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7	专业（方向）	计科
学号	18340166	姓名	王若琪
Email	wangrq29@mail2.sysu.edu.cn	完成日期	2020.10.30

1. 实验目的

- 通过 OpenMP 实现通用矩阵乘法的并行版本，OpenMP 并行线程从 1 增加至 8, 矩阵规模从 512 增加至 2048。
- 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。
- 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制。
    - 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。
    - 在 Linux 系统中将 parallel_for 函数编译为.so 文件,由其他程序调用。
    - 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)。

2. 实验过程和核心代码

2.1 OpenMP 实现通用矩阵乘法的并行版本

在原本的通用矩阵乘法的外层循环进行并行，关键代码如下：

```
void gemm_parallel(int **matrixA, int **matrixB, int **matrixC)
{ //OpenMP并行矩阵乘法
#pragma omp parallel for num_threads(ThreadNumber)
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < K; j++)
        {
            matrixC[i][j] = 0;
            for (int l = 0; l < N; l++)
```

```
            {
                matrixC[i][j] += matrixA[i][l] * matrixB[l][j];
            }
        }
    }
}
```

计时采用 gettimeofday(&t, NULL) 函数，更加准确，关键代码如下：

```
#define GET_TIME(now)                              \
    {                                              \
        struct timeval t;                          \
        gettimeofday(&t, NULL);                    \
        now = t.tv_sec + t.tv_usec / 1000000.0; \
    }
```

其余细节请参照源文件 gemm_1.c，编译运行指令如下：

```
gcc -o gemm_1 gemm_1.c -fopenmp
./gemm_1 <number of threads>
```

2.2 基于 OpenMP 的通用矩阵乘法优化

采用默认的任务调度机制的并行指令如下：

```
#pragma omp parallel for num_threads(ThreadNumber)
```

采用静态调度 schedule(static, 1) 的并行指令如下：

```
#pragma omp parallel for num_threads(ThreadNumber)\
schedule(static, 1)
```

采用动态调度 schedule(dynamic,1) 的并行指令如下：

```
#pragma omp parallel for num_threads(ThreadNumber)\
schedule(dynamic, 1)
```

这些指令都放在通用矩阵乘法的外层循环之前。其他细节请见源文件 gemm_schedules_compare.c，编译运行指令如下：

```
gcc -o gemm_schedules_compare gemm_schedules_compare.c -fopenmp
./gemm_schedules_compare <number of threads>
```

2.3 构造基于 Pthreads 的并行 for 循环机制，封装动态库，并改造并行通用矩阵乘法

2.3.1 构建 parallel_for() 函数

首先定义传参需要的结构体：

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

定义 parallel_for(int start, int end, int increment, void *(functor)(void*), void *arg, int num_thread) 函数，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。在此函数中有以下关键模块：

先分配线程空间：

```
    long thread;
    pthread_t *thread_handles;
    thread_handles = malloc(num_thread * sizeof(pthread_t)); //为每个线程的pthread_t对象分配内存
```

再构建 parallel_for 函数对循环分解、分配和执行机制，并生成线程：

```
    for (thread = 0; thread < num_thread; thread++)           //生成线程
    {
        // 确定每个线程的开始和结束
        int my_rank = thread;
        int my_first, my_last;
        int quotient = (end - start) / num_thread;
        int remainder = (end - start) % num_thread;
        int my_count;
        if (my_rank < remainder)
        {
            my_count = quotient + 1;
            my_first = my_rank * my_count;
        }
        else
        {
            my_count = quotient;
            my_first = my_rank * my_count + remainder;
        }
        my_last = my_first + my_count;
        struct for_index *index;
        index=malloc(sizeof(struct for_index));
        index->start=start+my_first;
        index->end=start+my_last;
        index->increment=increment;
        pthread_create(&thread_handles[thread], NULL, functor, index);
    }
```

