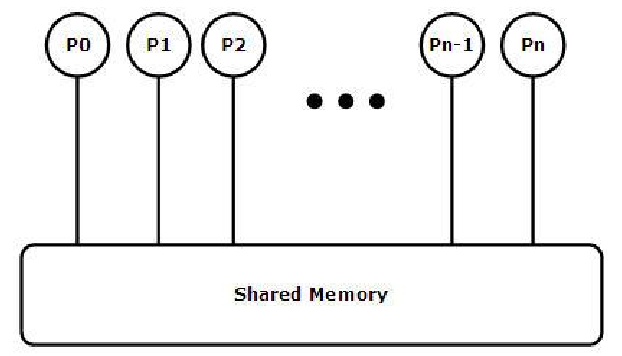
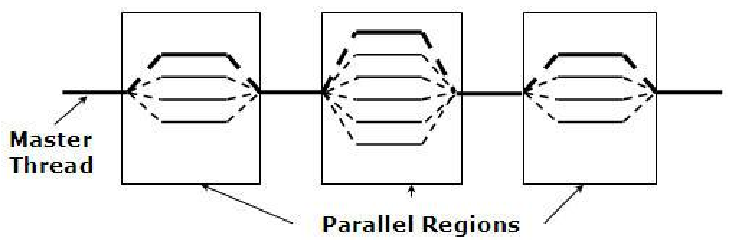
# 实验二 多核环境下OpenMP并行编程

## 一 自学知识：OpenMP简介

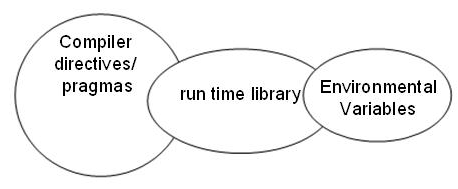
OpenMP是由OpenMP Architecture Review Board牵头提出的，并已被广泛接受的，用于共享内存并行系统的多线程程序设计的一套指导性注释(Compiler Directive)。OpenMP是一种面向共享内存以及分布式共享内存的多处理器多线程并行编程语言，能被用于显示指导多线程、共享内存并行的应用程序编程接口。其规范由SGI发起，具有良好的可移植性，支持多种编言，支持多种平台，包括大多数的类UNIX以及WindowsNT系统。OpenMP最初是为了共享内存多处理的系统结构设计的并行编程方法，与通过消息传递进行并行编程的模型有很大的区别，多个处理器在访问内存的时候使用的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。

#### 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

#### OMP在Linux实验环境说明

1. **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。

1. **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.

