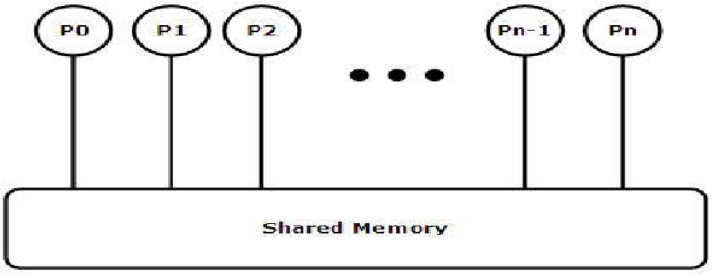
实验二 多核环境下 OpenMP 并行编程

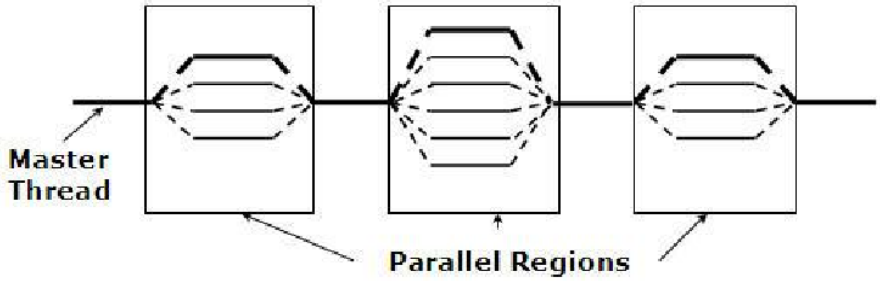
一 自学知识：OpenMP 简介

OpenMP 是由 OpenMP Architecture Review Board 牵头提出的，并已被广泛接受的，用于共享内存并行系统的多线程程序设计的一套指导性注释(Compiler Directive)。OpenMP 是一种面向共享内存以及分布式共享内存的多处理器多线程并行编程语言，能被用于显示指导多线程、共享内存并行的应用程序编程接口。其规范由 SGI 发起，具有良好的可移植性，支持多种编言，支持多种平台，包括大多数的类 UNIX 以及 WindowsNT 系统。OpenMP 最初是为了共享内存多处理的系统结构设计的并行编程方法，与通过消息传递进行并行编程的模型有很大的区别，多个处理器在访问内存的时候使用的是相同的内存编址空间。SMP 是一种共享内存的体系结构，同时分布式共享内存的系统也属于共享内存的多处理器结构，分布式共享内存将多机的内存资源通过虚拟化的方式形成一个相同的内存空间提供给多个机子上的处理器使用，OpenMP 对这样的机器也提供了一定的支持。所以 OpenMP 提供了对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的 pragma 来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。当选择忽略这些 pragma，或者编译器不支持 OpenMP 时，程序又可退化为通常的程序(一般为串行)，代码仍然可以正常运作，只是不能利用多线程来加速程序执行。

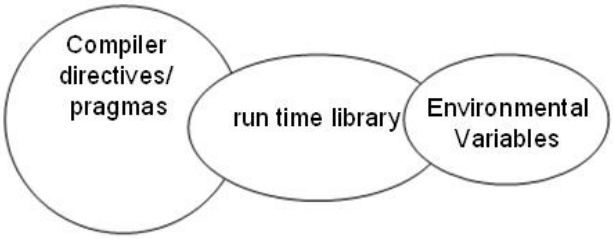


OpenMP 的编程模型以线程为基础，通过编译指导语句来显示地指导并

行化，为编程人员提供了对并行化的完整控制。
OpenMP 的执行模型采用 Fork-Join 的形式，Fork-Join 执行模式在开始执行的时候，只有一个叫做“主线程“的运行线程存在。主线程在运行过程中，当遇到需要进行并行计算的时候，派生出线程来进行并行执行，在并行执行的时候，主线程和派生线程共同工作，在并行代码结束后，派生线程退出或者是挂起，不再工作，控制流程回到单独的主线程中。



