实验三：多线程协作实验

17341046 郭梓煜

实验目的

- pthread线程间的同步函数的实现
  - 忙等待
  - 阻塞等待
- 多线程矩阵分块乘法
  - 根据线程数分块
  - 对分块进行先乘后加的计算
    - 加法分两种: 单线程直接加，多线程树形加法
    - 在考虑线程时使用前面的同步函数并比较效率
- 列出给出不同规模的加速比和效率
  - 两种同步方法* 两种不同加法 *两个表= 8个表

实验环境

- 图形化管理文件: winscp
- 编辑器: vscode
- 语言: c++
- 打开命令行界面: putty
- 编译器: g++
- 集群: 222.200.180.115

实验代码实现与分析

实现**pthread**线程同步函数

/这里暂时不展示在后面求矩阵时的同步。展示简单的线程同步以展示方法。/

1、 阻塞等待

- 即使用信号量实现barrier
- sem_t 创建信号量 sem_init 初始化信号量
- sem_post 、sem_wait即操作系统中的P,V操作。
- 通过对count_sem、barrier_sem两个信号量的值的改变，实现将多个线程进行同步。
  关键代码如下(详见sync_1.cpp):

```
...
#define NUM_THREADS 6
int counter = 0;
```

```
sem_t count_sem;
sem_t barrier_sem;
void *Thread_work(void *rank)
{
    long my_rank = (long)rank;
    printf("Before barrier: %ld\n", my_rank);
    /*Barrier*/
    sem_wait(&count_sem);
    if (counter == NUM_THREADS - 1)
    {
        counter = 0;
        sem_post(&count_sem);
        for (int j = 0; j < NUM_THREADS - 1; j++)
            sem_post(&barrier_sem);
    }
    else
    {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    printf("After barrier: %ld\n", my_rank);
}
int main()
{
    sem_init(&count_sem, 0, 1);
    sem_init(&barrier_sem, 0, 0);
    ...
    /* Start threads */
    for (thread = 0; thread < NUM_THREADS; thread++)
        pthread_create(&thread_handles[thread], NULL, Thread_work, (void
*)thread);

    /* Wait for threads to complete */
    for (thread = 0; thread < NUM_THREADS; thread++)
        pthread_join(thread_handles[thread], NULL);
}
```

运行结果如下:

```
设置了barrier时                    注释掉barrier后
Before barrier: 1                  Before barrier: 0
Before barrier: 0                  After barrier: 0
Before barrier: 2                  Before barrier: 1
Before barrier: 3                  After barrier: 1
Before barrier: 4                  Before barrier: 2
Before barrier: 5                  Before barrier: 4
After barrier: 1                   After barrier: 4
After barrier: 5                   After barrier: 2
After barrier: 2                   Before barrier: 3
After barrier: 3                   After barrier: 3
```

```
After barrier: 4                    Before barrier: 5
After barrier: 0                    After barrier: 5
```

可以看到设置了barrier后，只有当所有线程到达barrier时，才会一起出来，即实现了线程的同步。

2、 忙等待

- pthread_mutex_t 创建互斥量
- pthread_mutex_lock()、pthread_mutex_unlock() 互斥锁类似于操统中的临界区
- 通过使用counter记录线程数，在线程未全部到达时，使用循环无限等待，直至所有线程到达，来实现将多个线程进行同步。
  关键代码如下(详见sync_2.cpp):

```
/主函数与上一点无异，此处省去/
...
#define NUM_THREADS 6
int counter = 0;
int thread_count;
pthread_mutex_t barrier_mutex;

void *Thread_work(void *rank)
{
    long my_rank = (long)rank;
    printf("Before barrier: %ld\n", my_rank);
    /*Barrier*/
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < NUM_THREADS)
        ;
    printf("After barrier: %ld\n", my_rank);
}
...
```

实现之后的运行结果与上一点一致，这里便不再多展示。

- 对两种方法实现barrier的效率见下一点的报告中。

多线程矩阵分块乘法

- 由于有两种实现barrier的方法，两种不同的加法，所以会有四份代码。

1. 统一部分

- 矩阵存储及空间申请
  - 使用一个二维矩阵存放结果，三维矩阵存放中间值，中间值即加法产生的n* n*n个数。
  - 使用vector表示这些矩阵，申请空间并初始化为0
    代码如下：

```
vector<vector<double> > res; //存放结果
vector<vector<vector<double> > > cube; //存放中间值

void malloc_for_multi()
{
    //malloc_for_cube
    cube.resize(n);
    for (int i = 0; i < n; i++)
        cube[i].resize(n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cube[i][j].resize(n, 0);

    //malloc_for_result_matrix
    res.resize(n);
    for (int i = 0; i < n; ++i)
        res[i].resize(n, 0);
}
```

