高性能计算程序设计基础(4) 秋季2021

1. 通过OpenMP实现通用矩阵乘法

> 通过OpenMP实现通用矩阵乘法（Lab1）的并行版本，OpenMP并行线程从1增加至8，矩阵规模从512增加至2048。
>
> 通用矩阵乘法（GEMM）通常定义为：
>
> $C = A * B$
>
> $C_{m,n} = \sum_{n=1}^{N} A_{m,n} B_{n,k}$
>
> 输入：M , N, K三个整数（512 ~2048）
>
> 问题描述：随机生成M、*N和N*K的两个矩阵A,B,对这两个矩阵做乘法得到矩阵C.
>
> 输出：A,B,C三个矩阵以及矩阵计算的时间

代码

> 完整代码见lab4_1.c、lab4_1_2.c

1. 头文件以及一些全局变量

```
1 #include<stdio.h>
2 #include<omp.h>
3 #include<time.h>
4 #include<stdlib.h>
5
6 int thread_num, m, n, k;
```

- 第1-4行，是我们需要用到的一些头文件，其中`#include<omp.h>`是我们利用`openmp`实现并行计算需要用到的头文件。
- 第6行，4个全局变量：m，n，k是矩阵A(m*n)、B(n*k)、C(m*k)的大小，thread_num是接下来要创建的线程个数，本次实验只用到一次omp并行因此不用专门调用函数去设置线程

数，ompui自动调用。

2. 通过一个函数 `Get_args()` 来获得我们在终端输入运行指令时同时输入的参数m、n、k。thread_num会被自动设置为omp并行的线程数。

```
void Get_args(int argc,char *argv[]){
    //thread_num = strtol(argv[1], NULL, 10); //the first  omp will automatically set the first as the number of threads
    m = strtol(argv[2], NULL, 10);
    n = strtol(argv[3], NULL, 10);
    k = strtol(argv[4], NULL, 10);
}
```

3. 一个用来输出矩阵的函数 `PRINT()`：

```
void PRINT(double *x,int m,int n){
    for(int i=0;i<m;++i){
        for(int j=0;j<n;++j) printf("%lf  ",x[i*n+j]);
        printf("\n");
    }
    printf("\n");
}
```

4. 主函数中：
   - 首先调用函数获得矩阵尺寸，接着定义一些所需的变量：
     - 第5行，定义三个double类型的指针，并在第8-10行为他们分配空间；
     - 第6行，定义两个变量来记录时间；

```
int main(int argc,char *argv[]){
    Get_args(argc,argv);

    //定义和分配空间
    double *A,*B,*C;
    int start_time,finish_time;

    A = (double*)malloc(sizeof(double)*m*n);
    B = (double*)malloc(sizeof(double)*n*k);
    C = (double*)malloc(sizeof(double)*m*k);

    ...
}
```

   - 给矩阵A、B赋值50以内的随机数作为矩阵元素：

```
int main(int argc,char *argv[]){
    ...
    //给矩阵赋值
    srand(time(NULL));
    for(int i=0; i<m; ++i)
        for(int j=0; j<n; ++j) A[i*n+j]=rand()%50;
    for(int i=0; i<n; ++i)
        for(int j=0; j<k; ++j) B[i*k+j]=rand()%50;
    ...
}
```

- 开始利用openmp并行计算并给此过程计时（最简单的方案）。

  > 见lab4_1.c

  - 第4行表示接下来的for循环用并行实现，`temp_c`为私有变量，`A、B、C`都是共有变量。
  - 第8-14行，根据通用矩阵乘法的公式计算两矩阵相乘的各元素。
  - 第9-12行，由于要将计算好的元素放入矩阵C中，而存放矩阵C的数组是个共享变量，为了预防数据竞争，给赋值的过程加上缓冲区。

```
1  int main(int argc,char *argv[]){
2      ...
3      start_time = clock();
4      #pragma omp parallel for private(temp_c) shared(A, B, C)
5          for(int i=0; i<m; ++i){
6              for(int j=0; j<k; ++j){
7                  temp_c = 0;
8                  for(int x=0; x<n; ++x) temp_c+=A[i*n+x]*B[x*k+j];
9                  #pragma omp critical
10                 {
11                     C[i*n+j] = temp_c;
12                 }
13             }
14         }
15     finish_time = clock();
16     ...
17 }
```

- 这里可以进行一定的优化，再第三层循环处再进行一次并行分配，并利用规约简化运算(第8行):

  > 见lab4_1_2.c

  - 此时i、j都作为私有变量，而矩阵A、B、C仍为公有变量，规约运算的操作对象是`temp_c`，运算方法是加法
  - 注意这里第二个并行化设置私有变量用的是`firstprivate`，而不是`private`。这是由于在第二个并行过程中要用到的i,j都必须继承之前的初始值，而`private`设置的变量并不继承之前的初始化的值。如果用`private`，会使计算出错。这里我在刚开始的时候并没有注意到，发现计算结果错误后输出了并行过程中的一些变量才发现i，j的值都有问题，最后才发现是设置私有变量的问题，改过来以后就运算结果正确了。

