5 000	1 704.461 00

表 3 是各种情况下的加速比,图 2 是测得的加速比曲线图. 观察加速比曲线图,对于阶数较小的方程组,如1000阶方程组,由于计算-通信比较小,当进程数增多时,进程间的通信开销增大较快,因此加速比不理想,有时甚至进程数增加,加速比反而下降;随着方程组阶数的增加,计算-通信比也逐渐增加,并行计算的优势开始显现出来,加速比也更加理想,5 000 阶方程组当进程数为 8 时的加速比已达到 6.68. 通过对加速比的分析可以看出,随着问题规模的扩大,加速比更加趋近线性加速比,这表明改进后的算法具有很好的扩展性.

表 3　不同情况下并行程序的加速比

Tab. 3　Speedups of parallel programs in different situations

进程数	阶数				
	1 000	2 000	3 000	4 000	5 000
2	1.764 3	1.810 8	1.721 8	1.728 2	1.674 6
3	2.544 1	2.645 1	2.575 9	2.589 5	2.560 9
4	2.610 0	3.327 3	3.426 6	3.536 5	3.530 6
5	3.042 6	4.143 3	4.090 5	4.185 5	4.182 2
6	2.890 6	4.542 8	4.835 1	5.047 7	5.161 7
7	3.261 1	5.051 0	5.543 9	5.807 8	6.006 8
8	3.162 5	5.535 7	6.136 9	6.487 4	6.682 1

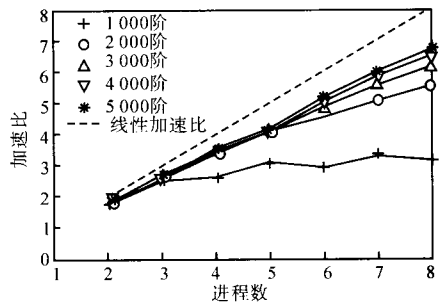

图 2　加速比

Fig. 2　Speedups

表 4 是各种情况下的并行效率,与加速比的情况类似,对于阶数较小的方程组,随着进程数增加,通信开销增长较快,并行效率也下降较快;对于阶数较大的方程组,通信所占的比重较小,因此加速比的减少也较小. 通过表 4 可以发现对于 5 000 阶的方程组,不论进程数如何变化,并行效率总能维持在 0.85 左右. 对并行效率的分析可以看出,改进后的算法具有稳定的并行效率. 通过以上分析表明,该算法适合于大规模并行计算.

表 4　不同情况下并行程序的并行效率

Tab. 4　Parallel efficiency of parallel programs in different situations

进程数	阶数				
	1 000	2 000	3 000	4 000	5 000
2	0.882 2	0.905 4	0.860 9	0.864 1	0.837 3
3	0.848 0	0.881 7	0.858 6	0.863 2	0.853 6
4	0.652 5	0.831 8	0.856 7	0.884 1	0.882 7
5	0.608 5	0.828 7	0.818 1	0.837 1	0.836 4
6	0.481 8	0.757 1	0.805 8	0.841 3	0.860 3
7	0.465 9	0.721 6	0.792 0	0.829 7	0.858 1
8	0.395 3	0.692 0	0.767 1	0.810 9	0.835 3

4　结　语

对 Gauss 全主元消去法进行了改进,在输入数据阶段利用 MPI 并行 I/O 技术并行读取文件,减少文件读取时间,降低对内存的需求;在选主元阶段使用标志数组避免了进程之间的通信;在消去阶段采用线程模型提高运算速度;在回代求解阶段使用预先发布机制将这一步的复杂度从 $O(n^2)$ 下降到 $O(n)$. 通过实际运行表明该算法有可扩展性和稳定的并行效率.

参考文献:

[1] McGinn S F, Shaw R E. Parallel Gaussian elimination using openMP and MPI [C] // *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications*. Moncton, NB, Canada, 2002: 169—174.

[2] Quinn M J. *Parallel Programming in C with MPI and OpenMP*[M]. Beijing: Tsinghua University Press, 2005.

[3] Murthy K N B, Murthy C S R. A new Gaussian elimination-based algorithm for parallel solution of linear equations[C] // *Proceedings of IEEE Region 10's Ninth Annual International Conference*. Singapore, 1994: 82—85.

[4] Zhang J, Maple C. Parallel solutions of large dense linear systems using MPI [C] // *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*. Warsaw, Paland, 2002: 312—317.

[5] Wasilewski M. Project: Parallel Gaussian Elimination[EB/OL]. http: // www. cgl. uwaterloo. ca/ ~ mmwasile/cs775/project. pdf, 2004-04.

[6] 都志辉. 高性能计算并行编程技术——MPI 并行程序设计[M]. 北京:清华大学出版社,2001.
Du Zhihui. *High Performance Computing Parallel Programming—MPI Parallel Programming* [M]. Beijing: Tsinghua University Press, 2001 (in Chinese).

[7] 陈国良. 并行计算[M]. 北京:高等教育出版社,1999.
Chen Guoliang. *Parallel Computing* [M]. Beijing: Higher Education Press, 1999 (in Chinese).

[8] 李庆杨,王能超,易大义. 数值分析[M]. 武汉:华中科技大学出版社,1986.
Li Qingyang, Wang Nengchao, Yi Dayi. *Numerical Analysis* [M]. Wuhan: Huazhong University of Science and Technology Press, 1986 (in Chinese).

[9] Barry Wilkinson, Michael Allen. 并行程序设计[M]. 陆鑫达,译. 北京:机械工业出版社,2002.
Barry Wilkinson, Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstation and Parallel Computers*[M]. Lu Xinda Trans. Beijing: China Machine Press, 2002 (in Chinese).