中山大学计算机院本科生实验报告

2022 学年秋季学期

课程名称：并行程序设计 批改人：

实验	并行程序设计 Lab2	专业（方向）	计算机科学与技术（人工智能与大数据方向）
学号	20337025	姓名	崔璨明
Emai	897369624@qq.com	完成日期	2023/4/13

1.实验目的

1. 通过 Pthreads实现通用矩阵乘法：通过Pthreads实现通用矩阵乘法（Lab1）的并行版本，Pthreads并行线程从1增加至8，矩阵规模从512增加至2048。
2. 编写一个多线程程序来实现面积计算，Monte-carlo方法参考课本137页4.2题和本次实验作业的补充材料。估算$y = x^2$曲线与x轴之间区域的面积，其中x的范围为[0,1]。
3. 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。
4. 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。
5. 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。
    - 在Linux系统中将parallel_for函数编译为.so文件，由其他程序调用。
    - 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。

2.实验过程和核心代码

2-1 Pthreads实现通用矩阵乘法

实现的过程为：通过输入的参数确定线程的数量，然后创建线程数组（为每个线程分配空间），然后在线程数组的每一项中使用`pthread_create()`创建一个线程，并指定该线程要执行的`gemm`函数，最后主进程等待所有线程计算结束即可。

使用Pthreads来实现通用矩阵乘法时，需要调用pthread.h头文件，并在编译时加上链接参数`-lpthreads`，实现的关键代码如下，具体代码和注释见`gemm_p.cpp`文件：

```
pthread_t *thread_handles;
// 创建一个线程数组，大小为线程数量
```

```
thread_handles = (pthread_t *)malloc(thread_count * sizeof(pthread_t));
clock_t start_time=clock();
for (int i = 0; i < thread_count; i++)
{
    // 创建一个线程，并指定该线程要执行的函数为 gemm
    // 将线程编号 i 传递给 gemm 函数
    pthread_create(&thread_handles[i], NULL, gemm, (void *)i);
}
// 等待所有线程执行完成
for (int i = 0; i < thread_count; i++)
{
    pthread_join(thread_handles[i], NULL);
}
```

每个线程的执行函数`gemm()`的关键代码如下，即根据传入的线程编号计算每个线程要计算的行数，然后执行矩阵乘法：

```
void *gemm(void *rank)
{
    // 获取线程编号
    int p_rank = (long)rank;
    // 指定每个线程要计算的行数
    int p_first_row, p_end_row;
    int quotient = M / thread_count;
    int remainder = M % thread_count;
    int p_cols=0;
    if (p_rank < remainder)
    {
        p_cols = quotient + 1;
        p_first_row = p_rank * p_cols;
    }
    else
    {
        p_cols = quotient;
        p_first_row = p_rank * p_cols + remainder;
    }
    p_end_row = p_first_row + p_cols;
    // 执行矩阵乘法
    for (int m = p_first_row; m < p_end_row; m++)
    {
        for (int k= 0; k < K; k++)
        {
            C[m][k] = 0;
            for (int n = 0; n < N; n++)
            {
                C[m][k] += A[m][n] * B[n][k];
            }
        }
    }
    return NULL;
}
```

2-2 Monte-carlo方法面积计算

使用Monte-carlo方法计算函数$y = x^2$曲线与x轴之间区域的面积（$x \in [0,1]$），即只需要在$[0,1] * [0,1]$的正方形内随机生成很多点，然后统计位于曲线$y = x^2$下方的点的数量，计算出这些点占所有点的总数的比例，该比例乘上正方形的面积即为函数$y = x^2$曲线与x轴之间区域的面积。

首先定义两个常量：线程数量`THREAD_COUNT`和迭代次数`ITERATIONS`。然后定义变量`sum`表示函数$y = x^2$在 [0,1] 区间内的面积估计值，以及互斥锁`lock`用于保证多个线程同时对`sum`进行修改时不会出现冲突。函数`calculate_area`的作用是计算函数$y = x^2$在 [0,1] 区间内的面积估计值，并将计算结果加到`sum`变量中。参数为当前线程的ID。在该函数的每次迭代中，随机生成两个在 [0,1] 区间内均匀分布的随机数 x 和 y，然后判断点 (x,y) 是否在函数 y = x^2 的下方，如果是，则将`local_sum`加 1。并在线程执行的最后将当前线程的计算结果加到`sum`变量中。

