**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. 通过OpenMP实现通用矩阵乘法
   2. 基于OpenMP的通用矩阵乘法优化
   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**
   1. **OpenMP实现通用矩阵乘法**
      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。

* + 1. **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。

* 1. **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];

}

}

}

* 1. **基于Pthreads的并行for循环分解、分配和执行机制**
     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。

* + 1. **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

* + 1. **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);

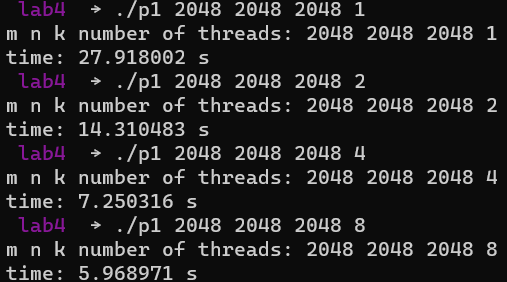
为了简化参数传递，将矩阵数据声明为全局变量，实现所有线程共享。

1. **实验结果**
   1. **OpenMP通用矩阵乘法**

矩阵大小

时间（s）

|  |  |  |  |
| --- | --- | --- | --- |
| 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。

* 1. **不同调度方式性能比较**

实验过程中统一使用四线程并行。

调度方式

时间（s）

|  |  |  |  |
| --- | --- | --- | --- |
| 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。

* 1. **parallel\_for函数**
     1. 样例测试

循环体代码段：向量

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){

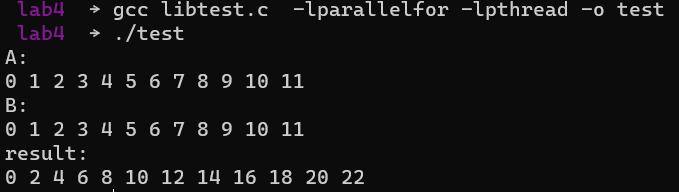
C[i] = A[i] + B[i];

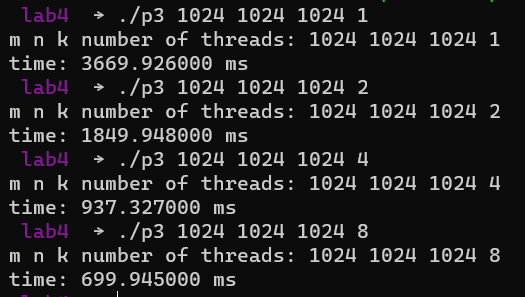
}

}

库函数调用

parallel\_for(0,11,1,code,NULL,4);



* + 1. **parallel\_for 并行计算矩阵乘法**

结果和使用OpenMP的结果相近。

1. **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。