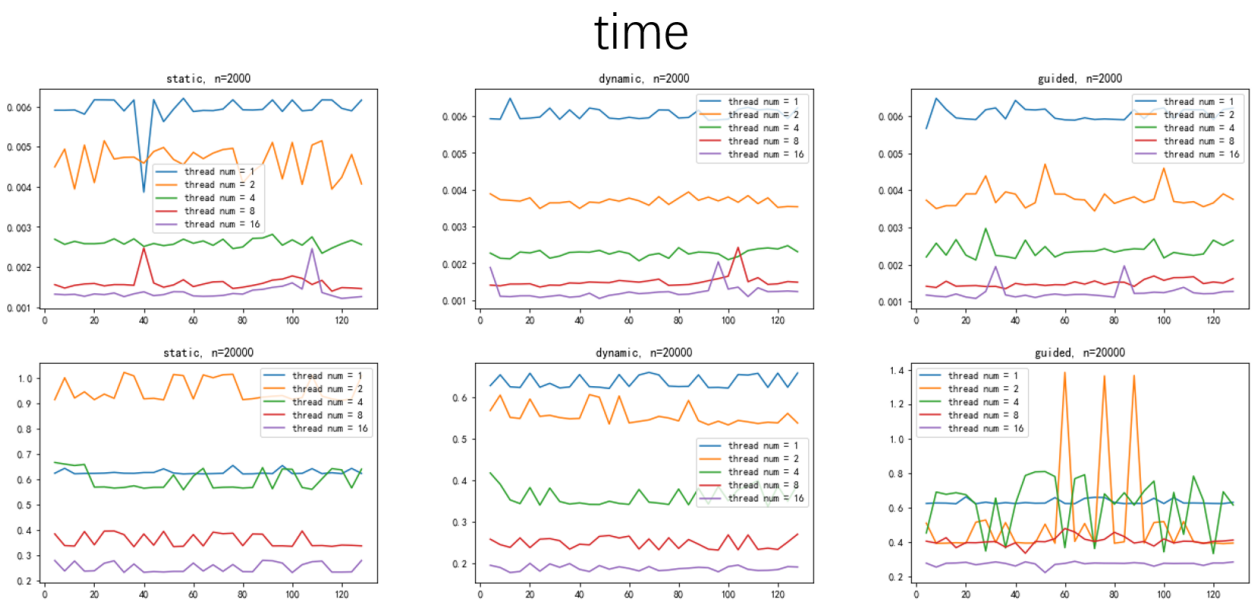
1# Programming Assignment

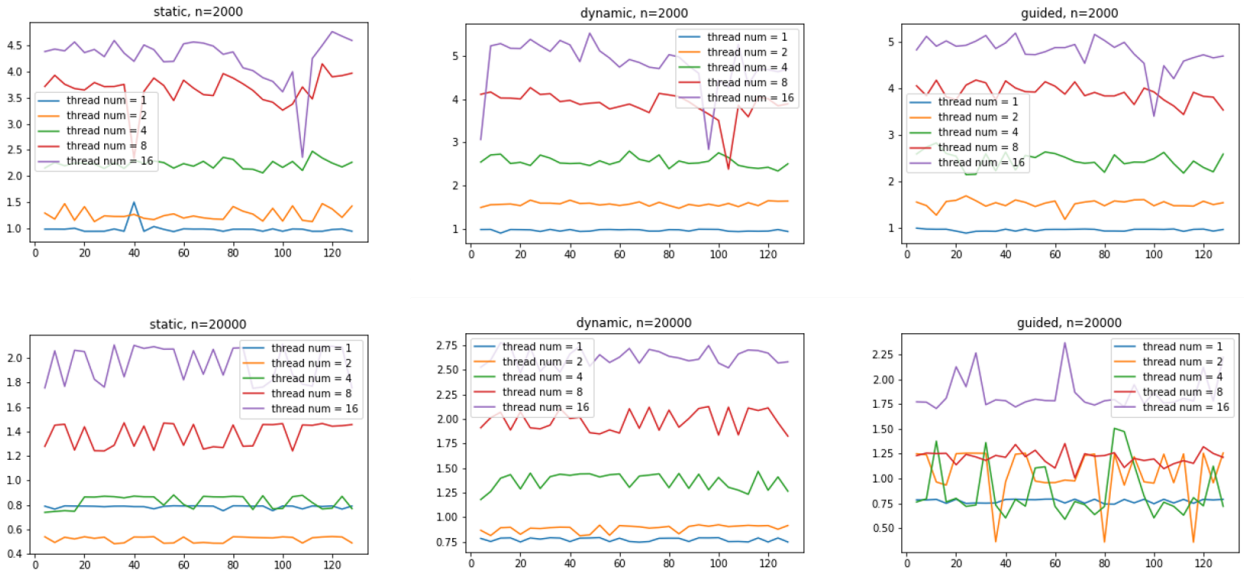
通过第三次作业中的测试，我发现使用MPI的情况下，进程数越多程序越慢，原因是通信开销过大。因此，在此次作业中，我选择纯openmp编程。正确性验证方法是在最后计算二范数。

```
void parallel(int n, double** matrix, double* vector, double** result) {
    #pragma omp parallel num_threads(thread_count)
    {
        int interval = (n-1)/thread_count+1;
        #pragma omp for
        for (int i=0; i<n; i+=interval) {
            memset((*result)+i, 0, sizeof(int) * std::min(n-i, interval));