主函数创建`THREAD_COUNT`个线程，并将线程 ID 存储在`thread_ids`数组中，然后使用`pthread_create`函数创建线程，每个线程都执行`calculate_area`函数进行计算。

最后使用`pthread_join`函数等待所有线程完成计算并输出计算结果`sum / THREAD_COUNT`，即为函数$y = x^2$在 [0,1] 区间内的面积估计值，关键代码如下：

```
#define THREAD_COUNT 8  // 线程数量
#define ITERATIONS 1000000  // 迭代次数

double sum = 0;
pthread_mutex_t lock;  //互斥锁

void *calculate_area(void *thread_id_ptr) {
    int thread_id = *(int *) thread_id_ptr;  // 获取线程ID
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0, 1);
    double local_sum = 0;

    for (int i = 0; i < ITERATIONS; i++) {
        double x = dis(gen);
        double y = dis(gen);
         //  (x, y) 在 y=x^2 曲线下方，需要计入面积
        if (y <= x * x) {
            local_sum++;
        }
    }

    //将当前线程的计算结果加到 sum 变量中
    pthread_mutex_lock(&lock);
    sum += local_sum / ITERATIONS;
    pthread_mutex_unlock(&lock);

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[THREAD_COUNT];  // 创建线程数组
    int thread_ids[THREAD_COUNT];  // 创建线程 ID 数组
    pthread_mutex_init(&lock, NULL);  // 初始化互斥锁

      // 创建线程并执行计算
```

```
    for (int i = 0; i < THREAD_COUNT; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, calculate_area, &thread_ids[i]);
    }

    // 等待所有线程完成计算
    for (int i = 0; i < THREAD_COUNT; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&lock);
    std::cout << "function: y=x^2\nEstimated area: " << sum / THREAD_COUNT <<
std::endl;  // 输出计算结果
    return 0;
}
```

2-3 OpenMP实现通用矩阵乘法

使用OpenMP实现通用矩阵乘法的过程较为简单，需要引入头文件`#include <omp.h>`，`#pragma omp pallel`语句用来表明之后的结构化代码块应该被多个线程并行执行。完成代码块前会有一个隐式阻塞，即先完成的线程必须等待线程组其他线程完成代码块。

`num_threads`子句允许我们指定执行后代码块的线程数，程序可以启动的线程数可能会受系统定义的限制。OpenMP标准并不保证实际能够启动thread_count个线程。关键代码如下：

```
//use openmp
void parallel_gemm(){
    //默认调度
    #pragma omp parallel for num_threads(Thread_NUM)
    for(int m=0;m<M;m++){
        for(int n=0;n<N;n++){
            for(int k=0;k<K;k++){
                C[m][k]+=A[m][n]*B[n][k];
            }
        }
    }
}
```

2-4 OpenMP的通用矩阵乘法优化

分别进行静态角度和动态调度，具体的实现语句为：

```
//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)
```

2-5 基于Pthreads的并行for循环分解、分配和执行机制

pthread_create函数创建num_threads个线程，每个线程执行一部分循环任务。其中，为了保证每个线程执行的循环任务大小相等，代码先计算了每个线程要处理的数据块大小block，然后分配给每个线程一段区间。最后，使用pthread_join函数等待所有线程执行完毕，并且在函数结束之前释放所分配的内存。具体代码如下：

```
//并行循环函数parallel_for
void parallel_for(int start, int end, int increment, void *(*functor)(void *),
void *arg, int num_threads){
    pthread_t *threads = (pthread_t *)malloc(num_threads * sizeof(pthread_t));
    for_index *index_arr = (for_index *)malloc(num_threads * sizeof(for_index));

    //每个线程要处理的数据块大小
    int block = (end - start) / num_threads;
    //为每个线程分配参数
    for (int i = 0; i < num_threads; i++){
        index_arr[i].args = arg;
        index_arr[i].start = start + i * block;
        index_arr[i].end = index_arr[i].start + block;
        //处理最后一个线程的数据块
        if (i == (num_threads - 1))
            index_arr[i].end = end;
        index_arr[i].increment = increment;
        pthread_create(&threads[i], NULL, functor, (void *)(index_arr + i));
    }
    //等待所有线程执行完成
    for (int thread = 0; thread < num_threads; thread++)
        pthread_join(threads[thread], NULL);
    free(threads);
    free(index_arr);
}
```

