南 开 大 学

NanKai University

OpenMP 编程实验——以高斯消去为例

姓名：文静静

学号：1811507

年级：2018级

专业：计算机科学与技术

2021年5月21日

摘要

本文使用所学 OpenMP 编程进行多线程实验，针对消去部分的两重循环，按照不同任务划分方式分别设计并实现多线程算法。并考虑将其与 SIMD 算法结合，对各算法性能进行分析。同时借助 Vune profiling 等工具分析算法过程中的同步开销和空闲等待等。

关键字：高斯消去　OpenMp　SIMD　VTune

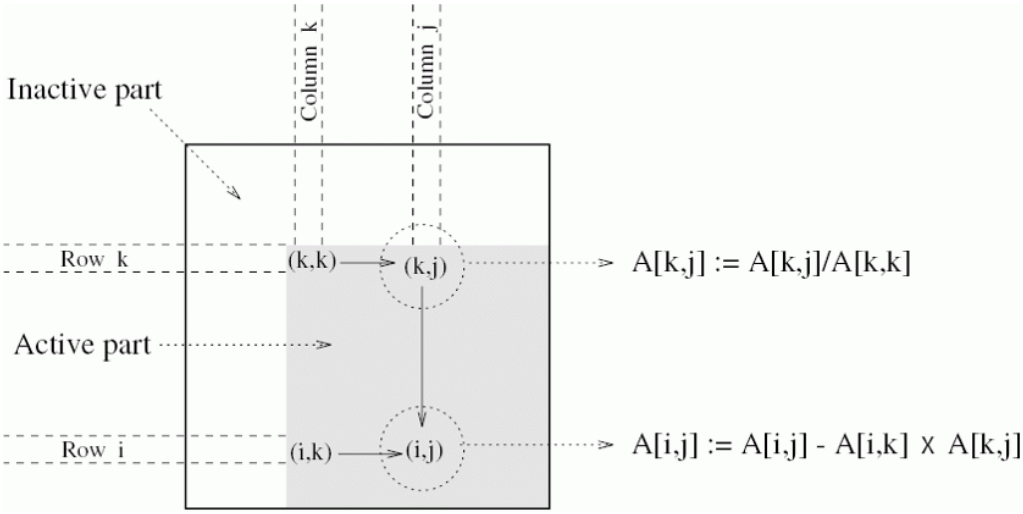
目录

1 绪论

本文通过高斯消去法探究 OpenMP 多线程编程对于性能的提升。具体探究了使用按行划分与按列划分、结合 SSE 与 AVX、改变矩阵大小、静态调度与动态调度等不同情况下的时间性能以及加速比，还探究了 pthread 与 OpenMP 的差别。使用了性能剖析根据 VTune 对几种算法进行了具体的剖析，对比了不同算法下最终所执行的指令数，周期数，CPI 等等。

2 实验选题

高斯消去的计算模式如图所示，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。



高斯消去法（LU 分解）OpenMP 并行化为例，具体要求如下：

1. 使用 OpenMP 编程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），分别设计并实现并行算法。

2. 与 SIMD (SSE/AVX) 算法结合。

3. 改变矩阵的大小、调度方式、线程数量等参数，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。

3 算法设计与编程

3.1 初始化

初始化系数矩阵：在使用高斯消去法的时候，有除法操作，为了避免出现除数为0以及无解的情况，将系数矩阵初始为上三角矩阵以便运算和验证。（N为系数矩阵规模大小）

```
// 初始系数矩阵，为了方便将其设置成上三角矩阵A
void init()
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < i; j++)
        {
            A[i][j] = 0;
        }
        for (int j = i; j < N; j++)
        {
            A[i][j] = i + j + 1;
        }
    }
}
```

