南开大学
Nankai University

南　开　大　学

计 算 机 学 院

并行程序设计实验报告

高斯消去法 OpenMP 并行化实验报告

周辰霏 1712991

年级：2017 级

专业：计算机科学与技术

2020 年 5 月 6 日

摘要

针对高斯消元法 (LU 分解) 的 OpenMP 结合 SSE 和 AVX 的并行化实验。并对每一种算法进行基于 Vtune 的微观 Profiling 验证实验结果。

关键字：LU 分解；OpenMP 编程；SSE/AVX；Vtune

目录

一、 概述

(一) 问题描述

高斯消元法 (Gaussian Elimination) 是线性代数中的一个算法, 用于线性方程组求解、求矩阵的秩、求可逆方阵的逆矩阵。[1]

高斯消元法实际上就是最朴素的线性方程组消元, 其时间复杂度是 $O(n^3)$, 是一个在矩阵规模 N 较大时非常费时的算法, 且在高斯消元法的串行平凡算法中, 并非每一步都是必须的, 或者某几步是可以进行向量化的循环的, 在使用 OpenMP 的情形下, 两个循环都可以进行相应的优化。

OpenMP 并行化和前面 pthread 编程的区别主要在编译方式上,OpenMP 需要添加编译器预处理指令 #pragma, 而创建线程等后续工作由编译器完成; pthread 是通过使用库函数完成一系列操作, 整体上编程复杂度高于 OpenMP。

(二) 实验内容

针对高斯消去法 (LU 分解) 进行以下 OpenMP 并行化实现:

- 串行朴素 LU
- OpenMP 按行/列划分 LU 消去并行处理
- OpenMP 按行划分 LU 消去并行处理与 SSE/AVX 结合
- OpenMP dynamic 以及 guided 调度按行划分 LU 消去并行处理

并对各种并行化方法进行性能和参数上的比较包括但不限于数据测试规模、执行指令数、周期数、CPI、线程运行情况等的测算比对。

(三) 实验环境

- 系统环境: Windows10 家庭中文版 (64 位)
- 编译环境: gcc 8.1.0 x86_64-posit-seh, Codeblocks17.12, SSE3 指令, pthread
- 实验测试工具: Intel Vtune profiler 2020

二、 算法设计与实现

(一) 串行朴素 LU 实现

根据 Wiki 以及实验指导书, 串行朴素 LU 实际上就是将矩阵化为一个上三角矩阵, 分别对其实现按行和按列划分:

1. 第 k 步的时候从第 k+1 个元素开始除以第 k 个元素
2. 第 k 步的时候从第 k+1 个元素开始减去第 k 个元素与上一行第 k+1 个元素的乘积 (按行)
3. 第 k 步的时候从第 k+1 个元素开始减去第 k 个元素与上一列第 k+1 个元素的乘积 (按列)

串行朴素

```
void naive_lu(float mat[][N])//按行
{
    for (int k = 0; k < N; k++)
    {
        for (int j = k + 1; j < N; j++)//除法
            mat[k][j] /=mat[k][k];
        mat[k][k] = 1.0;
        for (int i = k + 1; i < N; i++)//减去第k行
        {
            for (int j = k + 1; j < N; j++)
                mat[i][j] -= mat[i][k] * mat[k][j];
            mat[i][k] = 0;
        }
    }
}
void lu_col(float mat[][N])//按列
{
    for(int k = 0; k<N; k++)
    {
        for(int j = k+1; j<N; j++)//除法
            mat[k][j] = mat[k][j] / mat[k][k];
        mat[k][k] = 1.0;
        for(int j = k+1; j<N; j++){//按列消去
            for(int i = k+1; i<N; i++)
                mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
        }
        for(int i = k+1; i<N; i++)
            mat[i][k] = 0;
    }
}
```