最后停止线程：

```
    for (thread = 0; thread < num_thread; thread++) //停止线程
    {
        pthread_join(thread_handles[thread], NULL);
    }
    free(thread_handles);
```

parallel_for() 函数设计完成，其余细节请见 4.3 文件夹下的 parallel_for.c 文件。

2.3.2 封装为动态库

用以下指令生成 .so 文件：

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

再将 .so 文件移到 /usr/lib 中去,以便于在不同位置的程序中都能够调用动态库。

```
sudo mv libparallel_for.so /usr/lib
```

2.3.3 改造成为基于 parallel_for() 的通用矩阵乘法

将之前调用通用矩阵乘法函数的语句改为：

```
parallel_for(0, M, 1, functor_gemm, NULL, ThreadNumber); //并行矩阵乘法改造
```

其中functor_gemm() 函数为：

```
void * functor_gemm (void *args){
    struct for_index * index = (struct for_index *) args;
    for (int i = index->start; i < index->end; i = i + index->increment){
        for (int j = 0; j < K; j++)
        {
            matrixC[i][j] = 0;
            for (int l = 0; l < N; l++)
            {
                matrixC[i][j] += matrixA[i][l] * matrixB[l][j];
            }
        }
    }
}
```

改造完成。其余细节请见 4.3 文件夹下的 main.c 文件。

编译连接：

```
gcc main.c -L. -lparallel_for -o main
```

用 ldd 命令测试是否动态链接成功,命令及输出如下:

```
wrq@wrq-Inspiron-7490:~/Desktop/4/4.3$ ldd ./main
    linux-vdso.so.1 (0x00007ffd089de000)
    libparallel_for.so => /usr/lib/libparallel_for.so (0x00007f01312a8000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0130eb7000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007f0130c98000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f01316ad000)
```

3. 实验结果

3.1 通用矩阵乘法的并行版本测试结果

OpenMP 并行线程从 1 增加至 8, 矩阵规模从 512 增加至 2048，测试得到的计算时间如下：

线程数\矩阵规模	512 512 512	1024 1024 1024	2048 2048 2048
1	0.623190 s	5.308025 s	97.418987 s
2	0.322328 s	2.834889 s	50.161433 s
3	0.221249 s	2.149693 s	44.923346 s
4	0.173716 s	1.481340 s	35.022641 s
5	0.224444 s	1.727908 s	41.903394 s
6	0.209410 s	1.430377 s	28.633694 s
7	0.190737 s	1.500002 s	24.967521 s
8	0.175461 s	1.570082 s	26.394639 s

由表格中的运行时间对比，可以看出一定规律，比如在线程数为 1~4 时，随着线程数的增加，运行时间逐渐变短，当线程超过 4 时，运行时间基本稳定，这可能因为我的电脑是 4 核心的。

3.2 基于 OpenMP 的通用矩阵乘法优化实验结果

当矩阵规模为 **M = N = K = 512** 时，测试计算时间如下表：

线程数\调度方式	default	static	dynamic
1	0.625444 s	0.606694 s	0.778732 s
2	0.323151 s	0.313343 s	0.331126 s
3	0.222833 s	0.202345 s	0.204496 s
4	0.174162 s	0.151017 s	0.150515 s
5	0.219844 s	0.189456 s	0.150692 s
6	0.196559 s	0.172757 s	0.153362 s
7	0.188789 s	0.167607 s	0.147389 s
8	0.174894 s	0.147073 s	0.147360 s

可见，当矩阵规模为 **M = N = K = 512** 时，当线程数较小（不超过核心数）时， 默认方式和static方式还有dynamic方式速度没有明显差别，当线程数较大（超过核心数）时， dynamic方式和static方式的速度比默认方式速度快。

