中山大学数据科学与计算机院本科生实验报告

(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**　　**批改人：**

年级+班级	18 级计科 8 班	专业（方向）	计科(超级计算)
学号	18340224	姓名	钟赫明
Email	zhonghm5@mail2.sysu.edu.cn	完成日期	9 月 24 日

1. 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：

$$C = AB$$

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

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

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：

```
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

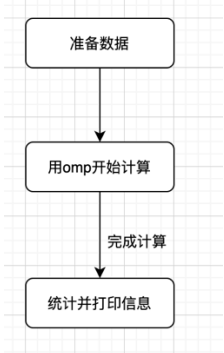
Thread1: start 和 end 分别为 5, 10

```
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];
}
```

2. 实验过程和核心代码

首先是问题 1 的实现过程。根据题目要求，这回我们要基于 OpenMP 来实现并行化的通用矩阵乘法。

该实现原理与之前用 MPI、Pthreads 的实现是基本完全一致的，整个程序的执行流程也是几乎没有变过（甚至 OpenMP 提供的 API 还让代码变得更加简洁）。流程依旧是： 准备数据->计算->打印数据。用流程图表示如下:



接下来我们分别给出这三个步骤的关键代码实现:

1.首先是数据准备。这里要做的事就是获取数据的输入（M、N、K、线程数），然后为矩阵开辟内存空间，并对开辟的空间进行随机化（矩阵 A、B）。这部分的代码实现如下：

首先是变量定义和输入：

```
// 准备数据
int * A, * B, * C;  //A: M*K B: K*N C: M*N
int M, N, K;
int thread_num = 4;
printf("请输入 M、N、K 的值:");
scanf("%d%d%d", &M, &N, &K);
printf("请输入指定的线程数量:");
scanf("%d", &thread_num);
```

然后是 A、B 矩阵的随机化，这里还是使用 srand+rand 的随机化方法：

```
// randomlize
srand(time(0));
A = new int[M * K];
for (int i = 0; i < M * K ; i++) {
    A[i] = rand() % 10;
}
B = new int[K * N];
for (int i = 0; i < K * N ; i++) {
    B[i] = rand() % 10;
}
C = new int[M * N];
```

这些就是数据准备部分的代码实现。

2.然后是计算部分，这里计算的实现十分简单，只需要先实现一个串行的三层嵌套 for 循环，然后用#pragma omp parallel for 语句将最外层循环并行化即可。同时，在计算过程中需要统计计算花费的时间，这里用 OpenMP 提供的函数 omp_get_wtime()来统计时间。具体代码实现如下：

```
double start = omp_get_wtime();
#pragma omp parallel for num_threads(thread_num)
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        int temp = 0;
        for (int k = 0; k < K; k++) {
            temp += A[i * K + k] * B[k * N + j]; // +=Aik * Bkj
        }
        C[i*N + j] = temp;
    }
}
double end = omp_get_wtime();
```

这就是计算部分的代码实现。

3.最后是信息统计和打印部分。这一部分的工作就是统计各种信息，并且将它们打印出来，比如，将 A、B、C 三个矩阵的值按照矩阵的格式打印出来，还有将 M、N、K、线程数、计算时间以结果的形式打印出来。具体的代码实现如下：

```
// 统计并打印信息
double time = end - start;
printf("矩阵 A 的值:\n");
for (int i = 0; i < M; i++) {
```

```
    for (int j = 0; j < K; j++) {
        printf("%d ", A[i * K + j]);
    }
    printf("\n");
}
printf("矩阵 B 的值:\n");
for (int i = 0; i < K; i++) {
    for (int j = 0; j < N; j++) {
        printf("%d ", B[i * N + j]);
    }
    printf("\n");
}
printf("矩阵 C 的值:\n");
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        printf("%d ", C[i * N + j]);
    }
    printf("\n");
}
printf("总结:\n");
printf("M: %d\nN: %d\nK: %d\n", M, N, K);
printf("线程数量: %d\n", thread_num);
printf("计算时间: %lfs\n", time);
```

可以看到，这里代码先计算出 GEMM 花费的时间，然后分别打印矩阵 A、B、C 的值，最后将 M、N、K、线程数和计算时间作为统计信息打印出来。

这就是数据统计和打印部分的实现代码。

以上就是问题 1 的全部实现过程。

然后是问题 2 的实现。根据问题要求，我们需要用#pragma omp for 实现并行 GEMM，并通过使用不同的调度机制，来对问题 1 中实现的 OpenMP GEMM 进行优化，并比较不同调度机制下的性能差异。

可以看到，在问题 1 的实现代码中我们已经用上了#pragma omp for，所以这里我们就不再需要去对问题 1 的原有代码进行任何修改了。也就是说，我们问题 2 的代码实现可以完全基于问题 1 的代码，而只需要修改调度方式 schedule 就行了。

所以我们在问题 2 中的实现就是：完全复用问题 1 的代码，然后在#pragma omp for 语句后加上不同的 schedule：

