OpenMP 多线程编程
——以高斯消去为例

曾泉胜

2022 年 5 月

目录

1 实验介绍

有一句调侃的话叫做，“并行程序设计就是索引分配的艺术”。同学们在 pthread 实验之后，应该或多或少感受到了最简单的并行化方式就是把一个好的串行化算法分配给不同线程，按行、按列或者按块划分等等。重复性的工作必然会被自动化的工具代替，于是我们来学习一种很方便的工具：OpenMP。

1.1 实验选题

同 SIMD 编程作业。

1.2 实验要求

1. 基本要求（最高获得 80% 分数）：ARM 平台上普通高斯消去计算的基础 OpenMP 并行化实验：
   - 设计实现适合的任务分配算法，分析其性能；
   - 与 SIMD（Neon、SSE/AVX/AVX-512）算法相结合；
   - 在 ARM 平台上编程实现、进行实验，测试不同问题规模、不同线程数下的算法性能（串行和并行对比），对比 Pthread 程序的性能对。
2. *** 进阶要求（最高可获得剩余 20% 分数）**：在基本要求基础上，进一步探讨特殊的高斯消去计算的 OpenMP（结合 SIMD）并行化、两种高斯消去不同平台（如 x86）上并行化实验、多线程并行化的不同算法策略（如矩阵水平划分、垂直划分等不同任务划分方法，不同算法策略下的一致性保证、线程管理代价优化等）及其复杂性分析、卸载到加速器设备、profiling 及体系结构相关优化（如 cache 优化）等。
3. 自主选题视难度和工作量与默认题目对等评分。
4. 撰写并提交研究报告，要求同 SIMD 编程作业。

2 实验设计指导

2.1 OpenMP 简介

2.1.1 学习链接

OpenMP（Open Multi-Processing）是一套支持跨平台共享内存方式的多线程并发的编程 API，使用 C,C++ 和 Fortran 语言，可以在大多数的处理器体系和操作系统中运行，包括 Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, 和 Microsoft Windows。包括一套编译器指令、库和一些能够影响运行行为的环境变量。

首先需要一个支持 OpenMP 的编译器，自 GCC 4.9 起，C、C++ 完全支持 OpenMP 4.0；自 GCC 6 起，C 和 C++ 完全支持 OpenMP 4.5。自 GCC 9 起，C 和 C++ 部分支持 OpenMP 5.0，并在 GCC 10 中得到扩展。更多编译器支持信息可见https://www.openmp.org/resources/openmp-compilers-tools/。

可以从官网获得文档 (https://www.openmp.org/resources/refguides/) 以及各种教程 (https://www.openmp.org/resources/tutorials-articles/)。课堂 PPT 中也给了大量的例子。

2.1.2 基本语法

OpenMP 的基本语法是将如下格式的预编译指令用在一个语句块之前：
#pragma omp <directive> [clause[[,] clause] ...]
其中，directive 可以是以下内容：

1. parallel: 创建线程，接下来的代码块是一个并行区域
2. single: 之后的代码块只会由一个线程（未必是主线程）执行
3. master: 之后的代码块只会由主线程执行
4. for: 之后的 for 循环将被并行化由多个线程划分执行，循环变量必须是整型
5. barrier: 所有线程在此同步
6. …

clause 可以是以下内容：

1. if(*scalar-expression*) 当跟在 parallel 指令之后使用时表示是否并行化
2. private(*list*) 指定并行区域中的变量为每个线程独立的存储
3. shared(*list*) 指定并行区域中的变量为各个线程共享
4. default(shared|none) 未指定变量缺省的存储方式，shared 表示共享，none 表示必须为所有变量指定存储方式
5. nowait 在离开下方的语句块时不进行线程同步
6. schedule(static|dynamic|guided[, *chunk_size*]) 用在 for 指令之后，指定循环任务的划分方法。
7. ...

此外，OpenMP 还提供了一些库函数：

```
// 设置之后一个并行区域的线程数，只能在串行代码区域使用
void omp_set_num_threads(int _Num_threads);

// 获得当前线程数目
int omp_get_num_threads(void);

// 返回当前线程id.id从1开始顺序编号,主线程id是0
int omp_get_thread_num(void);

// 获得当前墙上时钟时间，每个线程都有自己的时间
double omp_get_wtime(void);
```

2.1.3 编程范式

图2.1展示了使用 OpenMP 程序的运行模式，下面展示的是一种编程范式，以普通高斯消去为例，编写如下代码，并在使用 gcc 编译时使用-fopenmp 选项：

```
// 需要包含头文件omp.h
#include <omp.h>
// ...
// 在外循环之外创建线程，避免线程反复创建销毁，注意共享变量和私有变量的设置
#pragma omp parallel if(parallel), num_threads(NUM_THREADS), private(i, j, k, tmp)
for(k = 1; k < n; ++k){
```