- 线程数及矩阵大小的读入

```
int main(int arg_1, char **arg_2)
{
    num_threads = atoi(arg_2[1]), n = pow(2, atoi(arg_2[2]));

    cout << "num_threads: " << num_threads << "\nsize: " << n << "\n";
    ...
}
```

- 串行实现的矩阵相乘
  - 即实现一个n的三重循环，累加至一个二维结果矩阵即可。
    代码如下：

```
void serial_work()
{
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                res[i][j] += (i - 0.1 * k + 1) / (i + k + 1) *
                    ((j - 0.2 * k + 1) * (k + j + 1) / (k * k + j * j + 1));
}
```

- 分块方法以及分块消息记录
  - 按照给定线程数的多少，根据老师的指示，使用switch，case得到对A,B矩阵的行列分别是i_size，k_size，k_size，j_size的分块块数。

- 创建一个Block结构体，用于create线程时传递处理的块的i，k，j信息，即在原三维中间值矩阵的位置信息。
- 在线程函数中，只需得到i，k，j再加上偏移量即可得到操作数的具体位置。类似于数据在内存中的访问方式。
  代码如下：

```
int i_size, k_size, j_size;
...
switch (num_threads)
{
case 1:
    i_size = k_size = j_size = 1; break;
case 2:
    i_size = k_size = 1; j_size = 2; break;
case 4:
    i_size = 1; k_size = j_size = 2; break;
case 8:
    i_size = k_size = j_size = 2; break;
case 16:
    i_size = k_size = 2; j_size = 4; break;
case 32:
    i_size = 2; k_size = j_size = 4; break;
case 56:
    i_size = 2; k_size = 4; j_size = 7; break;
default:
    break;
}
...
struct Block
{
    Block(int i = 0, int k = 0, int j = 0) : i(i), k(k), j(j) {}
    int i, k, j;
};
```

- 创建线程
  - 由于对两个矩阵进行分块，即对三维的i，k，j创建线程，所以使用三维表示线程号。
    代码如下：

```
pthread_t threads[i_size][k_size][j_size];
...
for (int i = 0; i < i_size; ++i)
    for (int k = 0; k < k_size; ++k)
        for (int j = 0; j < j_size; ++j)
        {
            Block node = Block(i, k, j);
            pthread_create(&threads[i][k][j], NULL, parallel, (void
*)&node);
        }
for (int i = 0; i < i_size; ++i)
```

```
    for (int k = 0; k < k_size; ++k)
        for (int j = 0; j < j_size; ++j)
            pthread_join(threads[i][k][j], NULL);
```

- 计算程序时间
  - 利用omp.h库函数omp_get_wtime()来获取运行所需的时间。
    代码如下：

```
double start=omp_get_wtime();
... //所测程序
double finish=omp_get_wtime();
double elapsed = finish - start;
```

2. 忙等待&单线程完成分块加法

- 忙等待主要体现并实现于线程函数中，使用counter计数并循环等待，而且互斥地访问counter。
- 使用pthread_mutex_lock等函数实现 代码如下：

```
void *parallel(void *node)
{
    Block *t = (Block *)node;
    int a_row = t->i * a_row_size;
    int a_col = t->k * a_col_size;
    int b_col = t->j * b_col_size;
    for (int x = 0; x < a_row_size; ++x)
        for (int y = 0; y < a_col_size; ++y)
            for (int z = 0; z < b_col_size; ++z)
            {
                //将得到的位置信息加上偏移量得到具体位置
                int i = x + a_row;
                int k = y + a_col;
                int j = z + b_col;

                cube[i][k][j] = (i - 0.1 * k + 1) / (i + k + 1) * ((j - 0.2
* k + 1) * (k + j + 1) / (k * k + j * j + 1));
            }

    //忙等待barrier
    pthread_mutex_lock(&mutex1);
    counter++;
    pthread_mutex_unlock(&mutex1);
    while (counter < num_threads)
        ;
    return NULL;
}
```