```
//默认调度
#pragma omp parallel for num_threads(thread_num)
//静态调度
#pragma omp parallel for num_threads(thread_num) schedule(static, 1)
//动态调度
#pragma omp parallel for num_threads(thread_num) schedule(dynamic, 1)
```

当然，这里我为了方便进行比较，我将这三种调度都放入到一个源文件 OpenMP_gemm_opt.cpp 中，该程序会用这三种调度都算一遍，并分别统计其计算花费的时间。

以上就是问题 2 的全部实现过程。

然后是问题 3 的实现过程。根据问题要求，我们需要用 Pthread 提供的函数实现一个 parallel_for 函数，其功能类似于#pragma omp parallel for，也是对循环进行拆分和并

行化的。实现了 parallel_for 之后，我们还需要按照题目要求，将该函数编译成一个 so 文件，然后对之前的 OpenMP gemm 进行修改，用通过 so 文件调用的 parallel_for 函数代替之前使用的#pragma omp for 再完成一遍通用矩阵乘法实现。

问题 3 分为三个小问题，我们一个个来实现。

首先是第一小问。根据小问要求我们要实现一个基于 pthreads 的 for 循环拆分并行的函数 parallel_for。题目中已经给出了函数的参数及其含义，所以我们可以首先确定 parallel_for 函数的声明如下：

```
1.  void parallel_for(
2.      int start,
3.      int end,
4.      int increment,
5.      void *(*functor)(void*),
6.      void *arg,
7.      int num_threads);
```

但是根据我最终确定的思路，传入的用户函数 functor 只应该负责“for 循环”内部的代码执行，而这通常是离不开“for 循环”的当前 index 的，所以我决定给用户函数 functor 加上一个表示当前 index 的参数 i。那么 parallel_for 的函数头就变成了如下所示：

```
1.  void parallel_for(
2.      int start,
3.      int end,
4.      int increment,
5.      void *(*functor)(void*, int),
6.      void *arg,
7.      int num_threads);
```

在题目给出的例子中，对部分 for 循环的迭代是放在用户函数 functor 中完成的，我觉得这一点可以被优化。因此如上所述，我让 functor 只负责“for 循环”内部的计算。而部分“for 循环”的执行则交给一个新定义的函数__thread_function 来完成。这个是在库文件 Pthread_for.cpp 中定义的，也是 parallel_for 用于创建 pthread 线程时使用的函数，它是不对用户开放的内部函数。这里使用__thread_function 的目的就是尽量遵循库函数的设计思想，即尽量向用户隐藏内部实现原理，并保证接口的简单、易用。

在介绍__thread_function 的实现之前，我们需要先介绍它要用到的参数。正如前面所说，__thread_function 被用于创建 pthread 线程，所以它的参数和返回值都是一个 void*类型。在__thread_function 中，由于需要执行拆分后的一个部分的 for 循环、需要调用用户函数 functor，所以我们需要通过参数向__thread_function 传递 for 循环的部分起点 start、部分终点 end 和增量 increment，以及用户函数的函数指针，还有用户函数可能会用到的参数。与大多数 pthread 函数实现一样，这里我们需要建立一个结构体__for_index 来进行参数传递。同样__for_index 也是在库文件中实现的、不对用户开放的。__for_index 的实现如下：

```
1.  struct __for_index
2.  {
3.      void* arg;                  // functor 的真正的"参数"
4.      int start;                  // functor 负责的循环的起始位置
5.      int end;                    // functor 负责的循环的结束位置
6.      int increment;              // 循环增量
```

```
    void* (*functor)(void*, int);   // user function
};
```

确定了参数的传递之后，我们就可以开始进行__thread_function 的实现了：