（二） OpenMP 按行/列划分除法消去并行处理

由于高斯消去法做了较多的重复同类运算，故可以通过按行/列划分后对除法步骤、消去部分或两部分同时进行并行化多线程处理。此算法使用 critical 声明临界区保护数据，通过 for 将经过除法操作后的消去操作分配给多个线程。算法设计主要工作为：对除法部分声明临界区使其全部完成后再进行消去操作，而按行划分的时候只要当前行除法完成即可消去所以按行划分的时候除法部分和消去部分都可以并行化。使用 omp for 即可。

omp-lu

```
void lu_col_omp(float mat[][N])//按列omp
{
    #pragma omp parallel num_threads(thread_count)
    for(int k = 0; k<N; k++)
    {
        #pragma omp critical//临界区
```

```
            {for(int j = k+1; j<N; j++)
                mat[k][j] = mat[k][j] / mat[k][k];
            mat[k][k] = 1.0;}
            #pragma omp for//并行
            for(int j = k+1; j<N; j++){
                for(int i = k+1; i<N; i++)
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
            }
            for(int i=k+1; i<N; i++)
                mat[i][k] = 0.0;
        }
    }

    void omp_te_lu(float mat[][N])//按行消去omp
    {
        #pragma omp parallel num_threads(thread_count)
        for (int k = 0; k < N; k++)
        {
            #pragma omp critical//临界区
            {for (int j = k + 1; j < N; j++)
            {
                mat[k][j] = mat[k][j] / mat[k][k];
            }
            mat[k][k] = 1.0;}
            #pragma omp for//并行
            for (int i = k + 1; i < N; i++)
            {
                for (int j = k + 1; j < N; j++)
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
                mat[i][k] = 0;
            }
        }
    }

    void omp_lu(float mat[][N])//按行omp
    {
        #pragma omp parallel num_threads(thread_count)
        for (int k = 0; k < N; k++)
        {
            #pragma omp for//并行
            for (int j = k + 1; j < N; j++)
            {
                mat[k][j] = mat[k][j] / mat[k][k];
            }
            mat[k][k] = 1.0;
            #pragma omp for//并行
            for (int i = k + 1; i < N; i++)
            {
```

```
                for (int j = k + 1; j < N; j++)
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
                mat[i][k] = 0;
            }
        }
    }
```

(三) OpenMP 动态、guided 调度并行化

for 循环并行化的调度共有 static、dynamic、guided 三种, 一般的 omp for 即默认使用 static 调度——每个线程分配 $\frac{iteration}{thread}$ 次迭代,dynamic 和 guided 实际上都是动态分配给线程的迭代次数, 区别在于 guided 是从大到小的分配迭代次数, 其中 size 参数在 dynamic 调度中表示分配给每个线程的迭代次数, 而 guided 调度的 size 参数表示最小分配迭代次数。这两种调度的实现基于上述按行划分的除法消去并行 OpenMP 优化, 即将 omp for 增加子句 schedule。

omp-schedule

```
    void omp_lu_dynamic(float mat[][N])//dynamic 调度
    {
        #pragma omp parallel num_threads(thread_count)
        for (int k = 0; k < N; k++)
        {
            #pragma omp for schedule(dynamic, 24)//每个线程24次
            for (int j = k + 1; j < N; j++)
            {
                mat[k][j] = mat[k][j] / mat[k][k];
            }
            mat[k][k] = 1.0;
            #pragma omp for schedule(dynamic, 24)//每个线程24次
            for (int i = k + 1; i < N; i++)
            {
                for (int j = k + 1; j < N; j++)
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
                mat[i][k] = 0;
            }
        }
    }

    void omp_lu_guided(float mat[][N])//guided 调度
    {
        #pragma omp parallel num_threads(thread_count)
        for (int k = 0; k < N; k++)
        {
            #pragma omp for schedule(guided, 24)//每个线程最少迭代到24次
            for (int j = k + 1; j < N; j++)
            {
                mat[k][j] = mat[k][j] / mat[k][k];
            }
```

```
            mat[k][k] = 1.0;
            #pragma omp for schedule(guided, 24)// 每个线程最少迭代到24次
            for (int i = k + 1; i < N; i++)
            {
                for (int j = k + 1; j < N; j++)
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
                mat[i][k] = 0;
            }
        }
    }
```

