实验二 多核环境下 OpenMP 并行编程

18120482 蔡卓悦

实验 2-1. OpenMP 程序的编译和运行

1. 实验目的

1）在 Linux 平台或 Windows 平台上编译和运行 OpenMP 程序；

2）掌握 OpenMP 并行编程基础。

2. 实验环境

1）硬件环境：计算机一台；

2）软件环境：win10，VSCode+mingw64；

3. 实验内容

OpenMP 是一个共享存储并行系统上的应用编程接口，支持 C/C++和 FORTRAN 等语言，编译和运行简单的"Hello World"程序。在 VSCode 中编辑 hellomp.c 源程序，用"gcc -fopenmp -O2 -o ./hellomp ./hellomp.c"命令编译，用"./hellomp "命令运行程序。

gcc -fopenmp -O2 -o hellomp.out hellomp.c：

-o file 后接生成的可执行文件名；

-fopenmp Enable handling of OpenMP directives "#pragma omp" in C/C++；

-O2 Optimize even more. 使用 O2 优化选项

```
#include <omp.h>

#include <stdio.h>

int main()
```

```c
{
    int nthreads,tid;
    omp_set_num_threads(8);
    #pragma omp parallel private(nthreads,tid)
    {
        tid=omp_get_thread_num();
        printf("Hello World from OMP thread %d\n",tid);
        if(tid==0)
        {
            nthreads=omp_get_num_threads();
            printf("Number of threads is %d\n",nthreads);
        }
    }
}
```

实验结果

```
Hello World from OMP thread 3
Hello World from OMP thread 0
Number of threads is 8
Hello World from OMP thread 6
Hello World from OMP thread 5
Hello World from OMP thread 1
Hello World from OMP thread 2
Hello World from OMP thread 7
Hello World from OMP thread 4
```

omp_set_num_threads(8); 设置了子线程数为 8，即是可以有 8 个子线程并行运行。

#pragma omp parallel private(nthreads,tid) 为编译制导语句，每个线程都自己的

nthreads 和 tid 两个私有变量，线程对私有变量的修改不影响其它线程中的该变量。

如果将 private 改成 shared，tid 等于 0 时输出了 hello world 后，其他线程可能在输出总线程数的语句之前更改 tid，使得 if 判断无法为真，不能输出总线程数。

```
#include <omp.h>

#include <stdio.h>

int main()

{

    int nthreads,tid;

    omp_set_num_threads(8);

    #pragma omp parallel shared(nthreads,tid)

    {

        tid=omp_get_thread_num();

        printf("Hello World from OMP thread %d\n",tid);

        if(tid==0)

        {

            nthreads=omp_get_num_threads();

            printf("Number of threads is %d\n",nthreads);

        }

    }

}
```

实验结果

```
Hello World from OMP thread 3
Hello World from OMP thread 5
Hello World from OMP thread 4
Hello World from OMP thread 2
Hello World from OMP thread 0
Hello World from OMP thread 6
Hello World from OMP thread 7
Hello World from OMP thread 1
```

2-2 矩阵乘法的 OpenMP 实现及性能分析

1. 实验目的

1）用 OpenMP 实现最基本的数值算法“矩阵乘法”

2）掌握 for 编译制导语句

3）对并行程序进行简单的性能调优

2. 实验内容

1）运行并测试矩阵相乘程序

用 OpenMP 编写两个 n 阶的方阵 a 和 b 的相乘程序，结果存放在方阵 c 中，**其中乘法用 for 编译制导语句实现并行化操作**，并调节 for 编译制导中 schedule 的参数，使得执行时间最短。要求在 window 环境（不用虚拟机），在 linux 环境（用和不用虚拟机情况下）测试程序的性能，并写出详细的分析报告。

```
#include <stdio.h>

#include <omp.h>

#include <time.h>

void comput(float *A, float *B, float *C) //两个矩阵相乘传统方法
```

```
{
    int x, y;
    for (y = 0; y < 4; y++)
    {
        for (x = 0; x < 4; x++)
        {
            C[4 * y + x] = A[4 * y + 0] * B[4 * 0 + x] + A[4 * y + 1] * B[4 * 1 + x] +
                           A[4 * y + 2] * B[4 * 2 + x] + A[4 * y + 3] * B[4 * 3 + x];
        }
    }
}

int main()
{
    double duration,
        s, f;
    int x = 0;
    int y = 0;
    int n = 0;
    int k = 0;
    float A[] = {1, 2, 3, 4,
                 5, 6, 7, 8,
```

```
            9, 10, 11, 12,
            13, 14, 15, 16};
float B[] = {0.1f, 0.2f, 0.3f, 0.4f,
            0.5f, 0.6f, 0.7f, 0.8f,
            0.9f, 0.10f, 0.11f, 0.12f,
            0.13f, 0.14f, 0.15f, 0.16f};
float C[16];


s = omp_get_wtime();
for (n = 0; n < 10000000; n++)
    comput(A, B, C);
f = omp_get_wtime();
duration = f - s;
printf("Serial    :%f\n", duration);
for (y = 0; y < 4; y++)
{
    for (x = 0; x < 4; x++)
        printf("%f,", C[y * 4 + x]);
    printf("\n");
}


//parallel 2
```

```
    s = omp_get_wtime();
#pragma omp parallel for num_threads(2)
    for (n = 0; n < 10000000; n++)
        comput(A, B, C);
    f = omp_get_wtime();
    duration = f – s;
    printf("Parallel 2 :%f\n", duration);


    //parallel 4
    s = omp_get_wtime();
#pragma omp parallel for num_threads(4)
    for (n = 0; n < 10000000; n++)
        comput(A, B, C);
    f = omp_get_wtime();
    duration = f – s;
    printf("Parallel 4 :%f\n", duration);


    s = omp_get_wtime();
#pragma omp parallel for num_threads(8)
    for (n = 0; n < 10000000; n++)
        comput(A, B, C);
    f = omp_get_wtime();
```

```
    duration = f - s;

    printf("Parallel 8 :%f\n", duration);

    for (y = 0; y < 4; y++)

    {

        for (x = 0; x < 4; x++)

            printf("%f,", C[y * 4 + x]);

        printf("\n");

    }



    s = omp_get_wtime();

#pragma omp parallel for num_threads(16)

    for (n = 0; n < 10000000; n++)

        comput(A, B, C);

    f = omp_get_wtime();

    duration = f - s;

    printf("Parallel 16 :%f\n", duration);



    return 0;

}
```

