中山大学计算机学院本科生实验报告

课程名称：并行程序设计与算法

实验	基于OpenMP的并行矩阵乘法	专业（方向）	计算机科学与技术
学号	21307035	姓名	邓栩瀛
Email	dengxy66@mail2.sysu.edu.cn	完成日期	2024.4.22

1、实验目的

1. OpenMP通用矩阵乘法

使用OpenMP实现并行通用矩阵乘法，并通过实验分析不同进程数量、矩阵规模、调度机制时该实现的性能。

输入：m,n,k三个整数，每个整数的取值范围均为[128, 2048]

问题描述：随机生成m×n的矩阵A及n×k的矩阵B，并对这两个矩阵进行矩阵乘法运算，得到矩阵C

输出：A,B,C三个矩阵，及矩阵计算所消耗的时间t

要求：使用OpenMP多线程实现并行矩阵乘法，设置不同线程数量（1-16）、矩阵规模（128-2048）、调度模式（默认、静态、动态调度），通过实验分析程序的并行性能。

2. 构造基于Pthreads的并行for循环分解、分配、执行机制

模仿OpenMP的omp_parallel_for构造基于Pthreads的并行for循环分解、分配及执行机制。此部分可在下次实验报告中提交。

问题描述：生成一个包含parallel_for函数的动态链接库（.so）文件，该函数创建多个Pthreads线程，并行执行parallel_for函数的参数所指定的内容。

函数参数：parallel_for函数的参数应当指明被并行循环的索引信息，循环中所需要执行的内容，并行构造等。

以下为parallel_for函数的基础定义，实验实现应包括但不限于以下内容：

```
parallel_for(int start, int end, int inc, void *(*functor)( int,void*), void *arg, int
num_threads)
```

> start, end, inc分别为循环的开始、结束及索引自增量；
>
> functor为函数指针，定义了每次循环所执行的内容；

> arg为functor的参数指针，给出了functor执行所需的数据；
>
> num_threads为期望产生的线程数量。
>
> 选做：除上述内容外，还可以考虑调度方式等额外参数。

示例：给定functor及参数如下：

```
struct functor_args{
  float *A, *B, *C;
};
void functor(int idx, void* args){
  functor_args *args_data = (functor_args*) args;
  args_data->C[idx] = args_data->A[idx] + args_data->B[idx];
}
```

调用方式如下：

```
functor_args args = {A, B, C};
parallel_for(0, 10, 1, functor, (void*)&args, 2)
```

该调用方式应当能产生两个线程，并行执行functor完成数组求和（$C_i = A_i + B_i$）。当不考虑调度方式时，可由前一个线程执行任务{0,1,2,3,4}，后一个线程执行任务{5,6,7,8,9}。也可以实现对调度方式的定义。

要求：完成parallel_for函数实现并生成动态链接库文件，并以矩阵乘法为例，测试其实现的正确性及效率

2、实验过程和核心代码

1.OpenMP通用矩阵乘法

执行命令

```
g++ -fopenmp main1.cpp -o main1
```

设置OpenMP参数

```
omp_set_num_threads(threads);
omp_set_schedule(omp_sched_dynamic, 0); // default
if (sched == 1)
    omp_set_schedule(omp_sched_static, 0); //static
else if (sched == 2)
    omp_set_schedule(omp_sched_dynamic, 0); // dynamic
```

2.构造基于Pthreads的并行for循环分解、分配、执行机制执行命令

执行命令

```
g++ -c -fPIC -o parallel_for.o parallel_for.cpp -lpthread
g++ -shared -o parallel_for.so parallel_for.o -lpthread
g++ -o main2 main2.cpp -L./ -lparallel_for -lpthread
```

相关结构体定义

```
struct for_index{
    void *args;
    int start;
    int end;
    int increment;
};
```

parallel_for函数的定义

```
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));

    int block = (end - start) / num_threads;

    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;
        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));
    }
    for (int thread = 0; thread < num_threads; thread++)
        pthread_join(threads[thread], NULL);
    free(threads);
    free(index_arr);
}
```

3、实验结果

1.OpenMP通用矩阵乘法

default

线程数	矩阵规模			
	128	256	512	1024
1	0.023	0.197	7.720	22.536
2	0.018	0.128	1.180	12.936
4	0.017	0.126	1.214	11.211
8	0.015	0.142	1.122	9.995
16	0.016	0.143	1.369	12.544

static

线程数	矩阵规模			
	128	256	512	1024
1	0.022	0.185	1.724	22.567
2	0.019	0.125	1.256	13.109
4	0.017	0.152	1.183	11.781
8	0.019	0.130	1.145	11.129
16	0.018	0.135	1.228	11.980

dynamic

线程数	矩阵规模			
	128	256	512	1024
1	0.025	0.185	1.711	22.564
2	0.015	0.124	1.139	12.807
4	0.017	0.141	1.350	11.560
8	0.016	0.152	1.176	11.929
16	0.022	0.137	1.253	10.678

实验结果分析：

1. 对于不同调度模式，在默认调度模式下，随着线程数量的增加，性能并没有线性增加；在静态调度模式下，性能随着线程数量的增加呈现出较好的线性提升趋势；在动态调度模式下，性能表现与默认调度模式类似，线程数量增加并未带来线性的性能提升。
2. 对与不同矩阵规模下的性能，随着矩阵规模的增加，计算量增大，耗时也相应增加，在较小的矩阵规模下，不同调度模式的性能差异不明显，但在较大的矩阵规模下，静态调度模式相对其他两种模式的性能表现更好。
3. 总体而言，在较小的矩阵规模下，三种调度模式的性能差异不大，但在较大的矩阵规模下，静态调度模式的性能优势较为明显。

2.构造基于Pthreads的并行for循环分解、分配、执行机制

线程数	矩阵规模			
	128	256	512	1024
1	0.010448	0.09501	0.8778	11.5687
2	0.009779	0.121958	1.07026	12.9785
4	0.009124	0.127034	1.04925	12.6656
8	0.009769	0.118113	1.29845	11.8409
16	0.009734	0.122188	1.11006	12.2015

实验结果分析：

1. 随着线程数的增加，对于较小的矩阵规模，执行时间并没有明显减少，甚至在某些情况下稍微有所增加。
2. 对于较大的矩阵规模，随着线程数增加，执行时间呈现出先减少后增加的趋势。由此可见，增加线程数可以带来一定程度的性能提升，但在某个临界点之后，继续增加线程数可能导致性能下降。
3. 在某些情况下，增加线程数并没有明显改善执行时间，可能是因为任务的并行性受到了某些限制，无法充分利用增加的线程资源。
4. 随着矩阵规模的增加，执行时间也呈现出增加的趋势。

4、实验感想

1. 通过本次实验，进一步了解了OpenMP及其相关操作，通过简单的编译器指令和库函数调用，就可以实现并行化。
2. 不同调度模式对并行程序的性能也会造成一定的影响，静态调度模式能够更好地均衡任务负载，而动态调度模式则更适用于任务负载不均衡、需要动态调整的情况，因此合理选择调度模式可以提高程序的性能。
3. 在不同的情况下，增加线程数量可能会带来不同程度的性能提升，但也可能会带来额外的开销。