（四） OpenMP+SSE

在按行划分 OpenMP 动静态调度优化的基础上进行 SSE 优化, 合并运算。算法设计主要工作为: 中间变量引入及其数据类型更改 float→__m128、_mm_loadu_ps 加载数据到寄存器 _mm_storeu_ps 寄存器赋给变量、对循环进行 SSE 合并计算除法 _mm_div_ps、乘法 _mm_mul_ps 和减法 _mm_sub_ps。

omp+SSE

```
    void omp_lu_sse(float mat[][N])
    {
        #pragma omp parallel num_threads(thread_count)
        for(int k = 0; k<N; k++)
        {
            #pragma omp critical//加载数据设临界区
            {
           __m128 A_k_k = _mm_set_ps1(mat[k][k]);
            for(int j = N-4; j>k; j-=4)
            {
                __m128 A_k_j = _mm_loadu_ps(mat[k]+j);
                A_k_j = _mm_div_ps(A_k_j, A_k_k);
                _mm_storeu_ps(mat[k]+j, A_k_j);
            }
            for(int j = k+1; j<k+1+(N-k-1)%4; j++)   //  不能被4整除的部分
                mat[k][j] = mat[k][j] / mat[k][k];
            mat[k][k] = 1.0;}

            #pragma omp for//并行化
            for(int i = k+1; i<N; i++)
            {
                __m128 A_i_k = _mm_set_ps1(mat[i][k]);
                for(int j = N-4; j>k; j-=4)
                    {
                        __m128 A_k_j = _mm_loadu_ps(mat[k]+j);
                        __m128 t = _mm_mul_ps(A_k_j, A_i_k);
                        __m128 A_i_j = _mm_loadu_ps(mat[i]+j);
                        A_i_j = _mm_sub_ps(A_i_j, t);
                        _mm_storeu_ps(mat[i]+j, A_i_j);
                    }
```

```
                for(int j = k+1; j<k+1+(N-k-1)%4; j++)  // 不能被4整除的部分
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
                mat[i][k] = 0.0;
            }
        }
    }

    void omp_lu_sse_dynamic(float mat[][N])
    {
        #pragma omp parallel num_threads(thread_count)
        for(int k = 0; k<N; k++)
        {
            #pragma omp critical//加载数据设临界区
            {
            __m128 A_k_k = _mm_set_ps1(mat[k][k]);
            for(int j = N-4; j>k; j-=4)
            {
                __m128 A_k_j = _mm_loadu_ps(mat[k]+j);
                A_k_j = _mm_div_ps(A_k_j, A_k_k);
                _mm_storeu_ps(mat[k]+j, A_k_j);
            }
            for(int j = k+1; j<k+1+(N-k-1)%4; j++)  // 不能被4整除的部分
                mat[k][j] = mat[k][j] / mat[k][k];
            mat[k][k] = 1.0;}

            #pragma omp for schedule(dynamic,24)//动态调度
            for(int i = k+1; i<N; i++)
            {
                __m128 A_i_k = _mm_set_ps1(mat[i][k]);
                for(int j = N-4; j>k; j-=4)
                    {
                        __m128 A_k_j = _mm_loadu_ps(mat[k]+j);
                        __m128 t = _mm_mul_ps(A_k_j, A_i_k);
                        __m128 A_i_j = _mm_loadu_ps(mat[i]+j);
                        A_i_j = _mm_sub_ps(A_i_j, t);
                        _mm_storeu_ps(mat[i]+j, A_i_j);
                    }
                for(int j = k+1; j<k+1+(N-k-1)%4; j++)  // 不能被4整除的部分
                    mat[i][j] = mat[i][j] - mat[i][k] * mat[k][j];
                mat[i][k] = 0.0;
            }
        }
    }
```

三、 高精度计时分析算法复杂性

本次实验的编译选项, 除了沿用上次基本设置外, 还需在”Other compiler options” 选项卡下添加”-fopenmp”, 以及”Linker Settings” 选项卡下”Other linker options” 一栏中填入”-lgomp -lpthread”

由于实验中可能出现种种误差, 为尽量保证实验的公平性, 作以下假设:

- Cache 对所有算法都是公平的, 均为按行获取按行划分, 仅有一例为按列划分以验证不同划分方式, 每次换算法之前通过 reset 函数令初始矩阵一致, 并验证结果正确性, 在最终结果一致 (以串行朴素 LU 的计算结果为基准) 的情况下才进行高精度计时
- 由于 N 较小的时候测出的计算时间也较小, 误差较大, 故采取多次测量取均值的方法确定较合理的性能测试结果, 同时保证几种算法重复次数一致, 减少误差
- 为保证一致性, 在需要控制数据依赖及其运算顺序保证计算准确性的情况下使用 critical 建立临界区, 在前 k 个执行结束 (单线程访问) 之后才放行。