        }

        #pragma omp for schedule(static, chunk)
        for (int i = 0; i < n; i++) {
            double temp = 0.0;
            for (int j = 0; j < n; j++) {
                temp += matrix[i][j] * vector[j];
            }
            (*result)[i] += temp;
        }

    }
}
```

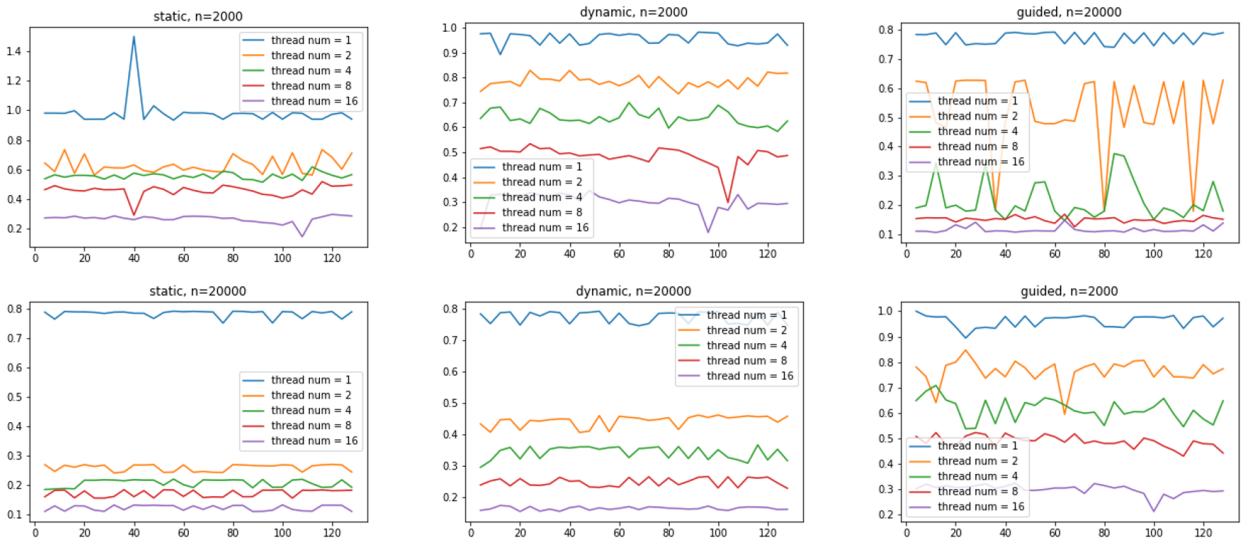
openmp编程可以很方便的选择各种调度策略与块大小，因此我测试了static，dynamic，guided三种调度方法在块大小是2-128的时候的运行速度。在n=2000时，三种调度策略差异不大，在n=20000时，dynamic效率较优，因而在提交代码中我选择dynamic，chunksize=48。



