pthread 多线程编程
——以高斯消去为例

杜忱莹 周辰霏

April 2021

目录

1 实验介绍

1.1 实验选题

1. 默认选题：高斯消去法（LU 分解）。
2. 鼓励自选与期末研究报告结合的题目，选题建议和要求同前次作业。

1.2 实验要求

1. 对选定题目设计实现多线程算法。
   - 设计实现适合的任务分配算法，分析其性能。
   - 与 SIMD(SSE/AVX、Neon) 算法结合。
2. 提交研究报告（问题描述、pthread+SSE/AVX 算法设计与实现、实验及结果分析）和源码（只将程序文件和工程文件提交，不要将编译出的目标文件和可执行文件也打包提交）。

1.3 思考

考虑更多算法设计、性能分析/测试的问题，以高斯消去为例：

1. 任务划分方法的思考：矩阵的水平划分、垂直划分等不同方法，相应的复杂性分析（并行时间、同步开销、加速比、效率等）。
2. 一致性保证，如何使用同步保证正确性？通过算法重构是否可以减少同步的使用？不同同步机制的性能是否有差异?
3. 实验设计、结果分析、profiling，与算法复杂性分析的对照，矩阵规模与 cache 关系对性能的影响、超线程对性能的影响、arm 平台上的实验、与 x86 平台的对比等等。

2 实验设计指导

2.1 实验总体思路

以高斯消去法为例：

1. 首先初始化生成矩阵元素值，按照上次作业给出的伪代码实现高斯消去法串行算法。

2. 使用课堂所学 pthread 编程任务划分和同步机制，针对消去部分的两重循环，按照不同任务划分方式（如按行划分和按列划分），分别设计并实现 pthread 多线程算法。并考虑将其与 SIMD 算法结合。对各算法进行复杂度分析（基本的运行时间、加速比的分析以及更深入的伸缩性分析），思考能否继续改进。

3. 实验方面，改变矩阵的大小、线程数等参数，观测各算法运行时间的变化，对结果进行性能分析。借助 Vtune profiling 等工具分析算法过程中的同步开销和空闲等待等。

2.2 Pthread 编程范式

Pthread 编程相比于 OpenMP 编程相对更底层一些，有助于我们更好地对线程进行管理和协调，我们也可以更加灵活地分配并行任务以及管理任务调度。

Pthread 程序需要包含头文件：<pthread.h>

GCC 编译选项：-lpthread（新版本改为-pthread）

主要基础 API：

```
int pthread_create(pthread_t *, const pthread_attr_t *,void * (*)(void *),void *);
//用于创建线程
//第一个参数为线程 id 或句柄（用于停止线程等）
//第二个参数代表各种属性，一般为空（代表标准默认属性）
//第三个参数为创建出的线程执行工作所要调用的函数
//第四个参数为第三个参数调用函数所需传递的参数
int pthread_join(pthread_t *, void **value_ptr)
//除非目标线程已经结束，否则挂起调用线程直至目标线程结束
//第一个参数为目标线程的线程 id 或句柄
//第二个参数允许目标线程退出时返回信息给调用线程，一般为空
void pthread_exit(void *value_ptr);
//通过 value_ptr 返回结果给调用者
int pthread_cancel(pthread_t thread);
//取消线程 thread 执行
```

Pthread 编程有两种范式：

- **静态线程**：程序初始化时创建好线程（池）。对于需要并行计算的部分，将任务分配给线程执行。但执行完毕后并不结束线程，等待下一个并

行部分继续为线程分配任务。直至整个程序结束，才结束线程。优点是没有频繁的线程创建、销毁开销，性能更优；缺点是线程一直保持，占用系统资源，可能造成资源浪费。

- **动态线程**：在到达并行部分时，主线程才创建线程来进行并行计算，在这部分完成后，即销毁线程。在到达下一个并行部分时，再次重复创建线程——并行计算——销毁线程的步骤。优点是再没有并行计算需求时不会占用系统资源；缺点是有较大的线程创建和销毁开销。

程序的具体结构，通常是在主函数（主线程）中使用 pthread_create 函数创建工作线程，将并行部分抽取出来形成线程函数（将计算任务——通常体现为循环结构——划分给多个线程），工作线程执行线程函数进行并行计算，主函数调用 pthread_join 函数等待工作线程完成。对于静态线程范式，在串行版本的基础上，通常是在主线程开始即创建所有工作线程，在主线程结束时等待线程结束，将串行程序的主体都纳入线程函数，在需要并行的地方进行任务划分。对于动态线程范式，在串行版本的基础上，在并行部分之前创建线程，将并行部分形成线程函数，紧接着等待线程结束。

