



南开大学
Nankai University

《并行程序与设计》实验报告

(2023~2024 学年第一学期)

实验名称: OpenMP 编程练习

学 院: 软件学院

姓 名: 陈高楠

学 号: 2112966

指导老师: 孙永谦

2023 年 12 月 2 日

目录

1 梯形积分法的 Pthread、OpenMP 版本	1
1 问题描述	1
2 算法设计与分析	1
2.1 pthread 方法	1
2.2 OpenMP 方法 (静态划分)	3
2.3 OpenMP 方法 (动态划分 (dynmaic))	4
2.4 OpenMP 方法 (动态划分 (guided))	4
3 结果统计与分析	5
3.1 结果正确性	5
3.2 两种编程方法的异同	5
2 “多个数组排序”的任务不均衡案例	7
1 问题描述	7
2 算法设计与分析	7
2.1 OpenMP 方法 (静态划分)	7
2.2 OpenMP 方法 (动态划分 (dynmaic))	8
2.3 OpenMP 方法 (动态划分 (guided))	8
3 结果统计与分析	9
3.1 三种分配方式运行时间比较	9
3.2 任务分配粒度对运行时间影响	10
3.3 线程数对运行时间影响	11
4 实验结论	12
3 附加题：高斯消去法的 OpenMP 编程	13
1 问题描述	13
2 算法设计与分析	13
2.1 OpenMP(静态划分)	13
2.2 OpenMP(动态划分 (dynmaic))	14
2.3 OpenMP(动态划分 (guided))	15
2.4 OpenMP(静态划分) 结合 SSE	16

2.5 OpenMP(动态划分 (dynmaic)) 结合 SSE	17
2.6 OpenMP(动态划分 (guided) 结合 SSE	18
2.7 OpenMP(静态划分) 结合 AVX	20
2.8 OpenMP(动态划分 (dynmaic)) 结合 AVX	21
2.9 OpenMP(动态划分 (guided) 结合 AVX	22
3 结果统计与分析	24
3.1 三种分配方式运行时间比较	24
3.2 分配任务不同粒度的比较	26
3.3 不同线程数的比较	26
4 实验结论	27

实验一 梯形积分法的 Pthread、OpenMP 版本

1 问题描述

分别实现课件中的梯形积分法的 Pthread、OpenMP 版本，熟悉并掌握 OpenMP 编程方法，探讨两种编程方式的异同。

2 算法设计与分析

2.1 pthread 方法

下面是使用 pthread 实现梯形积分法的代码，主要分为主函数 Cal(int n) 和线程函数 Cal_pthread(void*param)，先进行局部求和再对总的进行相加，使用动态任务分配的方法。

```
1 void *Cal_pthread(void *param) {
2     threadParam_t* p = (threadParam_t*)param;
3     int task = 0;
4     int n=p->n;
5     double h=p->h;
6     double x = 0;
7     double currArea = 0;
8     while (true) {
9         pthread_mutex_lock(&barrier_mutex);
10        next_task += 1;
11        task = next_task;
12        pthread_mutex_unlock(&barrier_mutex);
13        if (task ≥ n )
14            break;
15        else {
16            for (int i = task ; i < task+1; i++) {
```

```
17         x = a + i * h;
18         currArea = func(x) * h;
19         pthread_mutex_lock(&barrier_mutex);
20         total += currArea;
21         pthread_mutex_unlock(&barrier_mutex);
22     }
23 }
24 }
25 pthread_exit(nullptr);
26 return nullptr;
27 }
28 double Cal(int n) {
29     LARGE_INTEGER freq;
30     LARGE_INTEGER beginTime;
31     LARGE_INTEGER endTime;
32     QueryPerformanceFrequency(&freq);
33     QueryPerformanceCounter(&beginTime);
34     int thread_cnt = THREAD_NUM;
35     pthread_t* thread_handles = (pthread_t*)malloc(thread_cnt * sizeof(...
        pthread_t));
36     threadParam_t* param = (threadParam_t*)malloc(thread_cnt * sizeof(...
        threadParam_t));
37     double h=(b-a)/n;
38     total=(func(a)/2+func(b)/2)*h;
39     for (int t_id = 0; t_id < thread_cnt; t_id++) { //分配任务
40         param[t_id].t_id = t_id; //第几个线程
41         param[t_id].n=n;
42         param[t_id].h=h;
43
44     }
45
46     for (int t_id = 0; t_id < thread_cnt; t_id++) {
47         pthread_create(&thread_handles[t_id], NULL, Cal_pthread, &param[...
            t_id]);
48     }
49
50     for (int t_id = 0; t_id < thread_cnt; t_id++) {
```

```
51         pthread_join(thread_handles[t_id], NULL);
52     }
53     free(thread_handles);
54     free(param);
55     QueryPerformanceCounter(&endTime);
56     double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
57     cout << "普通Pthread方法耗时: " << time << "s" << endl;
58     return time;
59
60
61 }
```

2.2 OpenMP 方法 (静态划分)

使用 OpenMP 来进行并行计算,代码简洁很多。只需要在需要并行的语句上面加上 OpenMP 的命令就可以。这里采用的是静态划分任务的方法。

具体代码如下:

```
1 double Trap_static( double a, double b, int n)
2 {
3     total2 = 0.0;
4     double h = (b - a) / n;
5     double local_result = 0.0;
6
7     #pragma omp parallel for num_threads(THREAD_NUM) schedule(static) ...
        reduction(+:local_result)
8     for (int i = 0; i < n; i++) {
9         double local_x = a + i * h;
10        local_result += func(local_x) + func(local_x + h);
11    }
12
13    total2 = local_result * h / 2;
14
15 } /* Trap*/
```