同上一次实验, 本次的测试区间比较疏松, 但是仍考虑 cache 大小的情况进行 N 的取值设定。根据下图可以得知本机为 4 核 8 线程, 故线程池总数取取 4、8。另: 由于按列的朴素算法以及 OpenMP 优化在 N 较大时访存开销过大, 性能较差, 故整体画图数据忽略此项, 仅在按行按列小数据划分中进行对比。

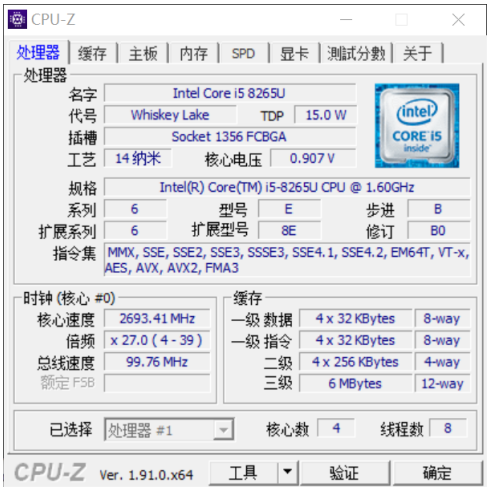

图 1: CPU 核心数

根据全部测试数据绘制运行时间、加速比对比图、效率对比图, 运行时间图可以看出各种方法对于朴素方法都节约了至少一半的运行时间, 八线程相比四线程并没有体现出优势, 反而由于建立线程等等开销问题而效果更差, 整体上辅以 SSE 的加速效果最好, 按列划分在数据较大时由于访存原因处于劣势,dynamic 调度相较其他两种调度加速效果更好。

加速比对比图则可以看出各个算法在加速上的趋势基本一致, 且与上次 pthread 相类似, 都在 N=1000 左右达到峰值, 八线程最高可达 6.8, 同时的四线程最高可达 6.03, 这之后加速比又逐渐下调到 2 附近。按照趋势来看, 八线程在前期表现良好, 而后期乏力不及四线程。

效率对比图, 整体上四线程和八线程趋势形态一致, 但显然四线程效率优于八线程, 可见四核八线程并不是真的可以很好的完成八线程的任务 (同步开销, 线程开销更大), 同运行时间反映的一致,SSE 效率更高, 而只优化了消去部分的由于同步开销相对较小反而效率更高。

dynamic 调度以及 guided 调度的 size 参数选取图可以看出,guided 的趋势和官方文档给出的并不一样,size 越大反而效果越好,dynamic 整体上表现为 size 越大时间越长。

N	naive	omp45	omp	dynamic	guided	omp-sse	sse-dynamic
16	0.000895	0.49529	0.77626	0.83125	0.77286	0.42623	0.41006
64	0.06331	1.64866	3.12332	3.10118	3.06728	1.51865	1.53128
100	0.5286	2.5045	4.7719	4.8045	4.8337	2.4128	2.37344
200	1.7188	4.997	10.3175	10.3517	9.9747	5.1727	5.2959
300	5.5783	6.6993	14.2927	13.8456	14.2294	6.7488	6.6974
400	14.0923	10.8272	20.2686	20.7427	29.8253	11.762	11.8384
500	26.9761	15.1919	27.1476	26.1629	26.0473	11.7499	11.3136
600	50.08	22.897	35.514	36.55	34.825	15.15	14.7292
700	74.638	31.542	48.707	46.667	48.475	19.677	18.898
800	115.4	44.5287	62.0949	60.5924	60.3512	23.9355	23.314
900	166.83	59.52	79.27	76.68	78.53	32.34	30.92
1000	230.83	76.71	96.81	96.01	96.37	39.92	38.23
2000	1955.17	764.38	944.82	908.93	968.72	694.99	686.445

表 1: 性能测试结果 (4 thread)(单位:ms)

按行按列划分对比可以看出按列划分的 OpenMP 优化反而效果优于按行划分, 但是当 N 达到 2000 的时候, 由于访存时间过长, 不论优不优化效果都比较差。

整体来看,OpenMP+SSE+dynamic+ 四线程的效果最好, 原因是四核 CPU 对八线程的处理并不能像八核那样理想,SSE 在数据打包解包上的开销做了优化,dynamic 和 guided 效果应该相差不多, 但是由于实验选取较中间的 size 故 dynamic 表现较 guided 好。

