

中山大学计算机院本科生实验报告

(2024 学年秋季学期)

课程名称：高性能计算程序设计

批改人：

实验	基于 OpenMP 的多线程编程	专业（方向）	信息与计算科学
学号	22336044	姓名	陈圳煌
Email	2576926165@qq.com	完成日期	2024. 11. 14

1. 实验目的

本次实验通过 OpenMP 实现通用矩阵乘法，并采用 OpenMP 的三种调度方法测试并比较性能，最后基于 Pthread 构造并行 for 循环分解、分配和执行机制，熟悉对 OpenMP 的使用。

2. 实验过程和核心代码

task1 代码：

```
#include<stdio.h>
#include<omp.h>
#include<stdlib.h>
#include<time.h>

int M,N,K;
double** A;
double** B;
double** C;
void matrix_multiply(){
    int my_rank=omp_get_thread_num();
    int thread_count=omp_get_num_threads();

    int batch=M/thread_count;
    int begin_row=batch*my_rank;
    int end_row=batch*(my_rank+1);

    for(int i=begin_row;i<end_row;i++){
        for(int j=0;j<K;j++){
            C[i][j]=0.0;
            for(int l=0;l<N;l++){
                C[i][j]+=A[i][l]*B[l][j];
            }
        }
    }
}
```

```

}

int main(int argc, char* argv){
    int thread_count=atoi(argv[1], NULL, 10);
    scanf("%d %d %d",&M,&N,&K);

    A=malloc(sizeof(double*)*M);
    B=malloc(sizeof(double*)*N);
    C=malloc(sizeof(double*)*M);

    for(int i=0;i<M;i++){
        A[i]=malloc(sizeof(double)*N);
    }
    for(int i=0;i<N;i++){
        B[i]=malloc(sizeof(double)*K);
    }
    for(int i=0;i<M;i++){
        C[i]=malloc(sizeof(double)*K);
    }

    srand(time(NULL));
    for(int i=0;i<M;i++){
        for(int j=0;j<N;j++){
            A[i][j]=rand()*5.53/(double)RAND_MAX;
        }
    }
    for(int i=0;i<N;i++){
        for(int j=0;j<K;j++){
            B[i][j]=rand()*3.35/(double)RAND_MAX;
        }
    }

    double start=omp_get_wtime();
    // #pragma omp parallel num_threads(thread_count)
    // matrix_multiply();
    int i=0,j=0,l=0;
    #pragma omp parallel for num_threads(thread_count) shared(A,B,C,M,N,K)
private(i,j,l)
    for(i=0;i<M;i++){
        for(j=0;j<K;j++){
            C[i][j]=0.0;
            for(l=0;l<N;l++){

```

```

        C[i][j]+=A[i][l]*B[l][j];
    }
}
}
double end=omp_get_wtime();

printf("Used time: %.2lf ms",(end-start)*1000);
for(int i=0;i<M;i++){
    free(A[i]);
}
for(int i=0;i<N;i++){
    free(B[i]);
}
for(int i=0;i<M;i++){
    free(C[i]);
}
free(A);
free(B);
free(C);
return 0;
}

```

在进行矩阵运算的三层 for 循环外增加语句：

```
#pragma omp parallel for num_threads(thread_count) shared(A,B,C,M,N,K)
private(i,j,l)
```

创建多线程按照默认调度的方法实现 for 循环的并行，其中 A,B,C,M,N,K 为共享变量，l,j,l 为各线程私有变量，使用 `omp_get_wtime()` 方法进行计时。

task2 代码：

```

#include<stdio.h>
#include<omp.h>
#include<stdlib.h>
#include<time.h>

int M=1024,N=1024,K=1024;
int thread_count;

void cal_mat(double** A,double** B, double **C, int op){
    int i=0,j=0,l=0;
    if(op==1){
        #pragma omp parallel for num_threads(thread_count)
        shared(A,B,C,M,N,K) private(i,j,l)
        for(i=0;i<M;i++){