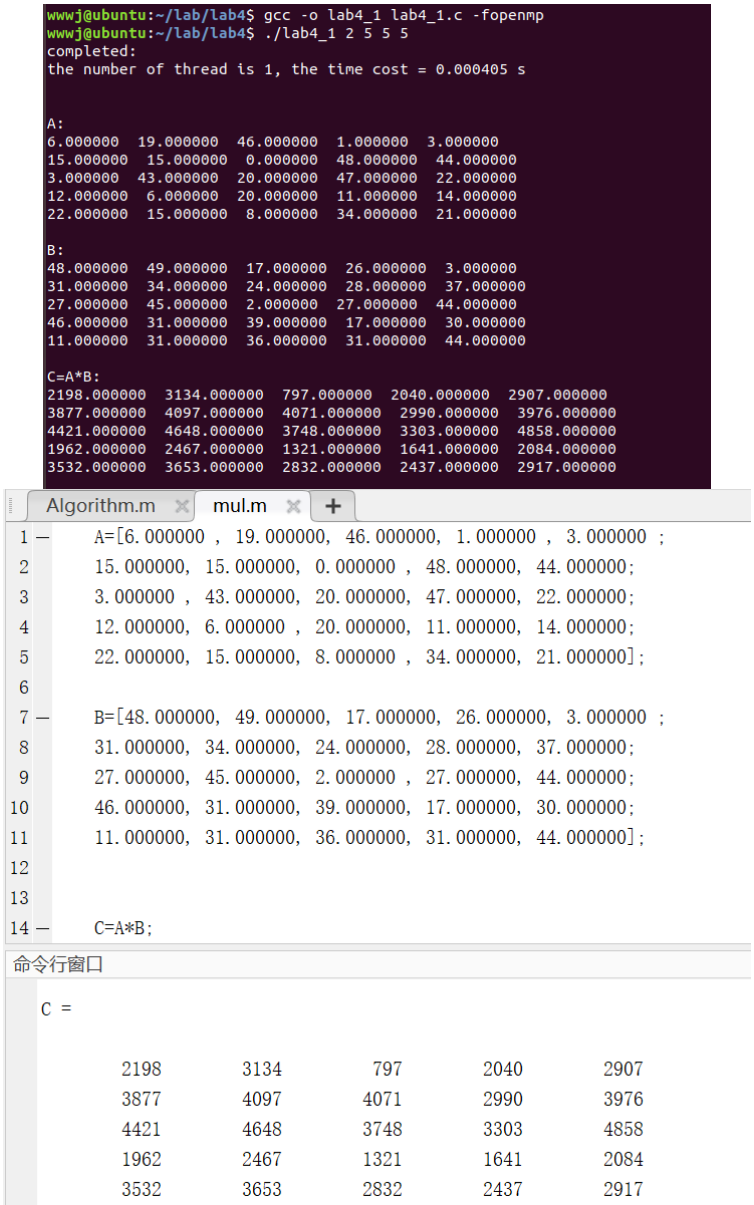
```
1  int main(int argc,char *argv[]){
2      ...
3      start_time = clock();
4      #pragma omp parallel for private(temp_c) shared(A, B, C)
5          for(int i=0; i<m; ++i){
6              for(int j=0; j<k; ++j){
7                  temp_c = 0;
8                  #pragma omp parallel for firstprivate(i,j)
   shared(A, B, C) reduction(+:temp_c)
9                  for(int x=0; x<n; ++x) temp_c+=A[i*n+x]*B[x*k+j];

10                 #pragma omp critical
11                 {
12                     C[i*n+j] = temp_c;
13                 }
14             }
15         }
```

```
    finish_time = clock();
    ...
}
```

运行结果

- 我们首先验证代码计算矩阵乘法的正确性：
  计算两个5 × 5矩阵的乘法，左边是代码的计算结果，右边是用matlab计算的结果。



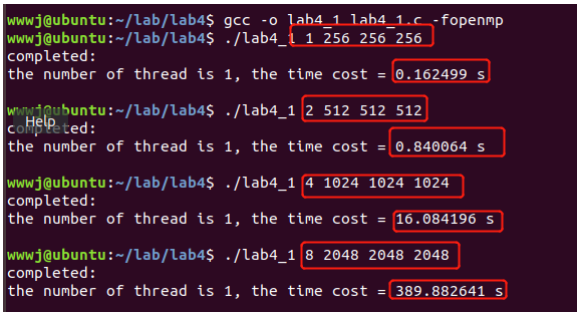
  计算结果正确。

- 优化前

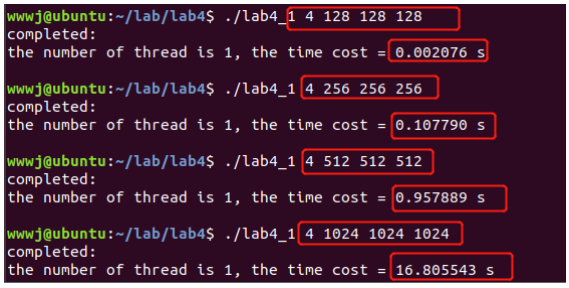
  下面是编译和运行这段代码的指令：

```
/* File:     lab4_1.c
 *
 * Purpose:  Use a layer of openmp to implement matrix multiplication
 *
 * Compile:  gcc -o lab4_1 lab4_1.c -lpthread
 *
 * Run:      ./lab4_1 <num> <m> <k> <n>
 *           num is the thread number that we will create
 *           m,k,n is size of matrix A,B,C
 *
 * Input:    none
 * Output:   the cost time. If you want ,you can also print matrix
A,B,C.
 */
```

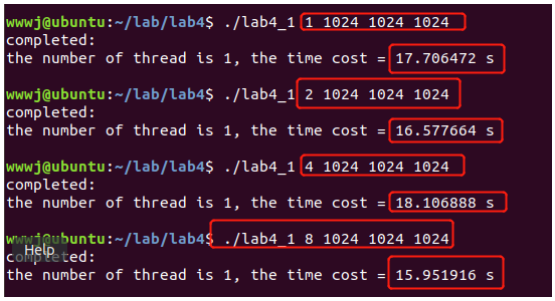
- 将并行线程从1增加至8，矩阵规模从256增加至2048：



或者可以在线程4下，将矩阵规模从128增加到1024：



在矩阵规模为1024的情况下，将线程数从1增加到8：



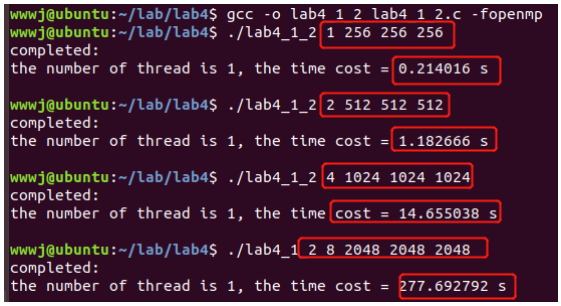
可以看到，并行的效果并不明显，随着线程的增加，用时可能会更多这可能是由于通信、数据量等原因，但整体来说用时会随着线程的增加而减少。

