



南開大學
Nankai University

计算机学院
并行程序设计

特殊高斯消去法的并行优化

并行程序设计期末研究报告

姓名：丁屹、卢麒萱

学号：2013280、2010519

专业：计算机科学与技术

2022 年 7 月 10 日

目录

1 问题描述	2
2 研究设计	2
2.1 测试用例	3
2.2 实验环境和相关配置	3
2.3 串行稀疏矩阵算法	3
2.4 串行位元矩阵算法	3
2.5 并行稀疏矩阵算法	4
2.6 并行位元矩阵算法	4
3 算法分析	6
3.1 正确性分析	6
3.2 正确性验证	6
3.3 复杂度分析	6
3.4 运行情况分析	6
3.4.1 计时方式	6

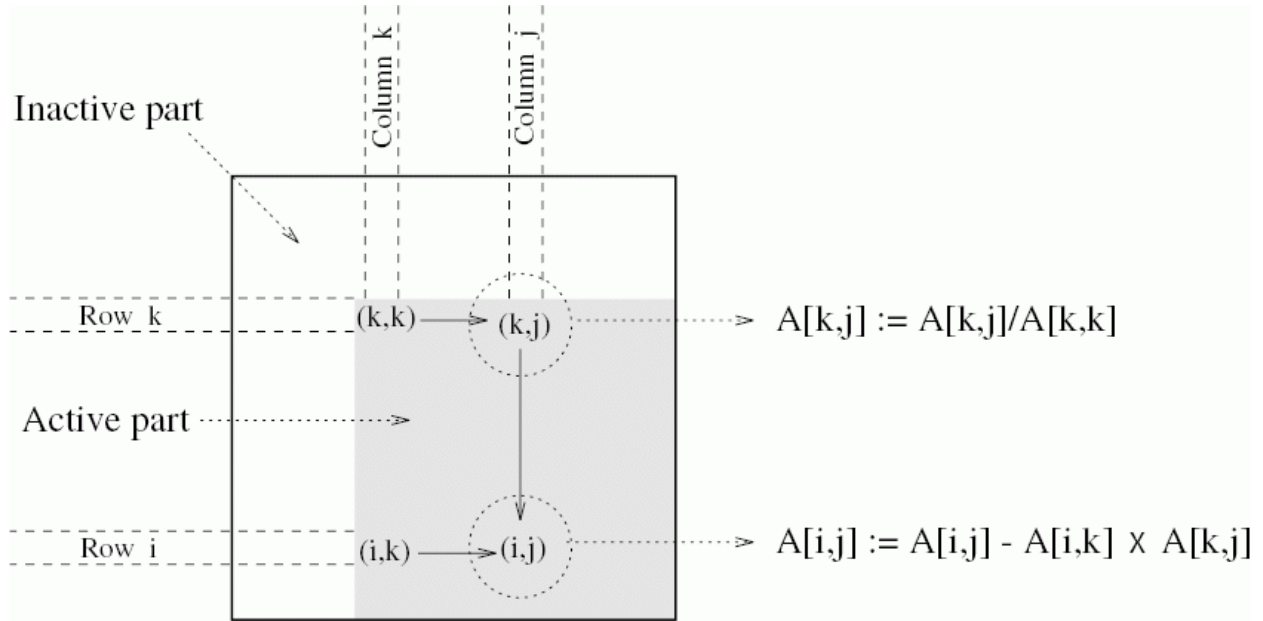


图 1.1: 高斯消去法示意图

1 问题描述

普通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。

Algorithm 1 普通高斯消元算法伪代码

```

1: function LU
2:   for  $k := 0$  to  $n$  do
3:     for  $j := k + 1$  to  $n$  do
4:        $A[k, j] := A[k, j]/A[k, k]$ 
5:     end for
6:      $A[k, k] := 1.0$ 
7:     for  $i := k + 1$  to  $n$  do
8:       for  $j := k + 1$  to  $n$  do
9:          $A[i, j] := A[i, j] - A[i, k] * A[k, j]$ 
10:      end for
11:       $A[i, k] := 0$ 
12:    end for
13:  end for
14: end function

```

特殊高斯消去法

2 研究设计

项目链接: <https://github.com/NeoWans/Parallel-Programming-Final>

2.1 测试用例

测试用例由老师提供的 Groebner.7z 压缩包解压后获得，总共 11 组数据，软链接至 res/目录下，命名规则为% 组号%.0 (非零消元子)、% 组号%.1 (被消元行)、% 组号%.2 (消元结果)

2.2 实验环境和相关配置

实验在本地 x86 Arch Linux 环境下完成，使用 Makefile 构建项目，开启 Ofast 加速；使用的 CPU 为 AMD Ryzen 4800HS (8C16T)，系统 RAM 大小为 38.6G，显卡为 nVIDIA RTX 2060 6G。