OpenMP 的功能由两种形式提供：编译指导语句和运行时库函数，并通过环境变量的方式灵活控制程序的运行。



编译指导语句：
指的是在编译器编译程序的时候，会识别特定的注释，而这些特定的注释就包含着 OpenMP 程序的一些语句。在 C/C++程序中，OpenMP 是以 #pragma omp 开始，后面跟具体的功能指令。即具有如下的形式：
#pragma omp<directive> [clause[[,]clause]…s]
其中的 directive 部分就包含了具体的编译指导语句，包括 parallel、for、parallel、for、section、sections、single、mater、ritical、flush、ordered 和 atomic。这些编译指导语句或者是用来分配任务，或者是用来同步。可选子句 clause 给出了相应的编译器指导语句的参数，子句可以影响到编译指导语句的具体行为，每个编译指导语句都有一系列适合它的子句。其中有 5 个编译指导语句不能跟别的子句：master、critical、flush、ordered、atomic。
运行时库函数：

OpenMP 运行时库函数原本用以设置和获取执行环境相关的信息。其也包含一系列用以同步的 API。要使用运行时函数库所包含的函数，应该在相应的源文件中包含 OpenMP 头文件即 omp.h。OpenMP 的运行时库函数的使用类似于相应编程语言内部的函数调用。
由编译指导语句和运行时库函数可见，OpenMP 同时结合了两种并行编程的方式，通过编译指导语句，可以将串行的程序逐步地改造成一个并行程序，达到增量更新程序的目的，从而减少程序编写人员的一定负担。同时，这样的方式也能将串行程序和并行程序保存在同一个源代码文件当中，减少了维护的负担。OpenMP 在运行的时候，需要运行函数库的支持，并会获取一些环境变量来控制运行的过程。这里提到的环境变量是动态函数库中用来控制函数运行的一些参数。
上面讲到的两种形式中，编译指导语句是 OpenMP 组成中最重要的部分，也是编写 OpenMP 程序的关键。

二 实验 2.1 多环境 OpenMP 程序的编译和运行

1 实验目的及要求

1) 掌握 OpenMP 并行编程基础；
2) 掌握在 Linux 平台上编译和运行 OpenMP 程序；
3) 掌握在 Windows 平台上编译和运行 OpenMP 程序。

2 Linux 下 OpenMP 程序的编译和运行

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 Linux 下编辑 hellomp.c 源程序，用"gcc -fopenmp -O2 -o hellomp.out hellomp.c"命令编译，用"./hellomp.out"命令运行程序。

注：在虚拟机中当使用 vi 编辑文件时，不是以 ESC 键退出插入模式，可以使用“Ctrl+c”进入命令模式，然后输入 wq 进行存盘退出。

1) 代码实现：

[xiaofei@master ompdemo]$ vim helloworld.c

```
#include <omp.h>
#include <stdio.h>
int main()
{
        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);
            }
        }
}
```

[xiaofei@master ompdemo]$ gcc -fopenmp -O2 -o helloomp.out helloworld.c
[xiaofei@master ompdemo]$ ./helloomp.out //执行
Hello World from OMP thread 0
Number of threads is 8
Hello World from OMP thread 4
Hello World from OMP thread 5
Hello World from OMP thread 6
Hello World from OMP thread 7
Hello World from OMP thread 3
Hello World from OMP thread 2
Hello World from OMP thread 1

2) OMP 在 Linux 实验环境说明

a) **首先需要安装 gcc ：** yum intall gcc (在 redhat, centOS 下使用，用 root 安装，默认安装)
sudo apt-get install gcc (在 ubuntu 下使用)

“#pragma omp parallel”是一条 OpenMP 标准的语句，它的含义是让它后面的语句按照多线程来执行。需要注意的是每个线程都去做相同的事情。

b) **编译语句 ” gcc -fopenmp -O2 -o helloomp.out helloworld.c ”** 的参数解释：

-o file 后接生成的可执行文件名。

Place output in file file. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.
If -o is not specified, the default is to put an executable file in a.out, the object file for source.suffix in source.o, its assembler file in source.s, a precompiled header file in source.suffix.gch, and all preprocessed C source on standard output.

-fopenmp

Enable handling of OpenMP directives "#pragma omp" in C/C++ and "!$omp" in Fortran. When -fopenmp is specified, the compiler generates parallel code according to the OpenMP Application Program Interface v2.5 <http://www.openmp.org/>. This option implies -pthread, and thus is only supported on targets that have support for -pthread.