2.3 OpenMP 方法 (动态划分 (dynmaic))

这里采用动态划分任务的方法，使用的是 dynmatic 的方法。

具体代码如下：

```
1 double Trap_static( double a, double b, int n)
2 {
3     total2 = 0.0;
4     double h = (b - a) / n;
5     double local_result = 0.0;
6
7     #pragma omp parallel for num_threads(THREAD_NUM) schedule(static) ...
        reduction(+:local_result)
8     for (int i = 0; i < n; i++) {
9         double local_x = a + i * h;
10        local_result += func(local_x) + func(local_x + h);
11    }
12
13    total2 = local_result * h / 2;
14
15 } /* Trap*/
```

2.4 OpenMP 方法 (动态划分 (guided))

这里采用动态划分的方法，采用的是 guided 的方法。

具体代码如下：

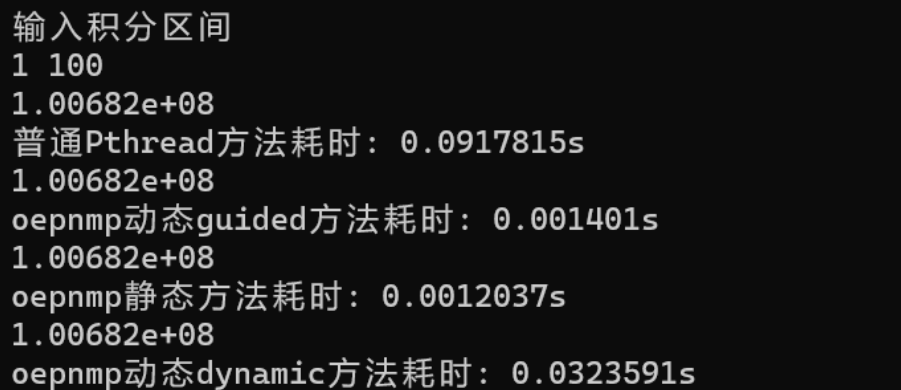
```
1 double Trap_guided( double a, double b, int n)
2 {
3     total2 = 0.0;
4     double h = (b - a) / n;
5     double local_result = (func(a)/2+func(b)/2) ;;
6
7     #pragma omp parallel for num_threads(THREAD_NUM) schedule(guided) ...
        reduction(+:local_result)
```

```
8   for (int i = 1; i < n; i++) {
9       double local_x = a + i * h;
10      local_result += func(local_x);
11  }
12
13  total2 = local_result * h ;
14
15 } /* Trap*/
```

3 结果统计与分析

3.1 结果正确性

将积分函数设为 $f(x) = 4x^3 + 2x^2 + 3x$, 划分一共 1000000 个梯形, 积分区间从 1 到 100, 测试结果正确性。



```
输入积分区间
1 100
1.00682e+08
普通Pthread方法耗时: 0.0917815s
1.00682e+08
oepnmp动态guided方法耗时: 0.001401s
1.00682e+08
oepnmp静态方法耗时: 0.0012037s
1.00682e+08
oepnmp动态dynamic方法耗时: 0.0323591s
```

图 1.1: 测试结果

由图可知, 使用 Pthread 和 OpenMP 来计算, 结果均正确。

3.2 两种编程方法的异同

相同点:

1. 两者都用于实现并行处理, 允许程序同时执行多个任务。
2. 两者采用共享内存模型, 线程可以访问相同的内存地址空间。这种模型便于数据共享, 但同时也增加了数据一致性和同步的复杂性。

不同点：

1.OpenMP 代码更为简略，写起来很方便，且便于拓展。Pthread 代码更为复杂，写起来更繁琐。

2.Pthread 是一个库，所有的并行线程创建都需要我们自己完成。OpenMP 是根植于编译器的，更偏向于将原来串行化的程序，通过加入一些适当的编译器指令变成并行执行。

实验二 “多个数组排序”的任务不均衡案例

1 问题描述

对于课件中“多个数组排序”的任务不均衡案例进行 OpenMP 编程实现（规模可自己调整），并探索不同循环调度方案的优劣。提示：可从任务分块的大小、线程数的多少、静态动态多线程结合等方面进行尝试，探索规律。

2 算法设计与分析

2.1 OpenMP 方法 (静态划分)

只需要在排序的循环前面加上 OpenMP 的指令就可以了。具体代码如下：

```
1 double arr_sort_static() {
2     LARGE_INTEGER freq;
3     LARGE_INTEGER beginTime;
4     LARGE_INTEGER endTime;
5     QueryPerformanceFrequency(&freq);
6     QueryPerformanceCounter(&beginTime);
7     #pragma omp parallel for num_threads(THREAD_NUM) schedule(static)
8     for (int i = 0; i < ARR_NUM; i++) {
9         stable_sort(arr[i].begin(), arr[i].end());
10    }
11    QueryPerformanceCounter(&endTime);
12    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
13    cout << "arr_sort_static方法耗时: " << time << "s" << endl;
14    return time;
15 }
```

2.2 OpenMP 方法 (动态划分 (dynmaic))

具体代码如下:

```
1 double arr_sort_dynamic() {
2     LARGE_INTEGER freq;
3     LARGE_INTEGER beginTime;
4     LARGE_INTEGER endTime;
5     QueryPerformanceFrequency(&freq);
6     QueryPerformanceCounter(&beginTime);
7     #pragma omp parallel for num_threads(THREAD_NUM) schedule(dynamic)
8     for (int i = 0; i < ARR_NUM; i++) {
9         stable_sort(arr[i].begin(), arr[i].end());
10    }
11    QueryPerformanceCounter(&endTime);
12    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
13    cout << "arr_sort_dynamic方法耗时: " << time << "s" << endl;
14    return time;
15 }
```

