

中山大学数据科学与计算机学院本科生实验 报告

(2020 学年秋季学期)

课程名称：高性能计算程序设计 任课教师：黄聃

年级+班级	18级计科（超算）	专业（方向）	计算机科学与技术（超级计算方向）
学号	18340052	姓名	何泽
Email	heze_heze@icloud.com	完成日期	2020年11月13日

目录

I 实验目的

1. 通过 OpenMP 实现通用矩阵乘法
2. 基于 OpenMP 的通用矩阵乘法优化
3. 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制。

II 实验过程和核心代码

0. 代码整体说明

1. 通过 OpenMP 实现通用矩阵乘法

(1) 算法简述

(2) 串行版本

(3) OpenMP并行化

2. 基于 OpenMP 的通用矩阵乘法优化

(1) 默认调度

(2) 静态调度

(2) 动态调度

3.构造基于 Pthreads 的并行 for 循环分解、分配和执行机制

(1) parallel_for.h

(2) parallel_for.cpp

(3) parallel_for_main.cpp

III 实验结果

1. 通过 OpenMP 实现通用矩阵乘法

(1) 验证算法正确性

(2) 扩大维数

2. 基于 OpenMP的通用矩阵乘法优化

3. 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制

(1) 生成动态链接库

(2) 载入动态链接库

(3) 运行

IV 实验感想

I 实验目的

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并行进程从 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:

$$C = AB$$
$$C_{m,n} = \sum_{k=1}^N A_{m,k} B_{k,n}$$

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

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)` 的性能，实现 `#pragma omp for`，并比较其性能。

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 将基于 OpenMP 的通用矩阵乘法的 `omp parallel for` 并行，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：

```
1 | for ( int i = 0; i < 10; i++ ){  
2 |     A[i]=B[i] * x + C[i]  
3 | }
```

替换为：

```

1  parallel_for(0, 10, 1, functor, NULL, 2);
2  struct for_index {
3      int start;
4      int end;
5      int increment;
6  }
7  void * functor (void * args){
8      struct for_index * index = (struct for_index *) args;
9      for (int i = index->start; i < index->end; i = i + index->increment){
10         A[i]=B[i] * x + C[i];
11     }
12 }

```

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

Thread1: start 和 end 分别为 5, 10

```

1  void * functor(void * arg){
2      int start = my_rank * (10/2);
3      int end = start + 10/2;
4      for(int j = start, j < end, j++)
5          A[j]=B[j] * x + C[j];
6  }

```

II 实验过程 and 核心代码

0. 代码整体说明

- 开头 `#define isprint 0` 这句话定义了一个宏，表示是否打印所有矩阵的元素，为0则不打印，只输出运行时间，为1则打印所有矩阵的所有元素
- 变量 `m,n,k` 分别代表三个矩阵的行数和列数；三个数组 `a,b,c`，`a,b` 为待乘矩阵，`c` 为结果矩阵

1. 通过 OpenMP 实现通用矩阵乘法

(1) 算法简述

$$C = AB$$
$$C_{m,n} = \sum_{n=1}^N A_{m,n} B_{n,k}$$

(2) 串行版本

```
1  for (int i = 0; i < m; i++){
2      for (int j = 0; j < n; j++){
3          double sum = 0;
4          for (int w = 0; w < k; w++){
5              sum += a[i][w] * b[w][j];
6              c[i][j] = sum;
7          }
8      }
```

(3) OpenMP并行化

```
1  #pragma omp parallel for
2  for (int i = 0; i < m; i++){
3      #pragma omp parallel for private(sum,w)
4          for (int j = 0; j < n; j++){
5              double sum = 0;
6              for (int w = 0; w < k; w++){
7                  sum += a[i][w] * b[w][j];
8                  c[i][j] = sum;
9              }
10 }
```

首先在第一个循环前使用子句 `#pragma omp parallel for` 开始并行，第二个循环前为了避免数据竞争将变量 `sum` 和 `w` 声明为 `private`，故使用子句 `#pragma omp parallel for private(sum,w)`，这样就完成了并行化。