Speed up



效率

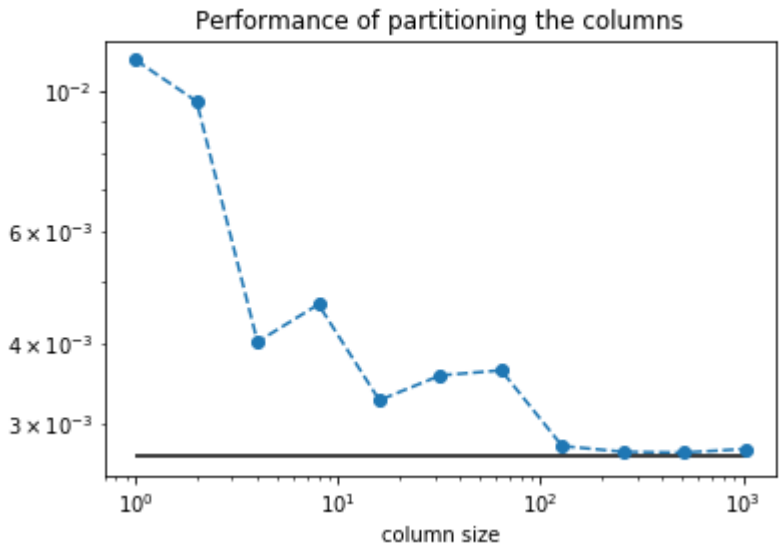


借鉴hw3中的矩阵分块思想，我也尝试了分块进行矩阵乘法，方法是首先将矩阵按列切成若干块，再在每一块内运行并行的矩阵向量乘，代码如下

```
void parallel(int n, double** matrix, double* vector, double** result) {
    #pragma omp parallel num_threads(thread_count)
    {
        int interval = (n-1)/thread_count+1;
        #pragma omp for
        for (int i=0; i<n; i+=interval) {
            memset((*result)+i, 0, sizeof(int) * std::min(n-i, interval));
        }

        for (int l = 0; l < n; l += chunk) {
            int r = min(l+chunk, n);
            #pragma omp for
            for (int i = 0; i < n; i++) {
```

```
                double temp = 0.0;
                for (int j = l; j < r; j++) {
                    temp += matrix[i][j] * vector[j];
                }
                (*result)[i] += temp;
            }
        }

    }
}
```

从理论上将，这么做会将对向量的访问限定在一个很小的范围内，从而获得性能的提升。然而，如下图对n=2000，4个线程的测试，其运行时间随列数的增加而减小，并不断接近于不进行这个优化的结果，因此判断这是一个负优化。



并行计算二范数也使用纯openmp。由于实数乘法很慢，我选择仅在并行结果和串行结果相差较大(绝对值之差>1e-10)的情况下将其计入二范数。然而，这一段代码是O(n)的，与O(n^2)的矩阵乘法相比时间可以忽略不计，故并行化这段代码并不会对总体效率带来明显提升。

```
#pragma omp parallel for num_threads(thread_count)
    for (int i=0; i<n; i++)
        if (mabs(ans_parallel[i] - ans_serial[i]) > eps) {
            printf("warning! not equal\n");
            L2 += (ans_parallel[i]-ans_serial[i]) * (ans_parallel[i]-ans_serial[i]);
        }
```