-O2 Optimize even more. 使用 O2 优化选项

GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code.

c) 控制并行执行的线程数：

首先使用系统默认线程数——即默认逻辑 CPU 数量(程序中没有设置线程数)

查看默认线程数（逻辑 CPU 的个数）：

grep 'processor' /proc/cpuinfo | sort -u | wc -l

```
[xiaofei@master ompdemo]$ grep 'processor' /proc/cpuinfo | sort -u | wc -l
1
```

执行程序，查看结果：

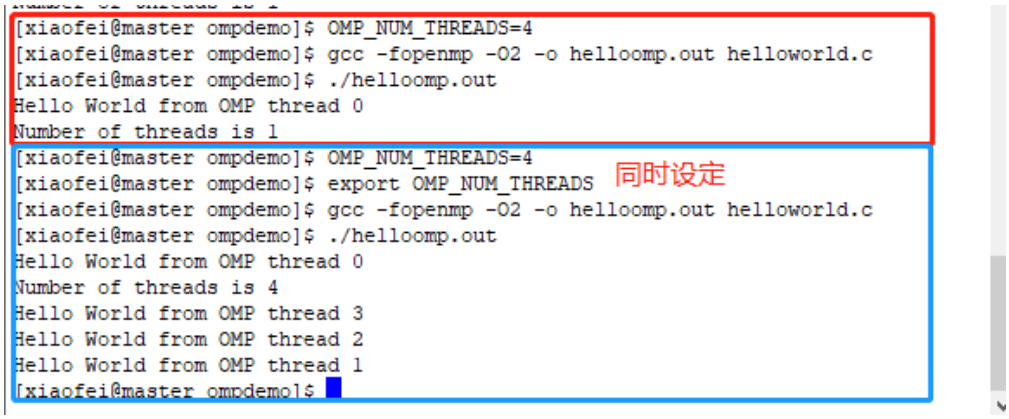
```
[xiaofei@master ompdemo]$ ./helloomp.out
Hello World from OMP thread 0
Number of threads is 1
```

其次使用系统命令设置线程数（程序中没有设置线程数）：

根据算法的要求和硬件情况，例如 CPU 数量或者核数，选择适合的线程数可以加速程序的运行。请按照下列的方法进行线程数量的设置。

[xiaofei@master ompdemo]$ OMP_NUM_THREADS=4 //设置线程数为 10

[xi… …emo]$ export OMP_NUM_THREADS //将线程数添加为临时环境变量
问题：如何才能将其设为长期换进变量？



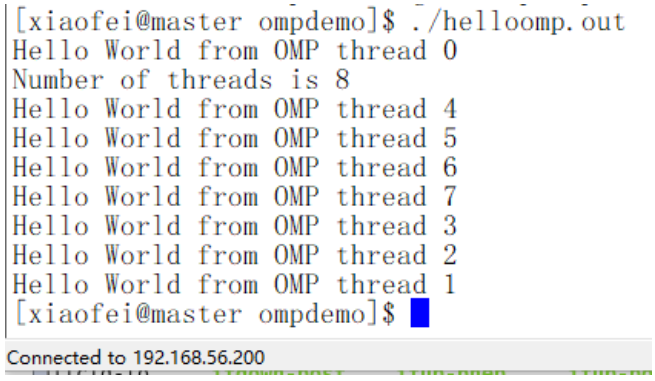
最后利用程序代码设置线程数：

omp_set_num_threads(8);