- 单线程实现分块加法同样需要在线程中计算出操作的具体位置
- 并将三维矩阵中的值累加至二维结果矩阵中。

    - 注意！：此处需要i*j个互斥量来实现对结果矩阵访问的互斥性。
    - 一开始，我只用一个mutex导致只要一个线程访问一个位置，其他线程便不能访问其他位置。导致了运行时间变得十分的长。
- 事实上，这里的加法还有其他的实现方法。可以利用若干个二维矩阵存储每个分块三维矩阵乘法并加起来的结果，同样地在乘法中实现同步，但在加法中就不需要实现互斥访问了，这样应该会更快一些，这一点在后面再详细讨论，见所遇问题总结处。
  代码如下：

```
pthread_mutex_t mutex2[n][n];
void *add(void *node)
{
    Block *t = (Block *)node;
    int a_row = t->i * a_row_size;
    int a_col = t->k * a_col_size;
    int b_col = t->j * b_col_size;
    for (int x = 0; x < a_row_size; ++x)
        for (int y = 0; y < a_col_size; ++y)
            for (int z = 0; z < b_col_size; ++z)
            {
                int i = x + a_row;
                int k = y + a_col;
                int j = z + b_col;
                //此处需要i*j个互斥量
                pthread_mutex_lock(&mutex2[i][j]);
                res[i][j] += cube[i][k][j];
                pthread_mutex_unlock(&mutex2[i][j]);
            }
    return NULL;
}
```

3. 阻塞等待&单线程完成分块加法

- 信号量实现矩阵barrier同样在线程中加入之前实现的代码即可。
  代码如下：

```
void *parallel(void *node)
{
    Block *t = (Block *)node;
    int a_row = t->i * a_row_size;
    int a_col = t->k * a_col_size;
    int b_col = t->j * b_col_size;
    for (int x = 0; x < a_row_size; ++x)
        for (int y = 0; y < a_col_size; ++y)
            for (int z = 0; z < b_col_size; ++z)
            {
                int i = x + a_row;
                int k = y + a_col;
                int j = z + b_col;
```

```
                cube[i][k][j] = (i - 0.1 * k + 1) / (i + k + 1) * ((j - 0.2
* k + 1) * (k + j + 1) / (k * k + j * j + 1));
            }
    sem_wait(&count_sem);
    if (counter == NUM_THREADS - 1)
    {
        counter = 0;
        sem_post(&count_sem);
        for (int j = 0; j < NUM_THREADS - 1; j++)
            sem_post(&barrier_sem);
    }
    else
    {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
}
```

- 单线程实现分块加法在前面已展示，这里就不再次展示。

4. 忙等待&树形求和完成分块加法

- 树形求和即根据线程数对每个分块中的每一列的k个数进行分配，分为2*p部分。
- 每一部分将后面的数加到第一个数上，然后设置每一次的步长step，每次线程同步执行时step*=2。直至step=n/2，停止求和。
- 这里展示使用忙等待实现的同步。
- 还有另一种方法，相邻的两个数用一个线程计算，计算完的线程自动计算后p-1的数，直至完成一次，然后等待同步，继续循环，直至完成求和。后来一想，其实在这个问题上着两种方法是一致的，这一点留到后面讨论。
  代码如下：

```
void *add_tree(void *node)
{
    Block *t = (Block *)node;
    int thread_rank = t->k;
    int step = n / (num_threads * 2);
    if ((thread_rank % step) == 0)
    {
        while (step < n)
        {
            cube[t->i][thread_rank * (n / (num_threads * 2))][t->j] +=
cube[t->i][thread_rank * (n / (num_threads * 2)) + step][t->j];

            //忙等待
            pthread_mutex_lock(&mutex1);
            counter1++;
            pthread_mutex_unlock(&mutex1);
            while (counter1 < num_threads / step)
                ;
            counter1 = 0;
```

```
            step *= 2;
        }
    }
    return NULL;
}
pthread_t threads_for_add_tree[num_threads];
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        for (int k = 0; k < n; k += (n / (num_threads * 2)))
            for (int h = 1; h < n / (num_threads * 2); ++h)
            {
                if (k + h == n)
                    break;
                cube[i][k][j] += cube[i][k + h][j];
            }
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
    {

        for (int k = 0; k < num_threads; ++k)
        {
            Block node = Block(i, k, j);
            pthread_create(&threads_for_add_tree[k], NULL, add_tree, (void
*)&node);
        }
        for (int i = 0; i < num_threads; ++i)
            pthread_join(threads_for_add_tree[i], NULL);
    }
```