2.3 OpenMP 方法 (动态划分 (guided))

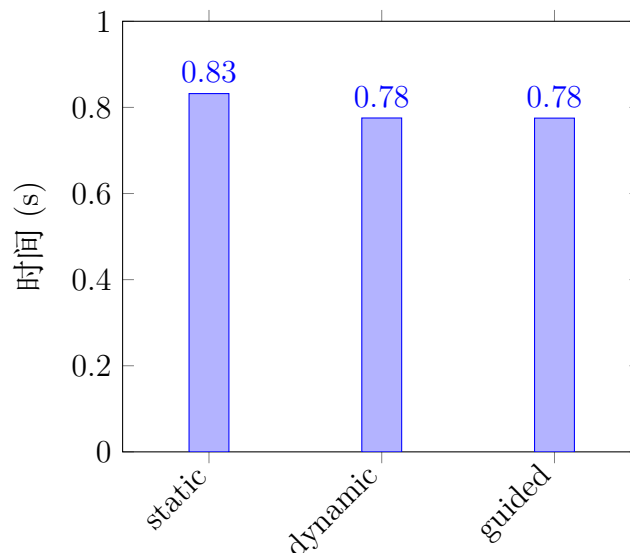
具体代码如下:

```
1 double arr_sort_guided() {
2     LARGE_INTEGER freq;
3     LARGE_INTEGER beginTime;
4     LARGE_INTEGER endTime;
5     QueryPerformanceFrequency(&freq);
6     QueryPerformanceCounter(&beginTime);
7     #pragma omp parallel for num_threads(THREAD_NUM) schedule(guided)
8     for (int i = 0; i < ARR_NUM; i++) {
9         stable_sort(arr[i].begin(), arr[i].end());
10    }
```

```
11     QueryPerformanceCounter(&endTime);
12     double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
13     cout << "arr_sort_guided方法耗时: " << time << "s" << endl;
14     return time;
15 }
```

3 结果统计与分析

3.1 三种分配方式运行时间比较

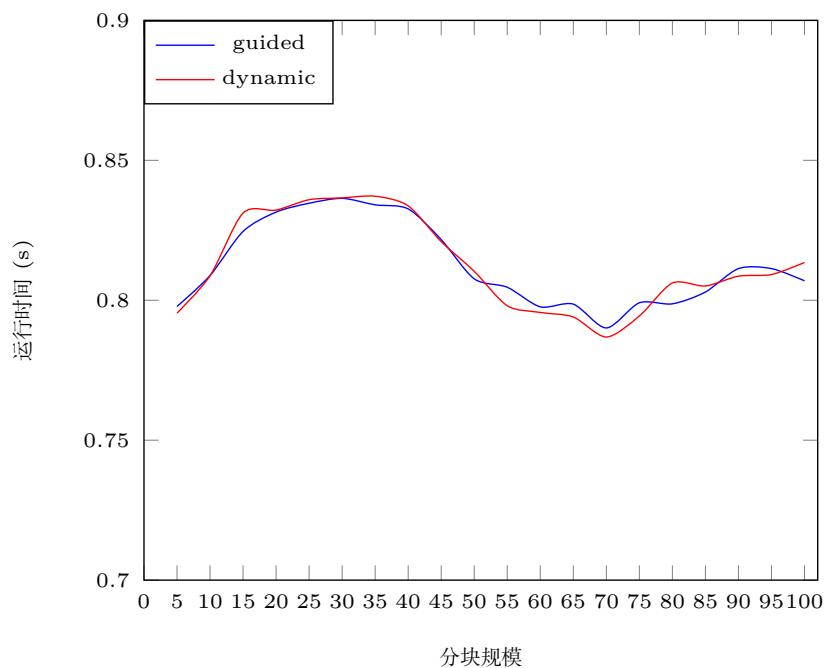


当矩阵是有规律地随机时, (1/4 升序, 1/4 降序, 1/4 一半升序一半降序, 1/4 一半降序一半升序) 不同线程之间存在难度的差距, 因此可以体现出动态线程分配的优势。而此时静态线程分配因为总的时间受到分配的任务重的线程的影响, 导致总运行时间上升。而 guided 分配和 dynamic 分配的效果几乎相同, 没有特别大的差别。

3.2 任务分配粒度对运行时间影响

分块数量	Guided	Dynamic
5	0.797776	0.795422
10	0.808834	0.808752
15	0.824588	0.831149
20	0.831492	0.83221
25	0.834662	0.835923
30	0.83641	0.836561
35	0.83409	0.837164
40	0.832625	0.833725
45	0.821666	0.820944
50	0.807656	0.810345
55	0.804642	0.798036
60	0.79765	0.795646
65	0.79861	0.794006
70	0.790106	0.786864
75	0.799088	0.794381
80	0.798721	0.806158
85	0.802942	0.805095
90	0.811305	0.808611
95	0.811308	0.809185
100	0.806943	0.813505