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。

程序的功能是对于每个线程都打印出它的 id 号，对于 id 号为 0 的线程打印出线程数目。

程序的运行结果下图所示：



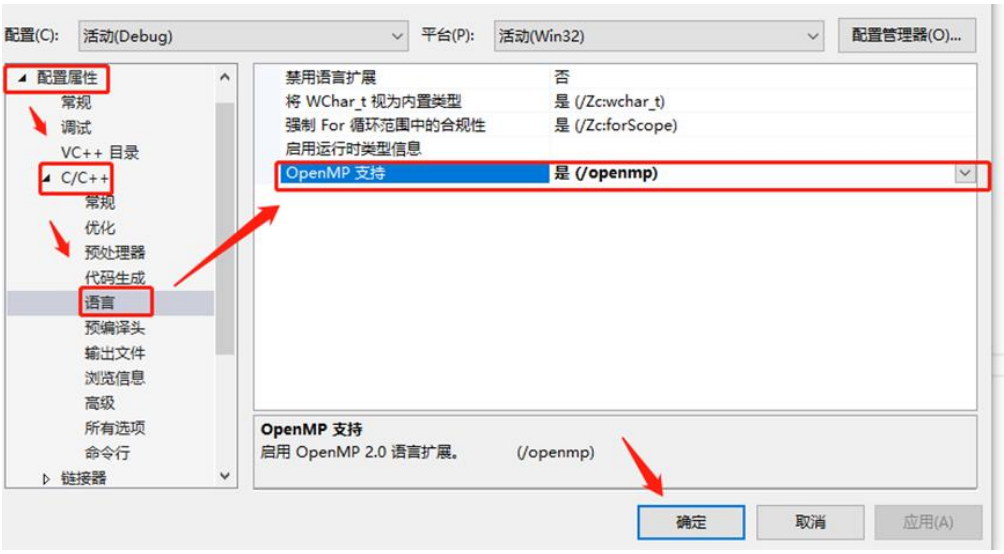
程序运行结果截图

3 Windows 下 OpenMP 程序的编译和运行。

用 VS2013 编辑上述的 hellomp.c 源程序，注意在菜单“项目->属性->C/C++->语言”选中“OpenMP 支持”，编译并运行程序。

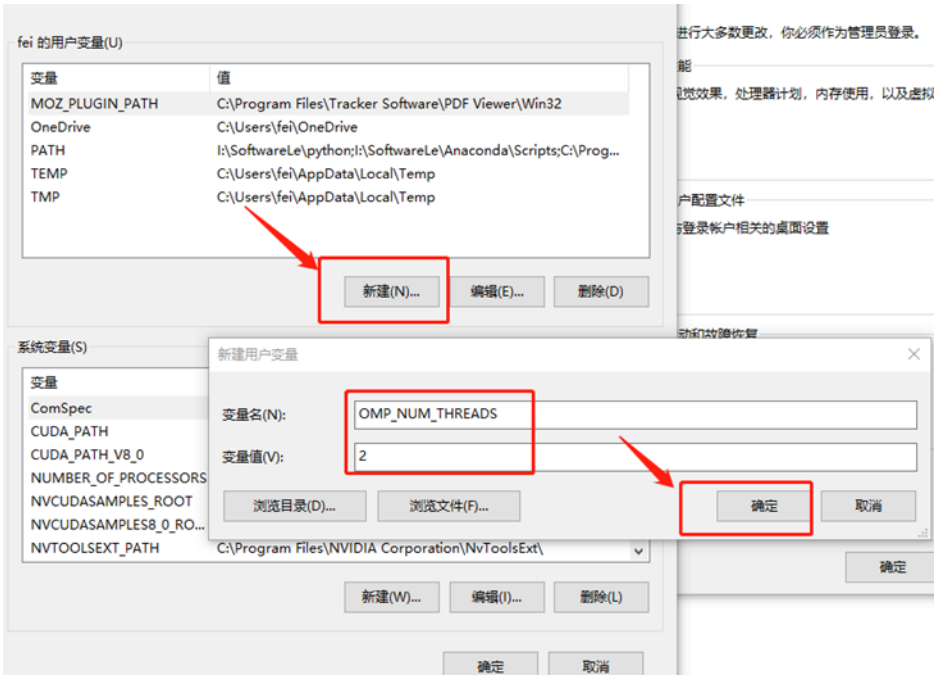
打开或者新建一个 c++项目，依次选择 Project -> 属性 -> 配置属性(configuration property) -> c/c++ -> 语言(Language)，打开 OpenMP 支持。

在 VS2017 中就可以使用，具体的：新建一个 C/C++程序，项目--属性--C/C++--语言--OpenMP 支持，把 OpenMP 打开。然后编写带编译指令的并行程序，注意一定要加上<omp.h>头文件。https://www.cnblogs.com/lfri/p/10111315.html