2# Programming Assignment 4.5

实现了一个元素按非降顺序排序的链表，采用读写锁进行控制。我查看了数据规模较小时的运行结果，发现没有问题。

- 链表部分

封装成一个链表类，在构造函数里对读写锁进行初始化，在析构函数里销毁读写锁。插入、删除用写锁，查询、输出用读锁。

为保证输出操作输出在一行上，首先将链表打印到一个字符串上，再将字符串输出。（这个类仅返回待输出字符串，不实际进行输出）

```
class ThreadSafeLinkedList {
public:
    ThreadSafeLinkedList() : head(NULL, INT_MIN) {
        pthread_rwlock_init(&rwlock, NULL);
    }
    ~ThreadSafeLinkedList() {
        pthread_rwlock_destroy(&rwlock);
    }

    void insert(int value) {
        pthread_rwlock_wrlock(&rwlock);
        Node *cur = &head;
        while (cur->next != NULL && cur->next->value < value) {
            cur = cur->next;
        }
        Node* newNode = new Node(cur->next, value);
        cur->next = newNode;
        pthread_rwlock_unlock(&rwlock);
    }

    bool erase(int value) {
        pthread_rwlock_wrlock(&rwlock);
        Node *pre = &head, *cur = pre->next;
        while (cur != NULL && cur->value < value) {
            pre = pre->next;
            cur = cur->next;
        }
        bool ret;
        if (cur != NULL && cur->value == value) {
            pre->next = cur->next;
            ret = 1;
        } else {
            ret = 0;
        }
        pthread_rwlock_unlock(&rwlock);
        return ret;
    }

    bool exist(int value) {
        pthread_rwlock_rdlock(&rwlock);
        Node* cur = &head;
        bool ret;
        while (cur != NULL && cur->value < value) {
            cur = cur->next;
        }
        ret = cur != NULL && cur->value == value;
        pthread_rwlock_unlock(&rwlock);
```

```
        return ret;
    }
    std::string toString () {
        pthread_rwlock_rdlock(&rwlock);
        std::string out = "head";
        for (ThreadSafeLinkedList::Node* cur = head.next; cur != NULL; cur =
cur->next) {
            out = out + "->" + std::to_string(cur->value);
        }
        pthread_rwlock_unlock(&rwlock);
        return out;
    }

private:
    struct Node {
        Node* next;
        int value;
        Node(Node* next_, int val) : next(next_), value(val) {}
    };
    pthread_rwlock_t rwlock;
    Node head;
} linkedList;
```

- 调度部分

主线程将生成的任务加入任务队列，其他线程从任务队列中取任务。队列操作需要加锁，用 `generateEnd` 变量指示主线程是否已经产生完所有任务。若队列为空，但没有 `generateEnd`，则使用条件变量等待；若队列为空，已经 `generateEnd`，则退出。主线程需要在设置 `generateEnd` 之后进行条件变量的广播。

为了保证线程的确是按照主线程生成任务的顺序执行，还需要使用一个条件变量来确认生成在这个操作之前的修改操作确实执行完毕。为了简化代码，我选择要求在某操作的所有操作都执行完之后，再执行这个操作。由于此题中读写操作的数量级相近，这么简化并不会明显拖慢程序速度。