串行的高斯消去法代码如下：

```
// 高斯消去法（串行）
void Gauss_plain()
{
    for (int k = 0; k < N; k++)
    {
        for (int j = k + 1; j < N; j++)
        {
            A[k][j] = A[k][j] / A[k][k];
        }
        A[k][k] = 1.0;
        for (int i = k + 1; i < N; i++)
        {
            for (int j = k + 1; j < N; j++)
```

```
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

3.2 OpenMP 编程

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，既可以采用水平划分（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）。因此本次实验，只对 8-13 行的消去进行水平划分和垂直划分的线程分配，进行多线程算法测试。

因此本次实验的做法是在最外层使用 omp parallel，统一创建线程，对于除法部分使用 omp single 执行，除法执行完成之后才能执行二重循环部分，不使用 critical 而使用 single 的原因是 single 是指定一段代码应由单线程(不一定是主线程)执行，而 critical 是指定代码一次由一个线程执行。因此，前者将只执行一次，而后者将被执行与线程数量相同的次数。所以为了避免不必要的时间开销，本次实验使用 single。然后对于双重循环使用 omp for，让各线程分别去执行任务，最外层循环执行完后在统一销毁线程。

3.2.1 按行划分

```
// 按行划分
void omp_row()
{
    #pragma omp parallel
    for (int k = 0; k < N; k++)
```

```
    {
        #pragma omp single
        {
            for (int j = k + 1; j < N; j++)
            {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
        }
        #pragma omp for
        for (int i = k + 1; i < N; i++)
        {
            for (int j = k + 1; j < N; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

3.2.2 按列划分

```
// 按列划分
void omp_col()
{
    #pragma omp parallel
    for (int k = 0; k < N; k++)
    {
        #pragma omp single
        {
            for (int j = k + 1; j < N; j++)
            {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
        }
        #pragma omp for
```

```
        for (int j = k + 1; j < N; j++)
        {
            for (int i = k + 1; i < N; i++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
        }
        #pragma omp single
        for (int i = k + 1; i < N; i++)
        {
            A[i][k] = 0;
        }
    }
}
```

3.2.3 静态调度

OpenMP 中任务调度主要针对并行的 for 循环，当循环中每次迭代的计算量不相等时，如果简单地给各个线程分配相同次数的迭代，则可能会造成各个线程计算负载的不平衡，影响程序的整体性能。

大部分编译器在没有使用 schedule 子句的时候，默认是 static 调度。static 在编译的时候就已经确定了，那些循环由哪些线程执行。

```
// 按行划分（静态）
void omp_row_static()
{
    #pragma omp parallel
    for (int k = 0; k < N; k++)
    {
        #pragma omp single
        {
            for (int j = k + 1; j < N; j++)
            {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
        }
```

```
        #pragma omp for schedule(static)
        for (int i = k + 1; i < N; i++)
        {
            for (int j = k + 1; j < N; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

3.2.4 动态调度

动态调度依赖于运行时的状态动态确定线程所执行的迭代，也就是线程执行完已经分配的任务后，会去领取还有的任务。由于线程启动和执行完的时间不确定，所以迭代被分配到哪个线程是无法事先知道的。当使用参数 size 时，逐个分配 size 个迭代给各个线程。本次实验使用 size 为 8。

```
// 按行划分（动态）
void omp_row_dynmaic()
{
    #pragma omp parallel
    for (int k = 0; k < N; k++)
    {
        #pragma omp single
        {
            for (int j = k + 1; j < N; j++)
            {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
        }
        #pragma omp for schedule(dynamic, 8)
        for (int i = k + 1; i < N; i++)
        {
            for (int j = k + 1; j < N; j++)
```

```
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

3.2.5 启发式调度

采用启发式调度 guided 方法进行调度，每次分配给线程迭代次数不同，开始比较大，以后逐渐减小。参数 size 表示每次分配的迭代次数的最小值，由于每次分配的迭代次数会逐渐减少，少到 size 时，将不再减少。如果不知道 size 的大小，那么默认 size 为1，即一直减少到1。本次实验使用默认参数1。

```
// 按行划分（启发式）
void omp_row_guided()
{
    #pragma omp parallel
    for (int k = 0; k < N; k++)
    {
        #pragma omp single
        {
            for (int j = k + 1; j < N; j++)
            {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
        }
        #pragma omp for schedule(guided, 1)
        for (int i = k + 1; i < N; i++)
        {
            for (int j = k + 1; j < N; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
```

```
            A[i][k] = 0;
        }
    }
}
```