设置环境变量 OMP_NUM_THREADS

操作：我的电脑 -> 属性 -> 高级 -> 环境变量，新建一个 OMP_NUM_THREADS 变量，值设为 2，即为程序执行的线程数。

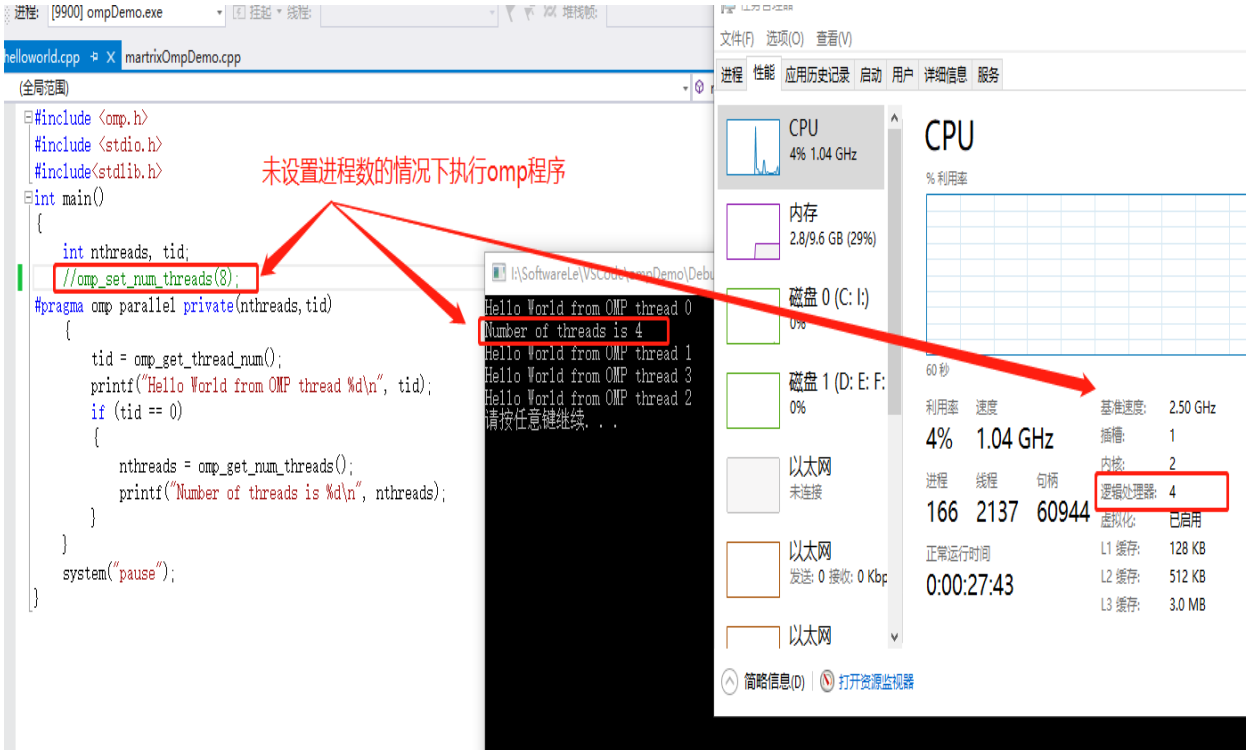


若程序里面没有设置线程数，执行上面的程序只有两个线程，结果如下：

```
int main()
{
    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);
        }
    }
    system("pause");
}
```

```
I:\SoftwareLe\VSCode\ompDemo\Debug\ompDemo.
Hello World from OMP thread 0
Number of threads is 2
Hello World from OMP thread 1
请按任意键继续. . .
```

没有设置 OMP_NUM_THREADS 的情况下，系统默认线程数为逻辑处理器数目，如下所示：



虽然线程都是一起开始运行，但实验中每次运行的结果都不一样，这个是因为每次每个线程结束的先后可能不一样的。所以每次运行的结果都是随机的。这是串行程序和并行程序不同的地方：串行程序可以重新运行，结果和之前一样；并行程序却因为执行次序无法控制可能导致每次的结果都不一样。

三 实验 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 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）测试程序的性能并分析，写在实验报告中。

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，分析性能，提出改进方法（可实现，也可以理论分析），写在实验报告中。

