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

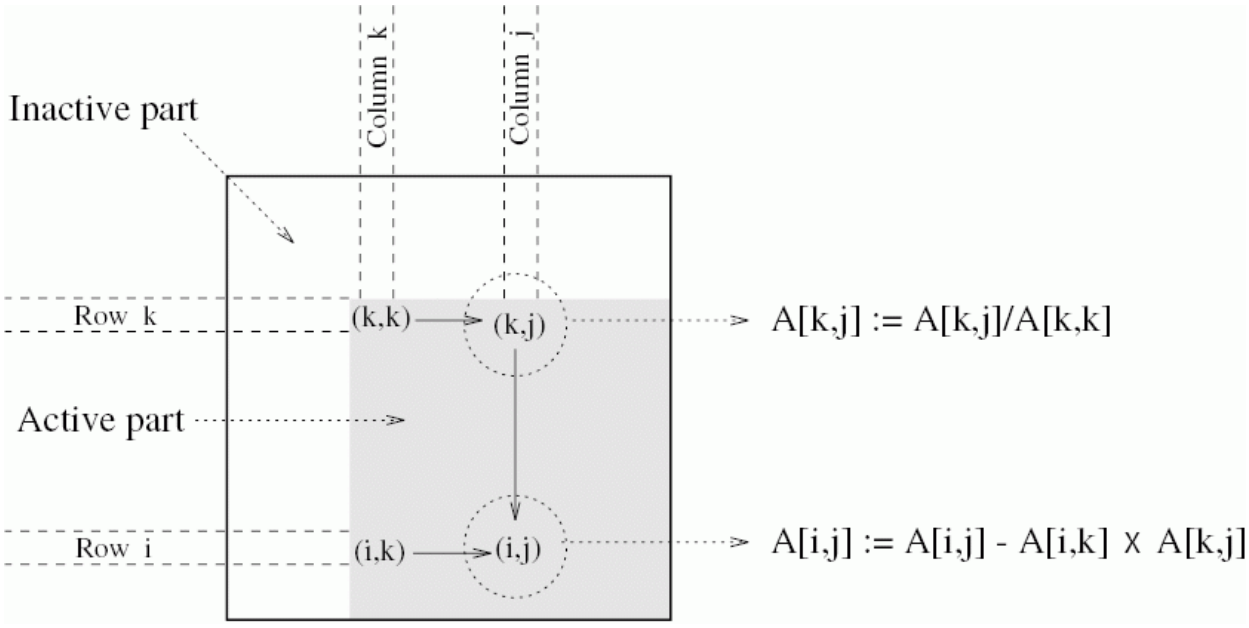

图 1.1: 高斯消去法示意图

1 问题描述

1.1 普通高斯消去法

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。

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
```

1.2 特殊高斯消去法

本算法源自布尔 Gröbner 基计算。Gröbner 基是一种广泛应用于复杂高次方程体系的计算方法，它是 Buchberger 首先提出的。它的实质是从多项式环的任何一个理想的生成元上，对一组“好的”生

成元进行描述和计算，从而对理想的构造进行分析，并对其进行一些理想的运算。由于数学、科学和工程学中的许多问题都可以用多元多项式方程组表示 (例如, 理想, 模块和矩阵),Gröbner 基的代数算法在理论物理学、应用科学和工程学中具有广泛的应用。在 HFE80 的 Gröbner 基计算过程中，高斯消元时间占比可以达到 90% 以上。考虑到此算法中高斯消元的特殊性，为加快高斯消元速度，设计此算法。

1.2.1 符号说明

R: 所有消元子构成的集合

$R[i]$：首项为 i 的消元子

E：所有被消元行构成的数组

$E[i]$：第 i 个被消元行

$lp(E[i])$：被消元行第 i 行的首项

这里首项的含义：首项是指某行下标最大的非零项的下标，如某行为 011000，从左到右下标分别为 5,4,3,2,1,0，那么首项为 4，因为该行非零项下标为 3,4，其中最大值为 4。

1.2.2 算法伪代码

Algorithm 2 串行算法伪代码

