电工电子实验中心

实 验 系 列 教 材

多 核 架 构 及 编 程 技 术

实 验 指 导 书

《多核架构及编程技术》实验课程组 编

武汉大学电子信息学院

目　　录

实验一 Windows 程序设计基础

【实验目的】

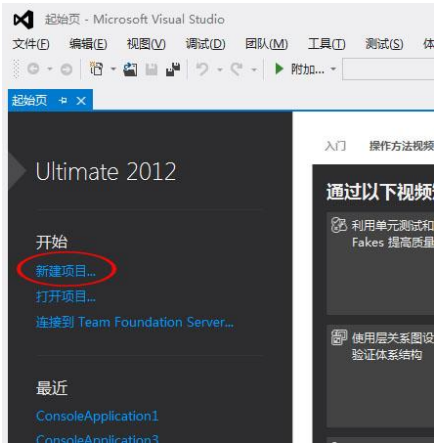
1、掌握 MS Visual Studio 集成开发环境的使用；

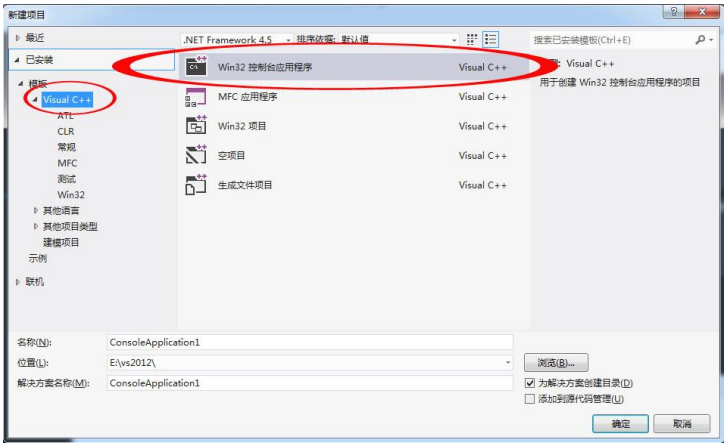
2、掌握使用 MS Visual Studio 进行 C/C++程序设计、编辑、调试和运行。