```
void * __thread_function(void *x) {
    __for_index * index = (__for_index*)x;
    void * arg = index->arg;
    int start = index->start;
    int end = index->end;
    int increment = index->increment;
    void *(*functor)(void*, int) = index->functor;

    // 部分 for 循环
    for (int i = start; i < end; i+=increment) {
        functor(arg, i); //调用循环体
    }
    return NULL;
}
```

可以看到，__thread_function 的代码做的工作正如我们之前所说：作为每个并行线程的执行载体，内部实现了子线程的部分循环执行，并且在这个过程中调用了用户函数。

有了上面的一系列实现之后，我们就可以开始进行 parallel_for 的实现了。按照我们的思路 parallel_for 的执行流程应该是：分割循环->准备数据，用__thread_function 创建目标数量个 pthread->检查是否有分配剩下的循环，如果有，在主线程内计算掉->pthread_join 结束子线程执行->结束。确定了流程之后，我们就可以给出 parallel_for 的代码实现了：

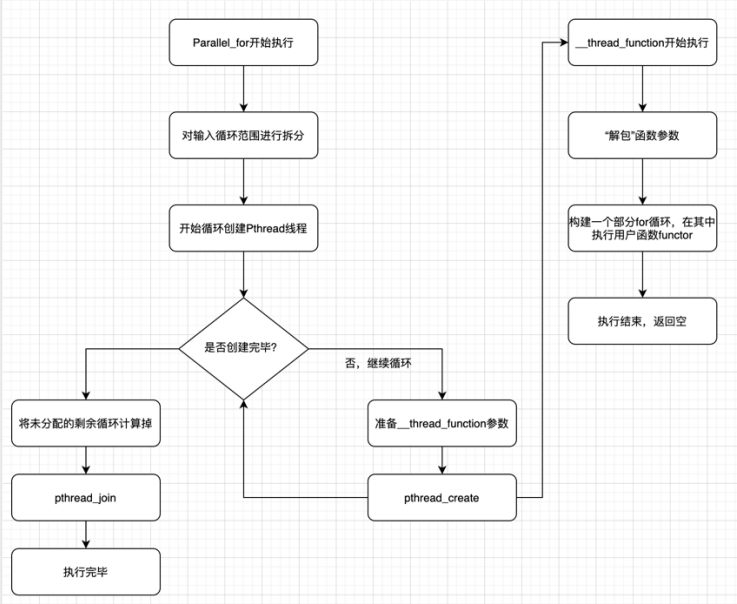
```
void parallel_for(
    int start,
    int end,
    int increment,
    void *(*functor)(void*, int),
    void *arg,
    int num_threads) {
        int iter_num = (end-start) / increment;
        int part = iter_num / num_threads;
        __for_index*  index_list = new __for_index[num_threads];
        // thread create
        pthread_t * handlers = new pthread_t[num_threads];

        for (int i = 0; i < num_threads; i++) {
            index_list[i].arg = arg;
            index_list[i].start = i * part;
            index_list[i].end = index_list[i].start + part;
            index_list[i].increment = increment;
            index_list[i].functor = functor;
            pthread_create(handlers+i, NULL, __thread_function, (void*)(index_list+i));
        }
        if(part * num_threads < iter_num) { //有剩余
            for (int i = part * num_threads * increment; i < iter_num * increment; i+=in
crement) {
                functor(arg, i);
            }
        }
        for (int i = 0; i < num_threads; i++) {
            pthread_join(handlers[i], NULL);
        }
        delete [] index_list;
        delete [] handlers;
}
```

可以看到，parallel_for 的代码实现基本上是按照前面提到的原理来实现的。

注：上面的代码中，parallel_for 的声明和__for_index 的定义是在头文件 pthread_for.h 中实现的，而__thread_function 和 parallel_for 的定义都是在对应源文件 pthread_for.cpp 中实现的。

到这里，parallel_for 的实现就基本完成了，小问一要求的内容也就基本上实现完毕了。最后作为总结再给出一张 parallel_for 执行的流程图：



以上就是第一小问的实现内容。

然后是第二个小问。题目要求我们将第一小问中实现的 parallel_for 函数在 Linux 下编译成 so 文件。像之前我们在实验二中做过的那样，我们可以创建一个 makefile 来自动化管理编译生成操作，其内容如下：