表 2.1: 分块对动态分配的影响



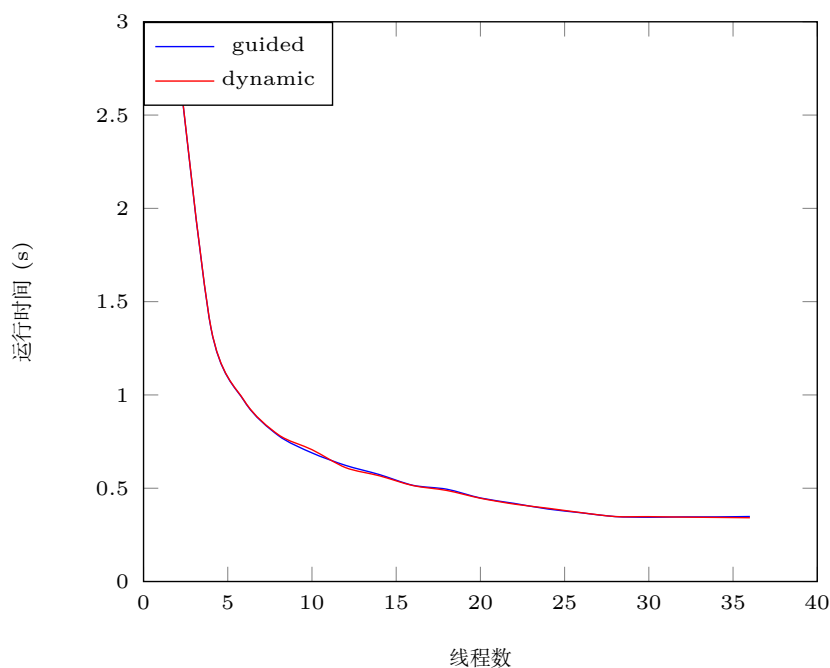
当分块规模处于 5 到 45 时，粒度越大执行时间越长，然后慢慢下降再波动。可以看出，当

任务分块粒度大概为 70 时效果比较好。

3.3 线程数对运行时间影响

线程数	Guided	Dynamic
2	2.84171	2.84496
4	1.35914	1.36205
6	0.964004	0.96658
8	0.782621	0.787265
10	0.689366	0.70671
12	0.622311	0.610241
14	0.573268	0.566678
16	0.515538	0.514147
18	0.494827	0.488096
20	0.447727	0.446275
22	0.41884	0.415032
24	0.388628	0.392826
26	0.368274	0.368921
28	0.347936	0.348274
30	0.344495	0.347705
32	0.346295	0.345157
34	0.346399	0.343575
36	0.348431	0.341794

表 2.2: 分块对动态分配的影响



从图中可以看出，当线程数慢慢增加时，刚开始加速的效果很好，而后慢慢变缓，线程数到 28 后基本停止变化。推测线程数增加导致线程管理成本变大，因此加速效果并不是特别好。因此线程数为 28 的效果比较好。

4 实验结论

通过以上测试可以得知，当线程数取 28，任务分配粒度采取 70 时效果比较好，guided 和 dynamic 在不均匀数组的排序上没有特别大的区别。

实验三 附加题：高斯消去法的 OpenMP 编程

1 问题描述

实现高斯消去法解线性方程组的 OpenMP 编程，与 SSE/AVX 编程结合，并探索优化任务分配方法。

2 算法设计与分析

2.1 OpenMP(静态划分)

OpenMP(静态划分)的方法，在需要并行计算的循环上面加上 OpenMP 相关的指令就可以。具体代码如下：

```
1 double LU_omp_static(int n, float a[][maxN]) { //
2     LARGE_INTEGER freq;
3     LARGE_INTEGER beginTime;
4     LARGE_INTEGER endTime;
5     QueryPerformanceFrequency(&freq);
6     QueryPerformanceCounter(&beginTime);
7
8     for (int i = 0; i < n - 1; i++) //对每一行
9     {
10         for (int j = i + 1; j < n; j++) //对这一行的每一个数
11         {
12             a[i][j] = a[i][j] / a[i][i];
13         }
14         a[i][i] = 1.0;
15         #pragma omp parallel for num_threads(THREAD_NUM) schedule(static)
16         for (int j = i + 1; j < n; j++) //每一行的下面的行
17         {
18             float tem = a[j][i] / a[i][i];
```



```
19         for (int k = i + 1; k ≤ n; k++) //这个是列，列是n+1个（有b）
20         {
21             a[j][k] -= a[i][k] * tem;
22         }
23         a[j][i] = 0.00;
24     }
25 }
26 QueryPerformanceCounter(&endTime);
27 double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
    freq.QuadPart;
28 // cout << "LU_omp_static方法耗时: " << time << "s" << endl;
29 return time;
30 }
```

2.2 OpenMP(动态划分 (dynmaic))

与上面同理，把划分任务的方法改成 dynamic 就好。具体代码如下：

```
1 double LU_omp_dynamic(int n, float a[][maxN]) { //
2     LARGE_INTEGER freq;
3     LARGE_INTEGER beginTime;
4     LARGE_INTEGER endTime;
5     QueryPerformanceFrequency(&freq);
6     QueryPerformanceCounter(&beginTime);
7
8     for (int i = 0; i ≤ n - 1; i++) //对每一行
9     {
10         for (int j = i + 1; j ≤ n; j++)//对这一行的每一个数
11         {
12             a[i][j] = a[i][j] / a[i][i];
13         }
14         a[i][i] = 1.0;
15         #pragma omp parallel for num_threads(THREAD_NUM) schedule(dynamic)
16         for (int j = i + 1; j < n; j++)//每一行的下面的行
17         {
18             float tem = a[j][i] / a[i][i];
```

```
19         for (int k = i + 1; k ≤ n; k++) //这个是列，列是n+1个（有b）
20         {
21             a[j][k] -= a[i][k] * tem;
22         }
23         a[j][i] = 0.00;
24     }
25 }
26 QueryPerformanceCounter(&endTime);
27 double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
    freq.QuadPart;
28 // cout << "LU_omp_dynamic方法耗时: " << time << "s" << endl;
29 return time;
30 }
```