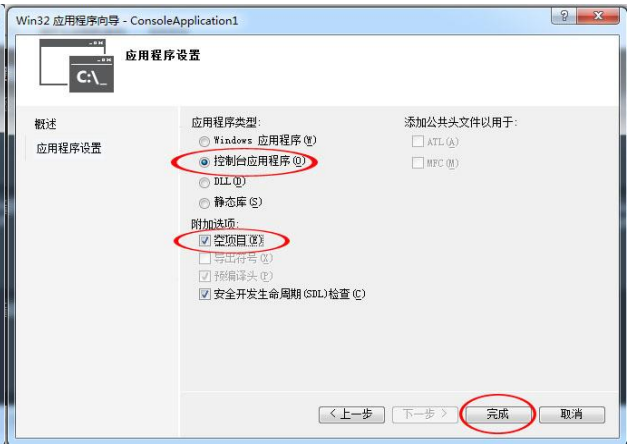
【实验内容】

内容一 MS Visual Studio 集成开发环境的使用

1、新建控制台应用程序

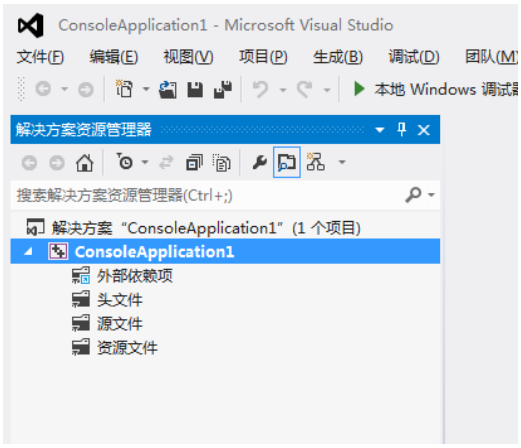


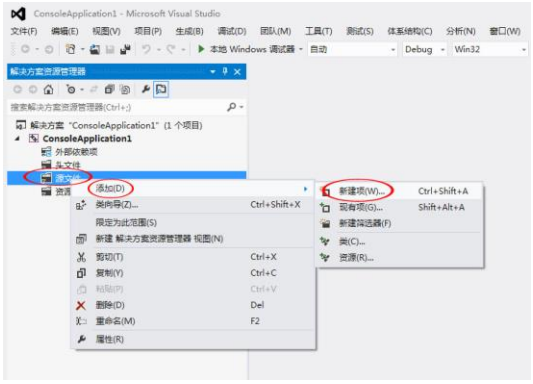


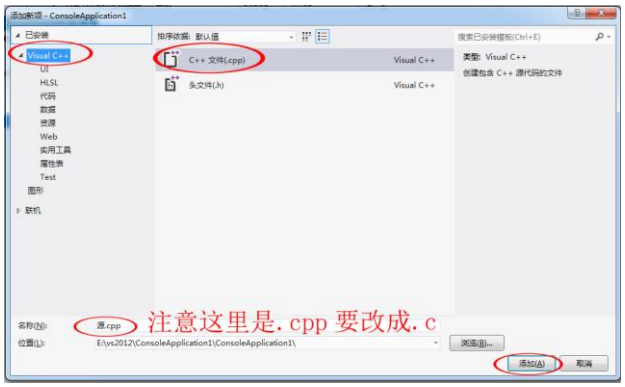


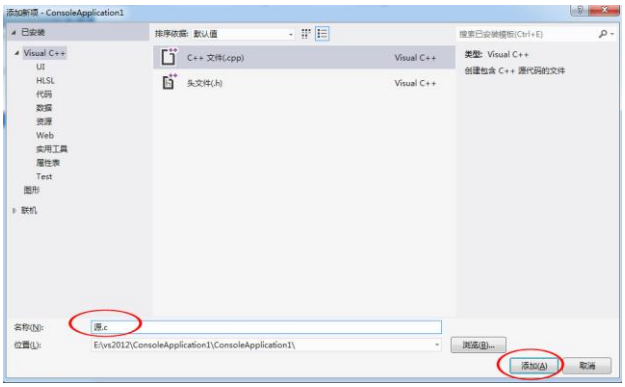
2、添加 C/C++文件


ConsoleApplication1 - Microsoft Visual Studio
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D) 团队(M) 工具(T) 测试(S)
解决方案资源管理器(P) Ctrl+Alt+L
团队资源管理器(M) Ctrl+\, Ctrl+M
体系结构资源管理器(A) Ctrl+\, Ctrl+R
书签窗口(B) Ctrl+K, Ctrl+W
调用层次结构(H) Ctrl+Alt+K
类视图(A) Ctrl+Shift+C
代码定义窗口(D) Ctrl+Shift+V
对象浏览器(J) Ctrl+Alt+J
错误列表(I) Ctrl+\, E
输出(O) Alt+2
资源视图(R) Ctrl+Shift+E
起始页(G)
工具箱(X) Ctrl+Alt+X
自动

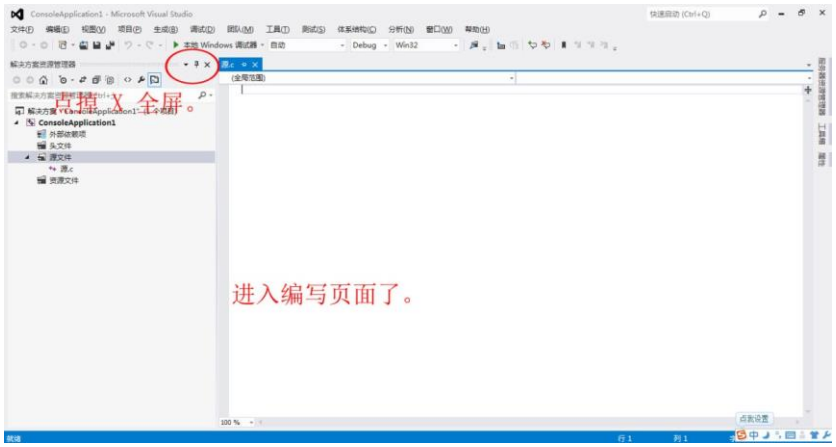

ConsoleApplication1 - Microsoft Visual Studio
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D) 团队(M)
本地 Windows 调试
解决方案资源管理器
搜索解决方案资源管理器(Ctrl+;)
解决方案 "ConsoleApplication1" (1 个项目)
ConsoleApplication1
外部依赖项
头文件
源文件
资源文件