3.3 结合 SIMD

除法部分和消去的过程都可以使用 SSE/AVX 加速。在之前的实验中，已经分析得到，按列划分的方法在使用 SSE/AVX 时，由于数据地址不连续，需要将矩阵元素一个个地装入寄存器中，而按行划分则利用了数据的连续性，可以直接将首地址作为参数传入，因此按列划分的性能并不好，所以本次实验不考虑按列划分的情况，只进行按行划分的 SIMD 向量化处理。

3.3.1 结合 SSE

```
// SSE
void omp_sse()
{
    __m128 t1, t2, t3, t4;
    int remain = 0;
    #pragma omp parallel private(t1, t2, t3, t4)
    for (int k = 0; k < N; k++)
    {
        #pragma omp single
        {
            // 不能凑够个的部分串行计算4
            remain = (N - k - 1) % 4;
            for (int j = k + 1; j <= k + remain; j++)
            {
                A[k][j] = A[k][j] / A[k][k];
            }
            // 并行计算剩下的倍数个，减少使用除法4
            t1 = _mm_set1_ps(1 / A[k][k]);
            for (int j = k + remain + 1; j < N; j += 4)
            {
                t2 = _mm_load_ps(A[k] + j);
```

```
                t2 = _mm_mul_ps(t1, t2);
                _mm_store_ps(A[k] + j, t2);
            }
            A[k][k] = 1;
        }
        #pragma omp for
        for (int i = k + 1; i < N; i++)
        {
            // 不能凑够个的部分串行计算4
            for (int j = k + 1; j <= k + remain; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            t1 = _mm_set1_ps(A[i][k]);
            // 并行计算剩下的倍数个4float
            for (int j = k + remain + 1; j < N; j += 4)
            {
                t2 = _mm_load_ps(A[k] + j);
                t3 = _mm_mul_ps(t1, t2);
                t4 = _mm_load_ps(A[i] + j);
                t4 = _mm_sub_ps(t4, t3);
                _mm_store_ps(A[i] + j, t4);
            }
            A[i][k] = 0;
        }
    }
}
```

3.3.2 结合 AVX

```
// AVX
void omp_avx()
{
    __m256 t1, t2, t3, t4;
    int remain = 0;
    #pragma omp parallel private(t1, t2, t3, t4)
    for (int k = 0; k < N; k++)
    {
```

```
        #pragma omp single
        {
            // 不能凑够个的部分串行计算8
            remain = (N - k - 1) % 8;
            for (int j = k + 1; j <= k + remain; j++)
            {
                A[k][j] = A[k][j] / A[k][k];
            }
            // 并行计算剩下的倍数个，减少使用除法8
            t1 = _mm256_set1_ps(1 / A[k][k]);
            for (int j = k + remain + 1; j < N; j += 8)
            {
                t2 = _mm256_loadu_ps(A[k] + j);
                t2 = _mm256_mul_ps(t1, t2);
                _mm256_storeu_ps(A[k] + j, t2);
            }
            A[k][k] = 1.0;
        }
        #pragma omp for
        for (int i = k + 1; i < N; i++)
        {
            // 不能凑够个的部分串行计算8
            for (int j = k + 1; j <= k + remain; j++)
            {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            t1 = _mm256_set1_ps(A[i][k]);
            // 并行计算剩下的倍数个8float
            for (int j = k + remain + 1; j < N; j += 8)
            {
                t2 = _mm256_loadu_ps(A[k] + j);
                t3 = _mm256_mul_ps(t1, t2);
                t4 = _mm256_loadu_ps(A[i] + j);
                t4 = _mm256_sub_ps(t4, t3);
                _mm256_storeu_ps(A[i] + j, t4);
            }
            A[i][k] = 0;
        }
```

```
    }
}
```

4 编译运行

在 CodeBlocks 中编译运行：通过 QueryPerformance 来精确计时，同时运行 epoch=10 次取平均值。

操作系统环境： Windows 10 专业版 (64 位)