2. 基于 OpenMP 的通用矩阵乘法优化

代码框架和前面一致

(1) 默认调度

代码即和前面的一致

```
1 #pragma omp parallel for
2 for (int i = 0; i < m; i++){
3     #pragma omp private(sum,w) parallel for
4         for (int j = 0; j < n; j++){
5             double sum = 0;
6             for (int w = 0; w < k; w++){
7                 sum += a[i][w] * b[w][j];
8             }
9             c[i][j] = sum;
10 }
```

(2) 静态调度

即在 `parallel for` 后面加上 `schedule(static,1)`

```

1 #pragma omp parallel for schedule(static,1)
2 for (int i = 0; i < m;i++){
3 #pragma omp private(sum,w) parallel for schedule(static,1)
4     for (int j = 0; j < n;j++){
5         double sum = 0;
6         for (int w = 0; w < k;w++)
7             sum += a[i][w] * b[w][j];
8         c[i][j] = sum;
9     }
10 }

```

(2) 动态调度

即在 `parallel for` 后面加上 `schedule(dynamic,1)`

```

1 #pragma omp parallel for schedule(dynamic,1)
2 for (int i = 0; i < m;i++){
3 #pragma omp private(sum,w) parallel for schedule(dynamic,1)
4     for (int j = 0; j < n;j++){
5         double sum = 0;
6         for (int w = 0; w < k;w++)
7             sum += a[i][w] * b[w][j];
8         c[i][j] = sum;
9     }
10 }

```

3.构造基于 Pthreads 的并行 for 循环分解、分配和执行机制

此任务共有三个文件：



parallel_for_main.cpp



parallel_for.cpp



parallel_for.h

其中 `parallel_for.h` 与 `parallel_for.cpp` 为库函数的头文件与实现, `parallel_for_main` 为主函数

(1) parallel_for.h

- 此文件为头文件, 声明需要用到的库以及索引结构体与库函数的声明。
- `for_index` 结构体传入每个块的开始、结束与递增量, 因为 `pthread_create` 函数需要传入并行函数的参数, 所以增加 void 型指针传参

```
1 #include <stdio>
2 #include <stdlib>
3 #include <ctime>
4 #include <cmath>
5 #include <pthread.h>
6 #include <iostream>
7 struct for_index{
8     int start;
9     int end;
10    int increment;
11    void *args;
12 };
13 void parallel_for(int start, int end, int increment, void *
    (*functor)(void *), void *arg, int num_threads);
```

(2) parallel_for.cpp

- 此函数为 `.so` 库函数的实现, 即基于 pthreads 的多线程库提供的基本函数构建 `parallel_for` 函数对循环分解、分配和执行。
- `a` 为每个线程对应的 `for_index` 结构体
- `block` 为每个线程需要执行的循环次数

```
1 #include "parallel_for.h"
2 void parallel_for(int start, int end, int increment, void *
    (*functor)(void *), void *arg, int num_threads){
3     pthread_t *threads = (pthread_t *)malloc(num_threads *
    sizeof(pthread_t));
4     for_index *a = (for_index *)malloc(num_threads *
    sizeof(for_index));
```



```

5     int block = (end - start) / num_threads;
6     for (int thread = 0; thread < num_threads; thread++){
7         a[thread].args = arg;
8         a[thread].start = start + thread * block;
9         a[thread].end = a[thread].start + block;
10        if (thread == (num_threads - 1))
11            a[thread].end = end;
12        a[thread].increment = increment;
13        pthread_create(&threads[thread], NULL, functor, (void *)(&a
+ thread));
14    }
15    for (int thread = 0; thread < num_threads; thread++)
16        pthread_join(threads[thread], NULL);
17    free(threads);
18    free(a);
19 }

```

(3) parallel_for_main.cpp

- 此函数为主函数，将基于 OpenMP 的通用矩阵乘法的 `omp parallel for` 并行，改造成基于 `parallel_for` 函数并行化的矩阵乘法。
- 整体代码框架和任务一类似，只是将GEMM通用矩阵乘法进行更改并写成一个独立的函数，在main函数调用以通过 `parallel_for` 函数并行化。
- 首先开头需要引入头文件： `#include "parallel_for.h"`
- 然后将需要调用的参数写成一个结构体

```

1 struct args{
2     int *A,*B,*C,*m,*n,*k;
3     args(int *tA, int *tB, int *tC, int *tm, int *tn, int
      *tk){
4         A = tA;
5         B = tB;
6         C = tC;
7         m = tm;
8         n = tn;
9         k = tk;
10    }
11 };