1. **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

C:\Users\fei\AppData\Local\Temp\1520992234(1).png

执行程序，查看结果：

C:\Users\fei\AppData\Local\Temp\1520992312(1).png

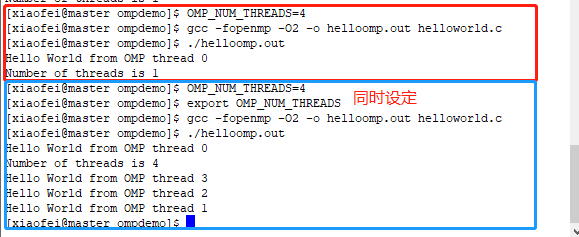
**其次使用系统命令设置线程数（程序中没有设置线程数）:**

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？



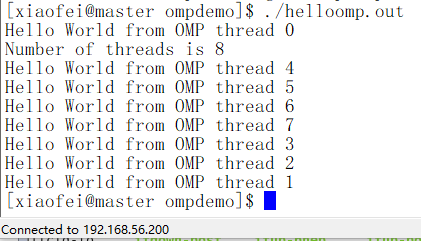
**最后利用程序代码设置线程数：**

**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：

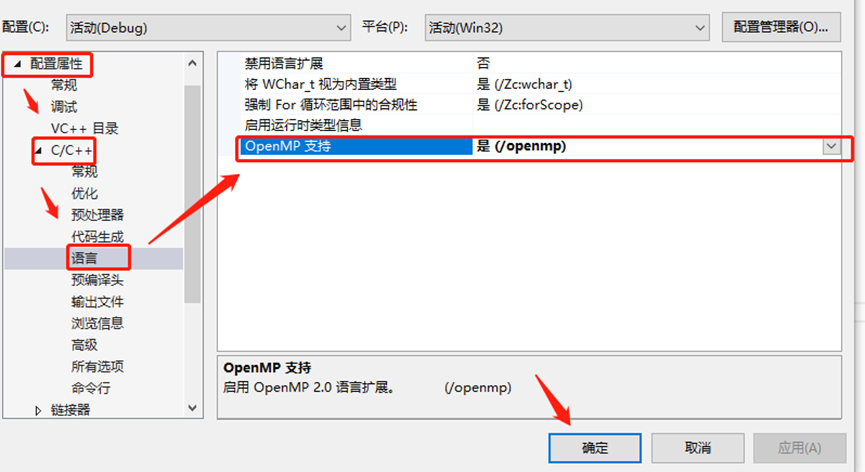


|  |
| --- |
|  |
| 程序运行结果截图 |

### 3 Windows下OpenMP程序的编译和运行。

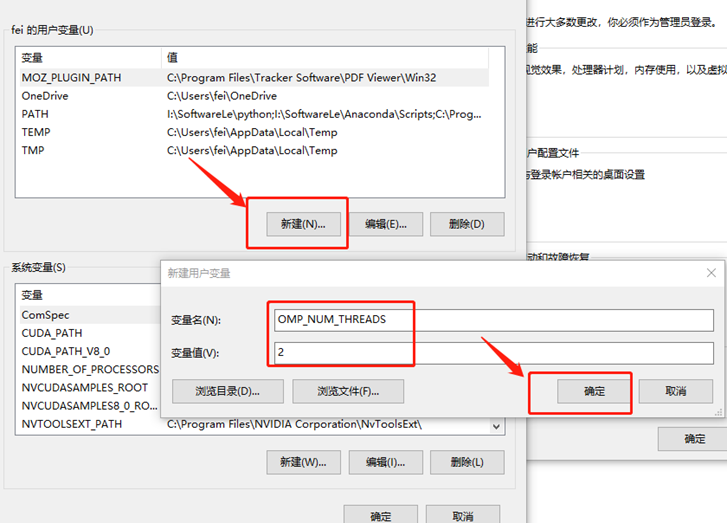
用VS2013编辑上述的hellomp.c源程序，注意在菜单“项目->属性->C/C++->语言”选中“OpenMP支持”，编译并运行程序。

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。

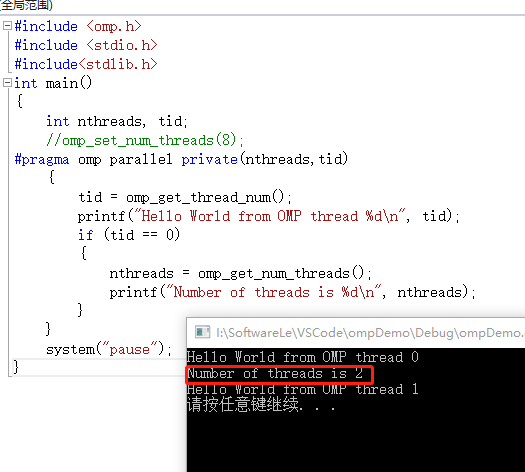
****

**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。

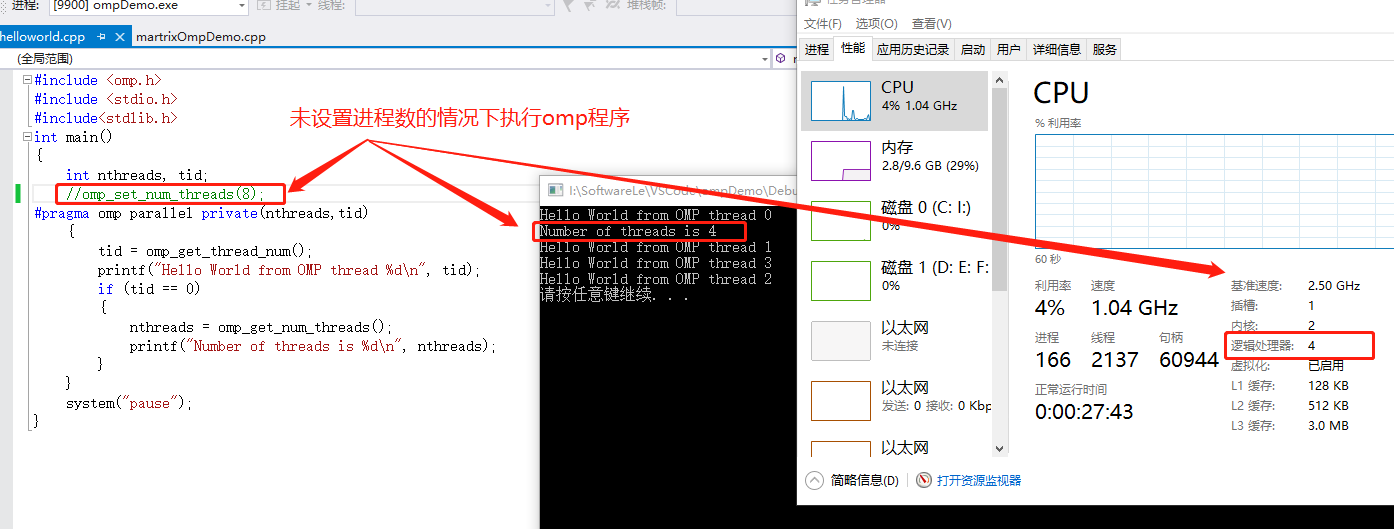


若程序里面没有设置线程数，执行上面的程序只有两个线程，结果如下：



|  |
| --- |
|  |

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，并写出详细的分析报告。

源代码见附件1

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（注：必须核对串行和并行的计算结果，保证正确性）

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;

}