

南開大學
Nankai University



《OpenMP 编程》
并行第04次作业

题 目：	OpenMP 编程
上课时间：	周一下午
授课教师：	孙永谦
姓 名：	张怡桢
学 号：	2013747
年 级：	2020级本科生
日 期：	2022/12/4

OpenMP 编程

张怡桢, 2013747

南开大学软件学院

摘 要: OpenMP是由OpenMP Architecture Review Board牵头提出的, 并已被广泛接受, 用于共享内存并行系统的多处理器程序设计的一套指导性编译处理方案(Compiler Directive) [1]。OpenMP支持的编程语言包括C、C++和Fortran; 而支持OpenMp的编译器包括Sun Compiler, GNU Compiler和Intel Compiler等。OpenMp提供了对并行算法的高层的抽象描述, 程序员通过在源代码中加入专用的pragma来指明自己的意图, 由此编译器可以自动将程序进行并行化, 并在必要之处加入同步互斥以及通信。当选择忽略这些pragma, 或者编译器不支持OpenMp时, 程序又可退化为通常的程序(一般为串行), 代码仍然可以正常运作, 只是不能利用多线程来加速程序执行。

关键词: OpenMP

1 实验内容

1. 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. 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。
3. 附加题: 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 实验环境

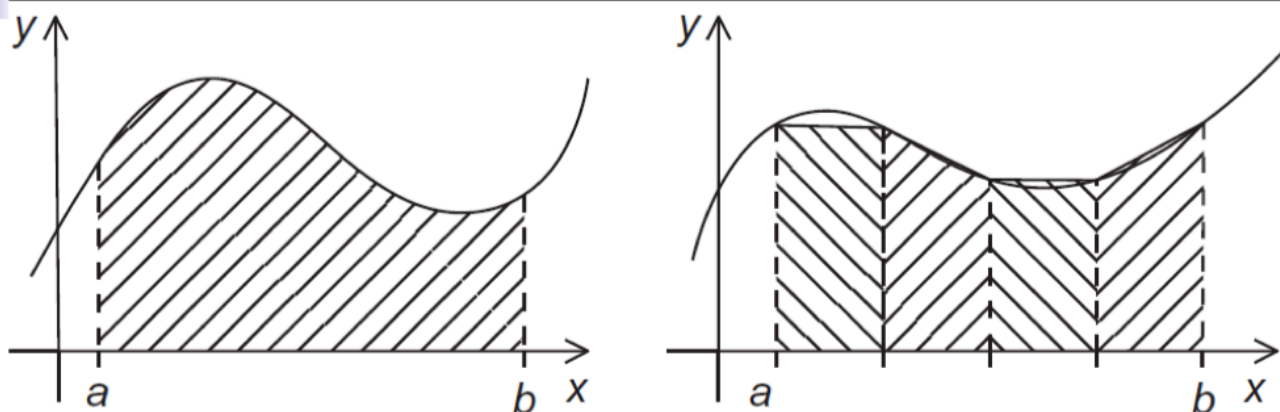
操作系统: MacOS, 编辑器: Clion, 编译器: lldw

3 梯形积分法

3.1 实验内容

参见ppt上的内容

梯形积分法：函数 $f(x)$ $[a,b]$ 间积分



- 间隔 $h=x_{i+1}-x_i=(b-a)/n$
- 每个梯形面积 $\frac{h}{2}[f(x_i)+f(x_{i+1})]$

$$x_0 = a, x_1 = a + h, x_2 = a + 2h, \dots, x_{n-1} = a + (n-1)h, x_n = b$$

- 梯形面积之和
: $h[f(x_0)/2 + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + f(x_n)/2]$

3.2 算法设计

分别实现课件中的梯形积分法的 Pthread 、 OpenMP 版本

3.2.1 Pthread版本

对于每个线程，先求局部和最后全局求和，避免过多通信。同时注意给全局和加锁保证结果的正确性。

代码实现如下：

//pthread编程线程控制程序

```
void pthread_trap(double a, double b, int n, double* global_result_p, int thread_count){  
    pthread_param params[thread_count];  
    pthread_t threads[thread_count];  
    pthread_mutex_t amutex;  
    pthread_mutex_init(&amutex, NULL);
```

```

for(int i=0;i<thread_count;i++){
    //为每个线程参数赋值
    params[i].a=a;
    params[i].b=b;
    params[i].n=n;
    params[i].global_result_p=global_result_p;
    params[i].thread_count=thread_count;
    params[i].my_rank=i;
    params[i].amutex=&amutex;
    //创建线程
    pthread_create(&threads[i],NULL,pthread_trap,(void*)&params[i]);
}
//销毁线程
for(int i=0;i<thread_count;i++){
    pthread_join(threads[i],NULL);
}
//销毁互斥量锁
pthread_mutex_destroy(&amutex);
}

```

其中每个线程执行的函数如下：

```

//pthread中每个线程调用的程序
void* pthread_trap(void* p){
    //恢复获取参数
    pthread_param* params=(pthread_param*) p;
    double h,x,my_result;
    double local_a,local_b;
    int local_n;
    int my_rank=params->my_rank;
    int thread_count=params->thread_count;
    h=(params->b-params->a)/params->n;
    local_n=params->n/thread_count;
    local_a=params->a+my_rank*local_n*h;
    local_b=local_a+local_n*h;
    my_result=(f(local_a)+f(local_b))/2.0;
    //先局部求和，避免过多通信
    for(int i=1;i<=local_n-1;i++){
        x=local_a+i*h;
        my_result+=f(x);
    }
    my_result=my_result*h;
}