3 实验报告

内容：1）和 2），其中 2）可以挑选 PPT 中的程序段及性能分析。

附件1-矩阵相乘代码示例：

```
#include<stdio.h>
#include<omp.h>
#include<time.h>
#include<stdlib.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, durations,speedrate;
    clock_t start, finalt;
    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];

    start = clock();

    //#pragma omp parallel if(false)
    for (n = 0; n<1000000; n++)
    {
        comput(A, B, C);
    }
    finalt = clock();
    durations = (double)(finalt - start) / CLOCKS_PER_SEC;  //duration
表示持续的时间
    printf("1,000,000次串行时间是:%f：%f\n", durations);

    //输出矩阵相乘结果
    printf("\n串行输出矩阵相乘结果：\n");

    for (y = 0; y<4; y++)
    {
        for (x = 0; x<4; x++)
        {
            printf("%f,", C[y * 4 + x]);
        }
        printf("\n");
    }

    printf("\n====================\n");


    //parallel 1 单线程并行
    start = clock();
#pragma omp parallel for
    for (n = 0; n<1000000; n++)
    {
        comput(A, B, C);
    }
    finalt = clock();
    duration = (double)(finalt - start) / CLOCKS_PER_SEC;

    speedrate = (durations / duration) * 100;
```

```
    printf("p1：1,000,000次单线程并行时间是:%f      加速比是：%f\n",
duration, speedrate);


    //parallel 2   双线程并行
    start = clock();
#pragma omp parallel for
    for (n = 0; n<2; n++)   //CPU是核线程的
    {
        for (k = 0; k<500000; k++)      //每个线程管个循环
        {
            comput(A, B, C);
        }
    }
    finalt = clock();
    duration = (double)(finalt - start) / CLOCKS_PER_SEC;
    speedrate = (durations / duration) * 100;
    printf("p2：1,000,000次双线程并行时间是:%f      加速比是：%f\n",
duration,speedrate);


    //parallel 3
    start = clock();
#pragma omp parallel for
    for (n = 0; n<4; n++)   //CPU是核线程的
    {
        for (k = 0; k<250000; k++)//每个线程管个循环
        {
            comput(A, B, C);
        }
    }
    finalt = clock();
    duration = (double)(finalt - start) / CLOCKS_PER_SEC;
    speedrate = (durations / duration)*100;
    printf("p3：1,000,000次四线程并行时间是:%f      加速比是：%f\n",
duration, speedrate);



    printf("矩阵相乘的结果如下：\n");
```

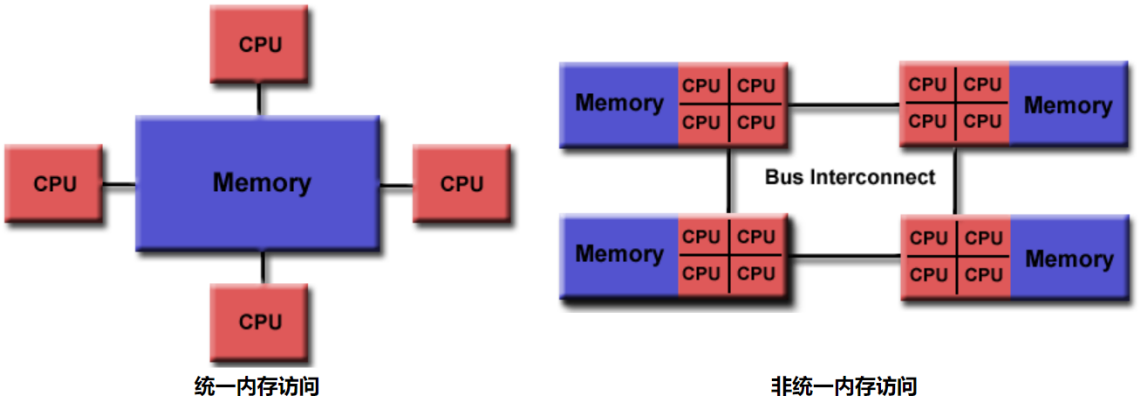
```
    for (y = 0; y<4; y++)
    {
        for (x = 0; x<4; x++)
        {
            printf("%f,     ", C[y * 4 + x]);
        }
        printf("\n");
    }


    system("pause");
    return 0;
}
```