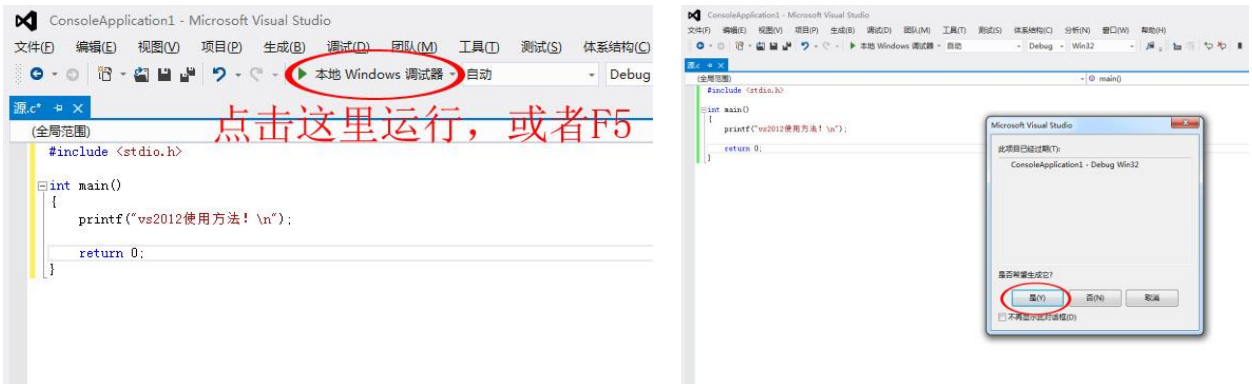

ConsoleApplication1 - Microsoft Visual Studio
解决方案资源管理器
ConsoleApplication1
外部依赖项
头文件
源文件
资源文件
添加(D)
新建项(W)... Ctrl+Shift+A
现有项(G)... Shift+Alt+A
新建筛选器(F)
类(C)...
资源(R)...
类向导(Z)... Ctrl+Shift+X
限定为此范围(S)
新建 解决方案资源管理器 视图(N)
剪切(T) Ctrl+X
复制(Y) Ctrl+C
粘贴(P) Ctrl+V
删除(D) Del
重命名(M) F2
属性(R)


添加新项 - ConsoleApplication1
已安装
Visual C++
UI
HLSL
代码
数据
资源
Web
实用工具
属性表
Test
图形
联机
排序依据: 默认值
C++ 文件(.cpp)
头文件(.h)
Visual C++
类型: Visual C++
创建包含 C++ 源代码的文件
名称(N): 源.cpp
位置(L): E:\vs2012\ConsoleApplication1\ConsoleApplication1\
浏览(B)...
添加(A)
取消
注意这里是.cpp 要改成.c


添加新项 - ConsoleApplication1
已安装
Visual C++
UI
HLSL
代码
数据
资源
Web
实用工具
属性表
Test
图形
联机
排序依据: 默认值
C++ 文件(.cpp)
头文件(.h)
Visual C++
类型: Visual C++
创建包含 C++ 源代码的文件
名称(N): 源.c
位置(L): E:\vs2012\ConsoleApplication1\ConsoleApplication1\
浏览(B)...
添加(A)
取消

3、编写及调试程序


点击 X 全屏。
进入编写页面了。



内容二 程序设计与调试

编写程序，实现以下功能要求：

2.1 编写程序求解 n 阶行列式值的程序，。

要求：

① 可任意输入行列式的阶数；

② 行列式中的各元素可手动输入或自动生成。

2.2 使用积分法计算 PI 值。

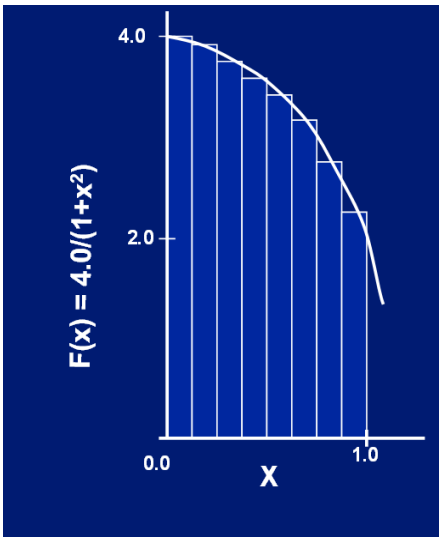
要求：

① 分段步长≥10^9；

② 采用时间函数统计计算 PI 值所耗费的时间。

设计思路：如右图所示：梯形积分——每一竖条都拥有固定宽度的“梯级”。每个竖条的高度便是被积函数的值。将所有竖条的面积加在一起便得出曲线下面积的近似值，即被积函数的值。