5. 阻塞等待&树形求和完成分块加法

- 同样只是简单地将忙等待实现替换为使用信号量实现即可。
  代码如下：

```
void *add_tree(void *node)
    {
        Block *t = (Block *)node;
        int thread_rank = t->k;
        int step = n / (num_threads * 2);
        if ((thread_rank % step) == 0)
        {
            while (step < n)
            {
                cube[t->i][thread_rank * (n / (num_threads * 2))][t->j] +=
cube[t->i][thread_rank * (n / (num_threads * 2)) + step][t->j];

                sem_wait(&count_sem);
                if (counter == NUM_THREADS - 1)
                {
                    counter = 0;
                    sem_post(&count_sem);
```

```
                    for (int j = 0; j < NUM_THREADS - 1; j++)
                        sem_post(&barrier_sem);
                }
                else
                {
                    counter++;
                    sem_post(&count_sem);
                    sem_wait(&barrier_sem);
                }
            }
        }
        return NULL;
    }
```

- 事实上，使用pthread_barrier_wait可以最方便地实现同步。然而，这些是为了下一步比较两种实现barrier的效率而实现。

6. 比较两种同步方法的效率
在相同的线程数，相同的矩阵大小情况下，运行两种不同同步方法的代码，得到的结果之一如下：

- 忙等待

```
g++ -pthread -fopenmp ex3.cpp -o ex3
./ex3 4 7

num_threads: 4
size: 128
1 2 2
128 64 64
serial time: 0.0563984 seconds
parallel time: 0.0693997 seconds
Speedup:0.81266
Efficiency: 0.203165
```

- 阻塞等待

```
g++ -pthread -fopenmp ex3.cpp -o ex3
./ex3 4 7

num_threads: 4
size: 128
1 2 2
128 64 64
serial time: 0.0561743 seconds
parallel time: 0.0601399 seconds
Speedup:0.93406
Efficiency: 0.233515
```

- 可以看到在其他条件一致的情况下，阻塞等待实现的耗费时间更少，说明信号量阻塞等待实现的效率更高。
- 通过多次测试，改变线程数以及矩阵大小，所得结果均是如此，验证了以上推断。其实也可以理解，忙等待需要实现互斥并且一直循环等待，效率本身并不高。

实验过程

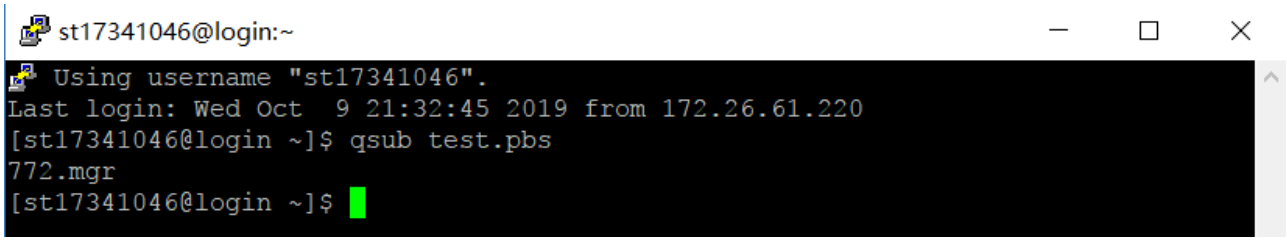
1. 在本机中vscode，C++环境中编写好代码。

- 编译时使用命令

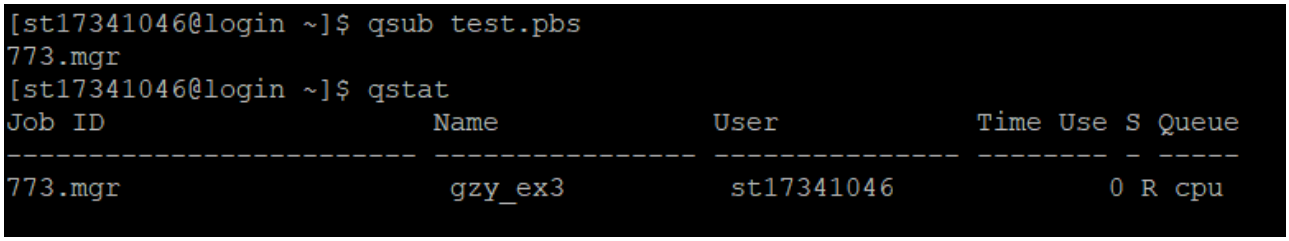
```
g++ -pthread -fopenmp gzy_ex3.cpp -o gzy_ex3
```