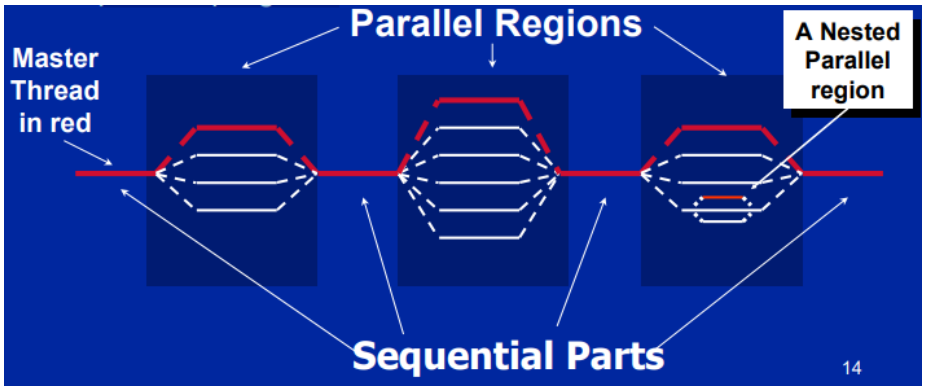

图 2.1: OpenMP 的 fork-join 并行机制，参考https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf

```
    // 串行部分，也可以尝试并行化
    #pragma omp single
    {
        tmp = mat[k][k];
        for(j = k + 1; j < n; ++j){
            mat[k][j] = mat[k][j] / tmp;
        }
        mat[k][k] = 1.0;
    }
    // 并行部分，使用行划分
    #pragma omp for
    for(i = k + 1; i < n; ++i){
        tmp = mat[i][k];
        for(j = k + 1; j < n; ++j){
            mat[i][j] = mat[i][j] - tmp * mat[k][j];
        }
        mat[i][k] = 0.0;
    }
    // 离开for循环时，各个线程默认同步，进入下一行的处理
}
```

2.1.4 更多细节

OpenMP 为我们提供了方便的 API，在此基础上，可以根据研究的问题进行更多的探索，在此简单罗列几个可以尝试的方向。

负载均衡问题 在为线程划分任务的时候，需要避免负载不均，以高斯消去为例，对于倒数第 k 行，消去过程的计算量为 $O(k^2)$，使用纯块划分会使得处理头几行的线程负载要高于处理最后几行的线程，在 OpenMP 的 schedule 子句默认的划分方式 (static) 就是这样，可以通过设置 chunk_size，使用循环划分或者块循环划分来均衡负载；当研究的问题负载不均并且不像高斯消去这样可以预先知道负载的分布时，可以使用动态的划分方式 (dynamic) 并设置一定的 chunk_size，该方式下某个线程结束了先前的任务之后，又会从任务队列中取出新的任务，而并不依赖于线程号与行号的某种映射关系；guided 方式把循环体的执行分组，分配给等待执行的线程。最初的组中的循环体执行数目较大，然后逐渐按指数方式下降到 chunk_size。

虚假共享问题 如果想要分析多线程程序的 cache，就必须要认识到在现代多核处理器中，可能是每个核都有独自的一级和二级 cache，然后有所有核共享的三级 cache（具体情况应当使用相关指令查阅自己的机器），这样一来，cache 不命中的原因除了冷启动、容量不足或者发生冲突之外，还会因为核 B 试图写入核 A 的 cache 中存在的某个高速缓存行，使得核 A 中的该缓存行失效，下一次核 A 访问该缓存行不命中。该现象被称为虚假共享。我们可以使用在共享访问的变量间适当添加填充，对齐到高速缓存行的大小（例如 64B），或者在多线程访问连续内存时，适当提高 chunk_size，使得一个高速缓存行放在一个线程里解决。

使用 OpenMP 提供的规约操作 在高斯消去的例子中，运算基本都是矩阵到矩阵的方式，因此不会用到规约操作。但是例如求和，求最大值这样从矢量到标量的计算，先前提供的范式不能得到正确的结果，而使用加锁等并发编程方式将大大增加计算开销，而 OpenMP 中为我们提供了规约操作，甚至还可以自定义规约的运算，需要的同学可以自行研究。

使用 SIMD 与 Pthread 实验一样，也可以在内层循环使用循环展开，SIMD 指令进一步加速；OpenMP 4.0 之后也提供了自动向量化的方式可以尝试。

嵌套的线程 我们知道在 fork() 中再次调用 fork()，可以得到 4 个一模一样的克隆，但是在缺省状态下，嵌套并行是禁用的，因此上文中在并行区域

中再写 #prama omp parallel 并不会使得线程数加倍，大多数情况下也并不推荐使用，容易造成资源的浪费。

2.2 实验总体思路

本次实验可以将上次 Pthread 的实验改写成 OpenMP 的方式，同样的可以在任务划分方式、与 SIMD 的结合，算法的复杂度分析（运行时间、加速比、伸缩性分析），进行 profiling 等等，也可以将 OpenMP 的实现与 Pthread 的实现进行对比，在保证结果正确的前提下尝试改进性能。

2.3 编译、运行

参见 SIMD 编程实验教学指导书。

3 Vtune 分析

此部分可见 Pthread 实验指导书。