```

```

//使用互斥量 amutex 上锁保证全局求和正确性
pthread_mutex_lock(params->amutex);
*params->global_result_p+=my_result;
pthread_mutex_unlock(params->amutex);
}

```

3.2.2 OpenMP版本

3.2.2.1 普通的OpenMP

利用 OpenMP 的相关方法实现多线程并行编程，思路与 Pthread 编程类似。代码实现如下：

```

//1:最普通的OpenMP版本
cout<<"Using OpenMP"<<endl;
#pragma omp parallel num_threads(thread_count)
    trap(a,b,n,&global_result);

```

其中每个线程执行的函数如下：

```

//每个线程执行的求和程序
void trap(double a,double b,int n,double*global_result_p){
    double h,x,my_result;
    double local_a,local_b;
    int local_n;
    int my_rank=omp_get_thread_num();
    int thread_count=omp_get_num_threads();
    h=(b-a)/n;
    local_n=n/thread_count;
    local_a=a+my_rank*local_n*h;
    local_b=local_a+local_n*h;
    my_result=(f(local_a)+f(local_b))/2.0;
    //先局部求和，避免过多通信
    for(int i=1;i<=local_n-1;i++){
        x=local_a+i*h;
        my_result+=f(x);
    }
    my_result=my_result*h;
    //用临界区保证全局求和正确性
#pragma omp critical
    *global_result_p+=my_result;
}

```

3.2.2.2 递归OpenMP

每个线程将计算得到的局部和返回，再在主进程中将这局部和求和得到最终结果。

//2:使用递归的OpenMP版本

```
cout<<"Using recurrence OpenMP"<<endl;
#pragma omp parallel num_threads(thread_count)
{
    double my_result=0.0;
    my_result+=local_trap(a,b,n);
    #pragma omp critical
    global_result+=my_result;
}
```

每个线程对局部求和的代码如下：

//返回局部和

```
double local_trap(double a,double b,int n){
    double h,x,my_result;
    double local_a,local_b;
    int local_n;
    int my_rank=omp_get_thread_num();
    int thread_count=omp_get_num_threads();
    h=(b-a)/n;
    local_n=n/thread_count;
    local_a=a+my_rank*local_n*h;
    local_b=local_a+local_n*h;
    my_result=(f(local_a)+f(local_b))/2.0;
    //局部求和，避免过多通信
    for(int i=1;i<=local_n-1;i++){
        x=local_a+i*h;
        my_result+=f(x);
    }
    my_result=my_result*h;
    return my_result;
}
```

3.2.2.3 归约简化OpenMP

//3:使用归约简化的递归OpenMP版本

```
cout<<"Using reduction OpenMP"<<endl;
#pragma omp parallel num_threads(thread_count)\
    reduction(+: global_result)
global_result+=local_trap(a,b,n);
```

3.3 实验结果

梯形设计

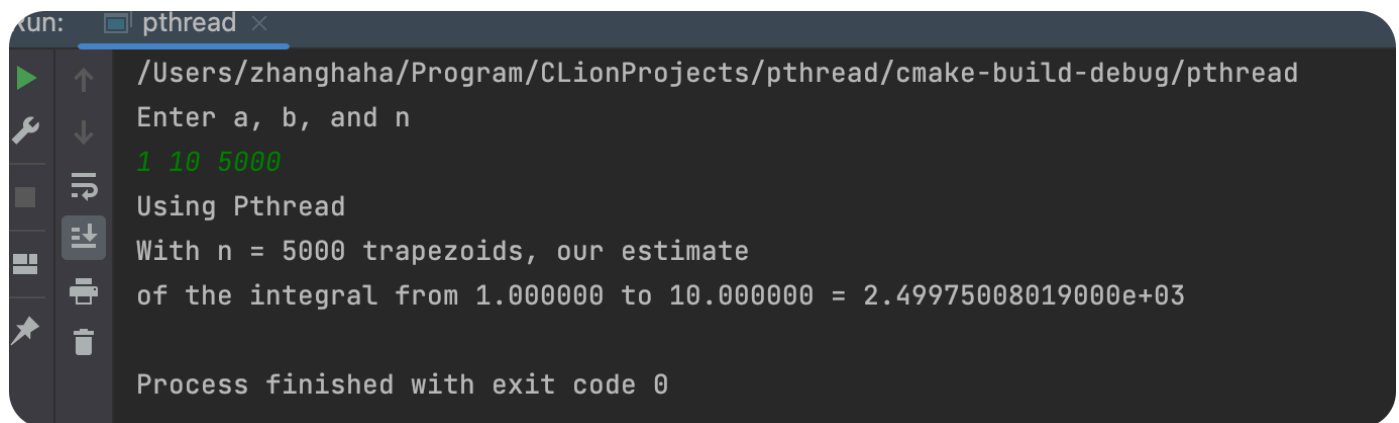
```
double f(double x){
    return (x*x*x);
}
```