- 优化后

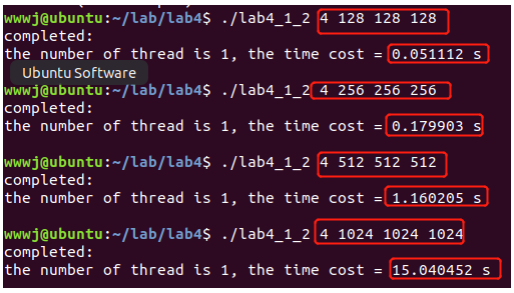
  编译的指令：

```
/* File:     lab4_1_2.c
 *
 * Purpose:  Use two layers of openmp to implement replacement of serial code
 *
 * Compile:  gcc -o lab4_1_2 lab4_1_2.c -lpthread
 *
 * Run:      ./lab4_1_2 <num> <m> <k> <n>
 *           num is the thread number that we will create
 *           m,k,n is size of matrix A,B,C
 *
 * Input:    none
 * Output:   the cost time. If you want ,you can also print matrix A,B,C.
 */
```

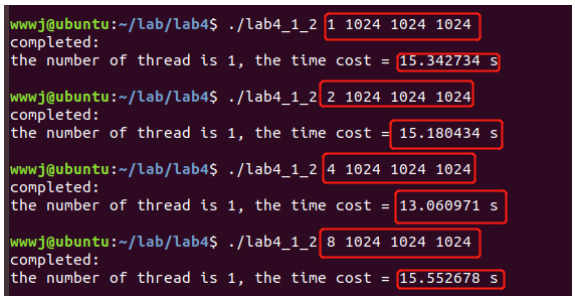
将并行线程从1增加至8，矩阵规模从256增加至2048：



或者可以在线程4下，将矩阵规模从128增加到1024：



在矩阵规模为1024的情况下，将线程数从1增加到8：



- 很明显，相同情况下`lab4_1_2`要比`lab4_1`加速更好一些，运算时间更短。但是同样对于优化后的代码自身来说，相同规模的矩阵，增加线程后的并行效果并不是很明显。

2. 基于OpenMP的通用矩阵乘法优化

> 分别采用OpenMP的默认任务调度机制、静态调度schedule(static, 1)和动态调度schedule(dynamic,1)，调度#pragma omp for的并行任务，并比较其性能。

默认任务调度机制

默认任务调度机制的实现和上面介绍的lab4_1中的内容一样，这里就不在赘述。

静态调度schedule(static, 1)

静态调度static方式。实际上就是：假设有n次循环迭代，t个线程，那么给每个线程静态分配大约n/t次迭代计算。由于n/t不一定是整数，因此实际分配的迭代次数可能存在差1的情况，如果指定了size参数的话，那么可能相差一个size。静态调度时可以不使用size参数，也可以使用size参数。

代码

> 完整代码见lab4_2_static.c

和最开始“用OpenMP实现通用矩阵的代码相比，发生变化的代码只有进行并行分配的这一部分，因此其他代码也就不再赘述。

```
1  int main(int argc,char *argv[]){
2      ...
3      start_time = clock();
4      #pragma omp parallel for private(temp_c) shared(A, B, C)
   schedule(static)
5          for(int i=0; i<m; ++i){
6              for(int j=0; j<k; ++j){
7                  temp_c = 0;
8                  #pragma omp parallel for firstprivate(i,j) shared(A, B, C)
   reduction(+:temp_c) schedule(static)
9                  for(int x=0; x<n; ++x) temp_c+=A[i*n+x]*B[x*k+j];
10
11                 #pragma omp critical
12                 {
13                     C[i*n+j] = temp_c;
14                 }
15             }
16         }
17     ...
18 }
```

- 静态调度加在了第4行和第8行上，`schedule()`表示设置调度方法，`static`表示使用的是静态调度机制，后面可以加参数size，表示有多少个迭代次数未被分配，可以作为第二个参数写入。比如在计算线程数为2，矩阵规模为5、5、5的时候，可以写成`schedule(static,1)`，表示会有一个迭代未被分配，不过不屑也没有什么问题，omp会自动处理。

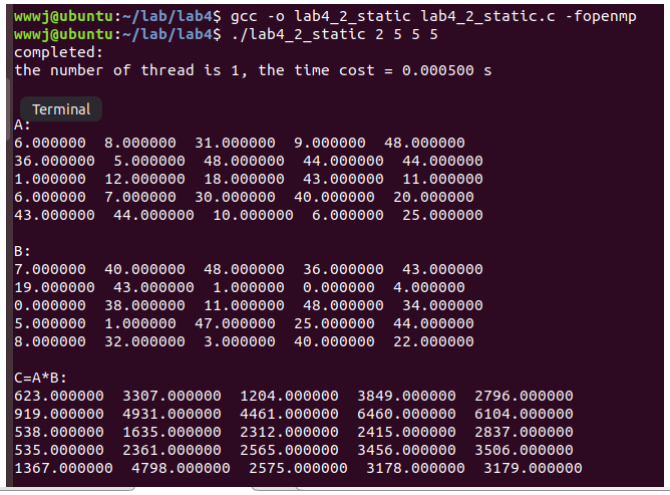
运行结果

- 下面是编译和运行这段代码的指令：

```
1  /* File:     lab4_2_static.c
2   *
3   * Purpose:  Use the static scheduling method to optimize
4   *           the general matrix multiplication implemented by openmp
5   *
6   * Compile:  gcc -o lab4_2_static lab4_2_static.c -fopenmp
7   *
8   * Run:      ./lab4_2_static <num> <m> <k> <n>
```

```
 *           num is the thread number that we will create
 *           m,k,n is size of matrix A,B,C
 *
 * Input:    none
 * Output:   the cost time. If you want ,you can also print matrix A,B,C.
 */
```