其中，`for_index`结构体如下，包括每个块的起始下标、终止下标、步长和并行函数需要的参数`args`，：

```
// for 循环参数的结构体
struct for_index{
    void *args;
    int start; // 起始下标
    int end; //终止下标
    int increment; // 步长
};
```

在Linux系统中将parallel_for函数编译为.so文件，由其他程序调用：

首先需要编写头文件`parallel_for.h`，头文件中包括了需要用到的库，并定义函数`paraller_for`和结构体`for_index`。然后编写函数实现的文件`parallel_for.cpp`，然后将其编译生成.so文件即可，具体过程见实验结果。

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：

原来版本的矩阵乘法如下：

```
for (int i=0;i<M;i++){
    for (int j=0;j<K;j++){
        C[i*K+j] =0;
        for (int k=0;k<N;k++)
            C[i*K+j] += A[i*N+k]*B[k*K+j];
    }
}
```

改造成基于parallel_for函数并行化的矩阵乘法如下：

```
void *gemm_fun(void *args){
    struct for_index *idx = (struct for_index *)args;
    struct args *matrix = (struct args *)(idx->args);
    int K=matrix->k;
    int N=matrix->n;
    for (int m = idx->start; m < idx->end; m = m + idx->increment){
        for (int k = 0; k < K; k++){
            matrix->C[m * K + k] =0;
            for (int n = 0; n < N; n++){
                matrix->C[m * K + k]  += matrix->A[m * N + n] * matrix->B[n* K +
k];
            }
        }
    }
    return NULL;
}
```

在main函数中调用：

```
parallel_for(0, M, 1, gemm_fun, arg, Thread_NUM);
```

3.实验结果

3-1 Pthreads实现通用矩阵乘法

运行程序，输入参数（M，N，K和线程数），运行结果如下，可以验证得到了正确的乘积结果：

```
cui@cui-VirtualBox:~/parall/lab3$ ./gemm_p 4 4 4 4
matrix A:
15.90   57.00   76.20   17.50
27.40   63.90   48.40   1.80
72.60   41.50   4.50    37.30
85.50   89.40   52.90   14.40
matrix B:
38.10   70.70   34.20   54.30
99.90   34.20   7.10    76.40
21.20   79.40   69.80   27.80
32.90   21.60   34.50   48.80
result:
8491.28         9501.81         6870.99         8190.53
8512.85         8004.40         4831.19         7803.14
8234.48         7715.10         4378.52         9058.12
13783.85        13613.63        7748.06         13646.15
uisng time:0.000653 s
cui@cui-VirtualBox:~/parall/lab3$
```

Pthreads并行线程从1增加至8，矩阵规模从512增加至2048，得到的实验结果如下表：

矩阵规模|线程数（单位ms）	1	2	4	8
512	675.3	669.2	621.4	631.2
1024	7538.4	7275.5	7390.3	7433.1
2048	127120	115004	102664	149976

分析：可以看到，当矩阵规模较大时，随着并行线程数目的增加，计算的时间也随之减少，即矩阵乘法的速度变快了。但在并行线程数目从4增大到8时，时间却没有明显变化，推测是由于虚拟机硬件的限制，即使设置了较多的线程数目，但不能满足。

3-2 Monte-carlo方法面积计算

运行程序，得到面积的计算结果如下：

```
cui@cui-VirtualBox:~/parall/lab3$ g++ Monte_carlo.cpp -o monte_carlo -pthread
cui@cui-VirtualBox:~/parall/lab3$ ./monte_carlo
function: y=x^2
Estimated area: 0.333503
cui@cui-VirtualBox:~/parall/lab3$ ./monte_carlo
function: y=x^2
Estimated area: 0.333379
cui@cui-VirtualBox:~/parall/lab3$ ./monte_carlo
function: y=x^2
Estimated area: 0.333325
cui@cui-VirtualBox:~/parall/lab3$
```

分析：可见运行结果是符合预期的，x的范围为[0,1]时，$y = x^2$曲线与x轴之间区域的面积为$\frac{1}{3}$。

3-3 OpenMP实现通用矩阵乘法

运行程序，输入参数（M，N，K和线程数），运行结果如下，可以验证得到了正确的乘积结果，在程序中，我还增加了与串行的矩阵乘法的对比，可见OpenMP实现的通用矩阵乘法确实要比串行的矩阵乘法要快。

```
cui@cui-VirtualBox:~/parall/lab3$ g++ gemm_openmp.cpp -o gemm_openmp
cui@cui-VirtualBox:~/parall/lab3$ ./gemm_openmp 4 4 4 2
result:
8491.28         9501.81         6870.99         8190.53
8512.85         8004.40         4831.19         7803.14
8234.48         7715.10         4378.52         9058.12
13783.85        13613.63        7748.06         13646.15
normal gemm uisng time:2e-06 s
openmp gemm uisng time:1e-06 s
cui@cui-VirtualBox:~/parall/lab3$
```