2.3 OpenMP(动态划分 (guided))

把划分任务的方法改成 guided。具体代码如下：

```
1 double LU_omp_guided(int n, float a[][maxN]) { //
2     LARGE_INTEGER freq;
3     LARGE_INTEGER beginTime;
4     LARGE_INTEGER endTime;
5     QueryPerformanceFrequency(&freq);
6     QueryPerformanceCounter(&beginTime);
7
8     for (int i = 0; i ≤ n - 1; i++) //对每一行
9     {
10         for (int j = i + 1; j ≤ n; j++)//对这一行的每一个数
11         {
12             a[i][j] = a[i][j] / a[i][i];
13         }
14         a[i][i] = 1.0;
15         #pragma omp parallel for num_threads(THREAD_NUM) schedule(guided)
16         for (int j = i + 1; j < n; j++)//每一行的下面的行
17         {
18             float tem = a[j][i] / a[i][i];
```

```
19         for (int k = i + 1; k ≤ n; k++) //这个是列，列是n+1个（有b）
20         {
21             a[j][k] -= a[i][k] * tem;
22         }
23         a[j][i] = 0.00;
24     }
25 }
26 QueryPerformanceCounter(&endTime);
27 double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
    freq.QuadPart;
28 // cout << "LU_omp_guided方法耗时: " << time << "s" << endl;
29 return time;
30 }
```

2.4 OpenMP(静态划分) 结合 SSE

在循环里面结合 SSE。具体代码如下：

```
1 double LU_omp_static_sse(int n, float a[][maxN]) { //
2
3     LARGE_INTEGER freq;
4     LARGE_INTEGER beginTime;
5     LARGE_INTEGER endTime;
6     QueryPerformanceFrequency(&freq);
7     QueryPerformanceCounter(&beginTime);
8     for (int i = 0; i ≤ n - 1; i++) //对每一行
9     {
10         for (int j = i + 1; j ≤ n; j++)//对这一行的每一个数
11         {
12             a[i][j] = a[i][j] / a[i][i];
13         }
14         a[i][i] = 1.0;
15         #pragma omp parallel for num_threads(THREAD_NUM) schedule(static)
16         for (int j = i + 1; j < n; j++)//每一行的下面的行
17         {
18
```

```

19         float tem = a[j][i] / a[i][i];
20         __m128 tem2=__mm_set1_ps(tem);
21         for (int k = i + 1; k ≤ n; k+=4) //这个是列，列是n+1个（有b）
22         {
23             if(k+3>n)break;
24             __m128 t1=__mm_loadu_ps(a[i]+k);
25             __m128 t2=__mm_loadu_ps(a[j]+k);
26             __m128 sub=__mm_sub_ps(t2,__mm_mul_ps(t1,tem2));
27             __mm_storeu_ps(a[j]+k,sub);
28
29         }
30         for (int k=n-(n-i)%4+1;k≤n;k+=1){
31             a[j][k] -= a[i][k] * tem;
32         }
33         a[j][i] = 0.00;
34     }
35 }
36 QueryPerformanceCounter(&endTime);
37 double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
    freq.QuadPart;
38 //     cout << "LU_omp_static_sse方法耗时： " << time << "s" << endl;
39     return time;
40 }

```

2.5 OpenMP(动态划分 (dynmaic)) 结合 SSE

在循环里面结合 SSE。具体代码如下：

```

1 double LU_omp_dynamic_sse(int n,float a[][maxN]) { //
2     LARGE_INTEGER freq;
3     LARGE_INTEGER beginTime;
4     LARGE_INTEGER endTime;
5     QueryPerformanceFrequency(&freq);
6     QueryPerformanceCounter(&beginTime);
7     for (int i = 0; i ≤ n - 1; i++) //对每一行
8     {

```

```

9      for (int j = i + 1; j ≤ n; j++)//对这一行的每一个数
10     {
11         a[i][j] = a[i][j] / a[i][i];
12     }
13     a[i][i] = 1.0;
14     #pragma omp parallel for num_threads(THREAD_NUM) schedule(dynamic)
15     for (int j = i + 1; j < n; j++)//每一行的下面的行
16     {
17
18         float tem = a[j][i] / a[i][i];
19         __m128 tem2 = _mm_set1_ps(tem);
20         for (int k = i + 1; k ≤ n; k+=4) //这个是列，列是n+1个（有b）
21         {
22             if(k+3>n)break;
23             __m128 t1 = _mm_loadu_ps(a[i]+k);
24             __m128 t2 = _mm_loadu_ps(a[j]+k);
25             __m128 sub = _mm_sub_ps(t2, _mm_mul_ps(t1, tem2));
26             _mm_storeu_ps(a[j]+k, sub);
27
28         }
29         for (int k = n - (n - i) % 4 + 1; k ≤ n; k += 1){
30             a[j][k] -= a[i][k] * tem;
31         }
32         a[j][i] = 0.00;
33     }
34 }
35 QueryPerformanceCounter(&endTime);
36 double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
    freq.QuadPart;
37 // cout << "LU_omp_guided_sse方法耗时: " << time << "s" << endl;
38 return time;
39 }