若选择一个被积函数及积分极限，那么该积分在数值上便等于 π。这样就构成了利用积分法计算圆周率的基本思路。



实验二 多线程程序设计基础

【实验目的】

1、掌握利用 Windows API 函数进行多线程编程；

2、掌握利用 MS MFC 类库进行多线程编程；

3、掌握利用 MS VS 集成开发环境编写、调试和运行 Windows 多线程程序。

4、掌握同步方法的使用；

5、掌握利用时间函数对比分析并行性能的优劣。

【实验内容】

内容一 基础函数及语法的使用

1.1：_beginthread 函数的使用

要求：

① 程序调试及运行，输出运行结果；

② 分析程序回答：程序输出结果是否有乱序和错位的情况？

```
void ThreadFunc1(PVOID param)
{  while(1)
   {  Sleep(1000);
      cout<<"This is ThreadFunc1\n"<<endl;
   }
}
void ThreadFunc2(PVOID param)
{  while(1)
   {  Sleep(1000);
      cout<<"This is ThreadFunc2\n"<<endl;
   }
}
int main()
{   int i=0;
   _beginthread(ThreadFunc1,0,NULL);
   _beginthread(ThreadFunc2,0,NULL);
   Sleep(3000);
   cout<<"end"<<endl;
   return 0;
}
```

1.2：线程状态的切换，线程的运行与挂起

要求：

① 程序调试及运行，输出运行结果；

② 分析程序回答：程序输出结果是否有乱序和错位的情况？

```
DWORD WINAPI FunOne(LPVOID param)
{
  while(true)
   {
       Sleep(1000);
       cout<<"hello! ";
   }
   return 0;
}
DWORD WINAPI FunTwo(LPVOID param)
{  while(true)
   {
      Sleep(1000);
      cout<<"world! ";
   }
   return 0;
}

int main(int argc, char* argv[])
{
  int input=0;
  HANDLE hand1=
     CreateThread (NULL, 0, FunOne, (void*)&input, CREATE_SUSPENDED, NULL);
  HANDLE hand2=
     CreateThread (NULL, 0, FunTwo, (void*)&input, CREATE_SUSPENDED, NULL);
      while(true){
         cin>>input;
         if(input==1) {  ResumeThread(hand1);  ResumeThread(hand2);  }
         if(input==0) {  SuspendThread(hand1);  SuspendThread(hand2);  }
         if(input==2) {  return 0;     }
       };
  TerminateThread(hand1,1);
  TerminateThread(hand2,1);
  return 0; }
```

1.3：线程函数 CreateThread 及全局变量的使用

要求：

① 程序调试及运行，输出运行结果；
② 是否可采用其同步方式替代现有全局变量的功能。

```
int  globalvar = false;
DWORD WINAPI ThreadFunc(LPVOID pParam)
{
  cout<<"ThreadFunc"<<endl;
  Beep(5000,2000);   // beep and Sleep(200);
  globalvar = true;
  return 0;
}
int main()
{
  HANDLE hthread = CreateThread(NULL, 0, ThreadFunc, NULL, 0, NULL);
  if (!hthread)
  {
     cout<<"Thread Create Error ! "<<endl;
     CloseHandle(hthread);
  }
  while (!globalvar)
     cout<< "Thread while" <<endl;
  cout<<"Thread exit"<<endl;
  return 0;
}
```