为保证顺利测试到唤醒功能，主线程每产生20个任务进行1秒的sleep

```
std::queue<Task> tasks;

bool getTask(Task* task, int my_rank) {
    bool ret;
    pthread_mutex_lock(&mtx_tasks);
    while (tasks.empty() && !generateEnd) {
        while (pthread_cond_wait(&cv_wait, &mtx_tasks) != 0);
        if (generateEnd) {
            ret = 0;
            goto END;
        }
    }
    if (!tasks.empty()) {
        *task = tasks.front();
        tasks.pop();
        ret = 1;
    } else {
        ret = 0;
    }
```

```
20
21  END:
22      pthread_mutex_unlock(&mtx_tasks);
23      return ret;
24  }
25
26  void* work(void* rank) {
27      void* work(void* rank) {
28      long my_rank = (long) rank;
29      Task task;
30      while (getTask(&task, my_rank)) {
31          pthread_mutex_lock(&mtx_serial);
32          while (1) {
33              if (todo == task.id)
34                 break;
35              while (pthread_cond_wait(&cv_serial, &mtx_serial) != 0);
36          }
37          pthread_mutex_unlock(&mtx_serial);
38          // solve the task
39          pthread_mutex_lock(&mtx_serial);
40          todo++;
41          pthread_mutex_unlock(&mtx_serial);
42          pthread_cond_broadcast(&cv_serial);
43      }
44  }
45  }
46
47  int main() {
48      // ...
49      for (int i = 0; i < n; i++) {
50          pthread_mutex_lock(&mtx_tasks);
51          tasks.push(Task::randomGenerate());
52          pthread_mutex_unlock(&mtx_tasks);
53          pthread_cond_signal(&cv_wait);
54          if (i % 20 == 19)
55              sleep(1);
56      }
57      generateEnd = 1;
58      pthread_cond_broadcast(&cv_wait);
59      // ...
60  }
```

程序运行截图

```
[2]: 8 is in the linked list
[13]: head->0->1->1->2->2->2->3->3->3->3->3->3->5->6->6->7->8->8->8->8->8->8->8->8->
>9->9->9->9->9->9
[20]: head->0->1->1->2->2->2->3->3->3->3->3->3->5->6->6->7->8->8->8->8->8->8->8->8->
>9->9->9->9->9->9
[21]: Successfully delete 8
[1]: Successfully insert 8
[4]: Successfully insert 4
[22]: Successfully delete 5
[12]: Successfully insert 6
[7]: 3 is in the linked list
[17]: Successfully insert 7
[0]: head->0->1->1->2->2->2->3->3->3->3->3->3->4->6->6->6->7->7->8->8->8->8->8->8->
8->8->8->9->9->9->9->9
[23]: 6 is in the linked list
[10]: Successfully insert 2
[15]: 1 is in the linked list
[14]: 8 is in the linked list
[19]: Successfully delete 8
[18]: 6 is in the linked list
[9]: head->0->1->1->2->2->2->2->3->3->3->3->3->3->4->6->6->6->7->7->8->8->8->8->8->
8->8->8->9->9->9->9->9
[3]: Successfully delete 1
[17]: Successfully insert 4
[14]: Successfully insert 3
[21]: 1 is in the linked list
[13]: Successfully insert 0
[4]: head->0->0->1->2->2->2->2->3->3->3->3->3->3->3->4->4->6->6->6->6->7->7->8->8->8->
8->8->8->8->8->9->9->9->9->9
[22]: Successfully insert 0
[6]: Successfully insert 0
[12]: Successfully delete 9
[19]: 9 is in the linked list
[20]: head->0->0->0->0->1->2->2->2->2->2->3->3->3->3->3->3->3->4->4->6->6->6->6->7->7->8->
>8->8->8->8->8->8->8->8->9->9->9->9->9
[11]: 4 is in the linked list
[16]: Delete 5 fail
[2]: Successfully delete 2
[18]: Successfully insert 6
[8]: head->0->0->0->0->0->1->2->2->2->2->3->3->3->3->3->3->3->3->4->4->6->6->6->6->6->7->7->8->
8->8->8->8->8->8->8->8->9->9->9->9->9
[0]: head->0->0->0->0->0->1->2->2->2->2->3->3->3->3->3->3->3->3->4->4->6->6->6->6->6->7->7->8->8->8->8->8->8->8->8->9->9->9->9->9
[1]: head->0->0->0->0->0->1->2->2->2->2->3->3->3->3->3->3->3->3->4->4->6->6->6->6->6->7->7->8->8->8->8->8->8->8->8->9->9->9->9->9
[9]: Successfully insert 4
[11]: head->0->0->0->0->0->1->2->2->2->2->3->3->3->3->3->3->3->3->4->4->4->6->6->6->6->6->7->7->8->8->8->8->8->8->8->8->9->9->9->9->9
[15]: Successfully insert 3
[4]: head->0->0->0->0->0->1->2->2->2->2->3->3->3->3->3->3->3->3->3->4->4->4->6->6->6->6->6->7->7->8->8->8->8->8->8->8->8->9->9->9->9->9
[5]: head->0->0->0->0->0->1->2->2->2->2->3->3->3->3->3->3->3->3->3->4->4->4->6->6->6->6->6->7->7->8->8->8->8->8->8->8->8->9->9->9->9->9
```