注意：这里的并行部分考虑的是*动态的程序执行*，而非*静态的程序结构*。例如，程序主体是一个双重循环，外层循环内、内层循环之外基本没有实质性计算，主要计算操作都在内层循环内，我们选择将内层循环拆分分配给工作线程。此时如果机械地从程序结构角度出发，只将内层循环放入线程函数中，外层循环还保留主函数中，则有两种情况：要么采用动态线程范式，外层循环的每步执行都创建、销毁所有工作线程，带来严重的额外开销；采用静态线程范式的话，主线程和工作线程之间的通信（同步）就会很复杂，程序易读性、可维护性大大下降。而好的方式是，从程序动态执行的角度看，其实外层循环执行的全过程都是处于并行部分中（因为内外层循环之间并无计算，外层循环的执行实际上只是在持续执行内层循环而已），因此将双重循环都置于线程函数中，对外层循环保持不动，将内层循环拆分分配工作线程即可。本次作业即可采用这种思路。如内外层循环间还有其他代码，视具体情况处理一下，都可以采用上述思路实现多线程版本。

同步问题为 pthread 编程的重点与难点，方式也有很多：比如忙等待、互斥量和信号量、barrier 与条件变量等等。PPT 中已给出相关 API 及例子，请同学们复习参考 PPT，这里不再一一介绍讲解。

2.3 作业注意要点及建议

1. 矩阵数值初始化问题

根据有些同学们反映，自己初始化矩阵在计算过程中会出现 inf 或 nan 的问题。这是由于精度问题以及非满秩矩阵造成的。inf 或 nan 的情况无疑会影响结果正确性的判断，也会在并行计算性能上造成一定影响，而随机生成数据的方式很明显无法保证能避免该问题尤其在规模巨大的情况下。个人建议初始化矩阵时可以首先初始化一个上三角矩阵，然后随机的抽取若干行去将它们相加减然后执行若干次，由于这些都是内部的线性组合，这样的初始数据可以保证进行高斯消去时矩阵不会有 inf 和 nan。

2. 生成线程所要执行函数的参数问题

由于只有一个 void 指针的参数，可以创建一个数据结构，将所需要的各种参数如线程 id，问题规模大小等打包起来通过指针转换来传递。注意对于均分任务时，当问题规模不等于线程数倍数时，对于分配最后一部分计算任务的线程不要直接使用 my_first+my_n 计算 my_last，可根据线程 id 将 n 作为该线程的 my_last。对于推荐的范式来说，可直接在线程函数内完成这些工作，就无需传递参数了。