```
1:  function GAUSS
2:      for i := 0 to m − 1 do
3:          while E[i]! = 0 do
4:              if R[lp(E[i])]! = NULL then
5:                  E[i] := E[i]−R[lp(E[i])]
6:              else
7:                  R[lp(E[i])] := E[i]
8:                  break
9:              end if
10:         end while
11:     end for
12:     return E
13: end function
```

在这里，最外层循环代表了对每一个被消元的行的遍历。内层循环代表每一条被消元行，若行没有被消为 0，则按照其第一项选择消元；如果有适当的消元符，则用该消元子消元，或者将该被消元行作为消元子，参加下一步的高斯消元。

1.2.3 数据结构

可采用位向量方式存储每个消元子和被消元行，优点是消元操作变为位向量异或操作，算法实现简单，且适合并行化，易达到更高的并行效率，缺点是 Gröbner 基计算中产生的消元子和被消元行非常稀疏，非零元素（1 元素）在 5 % 以下，位向量存储和计算可能并非最优。也可采用类似倒排链表的存储及方式——可认为是稀疏 0 / 1 矩阵的紧凑存储方式，每个消元子和被消元行只保存 1 元素的位置，且按升序排列，从而类似倒排链表数据结构。优点是存储空间占用更少，缺点是算法设计更复杂，并行化难度高。

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 串行倒排链表算法

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: 串行倒排链表消元部分

```
void gauss(list_matrix_t& m) {
  for (auto& eliminatee : m.op) {
    while (!eliminatee.empty()) {
      auto  key        = *(eliminatee.cbegin());
      auto& eliminater = m.pool[key];
      if (eliminater.empty()) {
        eliminater = eliminatee;
        break;
      } else {
        auto jt = eliminatee.begin();
        auto it = eliminater.cbegin();
        while (it != eliminater.cend() && jt != eliminatee.end())
          if (*it > *jt) eliminatee.insert(jt, *it++);
          else if (*it == *jt) jt = eliminatee.erase(jt), ++it;
          else ++jt;
        for (; it != eliminater.cend(); ++it) eliminatee.push_back(*it);
      }
    }
  }
}
```

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。

因为 STL bitset 充分利用了 RAM 模型下字的操作，如果约定计算机字长为 w 的话，将原本用 bool 数组的 w 次操作统一成 1 次，大大减小了时间常数，相当于已执行了 SIMD 优化。而且在 O2 以上

优化后 SIMD 的幅度更加大。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 $\forall x \in [0, matrix\_max\_sz)\ bsmap(bsmap(x)) = x$。

Listing 2: bsmap 宏

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

Listing 3: 串行位元矩阵消元部分

```
void gauss(bitset_matrix_t& m) {
  for (auto& eliminatee : m.op) {
    while (eliminatee.any()) {
      auto  key        = bsmap(eliminatee._Find_first());
      auto& eliminater = m.pool[key];
      if (eliminater.none()) {
        eliminater = eliminatee;
        break;
      } else eliminatee ^= eliminater;
    }
  }
}
```

2.5 并行倒排链表算法

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_conc 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

```
const auto num_thread = thread::hardware_concurrency();
boost::shared_mutex mutex_main;
void thread_callback(size_t index, bitset_matrix_t& m) {
  for (size_t local_index = index; local_index < m.op.size();
       local_index += num_thread) {
    auto& eliminatee = m.op.at(local_index);
    while (eliminatee.any()) {
      auto key = bsmap(eliminatee._Find_first());
      boost::upgrade_lock<boost::shared_mutex> read_lock(mutex_main);
      auto&                                    eliminater = m.pool[key];
      if (eliminater.none()) {
        boost::upgrade_to_unique_lock<boost::shared_mutex> write_lock(
          read_lock);
        eliminater = eliminatee;
        break;
      } else eliminatee ^= eliminater;
    }
  }
}
void gauss(bitset_matrix_t& m) {
  list<thread> thread_pool;
  for (unsigned i = 0; i < num_thread; ++i)
    thread_pool.push_back(thread(thread_callback, i, ref(m)));
  for (auto& t : thread_pool) t.join();
}
```

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

```
const auto num_thread = thread::hardware_concurrency();

void thread_callback(size_t index, bitset_matrix_t& m) {
  for (size_t local_index = index; local_index < m.op_sz;
       local_index += num_thread) {
    auto& eliminatee = m.op[local_index];
    for (auto idx = eliminatee._Find_first(); idx < eliminatee.size();
         idx      = eliminatee._Find_first()) {
      auto key   = bsmap(idx);
      bool exist = true;
      call_once(m.flag_v[key], [&]() {
        exist       = false;
```

