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	2018 级五班	专业（方向）	计算机科学与技术
学号	18340126	姓名	罗仁良
Email	luorliang@mail2.sysu.edu.cn	完成日期	2020 年 11 月 12 日

1. 实验目的

1.1 通过 OpenMP 实现通用矩阵乘法

1.2 基于 OpenMP 的通用矩阵乘法优化

1.3 基于 Pthread 的并行 for 循环分解、分配和执行机制

2. 实验过程和核心代码

2.1 OpenMP 实现通用矩阵乘法

2.1.1 数据存储

矩阵数据使用一位数组保存矩阵，为提高高速缓存的命中率，部分矩阵采用存储转置矩阵的方式。

2.1.2 并行计算

```
#pragma omp parallel for num_threads(num_threads)
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < k; j++) {
            C[i*k+j] = 0;
            for(int l = 0; l < n; l++) {
                C[i*k+j] += A[i*n+l]*B[j*n+l];
            }
        }
    }
```

注意到这里一共有三层嵌套循环，只对最外层的循环进行了并行化处理。主要的原因是大型矩阵的规模很大，若对内层的循环进行展开，会导致每一次进入内层循环就会创建新的线程，最后导致线程过多，额外的线程创建调度开销过大，得不偿失。

2.2 OpenMP 通用矩阵乘法优化

主要实验内容是比较不同的任务调度方式对程序性能的影响，具体的结果见实验结果，其中有详细的分析。

```
    #pragma omp parallel for num_threads(num_threads) \
    schedule(static,1)
        for(int i = 0; i < m; i++) {
            for(int j = 0; j < k; j++) {
                C[i*k+j] = 0;
                for(int l = 0; l < n; l++) {
                    C[i*k+j] += A[i*n+l]*B[j*n+l];
                }
            }
        }
```

2.3 基于 Pthreads 的并行 for 循环分解、分配和执行机制

2.3.1 parallel_for 函数

参数

- start：循环初始索引
- end：循环结束索引 **注：一般默认索引取值区间**[start, end)
- stride：每次循环索引增加的步长
- 函数 function：循环体代码
- arg：函数 function 的参数
- num_threads:并行的线程数

```
void parallel_for(int start, int end, int stride, void *
(*function)(void *), void * arg, int num_threads) {
        // 平均分配任务
        // 总任务量
    int total = (end - start) / stride;
    int q = total /  num_threads;
    int r = total %  num_threads;
        // 每个线程的任务
    int count;

    pthread_t * threads = malloc(sizeof(pthread_t)*num_threads);
    struct arg_index * parg = malloc(sizeof(struct
arg_index)*num_threads);

        for(int i=0; i<num_threads; i++) {
            // 初始化线程入口参数 （任务分配）
            if(i < r) {
                count = q + 1;
                parg[i].first = i*count;
            }
            else {
                count = q;
```

```
            parg[i].first = rank*count + r;
        }
        parg[i].last = parg[i].first + count;
        parg[i].stride = stride;
        // 创建子线程
        pthread_create(threads+i,NULL,function,parg+i);
    }

    for (int i = 0; i < num_threads; i++){
        // 等待子线程运行结束
        pthread_join(threads[i], NULL);
    }

    free(threads);
    free(parg);

}
```

为保证负载均衡，根据循环的总次数和并行线程数量对每个线程执行的循环次数进行平均分配。

2.3.2 库函数封装

1） 编译生成库函数

使用编译指令，将源文件编译成一个动态连接库

```
gcc -fPIC -shared parallelfor.c - o libparallelfor.so
```

- -fPIC 生成位置无关代码
- -shared 生成 so 共享库

注： 在 linux 和 unix 中的 so 文件，其扩展名必须是 so，文件前缀也必须是 lib

2） 使用库函数

- 用户源文件需要包含对应库的头文件
- 使用编译器链接时添加 -Lpath （库函数目录）和 -llibname（libname 不含前缀 lib 和后缀.so）

```
gcc test.c -L. -lparallelfor -lphtread -o test
```

注：若出现错误：cannot open shared object file: No such file or directory 把生成的库文件复制到/usr/lib/目录下即可