处理器： Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz (4 CPUs), 2.5GHz

RAM: 8.0GB

多线程的编译选项：-lpthread

SSE的编译选项：-march=corei7、-march=native

AVX的编译选项：-march=corei7-avx、-march=native

5 结果分析

先对上述算法的正确性进行验证，检验结果正确后，对不同规模、不同算法、不同线程下的性能进行测试并分析。

正确性判断代码如下：

```
// 正确性比较
void comp(float *A[N], float *B[N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = i; j < N; j++)
        {
            if (fabs(A[i][j] - B[i][j]) > 1e-4)
            {
                cout << "i=" << i << ",j=" << j << ", 误差超出范围，误差为:
                    ";
                cout << fabs(A[i][j] - B[i][j]) << endl;
```

```
12                  cout << "A[i][j]=" << A[i][j] << ",B[i][j]=" << B[i][j]
                        << endl;
13                  return;
14              }
15          }
16      }
17      cout << "误差未超出范围" << endl;
18  }
```

5.1 规模

使用CPU-Z查看本机cache的信息如下：

因此分别设置测试规模 N = 16, 32, 64, 128, 256, 512, 1024, 2048。

5.2 水平划分与垂直划分

针对不同规模下，水平划分与垂直划分的时间性能如下：

规模	平凡算法	omp_row	omp_col
16	0.0058	2.07957	2.99835
32	0.04175	3.92397	6.03067
64	0.33488	7.50375	10.925
128	3.05493	14.1001	22.8307
256	22.7974	31.0867	61.0123
512	185.49	96.7408	244.694
1024	1416.99	483.835	1346.08
2048	11506.3	3523.48	11490.6

图 1: 水平划分与垂直划分时间比较

由此得到加速比如下：

规模	omp_row	omp_col
16	0.002789	0.001934
32	0.01064	0.006923
64	0.04463	0.03065
128	0.21666	0.133808
256	0.73335	0.373653
512	1.91739	0.75805
1024	2.92866	1.05268
2048	3.26561	1.00136

图 2: 水平划分与垂直划分加速比比较

矩阵规模小于 512 时，串行用时比所有并行策略的运行时间少得多。这是因为创建和销毁线程的代价很大，数据量较小时，这个代价远大于并行节省的时间。矩阵规模大于 256 之后，并行的优化效果逐渐显现，N 在 512 至 1024 之间时，加速比随数据量的增加而增加，这说明此时的数据量还不足，一些子线程处于空闲状态。如果每个子线程都能拿到充足的数据，就能充分发挥多线程并行的优势。

按行划分的加速比远高于按列划分的加速比，这是因为 C++ 按行储存矩阵，按列分配任务会导致访问的数据地址不连续，增加了 cache 缺失率，运行时间增长。

5.3 调度方式分析

不同调度方式的时间性能以及加速比对比如下：

规模	平凡算法	static	dynmaic	guided
16	0.0058	2.17419	3.17818	2.04985
32	0.04175	4.01433	3.89461	3.65475
64	0.33488	7.33609	7.05756	7.09885
128	3.05493	14.2815	14.3496	14.6151
256	22.7974	30.0343	32.1821	33.2525
512	185.49	99.5941	102.652	103.879
1024	1416.99	498.748	499.159	547.82
2048	11506.3	3242.26	3229.31	3421.19

图 3: 不同调度方式时间比较

规模	static	dynmaic	guided
16	0.00267	0.001825	0.00283
32	0.0104	0.01072	0.01142
64	0.04565	0.04745	0.04717
128	0.214	0.2129	0.21
256	0.759	0.7084	0.68558
512	1.86246	1.80697	1.78563
1024	2.84109	2.83875	2.5866
2048	3.54885	3.56309	3.36324

图 4: 不同调度方式加速比比较

当规模比较小的时候，静态调度相对于动态调度的性能要更好一些。这是因为对于静态调度 static，每次哪些循环由那个线程执行时固定的。而当规模比较大的时候，静态调度显然不如动态调度，是因为静态调度每个线