当矩阵规模为 **M = N = K = 1024** 时，测试计算时间如下表：

线程数\调度方式	default	static	dynamic
1	5.500588 s	5.488630 s	5.447189 s
2	2.704989 s	2.684325 s	3.364460 s
3	1.827248 s	1.811851 s	2.176945 s
4	1.519549 s	1.747935 s	1.851494 s
5	1.717580 s	1.718275 s	1.847821 s
6	1.514699 s	1.487519 s	2.533608 s
7	1.561538 s	1.744358 s	3.142864 s
8	1.626888 s	1.638058 s	3.558922 s

可见，当矩阵规模为 **M = N = K = 1024** 时，无论线程数大小，默认方式和static方式的速度大体上比dynamic方式速度快。

当矩阵规模为 **M = N = K = 2048** 时，测试计算时间如下表：

线程数\调度方式	default	static	dynamic
1	99.507109 s	99.410687 s	99.405024 s
2	51.660021 s	56.097031 s	53.822154 s
3	36.196065 s	40.595960 s	40.439743 s
4	31.069536 s	33.821501 s	33.898595 s
5	31.529682 s	35.537048 s	33.527845 s
6	26.810430 s	33.668964 s	33.914865 s
7	27.053382 s	30.814649 s	30.033597 s
8	27.781406 s	31.015668 s	30.837958 s

可见，当矩阵规模为 **M = N = K = 2048** 时，无论线程数大小， 默认方式总比其他两种调度防暑速度快。

综上所述，三种调度方式的性能优劣不能一概而论，当线程数和矩阵规模不同时，三种调度方式都各有优劣，需要分情况讨论。比如当矩阵规模较小且线程数较小时，三种调度方式无明显差别；当矩阵规模较小且线程数较多时，dynamic调度方式性能较好；当矩阵规模中等时，默认调度方式和static调度方式的性能优于dynamic调度方式；当矩阵规模较大时，默认调度方式性能较好。

3.3 构造基于 Pthreads 的并行 for 循环机制，封装动态库，并改造并行通用矩阵乘法

用 ldd 命令测试是否动态链接成功,命令及输出如下:

```
wrq@wrq-Inspiron-7490:~/Desktop/4/4.3$ ldd ./main
    linux-vdso.so.1 (0x00007ffd089de000)
    libparallel_for.so => /usr/lib/libparallel_for.so (0x00007f01312a8000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0130eb7000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007f0130c98000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f01316ad000)
```

可见动态链接成功，并且程序正确运行完毕。

测试动态链接库函数 paralell_for() 改造的并行矩阵乘法的 for 循环，在不同线程下和不同矩阵规模时的运行时间如下表所示：

线程数\矩阵规模	512 512 512	1024 1024 1024	2048 2048 2048
1	0.545623 s	4.980776 s	88.666998 s
2	0.284179 s	2.343041 s	45.379110 s
3	0.284179 s	1.577121 s	31.899414 s
4	0.152584 s	1.179936 s	27.225694 s
5	0.192110 s	1.464185 s	31.027813 s
6	0.173831 s	1.266200 s	28.142327 s
7	0.167243 s	1.309993 s	22.406526 s
8	0.145586 s	1.489615 s	30.081853 s

该实验结果符合上一次实验中 pthreads 并行的规律。

---

4. 实验感想

- 在做第一、二个任务时，实验过程并没有遇到困难，但是在观察分析实验结果时遇到了与我的猜测不同的现象，于是在实验结果部分进行了简要的概括和推测。
- 做第三个任务时，因为有之前 pthreads 实验的基础，将多个任务分配的机制已经在上次实验中实现过了，这次只需重新封装成新函数，也有老师附的代码例子，所以也没有遇到难以解决的问题。在做这个任务的过程中，我学到了一种新的传参方式，就是使用结构体，这是之前的学习中没有遇到过的新知识。
- 总而言之，这次实验相对于前几次顺利许多，但在其中也学到了许多，如 openmp 的指令，还有各种调度机制的对比，还有封装了自己用 pthreads 写的循环 for 并行函数，理论知识也从中得到了巩固，受益匪浅。