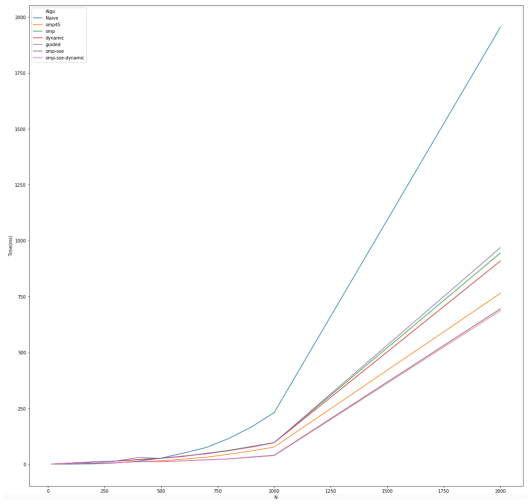

图 2: 四线程运行时间

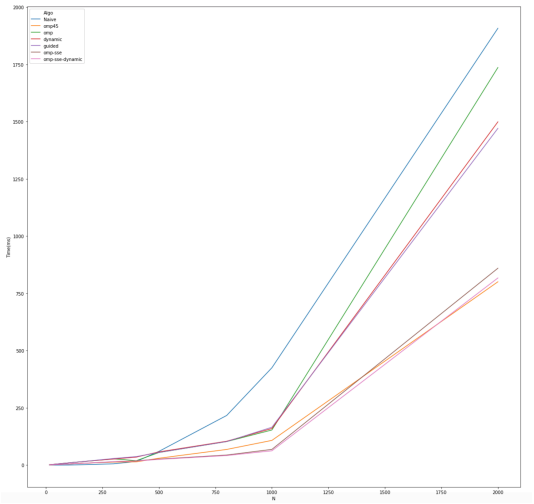

图 3: 八线程运行时间

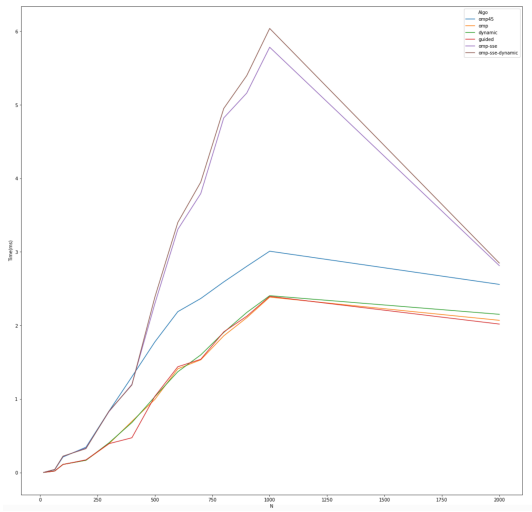

图 4: 四线程加速比

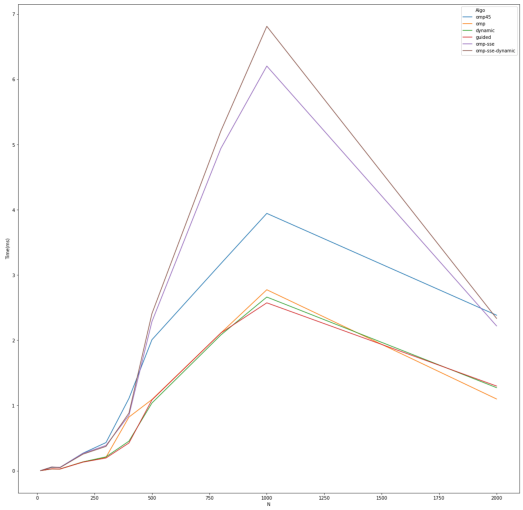

图 5: 八线程加速比

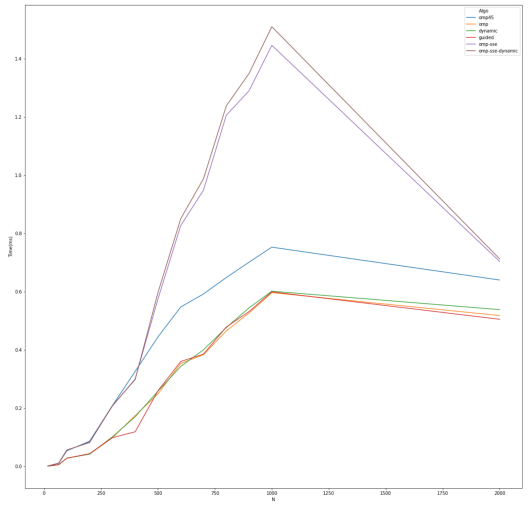

图 6: 四线程效率

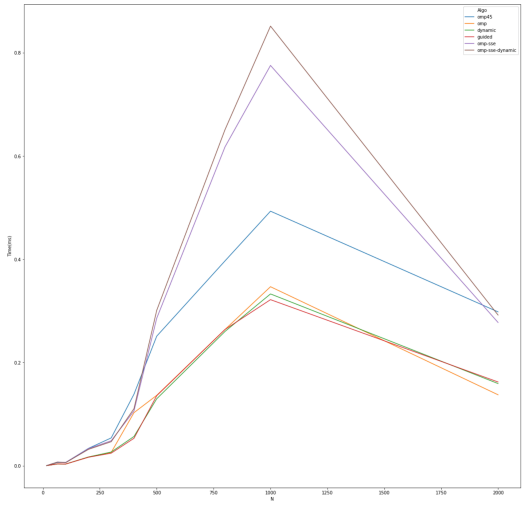

图 7: 八线程效率

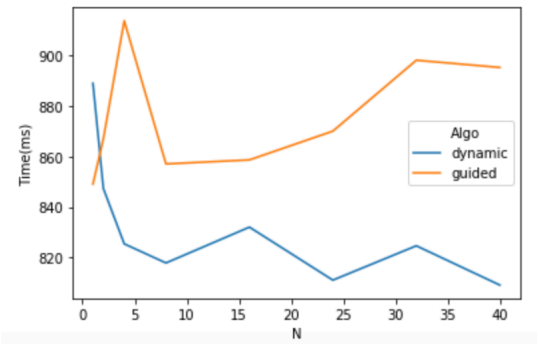

图 10: dynamic-guided 调度 size

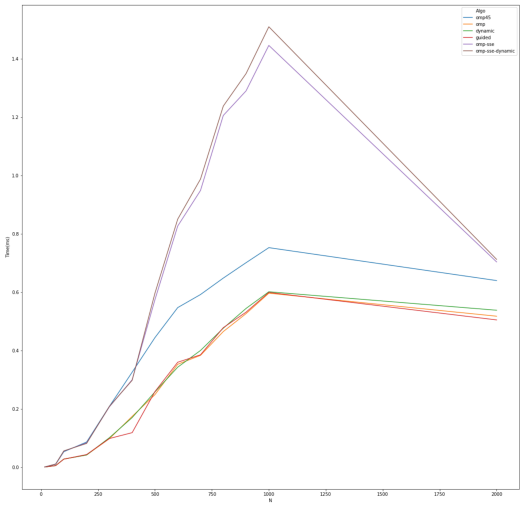

图 8: 四线程效率

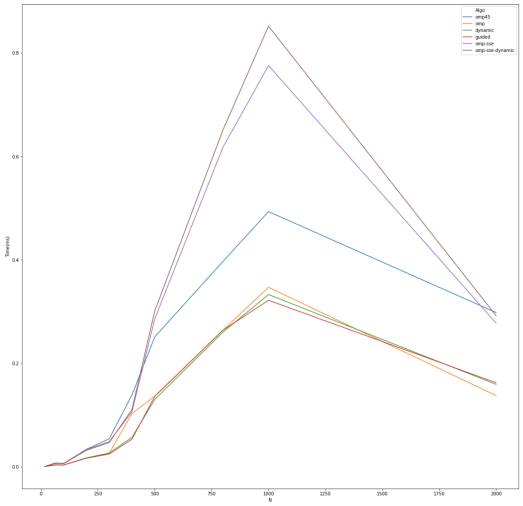

图 9: 八线程效率