程的任务是固定的，但是可能有的循环任务执行快，有的慢，不能达到最优。而动态调度 dynamic 会根据线程的执行快慢，已经完成任务的线程会自动请求新的任务或者任务块，每次领取的任务块是固定的。对于启发式调度guided，每个任务分配的任务是先大后小，指数下降。当有大量任务需要循环时，刚开始为线程分配大量任务，最后任务不多时，给每个线程少量任务，可以达到线程任务均衡。由于此次实验的规模有限，无法体现启发式调度在大量任务时的均衡优势。

5.4 SSE 与 AVX

针对不同规模下，SSE 与 AVX 的时间性能以及折线图如下：

规模	平凡算法	omp_row	omp_sse	omp_avx
16	0. 0058	2. 07957	1. 9353	2. 43746
32	0. 04175	3. 92397	3. 69165	4. 10942
64	0. 33488	7. 50375	6. 81279	7. 53786
128	3. 05493	14. 1001	13. 5749	14. 4673
256	22. 7974	31. 0867	27. 1151	27. 9524
512	185. 49	96. 7408	69. 437	60. 7309
1024	1416. 99	483. 835	266. 861	206. 675
2048	11506. 3	3523. 48	1806. 82	1372. 15

图 5: 结合 SSE 与 AVX 的时间比较

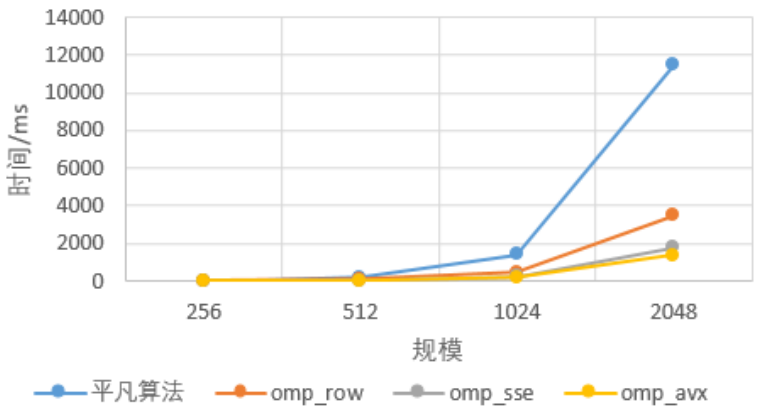


图 6: 结合 SSE 与 AVX 的时间折线图

由此得到加速比如下：

规模	omp_row	omp_sse	omp_avx
16	0.002789	0.003	0.002375
32	0.01064	0.01131	0.01016
64	0.04463	0.049	0.04443
128	0.21666	0.225042	0.211162
256	0.73335	0.840764	0.81558
512	1.91739	2.67134	3.05429
1024	2.92866	5.30983	6.856
2048	3.26561	6.36827	8.38558

图 7: 结合 SSE 与 AVX 的加速比比较

当矩阵规模小于 512 时，SSE/AVX 的使用基本没有任何优化效果，这是因为数据量太小，并行的开销超过了并行的收益。当数据量增大时，加速效果明显，当数据大于 1024 时，SSE 能获得 6 倍左右的加速比，AVX 能获得 8 倍左右的加速比。

5.5 线程数量对比

测试不同规模下，2、4、8线程数时按行访问的时间性能，得到加速比如下：

规模	2线程	4线程	8线程
16	0.003715	0.002789	0.001525
32	0.01415	0.01064	0.00603
64	0.0662	0.04463	0.0263
128	0.304186	0.21666	0.12188
256	0.8111	0.73335	0.4545
512	1.32821	1.91739	1.295
1024	1.65823	2.92866	2.33152
2048	1.745	3.26561	3.03575