```

```
for(j=0;j<K;j++){
    C[i][j]=0.0;
    for(l=0;l<N;l++){
        C[i][j]+=A[i][l]*B[l][j];
    }
}
}
}else if(op==2){
    #pragma omp parallel for num_threads(thread_count)
shared(A,B,C,M,N,K) private(i,j,l) schedule(static,16)
    for(i=0;i<M;i++){
        for(j=0;j<K;j++){
            C[i][j]=0.0;
            for(l=0;l<N;l++){
                C[i][j]+=A[i][l]*B[l][j];
            }
        }
    }
}
}else if(op==3){
    #pragma omp parallel for num_threads(thread_count)
shared(A,B,C,M,N,K) private(i,j,l) schedule(dynamic,16)
    for(i=0;i<M;i++){
        for(j=0;j<K;j++){
            C[i][j]=0.0;
            for(l=0;l<N;l++){
                C[i][j]+=A[i][l]*B[l][j];
            }
        }
    }
}
}else{
    #pragma omp parallel for num_threads(thread_count)
shared(A,B,C,M,N,K) private(i,j,l) schedule(guided,32)
    for(i=0;i<M;i++){
        for(j=0;j<K;j++){
            C[i][j]=0.0;
            for(l=0;l<N;l++){
                C[i][j]+=A[i][l]*B[l][j];
            }
        }
    }
}
}
```

```

int main(int argc, char* argv[]){
    thread_count=strol(argv[1], NULL, 10);

    double** A=malloc(sizeof(double*)*M);
    double** B=malloc(sizeof(double*)*N);
    double** C1=malloc(sizeof(double*)*M);
    double** C2=malloc(sizeof(double*)*M);
    double** C3=malloc(sizeof(double*)*M);
    double** C4=malloc(sizeof(double*)*M);

    for(int i=0;i<M;i++){
        A[i]=malloc(sizeof(double)*N);
    }
    for(int i=0;i<N;i++){
        B[i]=malloc(sizeof(double)*K);
    }
    for(int i=0;i<M;i++){
        C1[i]=malloc(sizeof(double)*K);
        C2[i]=malloc(sizeof(double)*K);
        C3[i]=malloc(sizeof(double)*K);
        C4[i]=malloc(sizeof(double)*K);
    }

    srand(time(NULL));
    for(int i=0;i<M;i++){
        for(int j=0;j<N;j++){
            A[i][j]=rand()*5.53/(double)RAND_MAX;
        }
    }
    for(int i=0;i<N;i++){
        for(int j=0;j<K;j++){
            B[i][j]=rand()*3.35/(double)RAND_MAX;
        }
    }

    double time0=omp_get_wtime();
    cal_mat(A,B,C1,1);
    double time1=omp_get_wtime();
    cal_mat(A,B,C2,2);
    double time2=omp_get_wtime();
    cal_mat(A,B,C3,3);
    double time3=omp_get_wtime();
    cal_mat(A,B,C4,4);
}

```

```

    double time4=omp_get_wtime();

    printf("Default schedule:\nFirst_num: %.4lf Used
time: %.4lfsec\n",C1[0][0],time1-time0);
    printf("Static schedule:\nFirst_num: %.4lf Used
time: %.4lfsec\n",C2[0][0],time2-time1);
    printf("Dynamic schedule:\nFirst_num: %.4lf Used
time: %.4lfsec\n",C3[0][0],time3-time2);
    printf("Guided schedule:\nFirst_num: %.4lf Used
time: %.4lfsec\n",C4[0][0],time4-time3);

    for(int i=0;i<M;i++){
        free(A[i]);
    }
    for(int i=0;i<N;i++){
        free(B[i]);
    }
    for(int i=0;i<M;i++){
        free(C1[i]);
        free(C2[i]);
        free(C3[i]);
        free(C4[i]);
    }
    free(A);
    free(B);
    free(C1);
    free(C2);
    free(C3);
    free(C4);
    return 0;
}