内容二 多线程程序设计

编写多线程程序，实现以下功能要求。

2.1 使用_beginthread 函数，创建 2 个（或以上）线程，实现打印各自线程号各 5 次。

2.2 使用 CreateThread 函数，创建 2 个（或以上）线程，采用至少 2 种同步方法，实现按确定顺序打印各自线程号 5 次。

2.3 创建 2 个线程，实现：按下键盘上“1”键时由线程 1 打印“Hello”，按下键盘上“2”键时由线程 2 打印“World”，按“0”键时程序退出。

2.4 编写程序计算 PI 值

要求：

① 采用至少两种**线程分配**方法；

② 使用时间函数统计每个线程的计算时间，对比分析哪种分配方法更优；

③ 程序调试及运行，输出运行结果。

实验三 OpenMP 程序设计

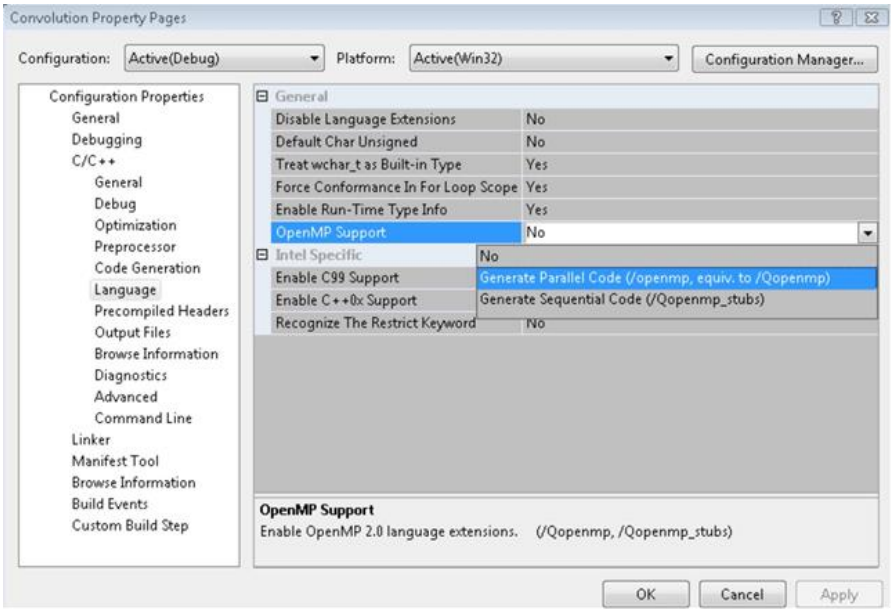
【实验目的】

1、掌握 OpenMP 指令、子句和库函数的使用；
2、掌握采用 OpenMP 并行程序设计的基本方法。

【预备知识】

1、OpenMP 编程环境配置

缺省情况下编译器并不支持 OpenMP。打开工程属性页，选择 C/C++, Language，在“OpenMP support” 下拉框中选择“Generate Parallel Code(/OpenMP, equiv to /QOpenMP)”。



2、fork/join 并行执行模式的概念

OpenMP 是一个编译器指令和库函数的集合，主要是为共享式存储计算机上的并行程序设计使用的。

从上一节实例中我们也可以发现 OpenMP 并行执行的程序要全部结束后才能执行后面的非并行部分的代码。这就是标准的并行模式 fork/join 式并行模式，共享存储式并行程序就是使用 fork/join 式并行的。

标准并行模式执行代码的基本思想是：程序开始时只有一个主线程，程序中的串行部分都由主线程执行，并行的部分是通过派生其他线程来执行，但是如果并行部分没有结束时是不会执行串行部分的，如以下代码：

```
int main(int argc, char* argv[])
{
     clock_t t1 = clock();
#pragma omp parallel for
     for ( int j = 0; j < 2; j++ ){
         test();
     }
     clock_t t2 = clock();
     printf("Total time = %d\n", t2-t1);
     test();
     return 0;
}
```

需要注意的是：在没有执行完 for 循环中的代码之前，后面的 clock_t t2 = clock();这行代码是不会执行的，如果和调用线程创建函数相比，它相当于先创建线程，并等待线程执行完，所以这种并行模式中在主线程里创建的线程并没有和主线程并行运行。

3、OpenMP 指令和库函数知识点归纳与回顾

在 C/C++中，OpenMP 指令使用的格式为：　　#pragma omp 指令 [子句[子句]…]

parallel for 就是一条指令，有些书中也将 OpenMP 的“指令”叫做“编译指导语句”，后面的子句是可选的。例如：

```
#pragma omp parallel private(i, j)
```