```

2.6 OpenMP(动态划分 (guided) 结合 SSE

在循环里面结合 SSE。具体代码如下：

```

1 double LU_omp_guided_sse(int n, float a[][maxN]) { //
2     LARGE_INTEGER freq;
3     LARGE_INTEGER beginTime;
4     LARGE_INTEGER endTime;
5     QueryPerformanceFrequency(&freq);
6     QueryPerformanceCounter(&beginTime);
7     for (int i = 0; i ≤ n - 1; i++) //对每一行
8     {
9         for (int j = i + 1; j ≤ n; j++)//对这一行的每一个数
10        {
11            a[i][j] = a[i][j] / a[i][i];
12        }
13        a[i][i] = 1.0;
14        #pragma omp parallel for num_threads(THREAD_NUM) schedule(guided)
15        for (int j = i + 1; j < n; j++)//每一行的下面的行
16        {
17
18            float tem = a[j][i] / a[i][i];
19            __m128 tem2=__mm_set1_ps(tem);
20            for (int k = i + 1; k ≤ n; k+=4) //这个是列，列是n+1个（有b）
21            {
22                if(k+3>n)break;
23                __m128 t1=__mm_loadu_ps(a[i]+k);
24                __m128 t2=__mm_loadu_ps(a[j]+k);
25                __m128 sub=__mm_sub_ps(t2, __mm_mul_ps(t1, tem2));
26                __mm_storeu_ps(a[j]+k, sub);
27
28            }
29            for (int k=n-(n-i)%4+1;k≤n;k+=1){
30                a[j][k] -= a[i][k] * tem;
31            }
32            a[j][i] = 0.00;
33        }
34    }
35    QueryPerformanceCounter(&endTime);
36    double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;

```

```
37 //      cout << "LU_omp_guided_sse方法耗时: " << time << "s" << endl;
38      return time;
39 }
```

2.7 OpenMP(静态划分) 结合 AVX

在循环里面结合 AVX。具体代码如下：

```
1 double LU_omp_static_avx(int n, float a[][maxN]) { //
2     LARGE_INTEGER freq;
3     LARGE_INTEGER beginTime;
4     LARGE_INTEGER endTime;
5     QueryPerformanceFrequency(&freq);
6     QueryPerformanceCounter(&beginTime);
7     for (int i = 0; i <= n - 1; i++) //对每一行
8     {
9         for (int j = i + 1; j <= n; j++)//对这一行的每一个数
10        {
11            a[i][j] = a[i][j] / a[i][i];
12        }
13        a[i][i] = 1.0;
14        #pragma omp parallel for num_threads(THREAD_NUM) schedule(static...
15        )
16        for (int j = i + 1; j < n; j++)//每一行的下面的行
17        {
18            float tem = a[j][i] / a[i][i];
19            __m256 tem2=__mm256_set1_ps(tem);
20            for (int k = i + 1; k <= n; k+=8) //这个是列，列是n+1个（有b）
21            {
22                if(k+7>n)break;
23                __m256 t1=__mm256_loadu_ps(a[i]+k);
24                __m256 t2=__mm256_loadu_ps(a[j]+k);
25                __m256 sub=__mm256_sub_ps(t2,__mm256_mul_ps(t1,tem2));
26                __mm256_storeu_ps(a[j]+k,sub);
27            }
```

```

28         }
29         for (int k=n-(n-i)%8+1;k≤n;k+=1){
30             a[j][k] -= a[i][k] * tem;
31         }
32         a[j][i] = 0.00;
33     }
34 }
35 QueryPerformanceCounter(&endTime);
36 double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
    freq.QuadPart;
37 // cout << "LU_omp_static_avx方法耗时: " << time << "s" << endl;
38 return time;
39 }

```

2.8 OpenMP(动态划分 (dynmaic)) 结合 AVX

在循环里面结合 AVX。具体代码如下：

```

1 double LU_omp_dynamic_avx(int n, float a[][maxN]) { //
2
3     LARGE_INTEGER freq;
4     LARGE_INTEGER beginTime;
5     LARGE_INTEGER endTime;
6     QueryPerformanceFrequency(&freq);
7     QueryPerformanceCounter(&beginTime);
8     for (int i = 0; i ≤ n - 1; i++) //对每一行
9     {
10         for (int j = i + 1; j ≤ n; j++)//对这一行的每一个数
11         {
12             a[i][j] = a[i][j] / a[i][i];
13         }
14         a[i][i] = 1.0;
15         #pragma omp parallel for num_threads(THREAD_NUM) schedule(dynamic)
16         for (int j = i + 1; j < n; j++)//每一行的下面的行
17         {
18

```



```

19         float tem = a[j][i] / a[i][i];
20         __m256 tem2=__mm256_set1_ps(tem);
21         for (int k = i + 1; k ≤ n; k+=8) //这个是列，列是n+1个（有b）
22         {
23             if(k+7>n)break;
24             __m256 t1=__mm256_loadu_ps(a[i]+k);
25             __m256 t2=__mm256_loadu_ps(a[j]+k);
26             __m256 sub=__mm256_sub_ps(t2,__mm256_mul_ps(t1,tem2));
27             __mm256_storeu_ps(a[j]+k,sub);
28
29         }
30         for (int k=n-(n-i)%8+1;k≤n;k+=1){
31             a[j][k] -= a[i][k] * tem;
32         }
33         a[j][i] = 0.00;
34     }
35 }
36     QueryPerformanceCounter(&endTime);
37     double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
38 //     cout << "LU_omp_dynamic_avx方法耗时：" << time << "s" << endl;
39     return time;
40 }

```

2.9 OpenMP(动态划分 (guided) 结合 AVX

在循环里面结合 AVX。具体代码如下：

```

1 double LU_omp_guided_avx(int n,float a[][maxN]) { //
2
3     LARGE_INTEGER freq;
4     LARGE_INTEGER beginTime;
5     LARGE_INTEGER endTime;
6     QueryPerformanceFrequency(&freq);
7     QueryPerformanceCounter(&beginTime);
8     for (int i = 0; i ≤ n - 1; i++) //对每一行