```

在 task1 的基础上分别在矩阵乘法的三层 for 循环之前加上不一样的语句，使得分别按照默认调度、静态调度、动态调度、Guide 的方法进行任务分配。

task3 代码：

首先在 parallel_fun.h 文件中定义函数

```

#ifndef PARALLEL_FOR_H
#define PARALLEL_FOR_H

void parallel_for(int start, int end, int increment, void*(functor)(void*),
void *arg, int num_threads);

```

```
#endif
```

并且在对应的 parallel_fun.c 文件中进行函数的实现:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<time.h>
#include"parallel_fun.h"

struct for_index{
    int start;
    int end;
    int increment;
    double **A;
    double **B;
    double **C;
    int scale1;
    int scale2;
};

void parallel_for(int start, int end, int increment,
void*(functor)(void*) ,void *arg , int num_threads){
    pthread_t* thread_handles=malloc(num_threads*sizeof(pthread_t));
    struct for_index* indice=malloc(num_threads*sizeof(struct
for_index));

    struct for_index *data=(struct for_index *)arg;

    for(int i=0;i<num_threads;i++){
        indice[i].start=start+i*(end-start)/num_threads;
        indice[i].end=start+(i+1)*(end-start)/num_threads;
        indice[i].increment=increment;
        indice[i].A=data->A;
        indice[i].B=data->B;
        indice[i].C=data->C;
        indice[i].scale1=data->scale1;
        indice[i].scale2=data->scale2;

        pthread_create(&thread_handles[i], NULL, functor, &indice[i]);
    }

    for(int i=0;i<num_threads;i++){
        pthread_join(thread_handles[i], NULL);
    }
}
```

```

    free(thread_handles);
    free(indice);
    free(data);
}

```

函数的参数 start 为循环开始的位置, end 为结束位置, increment 为步长, 函数 functor 为程序定义的函数, 用以执行多线程的任务, 参数 arg 用来传输矩阵信息, num_threads 为线程数。在函数中, 首先创建线程数组, 为每个数组分配对应线程计算所需要的矩阵信息, 之后创建对应线程, 并等待所有执行完成后结束。

接着在命令行输入指令

```
gcc -c parallel_fun.c
```





编译源文件生成 parallel_fun.o 文件, 接着运行

```
gcc -shared -o function.dll parallel_fun.c
```

生成一个名为 function.dll 的动态链接库, 接着运行

```
gcc -g -Wall -fopenmp -o task3 task3.c -L. -lfunction
```

编译主程序并连接动态库, 最后运行 task3.exe 文件。

 function.dll	2024/11/13 23:22	应用程序扩展	49 KB
 parallel_fun.c	2024/11/13 23:22	C 源文件	2 KB
 parallel_fun.h	2024/11/13 20:40	C Header File	1 KB
 parallel_fun.o	2024/11/13 23:22	O 文件	2 KB

3. 实验结果

task1:

矩阵运行时间

Comm_size (num of processes)	Order of Matrix (milliseconds)				
	128	256	512	1024	2048
1	6	52	460	5758	68686
2	3	27	230	2902	31423
4	2	14	124	1268	16767
8	2	11	85	946	10280
16	2	9	77	845	9818

加速比

Comm_size (num of processes)	Order of Matrix (Speedups)				
	128	256	512	1024	2048
1	1.0	1.0	1.0	1.0	1.0
2	2.0	1.93	2.0	1.98	2.19
4	3.0	3.71	3.71	4.54	4.10
8	3.0	4.73	5.41	6.09	6.68
16	3.0	5.78	5.97	6.81	7.0

并行效率

Comm_size (num of processes)	Order of Matrix				
	128	256	512	1024	2048
1	1.0	1.0	1.0	1.0	1.0
2	0.5	0.97	1.0	0.99	1.10
4	0.75	0.93	0.93	1.14	1.03
8	0.38	0.60	0.68	0.76	0.84
16	0.19	0.36	0.37	0.43	0.44

（以上表格中高亮部分为不符合理论的运行时间加速比和效率，可能是由于在运行过程中计算机性能不稳定或者初始化时给每个矩阵随机初始化从而出现浮点数的差异较大导致。）

根据表格中结果，随着增多线程和矩阵规模，运行加速比呈上升趋势，但是随着使用线程数的增大，并行效率逐渐降低，推测是因为在程序执行并行计算的过程中分配任务到多个线程的额外开销较大。

task2:

首先使用长度为 10 亿的数组求和的例子进行性能比较（线程数位为 16）
静态调度、动态调度每次分配的大小为 1 时：