```
        m.pool[key] = eliminatee;
      });
      if (exist) eliminatee ^= m.pool[key];
      else break;
    }
  }
}

void gauss(bitset_matrix_t& m) {
  list<thread> thread_pool;
  for (unsigned i = 0; i < num_thread; ++i)
    thread_pool.push_back(thread(thread_callback, i, ref(m)));
  for (auto& t : thread_pool) t.join();
}
```

2.7 CUDA 加速按行划分算法

由于 CUDA 是单纯的内存复制，使用复杂的数据结构需要自己实现对内存的操作，并不支持 bitset，并且也不支持 call_once 原语，因此实现过程十分复杂，这里只给出部分上层参考示例代码。

Listing 6: call_once 位元矩阵消元部分

```
__global__ void eliminate_kernel(bitset_matrix_t& m) {
  auto local_index = blockDim.x * blockIdx.x + threadIdx.x;
  if (local_index >= m.op_sz) return;
  auto& eliminatee = m.op[local_index];
  for (auto idx = eliminatee._Find_first(); idx < eliminatee.size();
       idx      = eliminatee._Find_first()) {
    auto key   = bsmap(idx);
    bool exist = true;
    call_once(m.flag_v[key], [&]() {
      exist       = false;
      m.pool[key] = eliminatee;
    });
    if (exist) eliminatee ^= m.pool[key];
    else break;
  }
}

void gauss(bitset_matrix_t& m) {
  dim3 grid(max(ceil(m.op_sz / 1024.0), 1.0));
  dim3 block(1024);
  eliminate_kernel<<<grid, block>>>(m);
  cudaDeviceSynchronize();
  if (auto ret = cudaGetLastError(); ret != cudaSuccess)
    cerr << "division kernel failed: " << cudaGetErrorString(ret) << endl;
}
```

2.8 CUDA 加速异或算法

bitset 的异或操作非常迅速：在 RAM 模型中，约定计算机字长为 w 的话，一次异或的复杂度为 $O(\frac{N}{w})$，而单纯拷贝到 GPU 就需要 $O(\frac{N}{)}$ 的时间，而且拷贝次数频繁，单次数据量不够大，与 GPU 大带宽高延迟的特性冲突。

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 7 即可判断正确性。经过验证，所有实现均保证了正确性。

Listing 7: 单行 Bash 脚本

```
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

对于使用 bitset 作为数据结构的串行算法来说，消去的时间复杂度为 $O(\frac{N^3}{w})$，其中 w 为计算机字长；对于多线程或多进程加速算法来说，消去的时间复杂度为 $O(\frac{N^3}{w \cdot P})$，其中 w 为计算机字长，P 为线程数。

3.4 运行情况分析

计时方式

使用了程序内计时器的方法，借助 C++11 的 chrono::high_resolution_clock 可以方便的实现极高精度的计时。为了避免测试平台 IO 速度波动的影响，不讨论不同 IO 方式的速度这种通用优化问题，并且不同实现中存放矩阵的数据结构相似，初始化时间复杂度相似；考虑到特殊的高斯消去只是布尔 Gröbner 基计算的中间步骤，保证仅在内存中完成。我们决定只记录高斯消去部分的用时，对于并行算法，我们将保证计时开始前所有线程没有工作，所有线程工作完成再结束计时。

Listing 8: 计时器参考代码

```
for (size_t i = 1; i <= cases; ++i) {
  bitset_matrix_t m;
  m.read(argv[1], i);
  auto t1 = chrono::high_resolution_clock::now();
  gauss(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

表 2: 不同数据结构与并行状态下最快运行情况

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

```
    auto t2 = chrono::high_resolution_clock::now();
    auto chrono_time =
      chrono::duration_cast<chrono::duration<double, std::milli>>(t2 - t1);
    cout.precision(9);
    cout << "N: " << matrix_sz[i] << " time: " << chrono_time.count() << " ms"
         << endl;
}
```

由表 2 可以发现对于矩阵的不同存储方法，位元存储相比于倒排链表存储不仅实现更简单，而且速度更加优秀，配合多线程加速可以实现百倍乃至千倍的加速比。

分析表 1 可以发现，不同加锁方式对性能的影响极其巨大。在程序运行时观察 CPU 占用也可以一窥端倪：使用读写锁方式的多线程算法 CPU 占用始终维持在 20% 左右的低位，而使用 once_call 方式的多线程算法 CPU 占用始终维持在 100% 左右的满载状态，可以说能够完全发挥 CPU 的全部实力。而且根据 CPU 核心数动态调整并发线程数也保证了程序的多线程伸缩性优异，能简单的通过扩展 CPU 硬件提高加速效果。