```
# 问题 3 编译并测试
dytest: Pthread_test
    LD_LIBRARY_PATH=. ./Pthread_test
Pthread_test: Pthread_test.cpp libPthread_for.so
    g++ Pthread_test.cpp -o Pthread_test -L. -lPthread_for
libPthread_for.so: Pthread_for.cpp
    g++ -fPIC -shared Pthread_for.cpp -o libPthread_for.so
```

可以看到，这里将动态库和测试源文件的编译和测试源文件的链接执行都整合到一个 make 指令中，只要在终端执行一个命令 make dytest 就可以完成编译+执行的工作。Makefile 的实现简化了我们的编译+测试的流程。

这就是第二小问的实现。

最后是第三个小问。根据要求，这里我们需要将问题 1 中的 OpenMP gemm 程序通过我们刚实现的 parallel_for 函数重新实现一遍。具体操作就是：将原先的#pragma omp parallel for 语句换成 parallel_for 的函数调用，然后将内部两层循环的代码内容转移到函数 mul 中，作为 parallel_for 的用户函数。此外，为了方便代码实现，将矩阵 A、B、C 和 M、N、K 修改为全局

变量（其实作为函数 mul 的参数传入更合适，但是为了节省工作量这里还是选择使用全局变量）。最后将原来的 OpenMP 时间测量函数 omp_get_wtime 换成 gettimeofday 函数。以上就是全部的代码修改内容，其他部分的代码可以完全不用修改直接复用。修改部分的代码汇总如下：

mul 用户函数实现：

```
void* mul(void * x, int i) {
    for (int j = 0; j < N; j++) {
        int temp = 0;
        for (int k = 0; k < K; k++) {
            temp += A[i * K + k] * B[k * N + j]; // +=Aik * Bkj
        }
        C[i*N + j] = temp;
    }
    return NULL;
}
```

全局变量：

```
int * A, *B, *C;
int M,N,K;
```

main 函数中的修改部分代码：

```
struct timeval start, end;
gettimeofday(&start, NULL);
parallel_for(0, M, 1, mul, NULL, thread_num);
gettimeofday(&end, NULL);
double time_use = (end.tv_sec-start.tv_sec)*1000000+(end.tv_usec-start.tv_usec);
```

这些就是第三小问的实现内容。到这里问题 3 的三个小问就已经全部实现完成了。

以上就是本次实验的全部实现过程。

3. 实验结果

首先是第一个问题的实验结果。

还是先给出一张 M、N、K 都为 512，线程数量为 4 时的程序执行结果截图如下（同样因为输出结果中矩阵过大只给出部分截图）：

```
9 9938 10105 10366 10567 11012 10355 1
435 10680 10163 10157 10778 10180 1059
3 9904 10522 10185 10372 10781 10433 1
总结：
M: 512
N: 512
K: 512
线程数量: 4
计算时间: 0.214320s
(base)
```

与之前的多次 GEMM 实现类似，这里同样给出 M、N、K 都为 512、1024、2048 时，线程数量分别为 1、2、4、8、16 时的计算时间统计表。表格内容如下：

矩阵\线程	1	2	4	8	16
512	0.758645s	0.391447s	0.216472s	0.117727s	0.107948s
1024	7.001287s	3.123060s	1.661300s	0.866676s	0.900823s
2048	94.178074s	39.498311s	16.106706s	8.837879s	7.741025s

以上就是问题 1 的实验结果。

然后是第二个问题的实验结果。

还是先给出一张 M、N、K 都为 512，线程数量为 8 时的程序执行结果截图（因为同样的原因也是部分截图）：