待积分的函数设置为 x^3 ，积分区间从 1 到 10。将整个图形划分为 5000 个小梯形

(即 取 $a=1$ $b=10$ $n=5000$)。

实验结果如下：计算结果一致。

3.3.1 pthread



```
run: pthread x
/Users/zhanghaha/Program/CLionProjects/pthread/cmake-build-debug/pthread
Enter a, b, and n
1 10 5000
Using Pthread
With n = 5000 trapezoids, our estimate
of the integral from 1.000000 to 10.000000 = 2.499750080190000e+03

Process finished with exit code 0
```

3.3.2 Common OpenMP

```
Run: pthread x
/Users/zhanghaha/Program/CLionProjects/pthread/cmake-build-debug/pthread
Enter a, b, and n
1 10 5000
Using Common OpenMP
With n = 5000 trapezoids, our estimate
of the integral from 1.000000 to 10.000000 = 2.49975008019000e+03
Process finished with exit code 0
```

3.3.3 recurrence OpenMP

```
Run: pthread x
/Users/zhanghaha/Program/CLionProjects/pthread/cmake-build-debug/pthread
Enter a, b, and n
1 10 5000
Using recurrence OpenMP
With n = 5000 trapezoids, our estimate
of the integral from 1.000000 to 10.000000 = 2.49975008019000e+03
Process finished with exit code 0
```

3.3.4 reduction OpenMP

```
Run: pthread x
/Users/zhanghaha/Program/CLionProjects/pthread/cmake-build-debug/pthread
Enter a, b, and n
1 10 5000
Using reduction OpenMP
With n = 5000 trapezoids, our estimate
of the integral from 1.000000 to 10.000000 = 2.49975008019000e+03
Process finished with exit code 0
|
```

3.4 Pthread 和 OpenMP的异同

3.4.1 同

openmp和pthread基于共享内存系统。不同线程之间的数据就无须传递，直接传送指针就行。

3.4.2 异

1. 线程数量

- (a) Pthread 在程序启动时创建线程，再将工作分配到线程上。这种方法需要相当多的线程指定代码，而且不能保证能够随着可用处理器的数量而合理地进行扩充。
- (b) OpenMP 不需要指定数量，在有循环的地方加上代码，修改设置文件即可。OpenMP 非常方便，因为它不会将软件锁定在事先设定的线程数量中，但是相对的查错更难也更麻烦。

2. 编译方式

- (a) OpenMP 的编译需要添加编译器预处理指令 `#pragma`，创建线程等后续工作要编译器来完成。
- (b) pthread 就是一个库，所有的并行线程创建都需要我们自己完成，较 OpenMP 麻烦一点。对于精细纹理的控制，Pthread 能够提供更大范围的原函数，属于更优的选择。

3. OpenMP随时禁用也可运行

- (a) OpenMP 的编译指示的一项重要优势：通过禁用 OpenMP 支持，代码可作为单一线程应用进行编译。当调试程序时，以这样的方式编译代码拥有巨大优势。如果没有这种选择，开发人员会经常发现很难说明复杂的代码是否能够正确工作，因为线程问题或因为与线程无关的设计错误。

4 多个数组排序（OpenMP）

4.1 实验内容

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。

- 1. 对 `ARR_NUM` 个长度为 `ARR_LEN` 的一维数组进行排序，使用 OpenMP 多线程编程，每个线程处理一部分数组。
- 2. 本实验考虑数组均衡以及不均衡情况下的多个数组排序的问题。
- 3. 本实验采取静态划分、动态划分、粗粒度动态划分多个角度，探索任务分块大小、线程数多少、静态动态多线程结合等方面因素对数组排序效率的影响。

4.1.1 数组均衡初始化

各个数组内数据顺序基本无差异。将数组长度设为 10000，生成 10000 个数组。

//正常情况下的初始化待排序数组

```
void init_1(void){
    srand(unsigned(time(nullptr)));
    for (int i = 0; i < ARR_NUM; i++) {
        arr[i].resize(ARR_LEN);
        for (int j = 0; j < ARR_LEN; j++){
            arr[i][j] = rand();
        }
    }
}
```

4.1.2 数组不均衡初始化

初始化 ARR_NUM 行，ARR_LEN 列的数组。

当行数 i 属于 $0 \sim 2499$ 时， $\text{ratio} = 0$ ，此时 2500 行内完全升序排列；当

行数 i 属于 $2500 \sim 4999$ 时， $\text{ratio} = 32$ 此时随机有 $2500 * 1/4$ 行升序， $2500 * 3/4$ 行降序；

当 行数 i 属于 $5000 \sim 7499$ 时， $\text{ratio} = 64$ ，此时随机有 $2500 * 1/2$ 行升序， $2500 * 1/2$ 行降序；

当行数 i 属于 $7500 \sim 9999$ 时， $\text{ratio} = 128$ ，此时 2500 行完全降序。此时数组可分为四块，数组负载不均衡。

```
// 初始化待排序数组，使得
// 第一段：完全升序
// 第二段：1/4逆序，3/4升序
// 第三段：1/2逆序，1/2升序
// 第四段：完全逆序
void init_2(void){
    int ratio;
    srand(unsigned(time(nullptr)));
    for (int i = 0; i < ARR_NUM; i++){
        arr[i].resize(ARR_LEN);
        if(i < seg)ratio = 0;
        else if(i < seg * 2)ratio = 32;
        else if(i < seg * 3)ratio = 64;
        else ratio = 128;
        if((rand() & 127) < ratio){
            for(int j = 0; j < ARR_LEN; j++){
                arr[i][j] = ARR_LEN - j;
            }
        }
        else{
            for(int j = 0; j < ARR_LEN; j++){
                arr[i][j] = j;
            }
        }
    }
}
```