3. 尽量避免频繁的创建和销毁线程

以高斯消去问题为例，回顾一下其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
```

一个简单直观的多线程思路就是：共进行 n 轮消去步骤（外层循环），第 k 轮执行完第 k 行除法后（第一个内层循环），对于后续的 $k+1$ 至 n 行进行减去第 k 行的操作（第二个内层循环），各行之间互不影响，可采用多

线程执行。每轮除法完成后创建线程，该轮消去完成后销毁线程。这样的话创建和销毁线程的次数过多，而线程创建、销毁的代价是比较大的，

可以通过信号量同步、barrier 同步等同步方式避免频繁的创建和销毁线程。以信号量同步思路为例，主线程执行除法，工作线程执行消去。主线程开始时建立多（4）个工作线程；在每一轮消去过程中，工作线程先进入睡眠，主线程完成除法后将它们唤醒，自己进入睡眠；工作线程进行消去操作，完成后唤醒主线程，自己再进入睡眠；主线程被唤醒后进入下一轮消去过程，直至任务全部结束销毁工作线程。barrier 同步（有助于后面 openmp 编程的学习与理解）和其他同步方式请同学们自行思考，这里不再讲解。

但实际上，如上一节所述，第二种方法也存在程序逻辑复杂的问题。更好的方式是采用上一节所述范式，将多重循环都纳入线程函数中，消去操作对应的内层循环拆分，分配给工作线程。对于除法操作，可以只由一个线程执行，也可拆分由所有工作线程并行执行。需要注意的是，除法和消去两个步骤后都要进行同步，以保证进入下一步骤（下一轮）之前上一步骤的计算全部完成，保证一致性。

4. 不同任务划分策略

可以看到，高斯消去过程中的计算集中在 4-5 的第一个内层循环（除法）和 8-13 行的第二个内层循环（双重循环，消去），对应矩阵右下角 $(n-k+1)\times(n-k)$ 的子矩阵。因此，任务划分可以看作对此子矩阵的划分。对于除法部分，因为只涉及一行，只可能采用垂直划分（列划分）。而对于消去部分，即可采用水平划分（将其外层循环拆分，即每个线程分配若干行），也可采用垂直划分（将其内层循环拆分，即每个线程分配若干列）。两种划分策略在负载均衡上可能会有细微差异，而在同步方面会有差异，cache 利用方面也会有不同。

此外，在与 SIMD 结合时，SIMD 只能将行内连续元素的运算打包进行向量化，即只能对最内层循环进行展开、向量化。多线程不同任务划分策略要注意与 SIMD 的结合方式。

5. 计算误差与程序正确性

有关问题规模和并行计算由于重排了指令执行顺序和计算机浮点数所导致误差问题说明参考之前实验。

多线程编程一次结果正确并不代表算法的正确性，相比错误的结果，算法错误结果正确无法暴露问题才是最可怕的，需要多进行实验降低错误概率，不仅是同阶矩阵数据的多次实验，还需要设计多组数据进行对比测试，

更好地验证并行优化算法对不同数据规模的加速效果，同学们也在代码逻辑上可多进行梳理。多线程编程不易调试 bug，错误也不好复现。需要大家更多的去思考各线程执行过程中的可能情况。

3 例题编程分析

下面将举一个 PPT 上估算 π 的小例子来讲解 pthread 多线程编程：

3.1 问题描述

估算 π 值可以借助反三角函数 arctan 的泰勒展开式，其数学公式如下：

$$\pi = 4[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + ...] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} \tag{1}$$

3.2 问题思路

公式 1本质上是一个多个正负数值求和问题，其中项数越多结果越能够逼近 π 值，我们可以将各项求和的任务划分成多个部分，每个部分求取对应的部分和，各部分之间互不影响，每个部分使用一个线程进行计算获得部分和，然后将部分和求和获得最终求和结果再乘 4 获得 π 的估算值。其中需要注意的是，在对部分和求和时，也就是将部分和加到保存最终求和结果变量 sum 时需要保证该临界区一个时刻只能一个线程进行处理，可以采取 PPT 上的忙等待和互斥量同步方法实现。

3.3 问题实现

问题实现的关键代码和不同规模的实验结果见 PPT。由于 PPT 中只给出了忙等待版本和互斥量版本中生成线程所要执行的函数代码。这里以互斥量版本为例给出包含线程创建部分的代码供大家参考。

```
1 void *pi_mutex(void *parm)
2 {
3   threadParm_t *p = (threadParm_t *) parm;
4   int r = p->threadId; int n = p->n; int my_n = n/THREAD_NUM;
5   int my_first = my_n*r; int my_last = my_first + my_n;
6   double my_sum = 0.0;
7   double factor = -1.0;
8   if (my_first % 2 == 0)
```