- 我们首先验证代码计算矩阵乘法的正确性：
  计算两个5×5矩阵的乘法，左边是代码的计算结果，右边是用matlab计算的结果。




```
A=[6.000000 , 8.000000 , 31.000000, 9.000000 , 48.000000 ;
36.000000, 5.000000 , 48.000000, 44.000000, 44.000000 ;
1.000000 , 12.000000, 18.000000, 43.000000, 11.000000 ;
6.000000 , 7.000000 , 30.000000, 40.000000, 20.000000 ;
43.000000, 44.000000, 10.000000, 6.000000 , 25.000000 ];

B=[7.000000 , 40.000000, 48.000000, 36.000000, 43.000000 ;
19.000000, 43.000000, 1.000000 , 0.000000 , 4.000000 ;
0.000000 , 38.000000, 11.000000, 48.000000, 34.000000 ;
5.000000 , 1.000000 , 47.000000, 25.000000, 44.000000 ;
8.000000 , 32.000000, 3.000000 , 40.000000, 22.000000 ];

C=A*B;
```

令行窗口

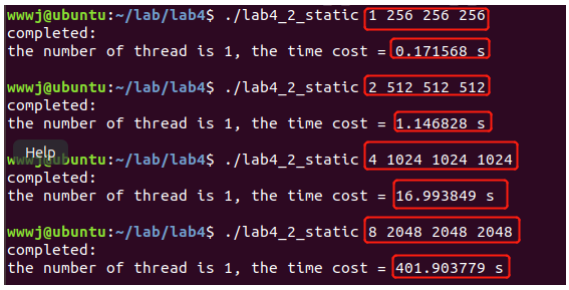
```
C =

   623   3307   1204   3849   2796
   919   4931   4461   6460   6104
   538   1635   2312   2415   2837
   535   2361   2565   3456   3506
  1367   4798   2575   3178   3179
```

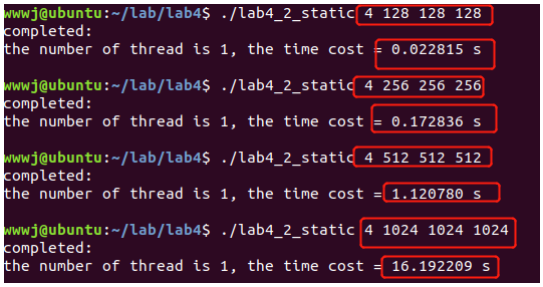
运算结果正确。

- 运行时间对比

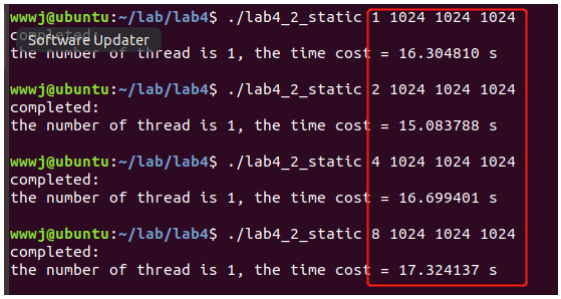
  将并行线程从1增加至8，矩阵规模从256增加至2048：



或者可以在线程4下，将矩阵规模从128增加到1024：



在矩阵规模为1024的情况下，将线程数从1增加到8：



动态调度schedule(dynamic,1)

动态调度是动态地将迭代分配到各个线程，动态调度可以使用size参数也可以不使用size参数。不使用size参数时是将迭代逐个地分配到各个线程，使用size参数时，每次分配给线程的迭代次数为指定的size次。

代码

> 完整代码见lab4_2_dyna.c

和上面一样这里只介绍并行部分：

```
1  int main(int argc,char *argv[]){
2      ...
3      start_time = clock();
4      #pragma omp parallel for private(temp_c) shared(A, B, C)
   schedule(dynamic)
5          for(int i=0; i<m; ++i){
6              for(int j=0; j<k; ++j){
7                  temp_c = 0;
8                  #pragma omp parallel for firstprivate(i,j) shared(A, B, C)
   reduction(+:temp_c) schedule(dynamic)
9                  for(int x=0; x<n; ++x) temp_c+=A[i*n+x]*B[x*k+j];
10
11                 #pragma omp critical
12                 {
13                     C[i*n+j] = temp_c;
14                 }
15             }
16         }
17     ...
18 }
```

- 静态调度加在了第4行和第8行上，`schedule()`表示设置调度方法，`dynamic`表示使用的是动态调度机制，后面可以加参数size，表示每次分配给一个线程多少个迭代次数，可以作为第二个参数写

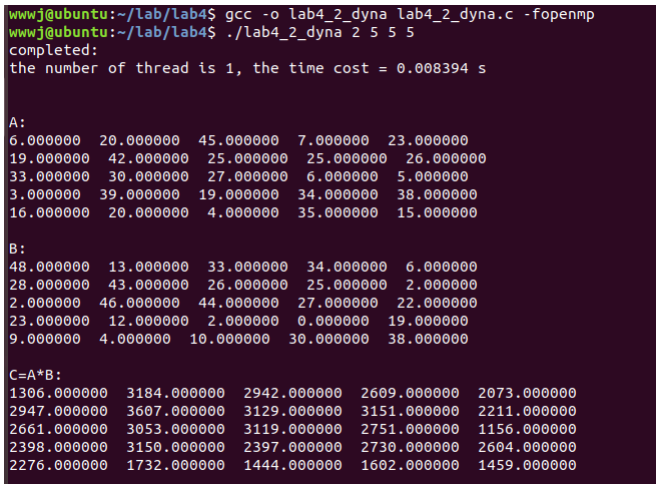
入。不写他自己会进行分配。

运行结果

- 下面是编译和运行这段代码的指令：

```
/* File:     lab4_2_dyna.c
 *
 * Purpose:  Use the dynamic scheduling method to optimize
 *           the general matrix multiplication implemented by openmp
 *
 * Compile:  gcc -o lab4_2_dyna lab4_2_dyna.c -fopenmp
 *
 * Run:      ./lab4_2_dyna <num> <m> <k> <n>
 *           num is the thread number that we will create
 *           m,k,n is size of matrix A,B,C
 *
 * Input:    none
 * Output:   the cost time. If you want ,you can also print matrix
A,B,C.
 */
```