图 8: 不同线程数的加速比比较

得到不同规模下不同线程数的加速比折线图如下：

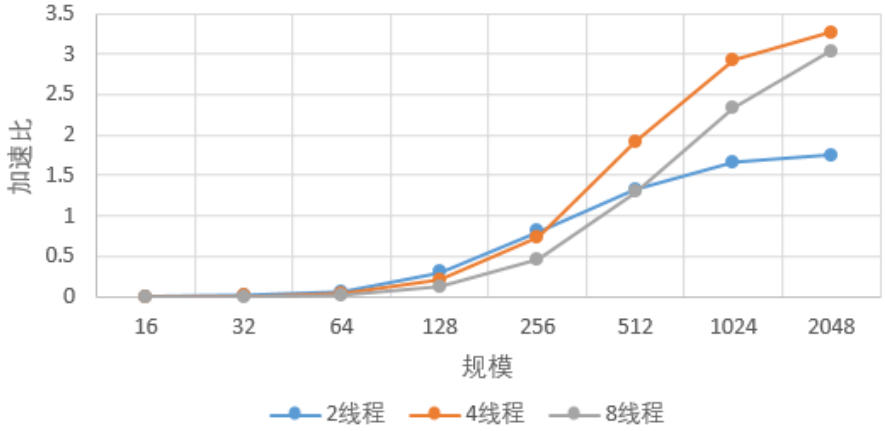


图 9: 不同线程数的加速比折线图

可以发现，当规模比较小的时候，2 线程的加速比是优于 4 线程和 8 线程的，因为此时数据量小，线程效率高。当数据量充足（大于 256）时，4 线程的加速比明显优于 2 线程，8 线程的加速比与 4 线程的接近，因此增加的子线程没能提高加速比。由于线程数越多，额外开销就越多，8 线程的效率会比较低。因此 4 线程并行是比较合适的，线程较少不能充分地利用 CPU 核心，若创建过多的线程数，只能增加上下文切换的次数，因此会带来额外的开销。

5.6 与 pthread 对比

得到按行访问、静态调度时 OpenMP 与 pthread 的加速比对比如下：

规模	pthread	omp_row	pthread_sse	omp_sse
16	0. 002859	0. 002789	0. 002612	0. 003
32	0. 011761	0. 01064	0. 0113665	0. 01131
64	0. 06637	0. 04463	0. 06888	0. 049
128	0. 24086	0. 21666	0. 268614	0. 225042
256	0. 84939	0. 73335	0. 9404	0. 840764
512	1. 794	1. 91739	2. 456	2. 67134
1024	2. 9188	2. 92866	4. 7025	5. 30983
2048	2. 9448	3. 26561	5. 0829	6. 36827