2.3.3 基于 parallel_for 函数并行化的矩阵乘法

借助 parallelfor 函数并行化 for 循环，通过调用前面封装的库函数即可。这里主要需要写的是循环主体相关的代码段。

```
    struct arg_index
    {
        int first;
        int last;
        int stride;
    };


    void * mat_mul(void * index) {
        struct arg_index *pindex = (struct arg_index *) index;
        for(int i=pindex->first; i < pindex->last; i +=
pindex->stride){
            for(int j = 0; j < k; j++) {
                C[i*n+j] = 0;
                for(int l = 0; l < n; l++) {
                    C[i*n+j] += A[i*n+j]*B[j*n+l];
                }
            }
        }
        return NULL;
    }

    . . .

    // 调用库函数
    parallel_for(0,m,1,mat_mul,NULL,num_threads);
```

为了简化参数传递，将矩阵数据声明为全局变量，实现所有线程共享。

3. 实验结果

3.1 OpenMP 通用矩阵乘法

时间（s） 矩阵大小 / 线程数量	512	1024	2048
1	0.429133	3.472808	27.918002
2	0.216224	1.735803	14.310483
4	0.109426	0.875043	7.250316
8	0.091639	0.643045	5.968971

对比上一个 pthread 实验中的结果，使用 OpenMP 的性能要稍微差一点，但差距并不大，主要原因可能是 OpenMP 函数调度过程中的额外开销。从表格的结果可以看出，1-2-4 的线程数量和计算时间提速成正比关系，但是 4 线程到 8 线程不符这一规律。该结果的主要原因是 CPU 的核心和线程数限制，我在 CPU 只支持 6 线程，8 线程程序运行时，8 个线程并没有实现同一时间运行，只是通过调度实现了并发而已所以加速比没有相应提高。

```
 lab4  → ./p1 2048 2048 2048 1
m n k number of threads: 2048 2048 2048 1
time: 27.918002 s
 lab4  → ./p1 2048 2048 2048 2
m n k number of threads: 2048 2048 2048 2
time: 14.310483 s
 lab4  → ./p1 2048 2048 2048 4
m n k number of threads: 2048 2048 2048 4
time: 7.250316 s
 lab4  → ./p1 2048 2048 2048 8
m n k number of threads: 2048 2048 2048 8
time: 5.968971 s
```

3.2 不同调度方式性能比较

实验过程中统一使用四线程并行。

时间（s） 调度方式 / 矩阵大小	默认调度	静态调度	动态调度
512	0.109483	0.107774	0.108502
1024	0.881733	0.873051	0.867641
2048	7.086199	7.040557	6.963594

从实验结果中可以看到，三种方式的性能差距并不是特别大。这与每次循环的任务量有一定的关系。在本次实验中，每次循环的任务是均衡的，三种调度方式都是能够负载均衡的，所以性能差距不大。当面临一些，任务不均衡，动态变换的过程，需要根据实际情况采取不同的调度方式。

3.3 parallel_for 函数

3.3.1 样例测试

循环体代码段：向量 A + B = C

```
    void * code(void * index) {
        struct arg_index *pindex = (struct arg_index *) index;
        printf("first:%d\n",pindex->first);
        for(int i=pindex->first; i < pindex->last; i +=
pindex->stride){
            C[i] = A[i] + B[i];
        }
    }
```

库函数调用

```
    parallel_for(0,11,1,code,NULL,4);
```

```
 lab4  → gcc libtest.c  -lparallelfor -lpthread -o test
 lab4  → ./test
A:
0 1 2 3 4 5 6 7 8 9 10 11
B:
0 1 2 3 4 5 6 7 8 9 10 11
result:
0 2 4 6 8 10 12 14 16 18 20 22
```

3.3.2 parallel_for 并行计算矩阵乘法

```
 lab4  → ./p3 1024 1024 1024 1
m n k number of threads: 1024 1024 1024 1
time: 3669.926000 ms
 lab4  → ./p3 1024 1024 1024 2
m n k number of threads: 1024 1024 1024 2
time: 1849.948000 ms
 lab4  → ./p3 1024 1024 1024 4
m n k number of threads: 1024 1024 1024 4
time: 937.327000 ms
 lab4  → ./p3 1024 1024 1024 8
m n k number of threads: 1024 1024 1024 8
time: 699.945000 ms
```

结果和使用 OpenMP 的结果相近。

4. 实验感想

经过本次实验，基本掌握了使用 OpenMP 的编程方法，熟悉了不同的调度方式，简单的比较了之间的不同。同时结合使用 pthread 实现 for 循环的并行化，进一步加深了对 OpenMP 内部机制的理解。