- 我们首先验证代码计算矩阵乘法的正确性：
计算两个5×5矩阵的乘法，左边是代码的计算结果，右边是用matlab计算的结果。



```
A=[6.000000 ,  20.000000,  45.000000,  7.000000 ,  23.000000;
19.000000,  42.000000,  25.000000,  25.000000,  26.000000;
33.000000,  30.000000,  27.000000,  6.000000 ,  5.000000 ;
3.000000 ,  39.000000,  19.000000,  34.000000,  38.000000;
16.000000,  20.000000,  4.000000 ,  35.000000,  15.000000];

B=[48.000000,  13.000000,  33.000000,  34.000000,  6.000000 ;
28.000000,  43.000000,  26.000000,  25.000000,  2.000000 ;
2.000000 ,  46.000000,  44.000000,  27.000000,  22.000000;
23.000000,  12.000000,  2.000000 ,  0.000000 ,  19.000000;
9.000000 ,  4.000000 ,  10.000000,  30.000000,  38.000000];


C=A*B;
```

令行窗口

```
C =

        1306        3184        2942        2609        2073
        2947        3607        3129        3151        2211
        2661        3053        3119        2751        1156
        2398        3150        2397        2730        2604
        2276        1732        1444        1602        1459
```

计算结果正确。

- 运算时间

  将并行线程从1增加至8，矩阵规模从256增加至2048：

```
wwwj@ubuntu:~/lab/lab4$ gcc -o lab4_2_dyna lab4_2_dyna.c -fopenmp
wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 1 256 256 256
completed:
the number of thread is 1, the time cost = 0.546367 s

wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 2 512 512 512
^[[Acompleted:
the number of thread is 1, the time cost = 4.363340 s

wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 4 1024 1024 1024
completed:
the number of thread is 1, the time cost = 54.078937 s

wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 8 2048 2048 2048
completed:
the number of thread is 1, the time cost = 981.575012 s
```

  或者可以在线程4下，将矩阵规模从128增加到1024：

```
Ubuntu Software
wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 4 128 128 128
completed:
the number of thread is 1, the time cost = 0.054149 s

wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 4 256 256 256
completed:
the number of thread is 1, the time cost = 0.544103 s

wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 4 512 512 512
completed:
the number of thread is 1, the time cost = 4.243539 s

wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 4 1024 1024 1024
completed:
the number of thread is 1, the time cost = 67.850861 s
```

  在矩阵规模为1024的情况下，将线程数从1增加到8：

```
wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 1 1024 1024 1024
completed:
the number of thread is 1, the time cost = 60.836549 s

wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 2 1024 1024 1024
completed:
the number of thread is 1, the time cost = 65.813963 s

wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 4 1024 1024 1024
completed:
the number of thread is 1, the time cost = 69.661108 s

wwwj@ubuntu:~/lab/lab4$ ./lab4_2_dyna 8 1024 1024 1024
completed:
the number of thread is 1, the time cost = 69.447817 s
```

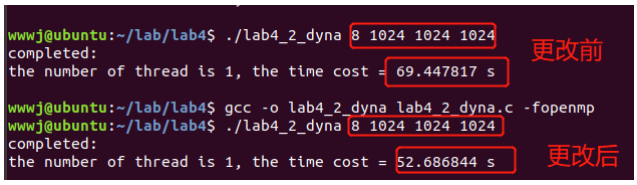
三种调度机制的比较

通过三种调度机制在相同线程不同矩阵规模以及不同线程相同矩阵规模的运行时间比较，发现他们的性能：

默认调度机制 > 静态调度机制 > 动态调度机制

动态调度机制的状态最差可能是由于不断分配任务造成的时间开销。

在网上一些文章说双层并行第一层使用默认调度机制，第二层使用动态调度机制的效果最好，但是在尝试以后，在我的代码中并不是这样的结果，性能并没有比之前更好，可能是一些其他方面的原因，比如数据大小、机器性能等。



文章链接：

C++性能优化系列——矩阵转置(四)OpenMP并行计算_yan31415的博客-CSDN博客

3. 构造基于Pthreads的并行for循环分解、分配和执行机制。

> 1）基于pthreads的多线程库提供的基本函数，如线程创建、线程join、线程同步等。构建parallel_for函数对循环分解、分配和执行机制，函数参数包括但不限于(int start, int end, int increment, void *(functor)(void*), void *arg , int num_threads)；其中start为循环开始索引；end为结束索引；increment每次循环增加索引数；functor为函数指针，指向的需要被并行执行循环程序块；arg为functor的入口参数；num_threads为并行线程数。
>
> 2）在Linux系统中将parallel_for函数编译为.so文件，由其他程序调用。
>
> 3）将通用矩阵乘法的for循环，改造成基于parallel_for函数并行化的矩阵乘法，注意只改造可被并行执行的for循环（例如无race condition、无数据依赖、无循环依赖等）。

> **举例说明：**
>
> 将串行代码：

```
for ( int i = 0; i < 10; i++ ){
 A[i]=B[i] * x + C[i]
}
```

> 替换为------>

```
parallel_for(0, 10, 1, functor, NULL, 2);

struct for_index {
    int start;
    int end;
    int increment;
}

```

```
void * functor (void * args){
    struct for_index * index = (struct for_index *) args;
    for (int i = index->start; i < index->end; i = i + index->increment){
        A[i]=B[i] * x + C[i];
    }
}
```

编译后执行阶段：

多线程执行

在两个线程情况下：

Thread0: start和end分别为0，5:

```
void * funtor(void * arg){
    int start = my_rank * (10/2)
    int end = start + 10/2;
    for(int j = start, j < end, j++)
        A[j]=B[j] * x + C[j];
}
```

Thread1: start和end分别为5，10

.....