2.3 串行稀疏矩阵算法

使用 STL list 存储矩阵中每行的非零位置，逐行放入嵌套的外层 STL list；使用 STL map 存储消元行首项与消元行的映射。

Listing 1: 串行稀疏矩阵消元部分

```

1 void gauss(list_matrix_t& m) {
2     for (auto& eliminatee : m.op) {
3         while (!eliminatee.empty()) {
4             auto key = *(eliminatee.cbegin());
5             auto& eliminator = m.pool[key];
6             if (eliminator.empty()) {
7                 eliminator = eliminatee;
8                 break;
9             } else {
10                auto jt = eliminatee.begin();
11                auto it = eliminator.cbegin();
12                while (it != eliminator.cend() && jt != eliminatee.end())
13                    if (*it > *jt) eliminatee.insert(jt, *it++);
14                    else if (*it == *jt) jt = eliminatee.erase(jt), ++it;
15                    else ++jt;
16                for (; it != eliminator.cend(); ++it) eliminatee.push_back(*it);
17            }
18        }
19    }
20 }

```

2.4 串行位元矩阵算法

使用 STL bitset 倒序存储矩阵每行，逐行放入外层 STL list；使用 STL map 存储消元行首项与消元行的映射。

其中值得注意的是，STL bitset 提供了快速查询最低真值位索引的内建成员函数 `__Find_first()`，与之对应的是算法需要 $lp(E[i])$ 操作，即获得被消元行第 i 行的首项，然而正序存储时 `__Find_first` 函数得到的是被消元行第 i 行的末项，因此需要倒序存储。具体实现使用了“bsmap”宏²处理映射关系。由于 STL bitset 需要使用常量模板参数声明，因此使用了“matrix_max_sz”常量，大小为 85401，即测试样例中的最大矩阵大小。bsmap 可以保证定义域和陪域在 $[0, matrix_max_sz)$ 内且为双射，同时满足 $\forall x \in [0, matrix_max_sz) \text{ bsmap}(\text{bsmap}(x)) = x$ 。

Listing 2: bsmmap 宏

```
1 #define bsmmap(i) (matrix_max_sz - 1 - (i))
```

Listing 3: 串行位元矩阵消元部分

```
1 void gauss(bitset_matrix_t& m) {
2     for (auto& eliminatee : m.op) {
3         while (eliminatee.any()) {
4             auto key = bsmmap(eliminatee._Find_first());
5             auto& eliminator = m.pool[key];
6             if (eliminator.none()) {
7                 eliminator = eliminatee;
8                 break;
9             } else eliminatee ^= eliminator;
10        }
11    }
12 }
```

2.5 并行稀疏矩阵算法

由于分析测试数据发现使用链表存储矩阵运行时间远大于 bitset 存储，常数过大，即使在稀疏矩阵下也仅有内存占用低的优势。然而并行位元矩阵算法在稀疏矩阵下的内存占用也是完全可以接受的，没有研究的必要。

2.6 并行位元矩阵算法

使用 STL bitset 倒序存储矩阵每行，逐行放入外层数组；由于矩阵的秩比较小，而且测试样例存在稠密矩阵，所以决定改用数组存储消元行首项与消元行的映射。

并行化库选择了 C++11 的标准库 thread，可以最大程度保证可移植性。并发线程数量读取 thread::hardware_concurrency 的提示，在本地机器上为 16。

经过观察可以发现，外层循环就是并行化改造的入手点。这里采取了按照循环划分的方法，将线程 id 与行对线程总数的模数对应。由于存在被消元子成为非零消元子的可能，非零消元子不是只读访问。因此使用可升级的读写锁最符合直觉。即访问非零消元子前需要获取读锁，如果需要对消元子复制，则将读锁升级为写锁。这样借助 boost::upgrade_lock 就有了一种使用读写锁实现 4，具体来说，其中：boost::upgrade_lock<boost::shared_mutex> 实现加读锁，boost::upgrade_to_unique_lock<boost::shared_mutex> 实现将读锁升级为写锁。

Listing 4: upgrade_lock 位元矩阵消元部分

```
1 const auto num_thread = thread::hardware_concurrency();
2 boost::shared_mutex mutex_main;
3 void thread_callback(size_t index, bitset_matrix_t& m) {
4     for (size_t local_index = index; local_index < m.op.size();
5         local_index += num_thread) {
6         auto& eliminatee = m.op.at(local_index);
7         while (eliminatee.any()) {
8             auto key = bsmmap(eliminatee._Find_first());
```

```

9         boost::upgrade_lock<boost::shared_mutex> read_lock(mutex_main);
10        auto& eliminator = m.pool[key];
11        if (eliminator.none()) {
12            boost::upgrade_to_unique_lock<boost::shared_mutex> write_lock(
13                read_lock);
14            eliminator = eliminatee;
15            break;
16        } else eliminator ^= eliminator;
17    }
18 }
19 }
20 void gauss(bitset_matrix_t& m) {
21     list<thread> thread_pool;
22     for (unsigned i = 0; i < num_thread; ++i)
23         thread_pool.push_back(thread(thread_callback, i, ref(m)));
24     for (auto& t : thread_pool) t.join();
25 }