图 10: OpenMP 与 pthread 加速比比较

由此得到的折线图：

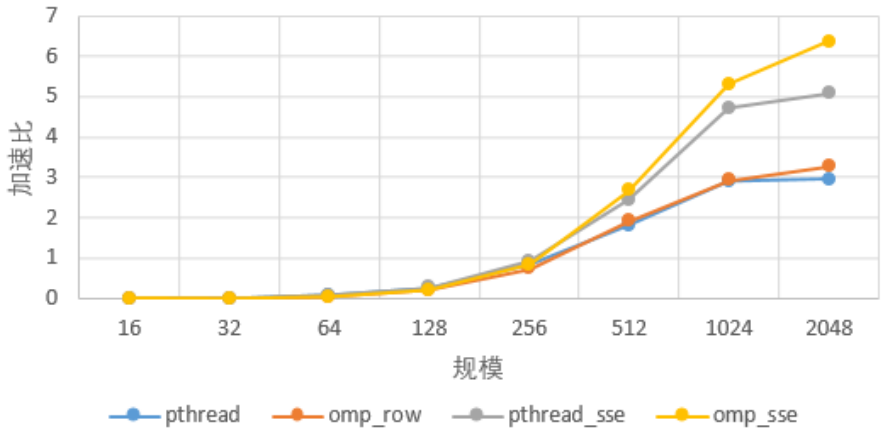


图 11: OpenMP 与 pthread 加速比折线图

可以发现 pthread 与 OpenMP 的加速比几乎差不多。OpenMP 提供大量其它的编译指示来识别需要进行线程化的代码块、在线程中共享的范围变量或本地化到单独线程，到同步线程，如何安排任务或循环叠代到线程等等。因此，最终它将通过线程功能提供中等级别纹理型控制。这种级别的纹理对于许多高性能计算（HPC）应用而言已经足够，在便携性和最佳执行能力的前提下，OpenMP 能够提供比大多数其它选择更好的选择，尤其是最大限度减少对于代码库的干扰。如果开发人员需要精细纹理的控制，则可以使用 OpenMP 的线程 API。API 包括一小组函数，分为以下三个领域：查询执行环境的线程资源并设置当前的线程数量；设置、管理并释放锁来解决线程之间的资源访问；以及一个小型的定时接口。但是使用这种 API 会取走纯编译指令方法提供的优势。在这个级别上，OpenMP API 是 Pthreads 提供的功能的一个小子集。这两个 API 都具有便携性，但是 Pthreads 能提供更大范围的原函数（primitive function），从而对线程化操作提供精细纹理的控制。因此，在必须单独管理线程的应用中，Pthreads 或本地的线程化 API（如 Windows 上的 Win32）将是更加自然的选择[1]。因此对于 pthread 和 OpenMP 不能简单用性能来衡量，需要在不同情况下考量使用不同的方法。

6 VTune 性能分析

使用 Vtune 分析静态调度与动态调度算法过程中的同步开销和空闲等待：对于 Microarchitecture Exploration，选择 Grouping: Function/thread/Logical Core/Call Stack，对于 4 个子线程、矩阵规模 N=1024 的情况如下：

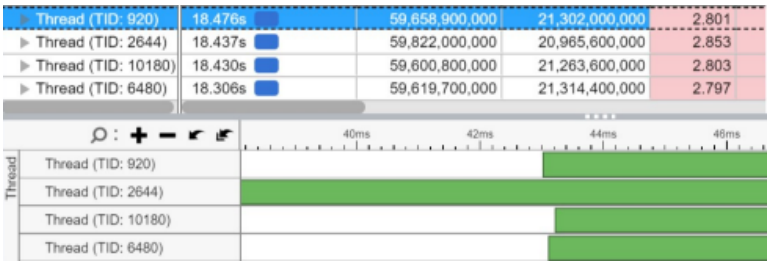


图 12: 静态调度

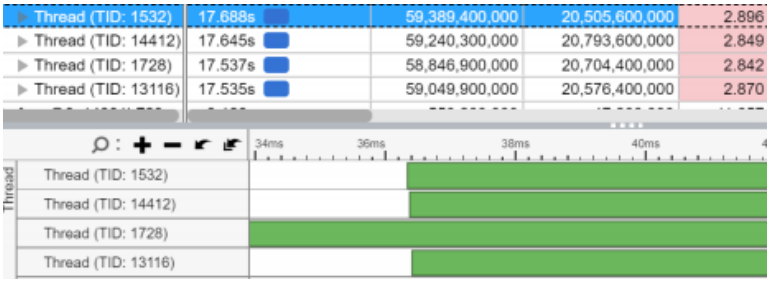
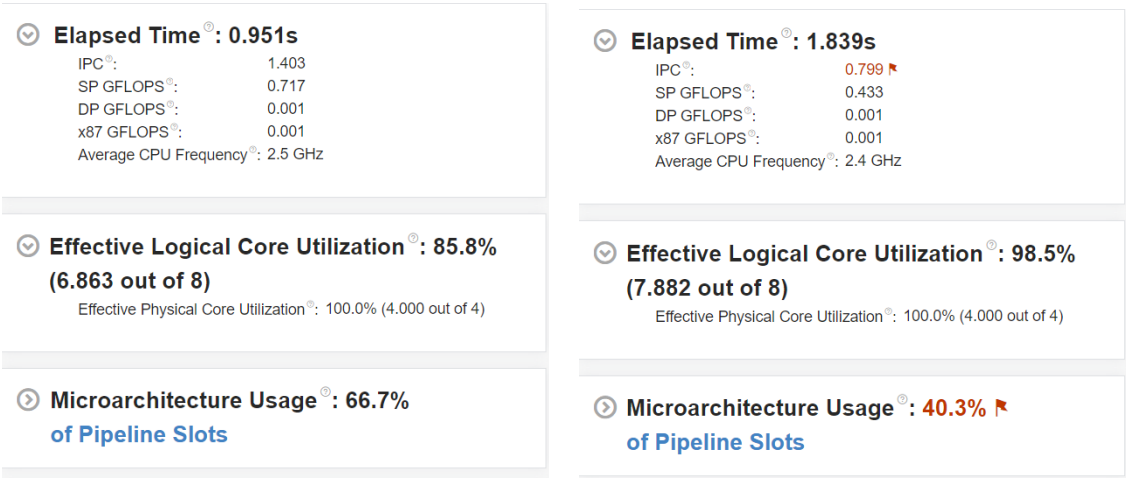