1)

代码

完整代码见lab4_3_1.c

- 头文件、一些全局变量以及用来获取命令行参数的函数：

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

double *A, *B, *C, x = 10;
int thread_num,n;//n是数组A[]、B[]、C[]共有多少个元素

void Get_args(int argc,char *argv[]){
    thread_num = strtol(argv[1], NULL, 10);
    n = strtol(argv[2], NULL, 10);
}
```

- 一个实例化一个结构体，用来传递每个线程进行需要的数据：start为循环开始索引；end为结束索引；increment每次循环增加索引数。

```
struct for_index {
    int start;
    int end;
    int increment;
};
```

- 构建parallel_for函数对循环分解、分配和执行机制：
  - 第2-3行，首先设置`num_threads`个标识符。并计算可以将这部分运算量划分成`divid`个部分，然后分配到各个线程上去。
  - 第4-14行，定义`num_threads`个结构体，保存每个线程的循环开始、结束索引以及每次循环增加索引数。给结构体里的变量赋值，这里可能运算量和线程数不能完全恰好被分配，将剩余的全部迭代次数都分配给最后一个线程。
  - 第16-18行然后创建`num_threads`个线程，每个线程都有对应的线程标识符，他们都调用`functor()`函数，并传入之前设置过的该线程计算需要用到的参数；
  - 第20-22行，线程计算结束，释放线程标识符对应的线程占用的资源，本次计算没有返回值，第二个参数为`NULL`；
  - 第24行，输出信息表示并行运算已经结束。

```
1  void parallel_for(int start, int end, int increment, void *(*functor)
   (void*), void *arg , int num_threads){
2      pthread_t pth[num_threads];
3      int divid = (end - start + 1)/num_threads;
4      struct for_index thread_assign[num_threads];
5      for(int i=0; i<num_threads; ++i){
6          thread_assign[i].start = i*divid ;
7          if(i == (num_thread - 1) ){
8              thread_assign[i].end = end;
9          }
10         else{
11             thread_assign[i].end = thread_assign[i].start + divid;
12         }
13         thread_assign[i].increment = increment;
14     }
15
16     for(int i=0; i<num_threads; ++i){
17         pthread_create(&pth[i], NULL, functor, &thread_assign[i]);
18     }
19
20     for(int i=0; i<num_threads; ++i){
21         pthread_join(pth[i], NULL);
22     }
23
24     printf("paraller finish\n");
25 }
```

- 每个线程中用来调用进行计算的函数`functor ()`：
  - 首先将收到的参数进行类型转化。
  - 然后根据`parallel_for()`分配的循环开始、结束的索引以及循环增加的索引数这些信息进行计算。

```
1 void * functor (void * args){
2     struct for_index * index = (struct for_index *) args;
3     for (int i = index->start; i < index->end; i = i + index->increment){
4         A[i] = B[i] * x + C[i];
5     }
6 }
```

- 主函数：

- 调用函数获取线程数`thread_num`和数组大小`n`，给数组A、B、C分配空间，并给数组A、B、C设置初始值。
- 第13行，调用`parallel_for()`，传入参数进行计算。
- 最后输出进行运算之后的数组，验证计算结果的正确性。

```
1  int main(int argc,char *argv[]){
2      Get_args(argc, argv);
3      A = (double*)malloc(sizeof(double)*n);
4      B = (double*)malloc(sizeof(double)*n);
5      C = (double*)malloc(sizeof(double)*n);
6
7      for(int i=0; i<n; ++i){
8          C[i]=i;
9          B[i]=i;
10         A[i]=0;
11     }
12
13     parallel_for(0, n, 1, functor, NULL, thread_num);
14
15     for(int i=0; i<n; ++i){
16         printf(" a[%d]=%f, b[%d]=%f, c[%d]=%f\n",i,A[i],i,B[i],i,C[i]);
17     }
18     return 0;
19
20 }
```

运算结果

- 下面是编译和运行这段代码的指令：

```
1  /* File:     lab4_3_1.c
2   *
3   * Purpose:  Use pthread_for to implement replacement of serial code
4   *
5   * Compile:  gcc -o lab4_3_1 lab4_3_1.c -lpthread
6   *
7   * Run:      ./lab4_3_1 <num> <n>
8   *           num is the thread number that we will create
9   *           n is size of array A,B,C
10  *
11  * Input:    none
12  * Output:   the array A,B,C.
13  */
```

- 运算结果正确，按照预期的结果输出：

```
wwwj@ubuntu:~/lab/lab4$ gcc -o lab4_3_1 lab4_3_1.c -fopenmp
wwwj@ubuntu:~/lab/lab4$ ./lab4_3_1 2 10
paraller finish
 a[0]=0.000000, b[0]=0.000000, c[0]=0.000000
 a[1]=11.000000, b[1]=1.000000, c[1]=1.000000
 a[2]=22.000000, b[2]=2.000000, c[2]=2.000000
 a[3]=33.000000, b[3]=3.000000, c[3]=3.000000
 a[4]=44.000000, b[4]=4.000000, c[4]=4.000000
 a[5]=55.000000, b[5]=5.000000, c[5]=5.000000
 a[6]=66.000000, b[6]=6.000000, c[6]=6.000000
 a[7]=77.000000, b[7]=7.000000, c[7]=7.000000
 a[8]=88.000000, b[8]=8.000000, c[8]=8.000000
 a[9]=99.000000, b[9]=9.000000, c[9]=9.000000
```

2)

代码

> 完整代码见lab4_3_2.c、parallel_for.h、parallel_for.c

- 首先设置一个头文件`parallel_for.h`，将函数`parallel_for()`以及结构体`for_index`写入这个头文件：

```
#ifndef _PARALLEL_FOR_
#define _PARALLEL_FOR_

#include<pthread.h>

struct for_index {
    int start;
    int end;
    int increment;
};

void parallel_for(int start, int end, int increment, void *(*functor)(void*), void *arg , int num_threads);

#endif
```

- 然后在文件`parallel_for.c`中对函数`parallel_for()`进行实现，这部分代码和1）中的函数`parallel_for()`完全相同，这里不在赘述：