什么是 OpenMP

Open Multi-Processing 的缩写，是一个应用程序接口（API），可用于显式指导多线程、共享内存的并行性。
在项目程序已经完成好的情况下不需要大幅度的修改源代码，只需要加上专用的 pragma 来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。当选择忽略这些 pragma，或者编译器不支持 OpenMp 时，程序又可退化为通常的程序(一般为串行)，代码仍然可以正常运作，只是不能利用多线程来加速程序执行。OpenMP 提供的这种对于并行描述的高层抽象降低了并行编程的难度和复杂度，这样程序员可以把更多的精力投入到并行算法本身，而非其具体实现细节。对基于数据分集的多线程程序设计，OpenMP 是一个很好的选择。
OpenMP 支持的语言包括 C/C++、Fortran；而支持 OpenMP 的编译器 VS、gcc、clang 等都行。可移植性也很好：Unix/Linux 和 Windows

OpenMP 编程模型

内存共享模型：OpenMP 是专为多处理器/核，共享内存机器所设计的。底层架构可以是 UMA 和 NUMA。即(Uniform Memory Access 和 Non-Uniform Memory Access)



基于线程的并行性

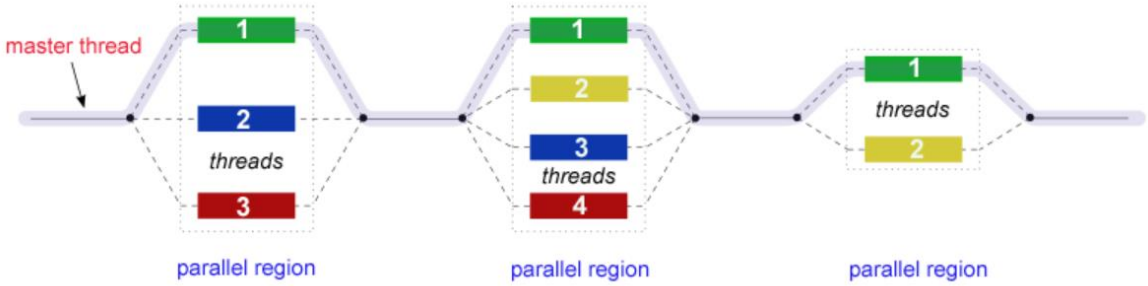
- OpenMP 仅通过线程来完成并行
- 一个线程的运行是可由操作系统调用的最小处理单
- 线程们存在于单个进程的资源中，没有了这个进程，线程也不存在了
- 通常，线程数与机器的处理器/核数相匹配，然而，实际使用取决与应用程序

明确的并行

- OpenMP 是一种显式（非自动）编程模型，为程序员提供对并行化的完全控制
- 一方面，并行化可像执行串行程序和插入编译指令那样简单

- 另一方面，像插入子程序来设置多级并行、锁、甚至嵌套锁一样复杂

Fork-Join 模型



- OpenMP 就是采用 Fork-Join 模型
- 所有的 OpenML 程序都以一个单个进程——master thread 开始，master threads 按顺序执行知道遇到第一个并行区域
- Fork：主线程创造一个并行线程组
- Join：当线程组完成并行区域的语句时，它们同步、终止，仅留下主线程

数据范围

- 由于 OpenMP 时是共享内存模型，默认情况下，在共享区域的大部分数据是被共享的
- 并行区域中的所有线程可以同时访问这个共享的数据
- 如果不需要默认的共享作用域，OpenMP 为程序员提供一种“显示”指定数据作用域的方法