3# Programming Assignment 5.3

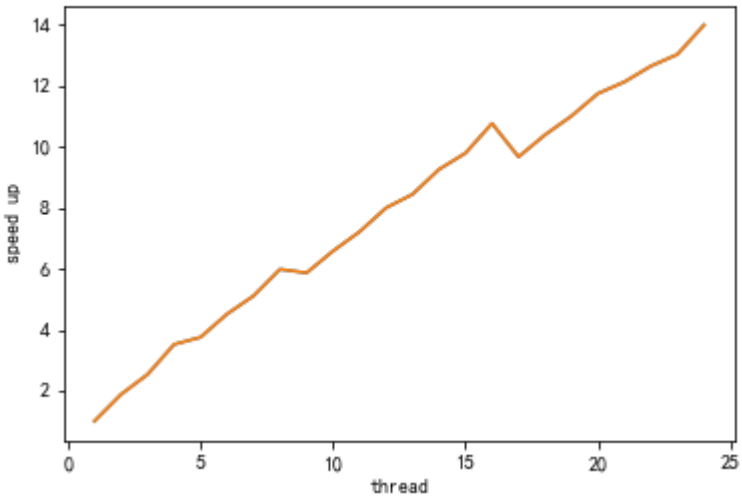
1. `i`, `j`, `count` 私有， `a`, `n`, `temp` 共享
2. 不存在循环依赖，对 `a` 和 `n` 是只读的，对 `temp` 是只写的，且保证不同线程不会写 `temp` 的同一位置
3. 不能并行，但可以修改代码进行并行。 由于 `memcpy` 比循环赋值要快很多，我采用了手动划分每个线程的任务，每个线程内部执行 `memcpy` 的方法。这对赋值阶段的性能优化十分明显，但由于程序的热点在排序阶段，此优化对总时间的影响不大。

```
        int interval = (n-1)/thread_count+1;
        // ......
#       pragma omp for
        for (i=0; i<n; i+=interval) {
            memcpy(a+i, temp+i, sizeof(int) * std::min(n-i, interval));
        }
```

4. 我使用了如下方法，将排序优化到$O(nlogn/p + p * log^2 n)$，其中n表示待排序数组中元素个数，p表示线程数。在n=1e8，p=24时可获得约14×的加速比。将排序与sort得到的结果进行对比，发现结果是正确的。
    1. 将整个序列切成p段，每段执行快速排序。
    2. 用归并排序的思想，每次合并相邻的段。合并的时候使用计数排序，即在合并后的序列中的排名=在本序列中的排名+在邻居序列中的排名。由于序列已经有序，在本序列中的排名可以通过位置直接得出，在邻居序列中的排名是不降的，因此获得这个排名是均摊O(1)的。若要并行化这一过程，可以让一个线程处理连续的一段，该线程首先通过二分查找，定位这一段的第一个元素（即最小元素）在邻居中的位置，剩下的元素在邻居中的位置可以均摊O(1)的计算出来。

下图是测量n=1e8时，与快速排序相比的加速比。可以看出，我的程序的扩展性较好。在8个线程、16个线程的位置发生了较为明显的抖动，是因为9个线程、17个线程较8个线程、16个线程要多进行一轮迭代，故按我的实现，应尽量选用2^n个线程。