图 13: 动态调度

可以看出在静态调度时显然没有动态调度负载均衡。

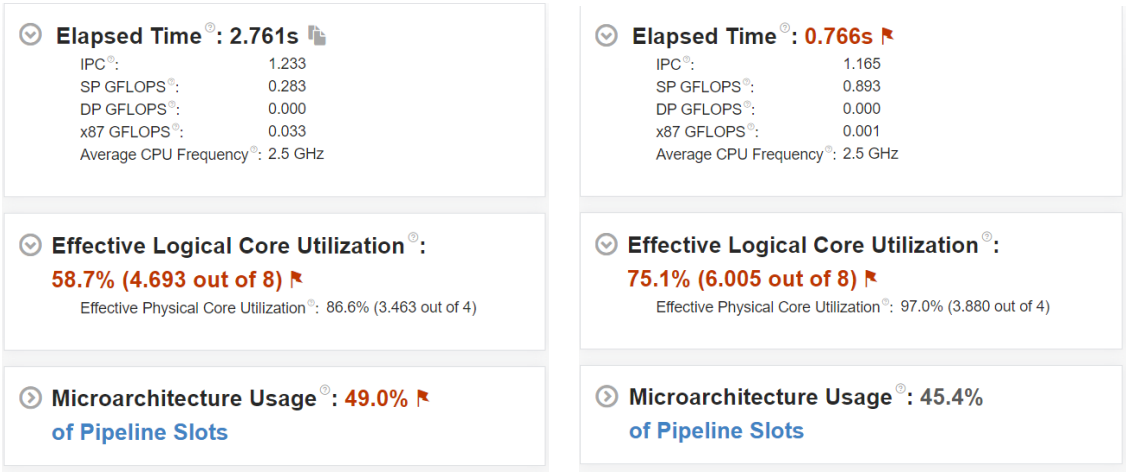
测试单次运行按行划分、按列划分多线程运行时间和相关数据如下：



(a) 按行划分多线程　　(b) 按列划分多线程

图 14: 按行划分与按列划分的数据对比

测试单次运行平凡算法和结合 SSE 多线程运行时间和相关数据如下：



(a) 平凡算法　　(b) SSE 结合多线程

图 15: 平凡算法和结合 SSE 多线程的数据对比

可以发现，按行访问大程度的提高了性能，而且较为接近结合 SSE 的性能，而按列访问花了将近按行访问的两倍时间，可以发现按列访问的 IPC 显然大于其他算法，因为按列访问增加了指令数量，导致每个时钟周期执行的指令数下降，时间性能也不如按行访问，因此我们在使用算法时，符合 cache 的存储规则，考虑空间局部性是十分重要的。

7 总结

这次实验理解了 OpenMP 的实现，了解了不同调度方式的区别与特点，同时通过 VTune 分析了算法过程中的同步开销和空闲等待等。也明白 pthread 与 OpenMP 不能简单用时间性能来衡量，需要在具体情况下具体分析。

参考文献

[1] https://software.intel.com/content/www/cn/zh/develop/articles/threading-models-for-high-performance-computing-pthreads-or-openmp.html