```

然而经过测试，这种实现的计算效率甚至不如串行算法（表1），因此有了下面的 `call_once` 实现。

再次观察可以注意到，实际上映射数组里的每一种消元子只会被初始化一次，能够对应保证线程安全的前提下初始化单例类型的模型，即双重检查锁定模型。换言之存在减少加锁次数提高性能的优化空间。对于支持 C++11 以上的编译器，存在 `call_once` 和 `once_flag` 原语。对 `call_once` 传入 `once_flag` 标志和回调函数 `func` 即可实现线程安全的前提下仅第一次成功调用回调函数 `func`。称为“积极调用”，并对 `once_flag` 翻转；此后再次向 `call_once` 传入 `once_flag` 就直接返回，称为“消极调用”。避免了双重检查锁定模型的繁琐和危险。因此我们有了另一种不使用 `boost` 的更快的实现 5。

Listing 5: `call_once` 位元矩阵消元部分

```

1 const auto num_thread = thread::hardware_concurrency();
2
3 void thread_callback(size_t index, bitset_matrix_t& m) {
4     for (size_t local_index = index; local_index < m.op_sz;
5         local_index += num_thread) {
6         auto& eliminatee = m.op[local_index];
7         for (auto idx = eliminatee._Find_first(); idx < eliminatee.size();
8             idx = eliminatee._Find_first()) {
9             auto key = bsmapi(idx);
10            bool exist = true;
11            call_once(m.flag_v[key], [&]() {
12                exist = false;
13                m.pool[key] = eliminatee;
14            });
15            if (exist) eliminatee ^= m.pool[key];
16            else break;
17        }
18    }
19 }
20
21 void gauss(bitset_matrix_t& m) {

```

表 1: 多线程不同实现运行情况

Matrix rank	serial bitset (ms)	16 threads rwlock bitset (ms)	16 threads call_once bitset (ms)
130	0.099817	1.406358	2.402953
254	3.173488	36.153601	2.272141
562	3.08886	24.075979	1.877817
1011	98.256365	1296.45696	12.519157
2362	426.11729	5158.95778	58.782182
3799	5026.35341	62177.1912	649.369009
8399	30331.4359	>1200000	5585.23381
23045	220484.326	>1200000	49605.6164
37960	322201.459	>1200000	77384.6553
43577	1016832.99	>1200000	252625.626
85401	440.782952	6335.83031	85.025176

```

22 list<thread> thread_pool;
23 for (unsigned i = 0; i < num_thread; ++i)
24     thread_pool.push_back(thread(thread_callback, i, ref(m)));
25 for (auto& t : thread_pool) t.join();
26 }

```

3 算法分析

3.1 正确性分析

3.2 正确性验证

由于% 组号%.2 (样例正确消元结果) 被链接到 res/目录下, 而% 组号%.out (程序计算结果) 被输出到 misc/ 目录下。

对于串行算法, 由于消元次序, 选取消元子的次序是固定的, 因此答案是固定的, 使用 diff -wB misc/% 组号%.out res/% 组号%.2 即可在忽略输出格式差异的前提下判断消元是否正确。每次运行完成只需运行单行脚本 6 即可判断正确性。经过验证, 所有实现均保证了正确性。

Listing 6: 单行 Bash 脚本

```

1 for i in {1..11}; do diff -wB "misc/$i.out" "res/$i.2"; done

```

而对于并行算法, 消元次序, 和选取消元子的次序都不固定。由于异或运算的特殊性, 以上两种次序不定不影响结果的正确性, 但是也意味着消元的等价结果很多, 不能简单的用 diff 程序判断正确性, 因此可以使用 Mathematica 等工具辅助检查。

3.3 复杂度分析

3.4 运行情况分析

3.4.1 计时方式

表 2: 不同方法运行情况

Matrix rank	serial list (ms)	serial bitset (ms)	16 threads call_once bitset (ms)
130	0.03233	0.099817	2.402953
254	1.867949	3.173488	2.272141
562	4.735001	3.08886	1.877817
1011	78.952469	98.256365	12.519157
2362	827.574329	426.11729	58.782182
3799	15708.6549	5026.35341	649.369009
8399	273776.479	30331.4359	5585.23381
23045	>1200000	220484.326	49605.6164
37960	>1200000	322201.459	77384.6553
43577	>1200000	1016832.99	252625.626
85401	96746.3559	440.782952	85.025176