```
Default schedule:
Sum: 500044978.6677 Used time: 0.4100sec
Static schedule:
Sum: 500044978.6677 Used time: 2.4040sec
Dynamic schedule:
Sum: 500044978.6677 Used time: 13.9180sec
Guided schedule:
Sum: 500044978.6677 Used time: 0.3610sec
```

可以看到，性能上 Guide>默认调度>静态调度>动态调度
静态调度、动态调度每次分配的大小为 1000 时：

```
Default schedule:
Sum: 500046379.5697 Used time: 0.3470sec
Static schedule:
Sum: 500046379.5698 Used time: 0.2680sec
Dynamic schedule:
Sum: 500046379.5698 Used time: 0.2480sec
Guided schedule:
Sum: 500046379.5698 Used time: 0.3140sec
```

性能上，动态调度>静态调度>Guide>默认调度

当每次分配大小为 1 时，要频繁地给每个线程分配任务，并且破坏了数据的连续访问（局部性原理），因为静态调度和动态调度性能比默认调度要差，而每次分配的较大的任务时，就会有较好地性能。

接下来用矩阵乘法进行验证（线程数为 8，矩阵规模为 1024*1024）

静态调度、动态调度每次分配的大小为 1 时：

```
Default schedule:
First_num: 4468.2439 Used time: 0.7640sec
Static schedule:
First_num: 4468.2439 Used time: 0.7480sec
Dynamic schedule:
First_num: 4468.2439 Used time: 0.7010sec
Guided schedule:
First_num: 4468.2439 Used time: 0.7690sec
```

性能上，动态调度>静态调度>默认调度>Guide

静态调度、动态调度每次分配的大小为 16 时：

```
Default schedule:
First_num: 4596.6873 Used time: 0.7660sec
Static schedule:
First_num: 4596.6873 Used time: 0.7390sec
Dynamic schedule:
First_num: 4596.6873 Used time: 0.7060sec
Guided schedule:
First_num: 4596.6873 Used time: 0.7840sec
```

性能上，动态调度>静态调度>默认调度>Guide

以上结果表明，在进行矩阵乘法运算时，静态调度和动态调度对性能提升比数组求和更大，并且受分配任务大小的影响较小。

task3:

将 task1 中的进行矩阵乘法部分的三层 for 循环替换成自建的 parallel_for 函数进行测试：

```
4 4 4
12.53 7.14 7.96 12.88
18.55 10.73 8.60 17.19
14.60 17.47 23.68 29.74
13.03 9.55 17.51 21.79
Used time: 0.0010 sec
```

经过验证，函数能够成功实现。

```
PS C:\Users\陈圳煌\Desktop\计算机学院\计算机学院（大三上）\高性能程序设计\test4> ./task3 8
1024 1024 1024
Used time: 0.8940 sec
```

进行 8 线程下 1024*1024 规模的矩阵乘法运算，运算时间与 task1 中的接近，说明改函数能够达到并行的效果。

4. 实验感想

在这次实验中，我了解并学习了 OpenMP 库的使用方法，并初步使用该库实现矩阵乘法、数组求和，与 MPI 库和 Pthread 库相比，OpenMP 库的使用更为方便，不需要自己手动分配每个线程的任务，只需要通过指令定义线程数以及调度方法等，感觉到了很大的便利。最后我用 Pthread 库自己实现了 parallel_for，刚开始我没有思路，不清楚如何对矩阵数据进行传入，并在自己的函数内定义多线程。后来经过思考和请教其他同学，我将矩阵数据和矩阵的规模作为参数 arg 的一部分传入，并按照 Pthread 实验中的方法进行创建线程、并行计算矩阵乘法。这次的实验让我对并行计算更加数据，也提升了我的实践能力。