```
#include<stdio.h>
#include"parallel_for.h"

void parallel_for(int start, int end, int increment, void *(*functor)(void*), void *arg , int num_threads){
    ...
}
```

- 最后写一个文件`lab4_3_2.c`来测试能否正确调用parallel_for函数：

  和1）中的代码相比只是删去了函数`parallel_for()`，以及结构体`for_index`，加上写好的头文件`#include"parallel_for.h"`

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include"parallel_for.h"

double *A, *B, *C, x = 10;
int thread_num,n;

void Get_args(int argc,char *argv[]){
    ...
}

void * functor (void * args){
    ...
}
```

```
16
17 int main(int argc,char *argv[]){
18     ...
19     parallel_for(0, n, 1, functor, NULL, thread_num);
20     ...
21
22 }
```

编译链接以及运行结果

1. 首先我们先将三个实现重载的函数文件编译为动态链接库，生成编译好的libparallel_for.so文件：

```
1 gcc parallel_for.c --shared -fPIC -o libparallel_for.so
```

2. 然后将`libparallel_for.so`文件挪入一般动态链接库所在的文件夹`/usr/lib/`中

```
1 sudo cp libparallel_for.so /usr/lib/
```

3. 然后编译调用了`parallel_for.h`的测试文件`lab4_3_2.c`：

```
1 gcc lab4_3_2.c -L. -lparallel_for -o lab4_3_2 -pthread
```

4. 然后再查看可执行文件能否找到我的动态链接文件：

```
wwwj@ubuntu:~/lab/lab4$ ldd lab4_3_2
        linux-vdso.so.1 (0x00007ffc451b2000)
        libparallel_for.so => /usr/lib/libparallel_for.so (0x00007f18d5440000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f18d522
1000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f18d4e30000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f18d5844000)
wwwj@ubuntu:~/lab/lab4$
```

5. 运行可执行文件：

```
wwwj@ubuntu:~/lab/lab4$ gcc parallel_for.c --shared -fPIC -o libparallel_for.so
wwwj@ubuntu:~/lab/lab4$ sudo cp libparallel_for.so /usr/lib/
wwwj@ubuntu:~/lab/lab4$ gcc lab4_3_2.c -L. -lparallel_for -o lab4_3_2 -pthread
wwwj@ubuntu:~/lab/lab4$ ./lab4_3_2 2 10
paraller finish
 a[0]=0.000000, b[0]=0.000000, c[0]=0.000000
 a[1]=11.000000, b[1]=1.000000, c[1]=1.000000
 a[2]=22.000000, b[2]=2.000000, c[2]=2.000000
 a[3]=33.000000, b[3]=3.000000, c[3]=3.000000
 a[4]=44.000000, b[4]=4.000000, c[4]=4.000000
 a[5]=55.000000, b[5]=5.000000, c[5]=5.000000
 a[6]=66.000000, b[6]=6.000000, c[6]=6.000000
 a[7]=77.000000, b[7]=7.000000, c[7]=7.000000
 a[8]=88.000000, b[8]=8.000000, c[8]=8.000000
 a[9]=99.000000, b[9]=9.000000, c[9]=9.000000
wwwj@ubuntu:~/lab/lab4$
```

3)

代码

> 完整代码见lab4_3_3.c、lab4_3_3_h.c

- 头文件、一些全局变量以及用来获取命令行参数、输出矩阵的函数：
  - 其中结构体`for_index`是用来存储每个线程执行的开始索引、结束索引以及每次的迭代数量。

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<pthread.h>
4
```

```
int m,n,k,thread_num;
double *A,*B,*C;

struct for_index {
    int start;//A start row
    int end;//A end row
    int increment;
};

void Get_args(int argc,char *argv[]){
    thread_num = strtol(argv[1], NULL, 10);
    m = strtol(argv[2], NULL, 10);
    n = strtol(argv[3], NULL, 10);
    k = strtol(argv[4], NULL, 10);
}

void PRINT(double *x,int m,int n){
    for(int i=0;i<m;++i){
        for(int j=0;j<n;++j) printf("%lf  ",x[i*n+j]);
        printf("\n");
    }
    printf("\n");
}
```

- 构建parallel_for函数对循环分解、分配和执行机制：
  - 第2-3行，首先设置`num_threads`个标识符。并计算可以将这部分运算量划分成`divid`个部分，然后分配到各个线程上去，这里我们是将矩阵A的行进行划分后分配给各个线程运算。
  - 第4-14行，定义`num_threads`个结构体，保存每个线程的循环开始、结束索引以及每次循环增加索引数。给结构体里的变量赋值，这里可能运算量和线程数不能完全恰好被分配，将剩余的全部迭代次数都分配给最后一个线程。
  - 第16-18行然后创建`num_threads`个线程，每个线程都有对应的线程标识符，他们都调用`functor()`函数，并传入之前设置过的该线程计算需要用到的参数；
  - 第20-22行，线程计算结束，释放线程标识符对应的线程占用的资源，本次计算没有返回值，第二个参数为`NULL`；
  - 第24行，输出信息表示并行运算已经结束。

```
void parallel_for(int start, int end, int increment, void *(*functor)(void*), void *arg , int num_threads){
    pthread_t pth[num_threads];
    int divid = (end - start + 1)/num_threads;
    struct for_index thread_assign[num_threads];
    for(int i=0; i<num_threads; ++i){
        thread_assign[i].start = i*divid ;
        if(i == num_threads-1){
            thread_assign[i].end = end;
        }
        else{
            thread_assign[i].end = thread_assign[i].start + divid;
        }
        thread_assign[i].increment = increment;
    }

    for(int i=0; i<num_threads; ++i){
        pthread_create(&pth[i], NULL, functor, &thread_assign[i]);

```

```
    }

    for(int i=0; i<num_threads; ++i){
        pthread_join(pth[i], NULL);
    }

    printf("paraller finish\n");
}
```

- 每个线程中用来调用进行计算的函数 `functor ()`：
  - 首先将收到的参数进行类型转化。
  - 然后根据 `parallel_for()` 分配的循环开始、结束的索引以及循环增加的索引数这些信息进行计算这里的计算方法主要就是运用通用矩阵乘法机型计算计算得到的元素放入数组C的对应位置。