嵌套并行

- API 提供在其它并行区域放置并行区域
- 实际实现也可能不支持

动态线程

- API 为运行环境提供动态的改变用于执行并行区域的线程数
- 实际实现也可能不支持

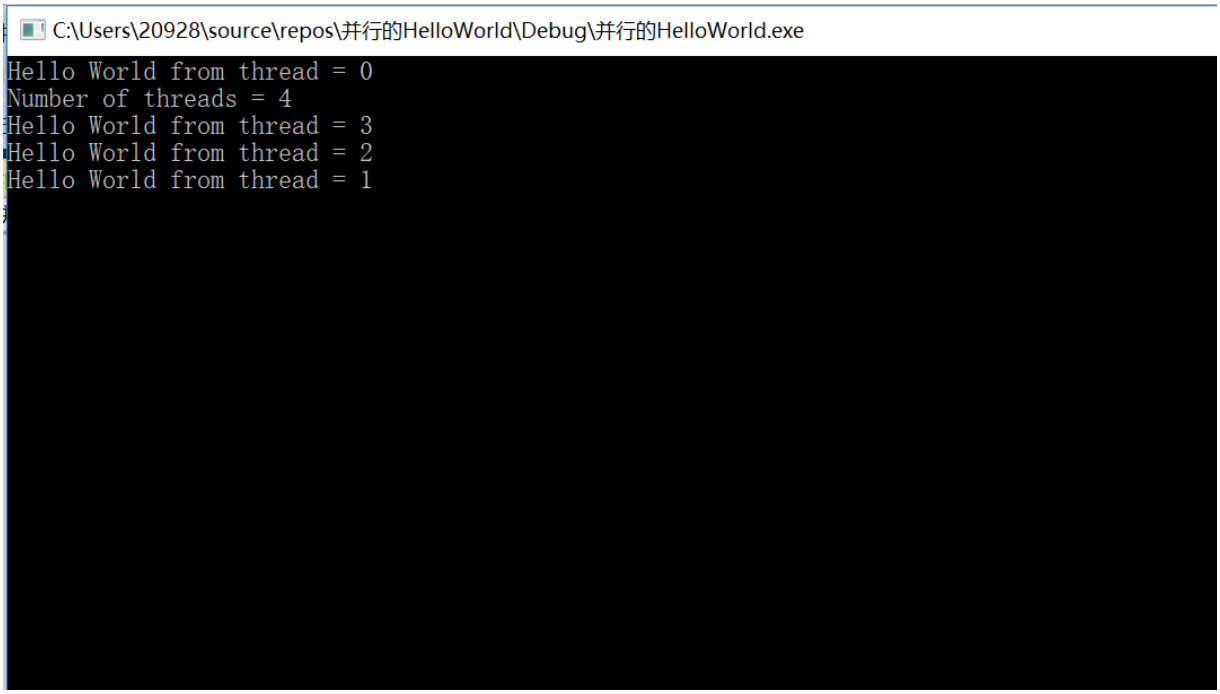
简单使用

在 VS2017 中就可以使用，具体的：新建一个 C/C++程序，项目--属性--C/C++--语言--OpenMP 支持，把 OpenMP 打开。然后编写带编译指令的并行程序，注意一定要加上<omp.h>头文件。
写一个并行的 Hello World

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
```

```
 4
 5 int main()
 6 {
 7     int nthreads, tid;
 8
 9     /* Fork a team of threads giving them their own copies of variables
*/
10     #pragma omp parallel private(nthreads, tid)
11     {
12
13         /* Obtain thread number */
14         tid = omp_get_thread_num();
15         printf("Hello World from thread = %d\n", tid);
16
17         /* Only master thread does this */
18         if (tid == 0)
19         {
20             nthreads = omp_get_num_threads();
21             printf("Number of threads = %d\n", nthreads);
22         }
23
24     }  /* All threads join master thread and disband */
25     return 0;
26 }
```

运行结果如下：



注：我的电脑默认是 4 个线程，不同的电脑运行结果不同，就算是同一部电脑每次运行的结果也可能不同（4 个线程并行执行，没有确定的先后顺序）

也可以直接使用 gcc 加上-fopenmp 编译，For example:

```
g++ test.cpp -o test -fopenmp
./test
```

(不知道我的 gcc 不行，只能用 g++，枯了)

补：直到原因了，gcc 默认编译链接不会链接 C++标准库，可以使用 g++编译链接（如上），也可以在 gcc 链接时显示指定链接 -lstdc++

```
gcc test.cpp -o test -fopenmp -lstdc++
```

至于 OpenMP 详细的编写格式和意义可以看这篇博客。

参考资料：

1、https://blog.csdn.net/wyjkk/article/details/6612108

2、*https://blog.csdn.net/HW140701/article/details/73716363*
3、*https://computing.llnl.gov/tutorials/openMP/#RunTimeLibrary*