在n足够大的情况下，快速排序快于计数排序，故我的实现也快于计数排序。




```
void count_sort(int n, int* cur, timeval* mid1, timeval* mid2) {
    int *pre = new int[n];
    int my_rank;
    bool needCpy = 0;
    int *left = new int[thread_count + 1];
#   pragma omp parallel num_threads(thread_count) \
    default(none) private(my_rank) shared(pre, cur, n, mid1, mid2, left,
needCpy, thread_count)
    {
#       pragma omp for
        for (my_rank = 0; my_rank < thread_count; my_rank++) {
            int l, r;
            getInterval(&l, &r, n, thread_count, my_rank);
            left[my_rank] = l;
            sort(cur+l, cur+r);
        }
#       pragma omp single
        left[thread_count] = n;
        for (int pw=0; (1<<pw) < thread_count; pw++) {
#           pragma omp single
            { swap(pre, cur); needCpy ^= 1; }

#           pragma omp for
            for (my_rank = 0; my_rank < thread_count; my_rank++) {
                int my_group = my_rank >> pw;
                int nei_group = my_group ^ 1;
                if ((nei_group<<pw) >= thread_count) { //最后一个，而且落单
                    int my_left = left[my_group << pw];
                    memcpy(cur+my_left, pre+my_left, sizeof(int)*(n-my_left));
                } else {
                    int my_left = left[my_group << pw];
```

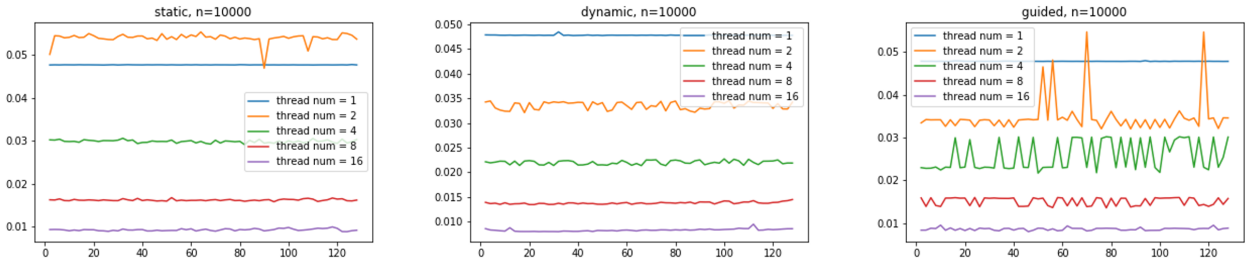
```
                    int nei_left = left[nei_group << pw];
                    int nei_right = left[min((nei_group+1) << pw,
thread_count)];
                    if (my_group & 1) { //组内后面的，要计入前面相等的
                        int pos = upper_bound(pre + nei_left, pre + nei_right,
pre[left[my_rank]]) - pre;
                        int rk = pos + left[my_rank] - my_left;
                        cur[rk] = pre[left[my_rank]];
                        for (int i = left[my_rank] + 1; i < left[my_rank+1];
i++) {
                            rk++; //不小于前一个数
                            while (pre[pos] <= pre[i] && pos < nei_right)
                                pos++, rk++;
                            cur[rk] = pre[i];
                        }
                    } else { // 组内前面的，不计入后面相等的
                        int pos = lower_bound(pre + nei_left, pre + nei_right,
pre[left[my_rank]]) - pre;
                        int rk = pos - nei_left + left[my_rank];
                        cur[rk] = pre[left[my_rank]];
                        for (int i = left[my_rank] + 1; i < left[my_rank+1];
i++) {
                            rk++; //不小于前一个数
                            while (pre[pos] < pre[i] && pos < nei_right)
                                pos++, rk++;
                            cur[rk] = pre[i];
                        }
                    }
                }
            }
        if (needCpy) {
#           pragma omp for
            for (my_rank=0; my_rank<thread_count; my_rank++) {
                memcpy(pre+left[my_rank], cur+left[my_rank], sizeof(int)*
(left[my_rank+1] - left[my_rank]));
            }
#           pragma omp single
            {
                swap(pre, cur);
            }
        }

    }
    delete[] pre;
    delete[] left;
}
```