```
 10201 10228 10205 10002 10134 10701
87 10982 10469 10379 10470 9774 10443
总结：
M: 512
N: 512
K: 512
线程数量： 8
default调用计算时间： 0.119345s
static调度计算时间： 0.115195s
dynamic调度计算时间： 0.113209s
```

可以看到这里程序的执行结果总结中，除了 M、N、K、和线程数量信息之外，还给出了三种调度模式下各自的计算时间，直观地显示出三种调度模式下的性能差异。

与问题 1 中类似，这里我们同样将 M、N、K 的值设置为 512、1024、2048、4096；线程数从 1-16 下的三种调度模式的执行时间表。

首先是 M、N、K 值都为 512 下的三种调度模式的执行时间表：

调度\线程	1	2	4	8	16
默认	0.756638s	0.397175s	0.211421s	0.119345s	0.105202s
静态	0.749991s	0.394538s	0.206537s	0.115195s	0.106386s
动态	0.763387s	0.394217s	0.207389s	0.113209s	0.102245s

然后是 M、N、K 值都为 1024 下的三种调度模式的执行时间表：

调度\线程	1	2	4	8	16
默认	6.107281s	3.300482s	1.642288s	0.841698s	0.876181s
静态	6.179653s	3.198505s	1.625174s	0.848382s	0.885920s
动态	7.246681s	3.094996s	1.610620s	0.842276s	0.884821s

接着是 M、N、K 值都为 2048 下的三种调度模式的执行时间表：

调度\线程	1	2	4	8	16
默认	116.636661s	37.383416s	16.347519s	8.790640s	8.088985s
静态	114.888830s	36.856444s	16.029363s	8.372890s	7.399074s
动态	115.163578s	36.747911s	16.027520s	8.338125s	7.394468s

最后是 M、N、K 值都为 4096 下的三种调度模式的执行时间表：

调度\线程	1	2	4	8	16
默认	x	737.781989s	328.113267s	177.922439s	170.551579s

静态	x	750.310956s	331.723085s	166.296230s	169.231650s
动态	x	747.464832s	331.816760s	165.519389s	168.291813s

通过以上数据我们可以看出，在线程数偏少、计算规模偏小的情况下，三种调度方法在计算时间上不分伯仲，排除误差带来影响之外，并未形成比较明显的差距，但是在数据规模偏大、线程数量偏多的情况下，计算时间上动态<静态<默认的形式就比较明显了（但其实差距也不是很大）。

以上就是问题 2 的全部实验结果。

然后是第三个问题的实验结果。当 M、N、K 和线程数量都为 4 的情况下，用于测试的 GEMM 的执行结果如下图所示：

```
$ ./Pthread_test
请输入M、N、K的值:4 4 4
请输入指定的线程数量:4
矩阵A的值:
2 7 6 8
0 9 7 1
8 1 8 4
8 9 3 9
矩阵B的值:
4 4 3 4
0 5 2 2
4 3 4 9
9 8 2 6
矩阵C的值:
104 125 60 124
37 74 48 87
100 93 66 130
125 158 72 131
总结:
M: 4
N: 4
K: 4
线程数量: 4
计算时间: 0.000127s
(base)
```

通过检验 A、B、C 的值可以确定，测试程序算出的结果是正确的。然后再测试一下 M、N、K 为 2048、线程数量为 16 时的执行结果，判断一下程序的并行情况：

```
总结:
M: 2048
N: 2048
K: 2048
线程数量: 16
计算时间: 5.808202s
(base)
```

通过该计算时间可以判断出，程序的并行情况十分良好（甚至比 OpenMP 下的时间开销更低）。

以上就是本次实验的全部实验结果。

4. 实验感想

本次实验中所做的工作大部分都是之前就接触过的（指用 OpenMP 重新实现一次 GEMM），所以这部分实现（指问题 1、2）没有遇到什么太大的问题。但是问题 3 的内容确实花了我一定的时间来理解和完成。主要是因为题目描述中给出的示例不是很明确，所以我花了一定的时间来思考和确定具体的实现方案。

最开始的时候我选择的问题 3 实现方案并不是向报告前面介绍的那样，用户函数与 parallel_for 函数基本隔离的。而是让用户函数承担__pthread_function 的工作，即完成部分 for 循环的执行，具体代码在 Pthread_test.cpp 和 Pthread_for.cpp 中以注释的形式保留，可以参考该代码来理解其具体实现。虽然这样实现不用给用户函数添加额外的参数，但是这样导致的后果就是用户在设计自己使用的函数的时候需要考虑如何兼容 parallel_for 库函数，

这样无疑是增加了 parallel_for 作为一个库函数的使用难度的，所以虽然这种实现方式也是可行的，但是明显不如我最后选择的那种实现方式。

上述就是我在本次实验过程中遇到的比较突出的问题。至于其他部分，因为对 OpenMP 框架的了解和 GEMM 的熟悉，所以这次实验基本上没有遇到太大的问题。

以上就是本次实验的全部内容。