4.1.3 时间计算

使用mac系统，选择的计时器为gettimeofday()，头文件为#include <sys/time.h>，它的精度可以达到微妙，是C标准库的函数。

```

struct timeval tpstart, tpend;
gettimeofday(&tpstart, NULL);
gettimeofday(&tpend, NULL);
double timeuse = 1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
printf("used time:%fus\n", timeuse);

```

4.2 算法设计

4.2.1 串行方法

//1: 串行方法

```

init_2(); //初始化数组
gettimeofday(&tpstart, NULL);
for(int i=0; i<ARR_NUM; i++){
    stable_sort(arr[i].begin(), arr[i].end());
}
gettimeofday(&tpend, NULL);
cout<<"serial: "<<(1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-
tpstart.tv_usec)/1000<<" ms"<<endl;

```

4.2.2 omp静态块划分

//2: omp静态块划分

```

init_2(); //初始化数组
gettimeofday(&tpstart, NULL);
#pragma omp parallel for num_threads(THREAD_NUM)
for(int i=0; i<ARR_NUM; i++){
    stable_sort(arr[i].begin(), arr[i].end());
}
gettimeofday(&tpend, NULL);
cout<<"omp static: "<<(1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-
tpstart.tv_usec)/1000<<" ms"<<endl;

```

4.2.3 omp动态块划分（一次划分一个任务，细粒度）

```

init_2(); //初始化数组
    gettimeofday(&tpstart,NULL);
#pragma omp parallel for num_threads(THREAD_NUM)\
    schedule(dynamic,1)
    for(int i=0;i<ARR_NUM;i++){
        stable_sort(arr[i].begin(),arr[i].end());
    }
    gettimeofday(&tpend,NULL);
    cout<<"omp dynamic(TASK_NUM=1): "<<(1000000*(tpend.tv_sec-
tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec)/1000<<" ms"<<endl;

```

4.2.4 omp动态块划分（一次划分TASK_NUM个任务，粗粒度）

动态分配任务，每个线程每次分配到 TASK_NUM 个任务后执行数组排序，完成后再次领取 新的任务，直至所有数组均已排序完成。通过设置 TASK_NUM 的值，控制任务分配时粒度 的粗细。粗粒度情况下，可以看作是静态动态多线程结合的实现。

```

//4:omp动态块划分（一次划分TASK_NUM个任务，粗粒度）
    init_2(); //初始化数组
    gettimeofday(&tpstart,NULL);
#pragma omp parallel for num_threads(THREAD_NUM)\
    schedule(dynamic,TASK_NUM)
    for(int i=0;i<ARR_NUM;i++){
        stable_sort(arr[i].begin(),arr[i].end());
    }
    gettimeofday(&tpend,NULL);
    cout<<"omp dynamic(TASK_NUM): "<<(1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-
tpstart.tv_usec)/1000<<" ms"<<endl;

```

4.2.5 omp使用guided调度

与动态分配任务类似，该实现方法也是线程动态领取任务。不同的是，分块开始大，随着迭代次数增加，分块越来越少。

```

//5:omp使用guided调度
    init_2(); //初始化数组
    gettimeofday(&tpstart,NULL);
#pragma omp parallel for num_threads(THREAD_NUM)\
    schedule(guided)
    for(int i=0;i<ARR_NUM;i++){
        stable_sort(arr[i].begin(),arr[i].end());
    }
    gettimeofday(&tpend,NULL);
    cout<<"omp guided dynamic: "<<(1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-
tpstart.tv_usec)/1000<<" ms"<<endl;
}

```

4.3 实验数据

4.3.1 数组均衡各算法比较

数组规模与线程数量等设置

```

const int ARR_NUM=10000;
const int ARR_LEN=10000;
const int THREAD_NUM=10;
const int TASK_NUM=50;