```
9       factor = 1.0;
10    for (int i = my_first; i < my_last; i++, factor = -factor) {
11      my_sum += factor/(2*i+1);
12    }
13    pthread_mutex_lock(&mutex);
14    sum += my_sum;
15    pthread_mutex_unlock(&mutex);
16  }
17
18  void Mutex(int n)
19  {
20    pthread_t thread[THREAD_NUM];
21    threadParm_t threadParm[THREAD_NUM];
22    pthread_mutex_init(&mutex, NULL);
23    sum=0.0;
24    for (m=0; m<THREAD_NUM; m++)
25    {
26      threadParm[m].threadId = m;
27      threadParm[m].n=n;
28      pthread_create(&thread[m], NULL, pi_mutex, (void *)&threadParm[m]);
29    }
30    for (m=0; m<THREAD_NUM; m++)
31    {
32      pthread_join(thread[m], NULL);
33    }
34    pthread_mutex_destroy(&mutex);
35    pi_approx=4.0*sum;
36  }
```

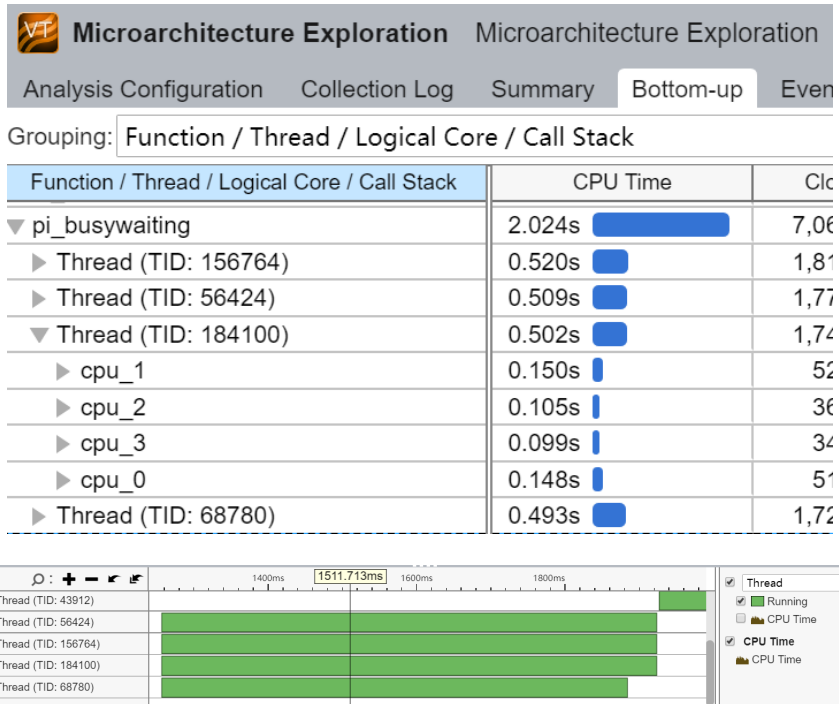
3.4 Vtune 分析

接下来简单举例说明如何使用 Vtune profiling 工具分析算法过程中的同步开销和空闲等待等问题。对于 Microarchitecture Exploration，可以选择含有 thread 的 Grouping，比如 Function/thread/Logical Core/Call Stack。我们就可以看到每个线程的 id 及其运行信息，以忙等待方法为例如下图所示：

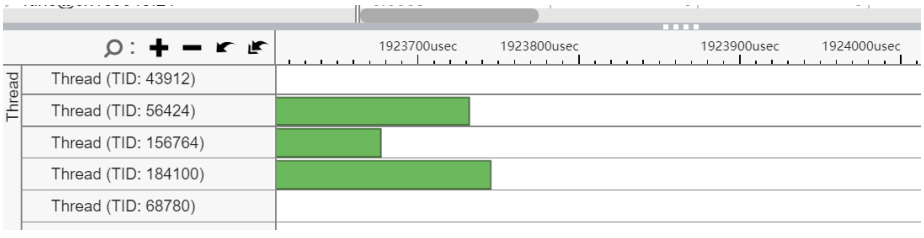
然后可以根据线程 id 找到各线程的运行时间以及对应事件状态：

很明显可以看出 68780 线程最先执行结束，为代码中的 0 号线程，其余线程结束时间十分接近，可以进行放大：

可以看出 156764 线程第二个结束为 1 号线程；56424 线程第三个线程为 2 号线程，184100 线程最后一个结束为 3 号线程。了解线程顺序后可以



查看到其他信息进行分析，比如 Sleep_Ex 函数，该函数用于中止当前线程等待恢复运行：可以看出 2 号线程和 3 号线程执行过该函数，结合 0 号线程相比其他三个线程很早就完成的情况，我们知道 1 号线程运行得很慢，由此造成了提早完成任务的 2 号线程和 3 号线程的空闲等待。这也是忙等待版本 take turn 相比于互斥量版本的缺陷。在线程数多的时候或是线程执行能力差异巨大的情况下尤为明显。对于 CPI，指令数，Cache 命中等其他功能的分析参考体系结构相关及性能测试实验指导书，不再赘述。



Grouping: Function / Thread / Logical Core / Call Stack

Function / Thread / Logical Core / Call Stack	CPU Time	Clockticks	Instructions Retired	CP
▶ RtlGetCurrentUmsThread	0.001s	5,400,000	540,000	
▶ func@0x1401c93e0	0.003s	7,020,000	1,080,000	
▼ SleepEx	0.004s	12,420,000	3,780,000	
▶ Thread (TID: 184100)	0.001s	3,780,000	3,780,000	
▶ Thread (TID: 56424)	0.003s	8,640,000	0	