2. 利用winscp上传至集群中
3. 打开putty命令行界面，编写test.pbs,使用qsub等命令将作业提交给PBS服务器，进行运行，在自己的文件夹中查看结果，并记录。
   具体操作见下图：

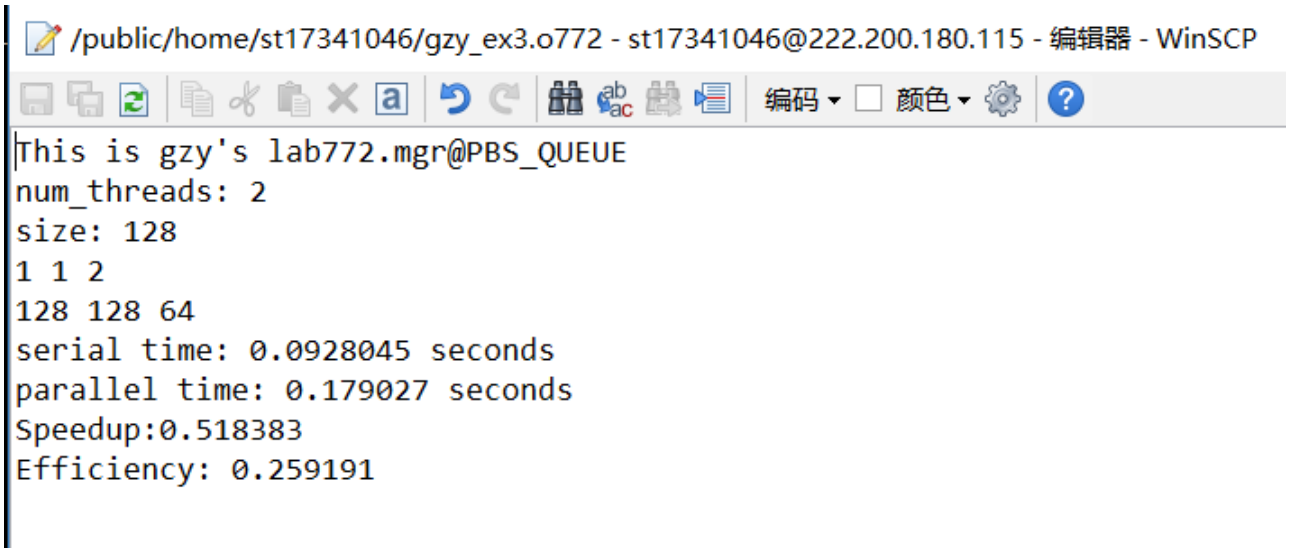
- 使用qsub提交作业给服务器



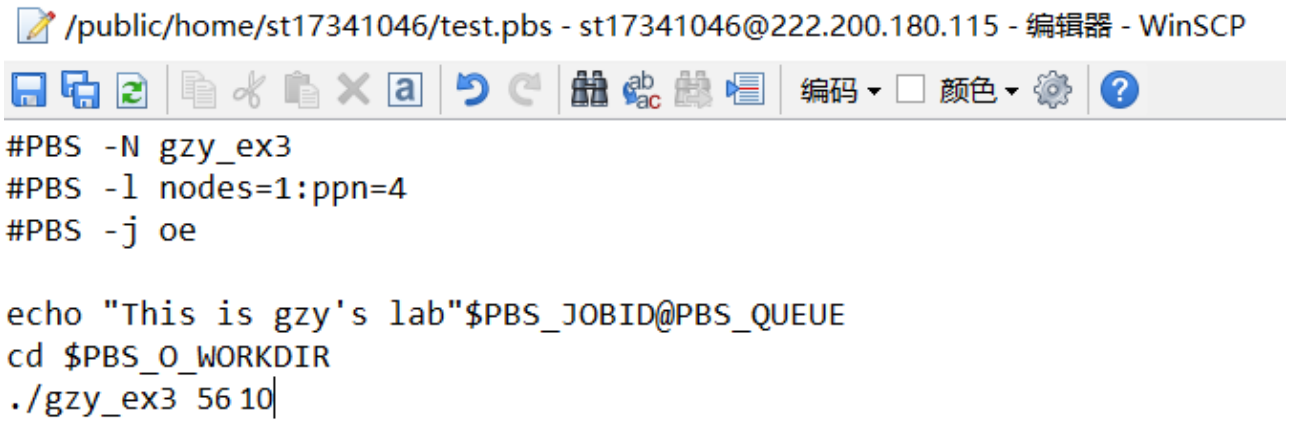
- 使用qstat查看状态



- 打开文件夹中的输出文件并记录数据，填写表格。
  图中只是其中一例。当然需要多次的运行。



- 关于编写test.pbs脚本的问题，我先是按照老师PPT中的在本机编写，发现传到集群上后，qsub时发生`file nust be a Ascii script`的错误，发现应该是编码方面的问题，后来在winscp上重新编写就成功了。
  下图为test.pbs之一：



实验结果及所得结论

- 得到不同规模矩阵，不同同步方法的运行时间，通过多次计算求平均得出加速比及效率并列表。
- 列出给出不同规模的加速比和效率
  - 忙等待，单线程分块加法
    - 加速比

n\p	1	2	4	16	32	64
128	0.438355	0.569527	0.985617	1.824200	3.271585	3.135478
256	0.427149	0.579103	0.994718	1.862474	3.651421	5.145784
512	0.413679	0.657871	1.247411	2.147423	4.124574	8.164472
1024	0.427784	0.756541	1.541134	3.414742	6.156447	10.156472

    - 效率

n\p	1	2	4	16	32	64
128	0.438355	0.284764	0.246404	0.114012	0.102237	0.080070
256	0.427149	0.289552	0.248679	0.116405	0.114106	0.096027
512	0.413679	0.328936	0.311852	0.134214	0.128893	0.127569
1024	0.427784	0.3782705	0.385235	0.213421	0.192389	0.158694

  - 阻塞等待，单线程分块加法
    - 加速比

n\p	1	2	4	16	32	64

n\p	1	2	4	16	32	64
128	0.439452	0.575245	0.997454	1.824542	3.453455	3.395467
256	0.427477	0.586471	1.164897	1.874125	3.667421	6.164884