```

在该规模设计下的不同算法对均衡数组排序用时如下：

```

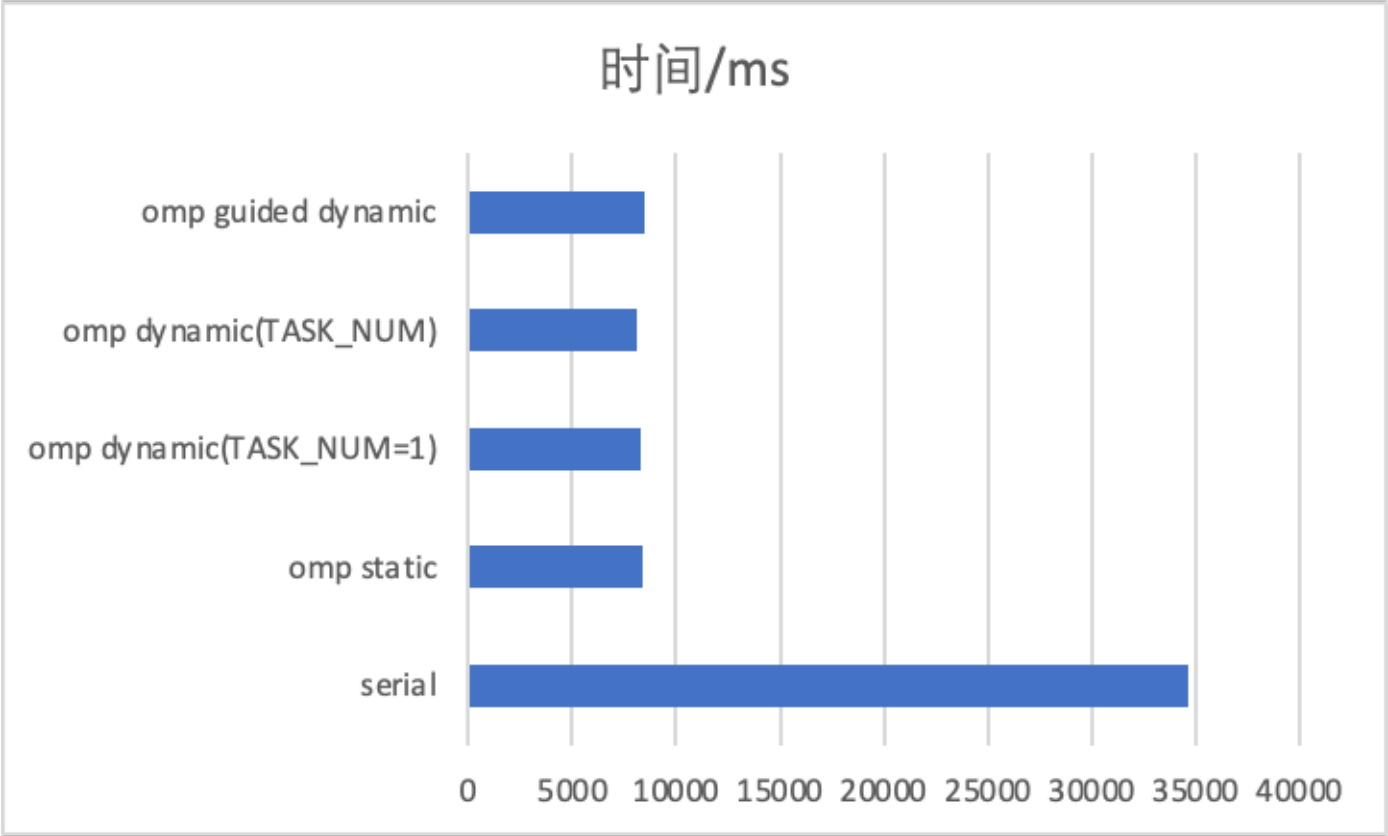
/Users/zhanghaha/Program/CLionProjects/pthread/cmake-build-debug/pthread
serial: 34663 ms
omp static: 8426 ms
omp dynamic(TASK_NUM=1): 8284 ms
omp dynamic(TASK_NUM): 8090 ms
omp guided dynamic: 8436 ms

Process finished with exit code 0

```

用excel进行图表统计：

算法	时间/ms
serial	34663
omp static	8426
omp dynamic(TASK_NUM=1)	8284
omp dynamic(TASK_NUM)	8090
omp guided dynamic	8436



结论：在数组完全随机、数组内数据分布均匀的情况下，几种循环调度方案的效率差别不大，且远远优于串行实现。

4.3.2 数组不均衡各算法比较

数组规模

```
const int ARR_NUM=5000;
const int ARR_LEN=1000;
const int TASK_NUM=50;
```