Pthreads并行线程从1增加至8，矩阵规模从512增加至2048，得到的实验结果如下表：

矩阵规模|线程数（单位ms）	1	2	4	8
512	543.6	512.7	490.9	560.2
1024	3936.1	4048.0	3898.9	4103.6
2048	33008.0	33381.7	33194.0	36063.6

分析：可见实验结果与预期一致，随着并行线程数目的增加，矩阵乘法的速度也有显著提升，但由于硬件的限制，当线程数提升到8时，OpenMP不能保证启用8个线程，因此速度有所下降。

3-4 OpenMP的通用矩阵乘法优化

固定矩阵规模为$1024 * 1024 * 1024$，并行线程数目为4，将默认调度、静态调度、动态调度三者进行对比：

默认调度：

```
openmp gemm uisng time:4.14009 s
cui@cui-VirtualBox:~/parall/lab3$
```

静态调度：

```
openmp gemm uisng time:4.11695 s
cui@cui-VirtualBox:~/parall/lab3$
```

动态调度：

```
openmp gemm uisng time:4.0923 s
cui@cui-VirtualBox:~/parall/lab3$
```

分析：可见动态调度效果最好，静态调度效果要比默认调度的好，但三种调度的速度相差不大，可能是因为矩阵的元素是随机生成的，调度影响不大。

3-5 基于Pthreads的并行for循环分解、分配和执行机制

要在Linux系统中将parallel_for函数编译为.so文件，首先编写`parallel_for.h`和`parallel_for.cpp`两个文件，然后通过以下命令进行编译：

```
cui@cui-VirtualBox:~/parall/lab3$ g++ -c -fPIC -o parallel_for.o parallel_for.cpp -lpthread
cui@cui-VirtualBox:~/parall/lab3$ g++ -shared -o libparallel_for.so parallel_for.o -lpthread
```

其中，`-c`表示只编译而不连接，`-fPIC`表示编译为位置独立的代码，`-shared`表示生成一个动态链接库，经过这两个命令后，将生成动态链接库文件`libparallel_for.so`。

为了验证是否能正常调用，我编写了一个测试程序`test.cpp`，主要运用改进后的矩阵乘法进行计算（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），结果在运行时，发生了如下错误：

```
cui@cui-VirtualBox:~/parall/lab3$ g++ -o test test.cpp -L./ -lparallel_for -lpthread
cui@cui-VirtualBox:~/parall/lab3$ ./test 2 2 2 2
./test: error while loading shared libraries: libparallel_for.so: cannot open shared object file: No such file or directory
```

使用`ldd`命令查看链接的动态库以分析原因，发现`libparallel_for.so`并没有被链接到，推测原因是编译时，在默认路径下没有这个库文件，因此没有找到，于是添加以下命令，将该库文件移动到链接的默认路径：

```
cui@cui-VirtualBox:~/parall/lab3$ sudo cp libparallel_for.so /usr/lib
[sudo] cui 的密码：
cui@cui-VirtualBox:~/parall/lab3$
```

如此一来便可以正常运行，结果如下：

```
cui@cui-VirtualBox:~/parall/lab3$ ./test 4 4 4 2
result:
8491.28         9501.81         6870.99         8190.53
8512.85         8004.40         4831.19         7803.14
8234.48         7715.10         4378.52         9058.12
13783.85        13613.63        7748.06         13646.15
uisng time:0.002203 s
cui@cui-VirtualBox:~/parall/lab3$
```

4.实验感想

在这次实验中，我学习了如何使用Pthreads和OpenMP这两种主流的并行化技术，掌握了使用Monte-carlo方法来估算函数面积的过程，以及如何构建一个基于Pthreads的并行for循环分解、分配和执行机制。除此之外，我还了解了不同线程数和矩阵规模对计算时间的影响，并学会了如何优化任务调度机制以提高性能。这次实验让我深入了解了课上所学的并行计算的原理，并在具体问题中付诸实践，这让我受益匪浅。