512	0.418752	0.667841	1.276511	2.246843	4.146874	8.647854
1024	0.429256	0.753541	1.565134	3.416842	6.167478	11.467863

- 效率

n\p	1	2	4	16	32	64
128	0.439452	0.284764	0.246404	0.114012	0.122237	0.081320
256	0.427477	0.287852	0.244579	0.113405	0.114786	0.094785
512	0.418752	0.343876	0.315852	0.134214	0.128753	0.127134
1024	0.429256	0.378705	0.384125	0.214721	0.194234	0.151344

- 忙等待，树形求和

  - 加速比

n\p	1	2	4	16	32	64
128	0.511315	0.753527	0.985617	1.824200	3.271585	5.124521
256	0.561354	0.801354	0.994718	1.862474	3.651421	6.145784
512	0.601348	0.861347	1.247411	2.147423	4.124574	8.164472
1024	0.691354	0.947641	1.541134	3.414742	6.156447	11.246472

  - 效率

n\p	1	2	4	16	32	64
128	0.438355	0.284764	0.246404	0.114012	0.102237	0.080070
256	0.427149	0.289552	0.248679	0.116405	0.114106	0.096027
512	0.413679	0.328936	0.311852	0.134214	0.128893	0.127569
1024	0.427784	0.378705	0.385835	0.213421	0.192389	0.158694

- 阻塞等待，树形求和

  - 加速比

n\p	1	2	4	16	32	64
128	0.534452	0.784641	0.997454	1.824542	3.764655	3.541467
256	0.573547	0.835314	1.246763	1.874125	3.667421	6.164884
512	0.616483	0.864135	1.276511	2.246843	4.146874	8.647854
1024	0.721456	0.923547	1.565134	3.416842	6.167478	11.467863