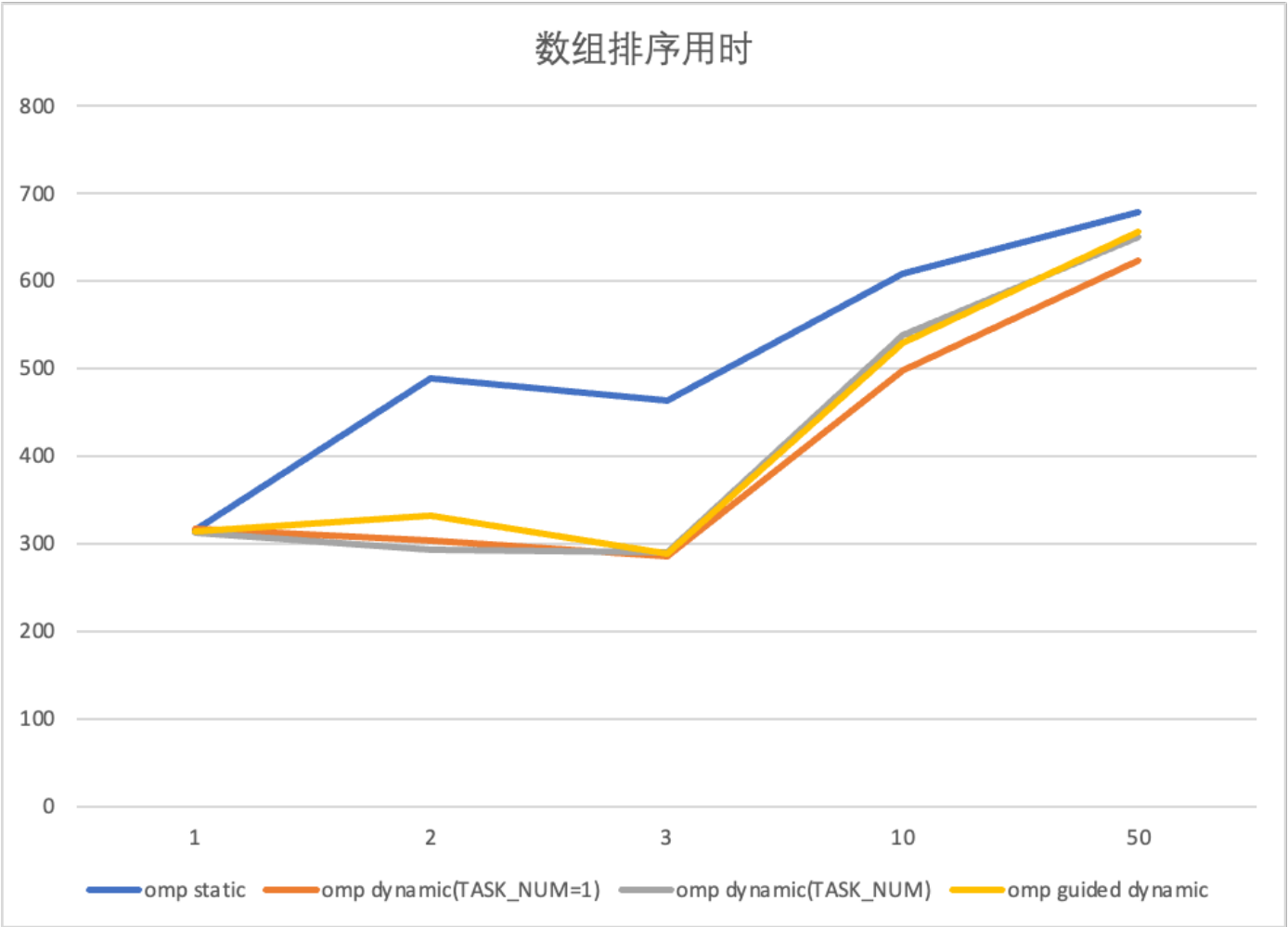
4.3.2.1 线程数多少

用时单位/ms

线程	1	2	3	10	50
omp static	315	488	463	608	678
omp dynamic(TASK_NUM=1)	317	303	285	498	623
omp dynamic(TASK_NUM)	313	293	290	538	650
omp guided dynamic	314	332	289	529	656