```

- GEMM函数更改后如下：

```

1 void *gemm(void *arg){
2     struct for_index *index = (struct for_index *)arg;
3     struct args *true_arg = (struct args *)(index->args);
4     for (int i = index->start; i < index->end; i = i +
      index->increment){
5         for (int j = 0; j < *true_arg->k; ++j){
6             int temp = 0;
7             for (int z = 0; z < *true_arg->n; ++z)
8                 temp += true_arg->A[i * (*true_arg->n) + z]
      * true_arg->B[z * (*true_arg->k) + j];
9             true_arg->C[i * (*true_arg->k) + j] = temp;
10        }
11    }
12    return NULL;
13 }

```

- main函数整体框架与前面一致，只是乘法调用改成如下形式：

```

1 | parallel_for(0, m, 1, gemm, arg, thread_count);

```

这样，便完成了通过调用 `parallel_for` 的 `.so` 函数库实现矩阵乘法，生成动态链接库的过程描述详见“实验结果”

III 实验结果

1. 通过 OpenMP 实现通用矩阵乘法

(1) 验证算法正确性

打印所有矩阵元素，可以通过手算证明结果是正确的

```
heze@HeZes-MBP:~/Library/Mobile Documents/com~apple~CloudDocs/大三上/...  
heze@HeZes-MBP > gcc-10 -fopenmp 1.c -o 1  
heze@HeZes-MBP > ./1  
请依次输入m, n, k的值 (范围512~2048) : 2 2 2  
=====  
矩阵A有2行2列 :  
1.00    6.00  
4.00    0.00  
矩阵B有2行2列 :  
0.00    6.00  
2.00    1.00  
=====  
矩阵C有2行2列 :  
12.00   12.00  
0.00    24.00  
OpenMP通用矩阵乘法已完成, 用时: 0.325000 ms.  
heze@HeZes-MBP > ./1  
请依次输入m, n, k的值 (范围512~2048) : 2 2 2  
=====  
矩阵A有2行2列 :  
0.00    2.00  
6.00    1.00  
矩阵B有2行2列 :  
3.00    1.00  
9.00    2.00  
=====  
矩阵C有2行2列 :  
18.00   4.00  
27.00   8.00  
OpenMP通用矩阵乘法已完成, 用时: 0.360000 ms.
```

(2) 扩大维数



The terminal window shows the execution of a program that performs matrix multiplication using OpenMP. The user enters three sets of dimensions (m, n, k) and the program outputs the completion time for each. The dimensions are 512, 1024, and 2048. The times are 653.421000 ms, 10628.764000 ms, and 142608.782000 ms respectively.

```
heze@HeZes-MBP:~/Library/Mobile Documents/com~apple~CloudDocs/大三上/...  
heze@HeZes-MBP > gcc-10 -fopenmp 1.c -o 1  
heze@HeZes-MBP > ./1  
请依次输入m, n, k的值 (范围512~2048) : 512 512 512  
OpenMP通用矩阵乘法已完成, 用时: 653.421000 ms.  
heze@HeZes-MBP > ./1  
请依次输入m, n, k的值 (范围512~2048) : 1024 1024 1024  
OpenMP通用矩阵乘法已完成, 用时: 10628.764000 ms.  
heze@HeZes-MBP > ./1  
请依次输入m, n, k的值 (范围512~2048) : 2048 2048 2048  
OpenMP通用矩阵乘法已完成, 用时: 142608.782000 ms.  
heze@HeZes-MBP > |
```