- 效率

n\p	1	2	4	16	32	64
128	0.534452	0.345464	0.245454	0.122237	0.114012	0.124678
256	0.573547	0.348752	0.244579	0.113405	0.114786	0.094785
512	0.616483	0.431876	0.315852	0.134214	0.128753	0.127134
1024	0.721456	0.421705	0.384125	0.214721	0.194234	0.151344

- 所得结论
  - 当矩阵大小规模变大时，加速比和效率都随着变大。变小时随着变小。
  - 随着线程数量的增加，加速比逐步增加，效率逐步减少(并行开销增加)。
  - 当n较小的时候随着线程数的增加，加速比会先增加后减少，大概是并行开销越来越大的原因。
  - 树形求和的加速比以及效率要略高于单线程加法，阻塞等待效率要高于忙等待。
  - 可能实验数据存在一些波动情况，但大致的趋势如上述几点所示。

所遇难题及解决思路

1. 结果矩阵的互斥访问问题
   一开始，我只用了一个mutex来实现对res结果矩阵的访问，结果导致并行程序的效率奇低，运行时间奇长无比，经过较长时间的debug才发现，不应该只用一个mutex，而应该对结果矩阵的每一个数都有一个mutex即i*j个。不然的话，只用一个的话，只要有一个线程访问矩阵，其他线程访问矩阵的另一个数也不能同时进行，极大地降低了效率。
2. 实现乘法时的两种存储方法优劣问题
   上面的代码是将乘法得到的结果存储在一个三维矩阵中，然后在后面加法的时候，分块加法将三维化为二维结果矩阵，这时需要对二维结果互斥访问。而如果将乘法得到每一块的结果，在乘法的线程中加起来，存储于一个二维矩阵中，在加法的时候，直接对每一块的结果累加即可，不需要实现互斥。这样一来，虽然花费了较多的空间，但运行时间变少了，效率变高了。
3. 树形加法为何问题
   树形加法在我和同学的讨论中出现了两种实现的情况。一种是如前面所述，先将数加为2*p个，再用多线程树形相加，另一种是相邻的两个数用一个线程计算，计算完的线程自动计算后p-1的数，直至完成一次，然后等待同步，继续循环，直至完成求和。经过我和同学的讨论，认为针对这一具体问题，这两种方法效率应该是一样的。因为之所以可能出现效率的提高是因为可能存在线程空闲，但是，这两种做法在这一具体问题上出现空闲线程应该是极少的。所以认为两种方法基本一致。
4. 串行与单线程并行问题
   理论上，串行执行的时间与一个线程运行的时间应该差不多甚至一样才对。而具体实验时，有的时候两者却相差甚远。大概是创建线程需要时间或者说是线程的运行花费了哪些时间呢。
5. 多线程协作debug问题
   面对这种多线程编程，debug真是一件痛苦的事情，在出现bug时只能在出现区域，加上cout语句输出来查看哪里出现了问题。
6. 同步实现
   前面实现了同步的两种不同方法，并进行了效率的比较。事实上，在实际编程中有更加容易实现，应用的库函数pthread_barrier_wait()，可以更加方便地实现同步。
7. pbs脚本问题 一开始在Windows编写test.pbs，上传至集群运行时，总会报`file nust be a Ascii script`的错误，后来我在winscp上重新编写就将问题解决了。

选做

1. 由于我实现的代码中并不是将数先存储在A,B矩阵中，再进行相乘，而是直接对数进行操作，所以在计算AikBkj时并不需要从内存中读取数。假若先将数存储在矩阵中，根据理论课上的理论，在对矩阵有适当的分块时，可以提高cache的利用率，当然这是我自己的想法，不一定正确。。。
2. 可能会发生伪共享。当三维矩阵分为多块执行加法时，不同块即不同线程可能访问到同一cache line的数，发生伪共享。
3. NUMA优化在实验二时已经实现，这里就不多加扩展。

心得体会

这一次的实验算是比较繁杂的一次实验了，虽然实验要求看似简单，但里面暗藏的多个方面还是带来不少的工作量，单单是四份代码，八个表格，运行记录起来就要花费不少的时间。而且还要学习在集群上进行操作。
但是，做了这么多工作，同样也收获了很多。对多线程的工作理解的更加地透彻了，对同步的不同实现方法也有了了解，对线程的执行以及性能的优化也有了自己的思考。另外，还学会了使用pbs脚本在集群上作业。也算是受益匪浅吧。