```

```
9      {
10          for (int j = i + 1; j ≤ n; j++)//对这一行的每一个数
11          {
12              a[i][j] = a[i][j] / a[i][i];
13          }
14          a[i][i] = 1.0;
15          #pragma omp parallel for num_threads(THREAD_NUM) schedule(guided)
16          for (int j = i + 1; j < n; j++)//每一行的下面的行
17          {
18
19              float tem = a[j][i] / a[i][i];
20              __m256 tem2=__mm256_set1_ps(tem);
21              for (int k = i + 1; k ≤ n; k+=8) //这个是列，列是n+1个（有b）
22              {
23                  if(k+7>n)break;
24                  __m256 t1=__mm256_loadu_ps(a[i]+k);
25                  __m256 t2=__mm256_loadu_ps(a[j]+k);
26                  __m256 sub=__mm256_sub_ps(t2, __mm256_mul_ps(t1, tem));
27                  __mm256_storeu_ps(a[j]+k, sub);
28
29              }
30              for (int k=n-(n-i)%8+1;k≤n;k+=1){
31                  a[j][k] -= a[i][k] * tem;
32              }
33              a[j][i] = 0.00;
34          }
35      }
36      QueryPerformanceCounter(&endTime);
37      double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
                 freq.QuadPart;
38      //      cout << "LU_omp_guided_avx方法耗时： " << time << "s" << endl;
39      return time;
40 }
```

3 结果统计与分析

3.1 三种分配方式运行时间比较

本次测试测试了九种 OpenMP 相关计算，分别是普通 LU(static), 普通 LU(guided), 普通 LU(dynamic), LU(static 与 SSE), LU(guided 与 SSE), LU(dynamic 与 SSE), LU(static 与 AVX), LU(guided 与 AVX), LU(dynamic 与 AVX)。

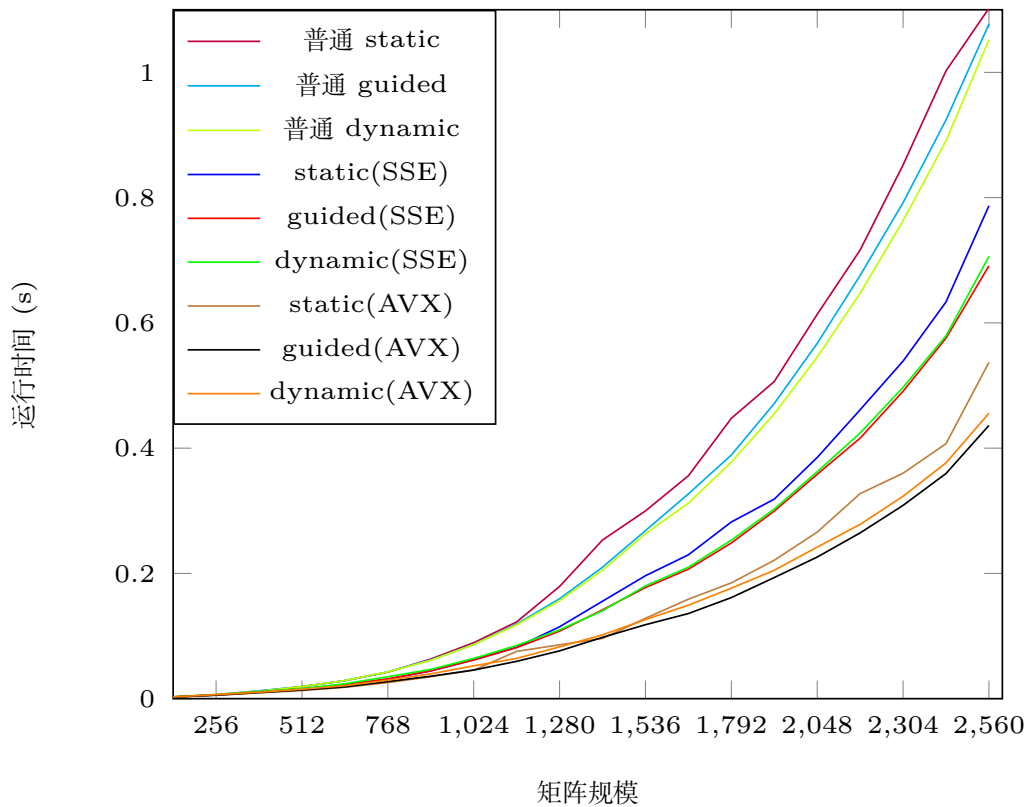
矩阵规模	static	guided	dynamic
128	0.00295702	0.00286672	0.0029147
256	0.00611565	0.00650837	0.00669373
384	0.0120125	0.0125682	0.011722
512	0.0192517	0.0192541	0.0193137
640	0.0287804	0.0288037	0.0289025
768	0.0423381	0.0427261	0.0428106
896	0.0633175	0.0620045	0.0612806
1024	0.0893394	0.0863307	0.08636
1152	0.122548	0.119227	0.117859
1280	0.17918	0.15959	0.156354
1408	0.253331	0.209841	0.204991
1536	0.299796	0.268326	0.263345
1664	0.355886	0.326965	0.312111
1792	0.447651	0.389015	0.377704
1920	0.506375	0.471307	0.454788
2048	0.613677	0.567271	0.545348
2176	0.716333	0.675764	0.64738
2304	0.852231	0.792008	0.763174
2432	1.00249	0.924109	0.890869
2560	1.10229	1.07714	1.05216