如图，在该数组规模下，观察发现：

线程数在2-3时，排序用时达到最低，随后随着线程增大，对于线程的维护成本变高，完全抵消了多线程带来的优势，反而用时更多。

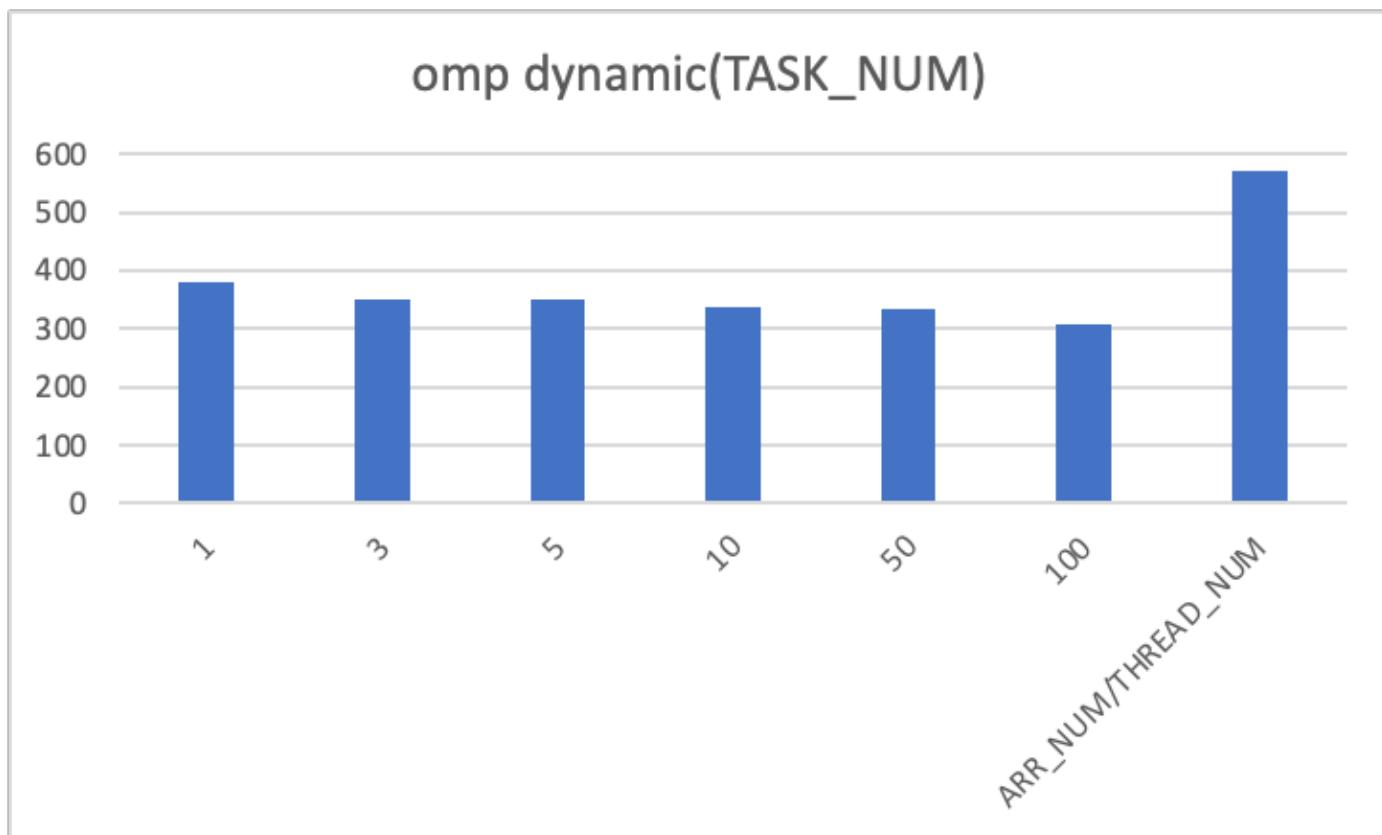


4.3.2.2 分块大小

数组规模：

```
const int ARR_NUM=5000;
const int ARR_LEN=1000;
const int THREAD_NUM=3;
```

TASK_NUM	1	3	5	10	50	100	ARR_NUM/THREAD_NUM
omp dynamic(TASK_NUM)	381	351	349	336	335	309	570



1. 粒度过细，可能会导致大量线程完成任务后等待再次分配任务的时间过长，效率较低；
2. 粒度过粗，会导致其更加接近静态分配，任务划分不均衡，效率较低。故应选择合适的分配粗细粒度。
3. 该实验中，当线程数为100时是效率最高的状态

5 附加题：高斯消去法（OpenMP）

5.1 实验内容

对于给定的方程组 $Ax=b$ ，使用 OpenMP 多线程编程，同时结合 SSE 向量计算，完成对于线性方程组的高斯消元和回代求解。

5.2 算法设计

5.2.1 高斯消元过程 OpenMP 结合 sse 实现

5.2.1.1 静态分配任务消元

```
void omp_sse_elimination_static(parameters* p){
    __m128 t1,t2,t3,t4;
    float tmp;
    for(int k=0;k<n-1;k++){
        int i,j;
# pragma omp parallel for num_threads(THREAD_NUM) \
        schedule(static) \
            default(none) shared(p,k,n) private(i,j,tmp,t1,t2,t3,t4)
        for(i=k+1;i<n+1;i++){
            tmp=p->b[i][k]/p->b[k][k];
            t3=_mm_load1_ps(&tmp);
            for(j=n+1-4;j>=k;j-=4){
                t1=_mm_loadu_ps(p->b[i]+j);
                t2=_mm_loadu_ps(p->b[k]+j);
                t3=_mm_mul_ps(t2,t3);
                t4=_mm_sub_ps(t1,t3);
                _mm_storeu_ps(p->b[i]+j,t4);
            }
            //解决不能被4整除的部分
            for (j=((n-k+1)%4)+k-1;j>= k;--j) {
                p->b[i][j] =p->b[i][j]-tmp*p->b[k][j];
            }
            p->b[i][k]=0;
        }
    }
}
```

5.2.1.2 动态分配任务消元

```
//动态分配任务消元
void omp_sse_elimination_dynamic(parameters* p,int task_size){
    __m128 t1,t2,t3,t4;
    float tmp;
    for(int k=0;k<n-1;k++){
        int i,j;
# pragma omp parallel for num_threads(THREAD_NUM) \
        schedule(dynamic,task_size) \
            default(none) shared(p,k,n,task_size) private(i,j,tmp,t1,t2,t3,t4)
        for(i=k+1;i<n+1;i++){
            tmp=p->b[i][k]/p->b[k][k];
```

```

    t3=_mm_load1_ps(&tmp);
    for(j=n+1-4;j>=k;j-=4){
        t1=_mm_loadu_ps(p->b[i]+j);
        t2=_mm_loadu_ps(p->b[k]+j);
        t3=_mm_mul_ps(t2,t3);
        t4=_mm_sub_ps(t1,t3);
        _mm_storeu_ps(p->b[i]+j,t4);
    }
    //解决不能被4整除的部分
    for (j=((n-k+1)%4)+k-1;j>= k;--j) {
        p->b[i][j] =p->b[i][j]-tmp*p->b[k][j];
    }
    p->b[i][k]=0;
}
}
}

```

5.2.1.3 guided分配任务消元

```

//guided分配任务消元
void omp_sse_elimination_guided(parameters* p){
    __m128 t1,t2,t3,t4;
    float tmp;
    for(int k=0;k<n-1;k++){
        int i,j;
        # pragma omp parallel for num_threads(THREAD_NUM) \
            schedule(guided) \
            default(none) shared(p,k,n) private(i,j,tmp,t1,t2,t3,t4)
        for(i=k+1;i<n+1;i++){
            tmp=p->b[i][k]/p->b[k][k];
            t3=_mm_load1_ps(&tmp);
            for(j=n+1-4;j>=k;j-=4){
                t1=_mm_loadu_ps(p->b[i]+j);
                t2=_mm_loadu_ps(p->b[k]+j);
                t3=_mm_mul_ps(t2,t3);
                t4=_mm_sub_ps(t1,t3);
                _mm_storeu_ps(p->b[i]+j,t4);
            }
            //解决不能被4整除的部分
            for (j=((n-k+1)%4)+k-1;j>= k;--j) {
                p->b[i][j] =p->b[i][j]-tmp*p->b[k][j];
            }
        }
    }
}