```
void *computer(void *arg){
    struct for_index *thread_assign;
    thread_assign = (struct for_index*)arg;

    for(int i= thread_assign->start; i<thread_assign->end;
i=i+thread_assign->increment){
        for(int j=0; j<k ;++j){
            int temp=0;
            for(int a=0;a<n;++a) temp+=A[i*n+a]*B[a*k+j];
            C[i*n+j] = temp;
        }
    }
}
```

- 主函数：
  - 第2-16行，主要是进行的工作是获取线程数、矩阵大小，并给矩阵按照他们的大小分配空间，并给矩阵赋值。
  - 第18-20行，调用函数 `parallel_for()`，利用pthread进行并行计算。传入的参数分别是矩阵A的行的其实索引和结束索引（我们是利用A的行进行划分实现并行）。
  - 第22-35行进行后续的一些信息输出以及空间释放。

```
int main(int argc,char *argv[]){
    Get_args(argc,argv);

    //定义
    int starttime,finishtime;
    A = (double*)malloc(sizeof(double)*m*n);
    B = (double*)malloc(sizeof(double)*n*k);
    C = (double*)malloc(sizeof(double)*m*k);


    //给矩阵赋值
    srand(time(NULL));
    for(int i=0; i<m; ++i)
        for(int j=0; j<n; ++j) A[i*n+j]=rand()%50;
    for(int i=0; i<n; ++i)
        for(int j=0; j<k; ++j) B[i*k+j]=rand()%50;

    starttime=clock();
```

```
    parallel_for(0, n, 1, computer, NULL, thread_num);//computer the 0~n row of C
    finishtime=clock();

    printf("the number of process is %d, the time cost = %lf s\n",thread_num,(double)(finishtime - starttime)/CLOCKS_PER_SEC);

    //print
        printf("\n");
        printf("A:\n");
        PRINT(A,m,n);
        printf("B:\n");
        PRINT(B,n,k);
        printf("C=A*B:\n");
        PRINT(C,m,k);

    free(A);
    free(B);
    free(C);

    return 0;

}
```

- 也可以直接调用之前写好的头文件`"parallel_for.h"`，来用`parallel_for()`函数，这里只写一个结构，里面的内容和上面介绍的实际上是一样的：

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<time.h>
int m,n,k,thread_num;
double *A,*B,*C;

void Get_args(int argc,char *argv[]){
    ...
}

void PRINT(double *x,int m,int n){
    ...
}

void *computer(void *arg){
    ...
}

int main(int argc,char *argv[]){
    ...
    parallel_for(0, n, 1, computer, NULL, thread_num);//computer the 0~n row of C
    ...
}
```

运算结果

- 下面是编译和运行这段代码的指令（不用头文件"parallel_for.h"）：

```
/* File:     lab4_3_3.c
 *
 * Purpose:  Use pthread_for to implement matrix multiplication
 *
 * Compile:  gcc -o lab4_3_3 lab4_3_3.c -lpthread
 *
 * Run:      ./lab4_3_3 <num> <m> <k> <n>
 *           num is the thread number that we will create
 *           m,k,n is size of matrix A,B,C
 *
 * Input:    none
 * Output:   the cost time. If you want ,you can also print matrix A,B,C.
 */
```

- 运算结果正确，用matlab进行验算：

```
wwwj@ubuntu:~/lab/lab4$ gcc -o lab4_3_3 lab4_3_3.c -fopenmp
wwwj@ubuntu:~/lab/lab4$ ./lab4_3_3 2 5 5 5
paraller finish
completed:
the number of process is 2, the time cost = 0.000864 s


A:
7.000000  9.000000  9.000000  30.000000  14.000000
38.000000  8.000000  3.000000  38.000000  1.000000
32.000000  28.000000  8.000000  41.000000  47.000000
33.000000  27.000000  21.000000  43.000000  4.000000
29.000000  41.000000  7.000000  12.000000  2.000000

B:
30.000000  24.000000  49.000000  11.000000  8.000000
20.000000  18.000000  19.000000  29.000000  0.000000
33.000000  19.000000  11.000000  38.000000  9.000000
12.000000  20.000000  37.000000  22.000000  13.000000
36.000000  5.000000  40.000000  8.000000  0.000000

C=A*B:
1551.000000  1171.000000  2283.000000  1452.000000  527.000000
1891.000000  1878.000000  3493.000000  1608.000000  825.000000
3968.000000  2479.000000  5585.000000  2746.000000  861.000000
2883.000000  2557.000000  4112.000000  2922.000000  1012.000000
2137.000000  1817.000000  2801.000000  2054.000000  451.000000
```

```
A=[7.000000 , 9.000000 ,  9.000000 , 30.000000,  14.000000;
38.000000, 8.000000 ,  3.000000 , 38.000000,  1.000000 ;
32.000000, 28.000000,  8.000000 , 41.000000,  47.000000;
33.000000, 27.000000,  21.000000, 43.000000,  4.000000 ;
29.000000, 41.000000,  7.000000 , 12.000000,  2.000000 ];

B=[30.000000, 24.000000,  49.000000, 11.000000,  8.000000 ;
20.000000, 18.000000,  19.000000, 29.000000,  0.000000 ;
33.000000, 19.000000,  11.000000, 38.000000,  9.000000 ;
12.000000, 20.000000,  37.000000, 22.000000,  13.000000;
36.000000, 5.000000 ,  40.000000, 8.000000 ,  0.000000 ];


C=A*B;
```

令行窗口

```
C =

        1551        1171        2283        1452         527
        1891        1878        3493        1608         825
        3968        2479        5585        2746         861
        2883        2557        4112        2922        1012
        2137        1817        2801        2054         451
```

- 使用头文件"parallel_for.h"，编译指令：

```
/* File:     lab4_3_3_3_h.c
 *
 * Purpose:  Use pthread_for.so to implement matrix multiplication
 *
 * Compile:  gcc lab4_3_3_h.c -L. -lparallel_for -o lab4_3_3_h -pthread
 *
 * Run:      ./lab4_3_3_h <num> <m> <k> <n>
 *           num is the thread number that we will create
 *           m,k,n is size of matrix A,B,C
 *
 * Input:    none
 * Output:   the cost time. If you want ,you can also print matrix A,B,C.
 */
```

- 运行结果正确：