表 3.1: 不同矩阵规模下不同分配策略的 LU 分解执行时间比较

在只使用 OpenMP 的情况下，使用 dynamic 来分配的效果是最好的。接下来比较结合 SSE/AVX 的情况。

矩 阵 规 模	static(SSE)	guided(SSE)	dynamic(SSE)	static(AVX)	guided(AVX)	dynamic(AVX)
128	0.0026	0.0027	0.0028	0.0027	0.0026	0.0028
256	0.0058	0.0062	0.0062	0.0054	0.0057	0.0064
384	0.0105	0.011	0.0112	0.0093	0.0098	0.0103
512	0.0154	0.0157	0.0161	0.013	0.0139	0.0151
640	0.0216	0.0225	0.0238	0.0182	0.0188	0.0207
768	0.0312	0.0317	0.0351	0.0257	0.0273	0.0296
896	0.0442	0.0446	0.0467	0.0351	0.0358	0.0397
1024	0.0622	0.0617	0.0643	0.0456	0.0457	0.0524
1152	0.0821	0.0815	0.0849	0.0755	0.0596	0.0643
1280	0.1148	0.1078	0.1099	0.086	0.0763	0.0827
1408	0.1556	0.1419	0.1404	0.0962	0.0979	0.1019
1536	0.1963	0.1774	0.18	0.1283	0.1181	0.1263
1664	0.2297	0.2068	0.2099	0.1588	0.1359	0.1493
1792	0.2821	0.2486	0.253	0.1849	0.1614	0.1764
1920	0.3186	0.2995	0.3028	0.221	0.1934	0.205
2048	0.3851	0.3583	0.3623	0.2659	0.2261	0.242
2176	0.4613	0.416	0.4242	0.3274	0.2647	0.2783
2304	0.539	0.4909	0.4971	0.36	0.3086	0.3232
2432	0.6335	0.5757	0.5794	0.4071	0.3595	0.3765
2560	0.787	0.6911	0.7062	0.537	0.4363	0.4558

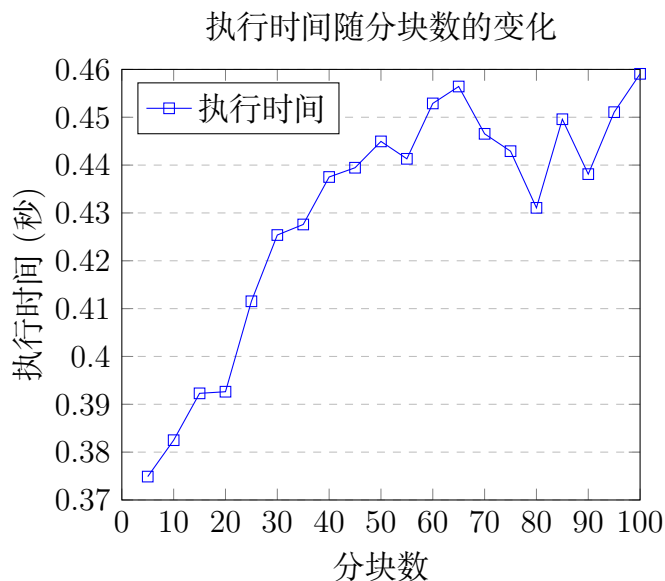
表 3.2: 不同矩阵规模下的 LU 分解算法执行时间比较 (SSE 和 AVX 版本)



从图中可以看出, 使用 AVX 比使用 SSE 效果更好。同时, 使用 guided 分配效果比 static, dynamic 分配更好。

3.2 分配任务不同粒度的比较

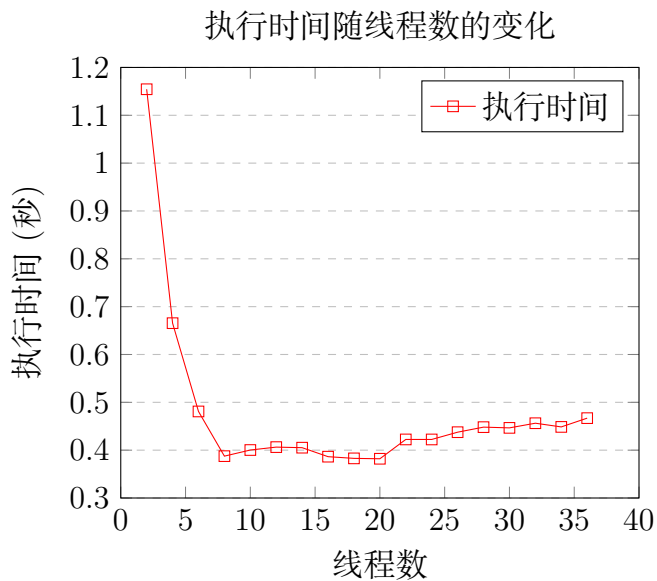
下面是线程数为 8, 使用 AVX 和 guided, 采取不同粒度分配的比较。



从图中可以看出, 当任务粒度越大时, 高斯消元所消耗的时间就越大。任务粒度的增加并没有提高执行效率, 因此, 任务粒度取 5 时是最合理的。

3.3 不同线程数的比较

下面比较了不同的线程数下, 分块规模为 5 的 guided 分配结合 AVX 的高斯消元算法的运行时间。



可以看出，线程数刚开始增加时，运行时间会减少很多。但是增加到线程数为 8 时便保持稳定，后面稍有起伏。这是因为线程数多了之后线程管理成本变大，因此加速效果并不是特别好。因此线程数为 8 的效果比较好。

4 实验结论

综上所述，当高斯消元使用 OpenMP，使用 guided 来动态分配任务，每次分配的任务粒度为 5，采用 8 线程来进行运算的效果是最好的，运行时间能够达到 0.359 秒，相比只用 OpenMP 的算法能够提高 3 倍。