4# Programming

1&2. 我手动检查了数据规模较小时的计算结果，发现结果正确。运行时间如下，发现在线程数较小时，动态调度策略速度稍快，其中dynamic较guided较为稳定。我多次运行程序，2个线程的static均慢于串行。综上，在提交的代码中选取dynamic。dynamic对块大小不敏感，因此我任选了一个48。

关键代码如下

```
#   pragma omp parallel for num_threads(thread_count) \
        default(none) private(i, j) shared(s, A, b, n) schedule(dynamic, 48)
    for(i = 0; i < n; i++) {
        s[i] = 0.0;
        for(j = 0; j < b[i]; j++) {
            s[i] += A[i][j];
        }
        s[i] = s[i] * 1.0 / b[i];
    }
```



3. 为使用相同的b进行测试，在gen中不应使用`srand(time(0));`

   pthread主要代码如下

```
void* run_default(void* rank) {
    long my_rank = (long) rank;
    int l, r, i;
    if (my_rank <= n % thread_count) {
        int upper = (n-1)/thread_count+1;
        l = my_rank * upper;
        r = (my_rank + 1) * upper;
    } else {
        int lower = n/thread_count;
        l = my_rank * lower + n % thread_count;
        r = (my_rank + 1) * lower + n % thread_count;
    }
    solve(l, r);
}

void* run_static(void* rank) {
    long my_rank = (long) rank;
    int der = thread_count * block_size;
    int i;
    for (i=block_size * my_rank; i<n; i+=der) {
        solve(i, min(i+block_size ,n));
    }
}
```

```
24
25  void dynamic_allocate(int *l, int *r) {
26      pthread_mutex_lock(&mtx_schedule);
27      if (remain >= n) {
28          *l = -1;
29      } else {
30          *l = remain;
31          remain += block_size;
32          *r = min(remain, n);
33      }
34      pthread_mutex_unlock(&mtx_schedule);
35  }
36
37  void* run_dynamic(void* rank) {
38      int l, r;
39      while (1) {
40          dynamic_allocate(&l, &r);
41          if (l == -1)
42              break;
43          solve(l, r);
44      }
45  }
46
47  void guided_allocate(int *l, int *r) {
48      pthread_mutex_lock(&mtx_schedule);
49      if (remain >= n) {
50          *l = -1;
51      } else {
52          *l = remain;
53          int alloc = max((n-remain)/thread_count, block_size);
54          remain += alloc;
55          *r = min(remain, n);
56      }
57      pthread_mutex_unlock(&mtx_schedule);
58  }
59
60  void* run_guided(void* rank) {
61      int l, r;
62      while (1) {
63          guided_allocate(&l, &r);
64          if (l == -1)
65              break;
66          solve(l, r);
67      }
68  }
```

提交文件路径

每个文件夹下都有写好的make，可直接make编译

题号	文件夹	可执行文件	源代码
1	/home/2017011307/hw5/1-openmp	/home/2017011307/hw5/1-openmp/target	/home/2017011307/hw5/1-openmp/main.cpp
2	/home/2017011307/hw5/2	/home/2017011307/hw5/2/target	/home/2017011307/hw5/2/2-4-5.cpp
3	/home/2017011307/hw5/3-opt	/home/2017011307/hw5/3-opt/target	/home/2017011307/hw5/3-opt/main.cpp
4(openmp)	/home/2017011307/hw5/4-openmp	/home/2017011307/hw5/4-openmp/target	/home/2017011307/hw5/4-openmp/main.c
4(pthread)	/home/2017011307/hw5/4-pthread	/home/2017011307/hw5/4-pthread/target	/home/2017011307/hw5/4-pthread/main.c