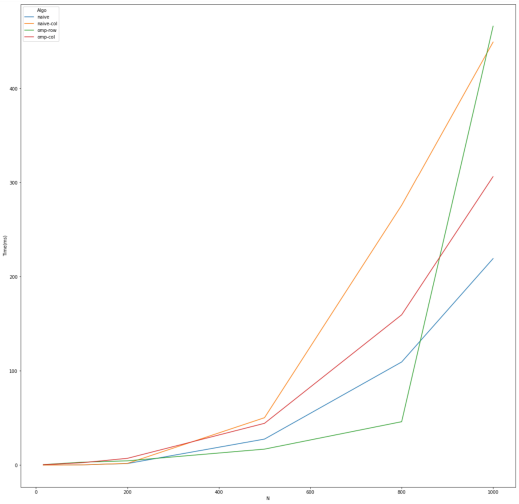

图 11: 按行按列划分

(一) 基于 Vtune 的程序性能剖析

由于这次的编程侧重点是编译后指令上的优化, 所以使用 Vtune profiling 的侧重点在微体系结构的指令数、周期数以及 CPI 以及线程执行情况上, 故按上次实验进行相同配置即可, 由于这次测试用例规模的变动不是重点, 所以 CPU 间隔取 0.1ms 即可。

如图为 N=2000 时四线程的 Profiling, 其他 N 取值表现相似。

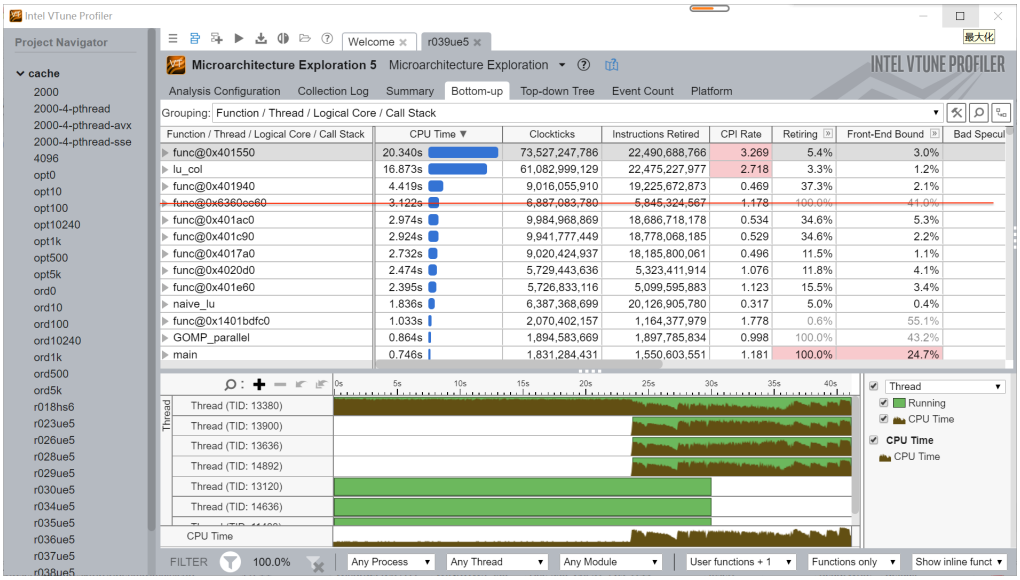

图 12: N=2000-4thread

这里的 Call Stack 无法显示包含 OpenMP 的函数名，只能根据线程时间进行推测，包括 naive 上方在内的 8 个 func 以及下方一个 func 就是我们要测的内容,static、guided 和 dynamic 优化的内容由于给每个线程分配的迭代次数基本一致故而线程占据的 CPU 时间形态是一致的，而按列划分由于前面要经过临界区等待的时间所以 4 个线程的形态不尽相同。时间相对较短的 SSE 优化在指令数上做了精简，所以最终效果也最佳。naive 占据的 CPU 时间不多但是指令数多 CPI 也尚可，可以认为是重复指令过多导致的,lu_col 指令数与 naive 类似但是所耗时间几乎到了 naive 的 8 倍，可见在 N 较大的时候 cache 的局部性有多么重要。

而对于四核八线程导致的八线程跟四线程的优化效果一致问题，根据 Vtune 的结果，可以发现其实在四线程的时候对每一个线程的操作也是通过八个虚拟核实现的，所以四线程和八线程的效果基本上是一致的。

四、 总结

这个实验整体较上次容易，掌握 OpenMP 的基本用法并实践即可。主要是划分的情形较多。

参考文献

[1] 高斯消元法 [EB/OL].https://zh.wikipedia.org/wiki/高斯消去法

[2] https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE3,AVX&expand=3931

[3] https://www.openmp.org

[4] 课件 algo1-2,openmp4-12