实验结果

```
Serial     :1.050000
4.320000,2.260000,2.630000,3.000000,
10.839999,6.420000,7.670000,8.920000,
17.359999,10.580001,12.710000,14.840000,
23.879999,14.740001,17.750000,20.759998,
Parallel 2 :0.676000
Parallel 4 :0.653000
Parallel 8 :0.510000
4.320000,2.260000,2.630000,3.000000,
10.839999,6.420000,7.670000,8.920000,
17.359999,10.580001,12.710000,14.840000,
23.879999,14.740001,17.750000,20.759998,
Parallel 16 :0.504000
```

2）请自己找一个需要大量计算但是程序不是很长的程序，实现 OMP 的多线程并行计算

要求写出并行算法，并分析并行的效果

```
#include <stdio.h>

#include <omp.h>

#include <time.h>

#include <stdlib.h>



double matrix[10000][10000] = {{0}};



double compute_serial()
```

```
{
    double sum = 0;
    //for (int k = 0; k < 1000; k++)
    //{
    //    sum=0;
        for (int i = 0; i < 10000; i++)
        {
            for (int j = 0; j < 10000; j++)
            {
                sum += matrix[i][j];
            }
        }
    //}
    return sum;
}


double compute_eight_private()
{
    double sum = 0;
    omp_set_num_threads(8);
#pragma omp parallel for reduction(+:sum)
        //for (int k = 0; k < 1000; k++)
```

```
        //{
        //    sum=0;
            for (int i = 0; i < 10000; i++)
            {
                for (int j = 0; j < 10000; j++)
                {
                    sum += matrix[i][j];
                }
            }
        //}
    return sum;
}


double compute_eight()
{
    double sum = 0, x = 0;
    omp_set_num_threads(8);
    #pragma omp parallel for shared(sum)
        //for (int k = 0; k < 1000; k++)
        //{
        //    sum=0;
            for (int i = 0; i < 10000; i++)
```

```
            {
                for (int j = 0; j < 10000; j++)
                {
                    sum += matrix[i][j];
                }
            }
        //}
    return sum;
}


int main()
{
    double start, end, duration;
    double sum = 0;
    for (int i = 0; i < 10000; i++)
    {
        for (int j = 0; j < 10000; j++)
        {
            matrix[i][j] = i / (j + 1);
        }
    }
```

```
start = omp_get_wtime();

sum = compute_serial();

printf("%f\n", sum);

end = omp_get_wtime();

duration = end – start;

printf("Serial     :%f\n", duration);


start = omp_get_wtime();

sum = compute_eight_private();

printf("%f\n", sum);

end = omp_get_wtime();

duration = end – start;

printf("Parallel 8 with reduction:%f\n", duration);


start = omp_get_wtime();

sum = compute_eight();

printf("%f\n", sum);

end = omp_get_wtime();

duration = end – start;

printf("Parallel 8 without reduction:%f\n", duration);


return 0;
```

```
}
```

实验结果：

```
443241016.000000
Serial     :0.280000
443241016.000000
Parallel 8 with reduction:0.056000
63396564.000000
Parallel 8 without reduction:0.285000
```

算法：

建立一个 10000*10000 的矩阵，每个元素都由行号除以列号得到，将矩阵中的元素相加，分别进行：

① 串行计算；

② 带有规约语句 reduction（对指定变量，在每个线程创建一份副本，每个线程完成时，都将副本的结果加起来，以此保证数据不出错）的八线程并行计算；

③ 不对数据进行保护的八线程并行计算。

结果：①、②结果相同，③数据有误，且②的运算速度比①和③快了一个量级；说明规约语句可以保护数据，同时减少了线程创建、分配、切换、删除等操作的开销。