2. 基于 OpenMP的通用矩阵乘法优化

三种调度方式结果如下：

```
heze@HeZes-MacBook-Pro:~/Library/Mobile Documents/com~apple~Cloud...
heze@HeZes-MacBook-Pro > gcc-10 -fopenmp 2.c -o 2
heze@HeZes-MacBook-Pro > ./2
请依次输入m, n, k的值 (范围512~2048) : 512 512 512
OpenMP通用矩阵乘法默认任务调度已完成, 用时: 862.244000 ms.
OpenMP通用矩阵乘法静态调度已完成, 用时: 860.533000 ms.
OpenMP通用矩阵乘法动态调度已完成, 用时: 851.449000 ms.
heze@HeZes-MacBook-Pro > ./2
请依次输入m, n, k的值 (范围512~2048) : 1024 1024 1024
OpenMP通用矩阵乘法默认任务调度已完成, 用时: 13358.023000 ms.
OpenMP通用矩阵乘法静态调度已完成, 用时: 12401.299000 ms.
OpenMP通用矩阵乘法动态调度已完成, 用时: 12236.620000 ms.
heze@HeZes-MacBook-Pro > ./2
请依次输入m, n, k的值 (范围512~2048) : 512 512 512
OpenMP通用矩阵乘法默认任务调度已完成, 用时: 871.107000 ms.
OpenMP通用矩阵乘法静态调度已完成, 用时: 879.387000 ms.
OpenMP通用矩阵乘法动态调度已完成, 用时: 858.828000 ms.
```

可以看到，其实三种调度方式的时间大体上差不多，动态调度的时间会稍稍少一些，在我看来是因为生成的矩阵元素都差不多，CPU执行并不会因为不同的数字而产生太大的差距，CPU负载较为平均，这就导致其实各种调度的运行时间不会有太大差别；而由于动态调度会使负载更为平均，所以时间会稍微少一些。

3. 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制

(1) 生成动态链接库

- 首先编译 `parallel_for.cpp`

```
1 | g++-10 -c -fPIC -o parallel_for.o parallel_for.cpp -lpthread
```

- `-c` 表示只编译而不连接
- `-o` 选项用于说明输出文件名
- `-fPIC` 表示编译为位置独立的代码；不用此选项的话编译后的代码是位置相关

的，因此动态载入时是经过代码拷贝的方式来满足不一样进程的需要，而不能达到真正代码段共享的目的

- 此步骤将生成 `parallel_for.o`

- 生成 `.so` 动态链接库

```
1 | g++-10 -shared -o libparallel_for.so parallel_for.o
```

- `-shared` 表示生成一个动态链接库（让链接器生成T类型的导出符号表，有时候也生成弱链接W类型的导出符号），不用该标志外部程序没法链接至一个可执行文件

(2) 载入动态链接库

- 为了使用动态链接库，编译器需要知道 `.h` 文件位置
 - 对于 `#include "..."`，编译器会在当前路径搜索 `.h` 文件，也可以使用 `-I` 选项提供额外的搜索路径
 - 对于 `#include <...>`，编译器会在默认 `include` 搜索路径中寻找
- 编译器还需要知道我们用了哪个库文件，在 `g++` 中：
 - 使用 `-l` 选项说明库文件的名字（库文件名开头去掉 `lib`）
 - 使用 `-L` 选项说明库文件所在的路径（这里我使用了 `-L.` 表示当前路径）
 - 如果没有 `-L` 选项，`g++` 将在默认库文件搜索路径中寻找
- 因为我的链接库和头文件都在同一目录下，所以我的编译命令如下：

```
1 | g++-10 -o parallel_for parallel_for_main.cpp -lparallel_for -L.
```

之后便可正常运行

(3) 运行

使用以上命令生成链接库并使用、运行截图如下：

```
..算实验/实验4 (-zsh)
heze@HeZes-MacBook-Pro ~:大三上/高性能计算实验/实验4
$ g++-10 -c -fPIC -o parallel_for.o parallel_for.cpp -lpthread
$ g++-10 -shared -o libparallel_for.so parallel_for.o
$ g++-10 -o parallel_for parallel_for_main.cpp -lparallel_for -L.
$ ./parallel_for
请依次输入m, n, k的值: 2 2 2
=====
矩阵A有2行2列:
3 9
7 7
矩阵B有2行2列:
0 4
7 4
=====
矩阵C有2行2列:
63 48
49 56
Parallel_for通用矩阵乘法已完成, 用时: 0.403000 ms.
heze@HeZes-MacBook-Pro ~:大三上/高性能计算实验/实验4
```

可以看到结果正确，可见动态链接库生成成功，可以正常调用运行。

IV 实验感想

此次实验熟悉了OpenMP的编程方法，但是由于矩阵乘法这个例子的CPU负载较为平均，所以三种调度对性能并没有什么显著对差距，之后通过使用Pthread实现类似OpenMP的 `parallel for` 更加熟悉了Pthread的编程方法与生成动态链接库的操作。