```

```

        p->b[i][k]=0;
    }
}
}

```

5.2.2 回代求解 OpenMP 结合 sse 实现

//回代求解

```

void omp_sse_back(parameters* p) {
    __m128 t1, t2, sumSSE;
    float sum;
    //先处理后四个解, 以使sse并行计算能够启动
    for(int i=n-1;i>=0;i--){
        if((n-1-i)==4)break;
        float sum = 0;
        for(int j=n-1;j>i;j--){
            sum+=p->b[i][j]*p->x[j];
        }
        p->x[i]=(p->b[i][n]-sum)/p->b[i][i];
    }
    //OpenMP结合SSE计算
    for(int i=n-5;i>=0;i--){
        sumSSE=_mm_setzero_ps();
        sum=0;
        int forSerial=(n-i-1)%4;
        int j;
        # pragma omp parallel for num_threads(THREAD_NUM) \
            schedule(dynamic,TASK_NUM) \
            default(none) shared(p,i,n,sumSSE,forSerial) private(j,t1,t2)
        for(j=i+forSerial+1;j<=n-4;j+=4){
            t1=_mm_loadu_ps(p->b[i]+j);
            t2=_mm_loadu_ps(p->x+j);
            t1=_mm_mul_ps(t1,t2);
            sumSSE=_mm_add_ps(sumSSE,t1);
        }
        sumSSE=_mm_hadd_ps(sumSSE,sumSSE);
        sumSSE=_mm_hadd_ps(sumSSE,sumSSE);
        _mm_store_ss(&sum,sumSSE);
        //解决不能被4整除的部分
        for(int j=i+1;j<=i+forSerial;j++) {
            sum+=p->b[i][j]*p->x[j];
        }
        p->x[i]=(p->b[i][n]-sum)/p->b[i][i];
    }
}

```

5.3 实验结论

结合了sse的OpenMP充分利用 cache 缓存的特性，提高了程序的并行效率。

6 实验问题

对于MacOS上的omp头文件，需要做如下的makefile配置，才能完成该实验：

```
cmake_minimum_required(VERSION 3.23)
project(pthread)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread")

#set(CMAKE_C_COMPILER "/usr/bin/gcc") #这里写你的gcc路径
#set(CMAKE_CXX_COMPILER "/usr/bin/g++") #这里写你的g++路径
set(OPENMP_LIBRARIES "/opt/homebrew/Cellar/libomp/15.0.4/lib") #这里写你的libomp路径，通过
brew安装的默认地址
set(OPENMP_INCLUDES "/opt/homebrew/Cellar/libomp/15.0.4/include")#这里写你的libomp路径

OPTION (USE_OpenMP "Use OpenMP to enable <omp.h>" ON)

# Find OpenMP
if(APPLE AND USE_OpenMP)
    if(CMAKE_C_COMPILER_ID MATCHES "Clang")
        set(OpenMP_C "${CMAKE_C_COMPILER}")
        set(OpenMP_C_FLAGS "-Xpreprocessor -fopenmp -lomp -Wno-unused-command-line-
argument")
        #注意以上需要增加-Xpreprocessor 以及不能直接-llibomp 在这里不需要前缀lib只需要-lomp即
        可，下面相似的地方也是同个道理
        set(OpenMP_C_LIB_NAMES "libomp" "libgomp" "libiomp5")
        set(OpenMP_libomp_LIBRARY ${OpenMP_C_LIB_NAMES})
        set(OpenMP_libgomp_LIBRARY ${OpenMP_C_LIB_NAMES})
        set(OpenMP_libiomp5_LIBRARY ${OpenMP_C_LIB_NAMES})
    endif()
    if(CMAKE_CXX_COMPILER_ID MATCHES "Clang")
        set(OpenMP_CXX "${CMAKE_CXX_COMPILER}")
        set(OpenMP_CXX_FLAGS "-Xpreprocessor -fopenmp -lomp -Wno-unused-command-line-
argument")
        set(OpenMP_CXX_LIB_NAMES "libomp" "libgomp" "libiomp5")
        set(OpenMP_libomp_LIBRARY ${OpenMP_CXX_LIB_NAMES})
        set(OpenMP_libgomp_LIBRARY ${OpenMP_CXX_LIB_NAMES})
        set(OpenMP_libiomp5_LIBRARY ${OpenMP_CXX_LIB_NAMES})
    endif()
endif()
```

```
        endif()
    endif()

    if(USE_OpenMP)
        find_package(OpenMP REQUIRED)
    endif(USE_OpenMP)

    if (OPENMP_FOUND)
        include_directories("${OPENMP_INCLUDES}")
        link_directories("${OPENMP_LIBRARIES}")
        set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
        set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
    endif(OPENMP_FOUND)

    add_executable(pthread Array.cpp)
```