parallel 就是指令， private 是子句

为叙述方便把包含#pragma 和 OpenMP 指令的一行叫做语句，如上面那行叫 parallel 语句。OpenMP 的指令有以下一些：

parallel，用在一个代码段之前，表示这段代码将被多个线程并行执行

for，用于 for 循环之前，将循环分配到多个线程中并行执行，必须保证每次循环之间无相关性。

parallel for， parallel 和 for 语句的结合，也是用在一个 for 循环之前，表示 for 循环的代码将被多个线程并行执行。

sections，用在可能会被并行执行的代码段之前

parallel sections，parallel 和 sections 两个语句的结合

critical，用在一段代码临界区之前

single，用在一段只被单个线程执行的代码段之前，表示后面的代码段将被单线程执行。

barrier，用于并行区内代码的线程同步，所有线程执行到 barrier 时要停止，直到所有线程都执行到 barrier 时才继续往下执行。

atomic，用于指定一块内存区域被制动更新

master，用于指定一段代码块由主线程执行

ordered，用于指定并行区域的循环按顺序执行
threadprivate，用于指定一个变量是线程私有的。

OpenMP 除上述指令外，还有一些库函数，下面列出几个常用的库函数：

omp_get_num_procs，返回运行本线程的多处理机的处理器个数。
omp_get_num_threads，返回当前并行区域中的活动线程个数。
omp_get_thread_num，返回线程号
omp_set_num_threads，设置并行执行代码时的线程个数
omp_init_lock，初始化一个简单锁
omp_set_lock， 上锁操作
omp_unset_lock， 解锁操作，要和 omp_set_lock 函数配对使用。
omp_destroy_lock， omp_init_lock 函数的配对操作函数，关闭一个锁

OpenMP 的子句包括：

private，指定每个线程都有它自己的变量私有副本。
firstprivate，指定每个线程都有它自己的变量私有副本，并且变量要被继承主线程中的初值。
lastprivate，主要是用来指定将线程中的私有变量的值在并行处理结束后复制回主线程中的对应变量。
reduce，用来指定一个或多个变量是私有的，并且在并行处理结束后这些变量要执行指定的运算。
nowait，忽略指定中暗含的等待
num_threads，指定线程的个数
schedule，指定如何调度 for 循环迭代
shared，指定一个或多个变量为多个线程间的共享变量
ordered，用来指定 for 循环的执行要按顺序执行
copyprivate，用于 single 指令中的指定变量为多个线程的共享变量
copyin，用来指定一个 threadprivate 的变量的值要用主线程的值进行初始化。
default，用来指定并行处理区域内的变量的使用方式，缺省是 shared

【实验内容】

内容一 基本语法的使用

请调试并写出以下程序的运行结果，并分析程序中 OpenMP 各语句的功能。

1、parallel for 指令的用法

程序一：

```
int main(int argc, char* argv[])
{
#pragma omp parallel for
```

```
        for (int i = 0; i < 10; i++ )
        {
            printf("i = %d\n", i);
        }
        return 0;
    }
```

运行结果：

程序二：

```
    void test()
    {
        int a = 0;
        clock_t t1 = clock();
        for (int i = 0; i < 100000000; i++)
        {
            a = i+1;
        }
        clock_t t2 = clock();
        printf("Time = %d\n", t2-t1);
    }
    int main(int argc, char* argv[]){
        clock_t t1 = clock();
     #pragma omp parallel for
        for ( int j = 0; j < 2; j++ ){
            test();
        }
        clock_t t2 = clock();
        printf("Total time = %d\n", t2-t1);
        test();
        return 0;
    }
```

1、运行结果

2、分析并行计算后效率提高程度？

2、parallel 指令的用法

parallel 是用来构造一个并行块的，也可以使用其他指令如 for、sections 等和它配合使用。在 C/C++中，parallel 的使用方法如下：

```
#pragma omp parallel [for | sections] [子句[子句]…]
{        ←————————括号遇到单语句的时候可以不用
        //代码
```

```
}
```

parallel 语句后面要跟一个大括号对将要并行执行的代码括起来。

程序一：

```
void main(int argc, char *argv[])
{
    #pragma omp parallel
    {
       printf(“Hello, World!\n”);
    }
}
```

运行结果：

程序二：

```
void main(int argc, char *argv[])
 {
    #pragma omp parallel num_threads(8)
    {
        printf(“Hello, World!, ThreadId=%d\n”, omp_get_thread_num() );
    }
}
```

运行结果：

3、for 指令的使用方法

for 指令则是用来将一个 for 循环分配到多个线程中执行。for 指令一般可以和 parallel 指令合起来形成 parallel for 指令使用，也可以单独用在 parallel 语句的并行块中。

```
#pragma omp [parallel] for [子句]
      for 循环语句
```

程序一：

```
int j = 0;
#pragma omp for
     for ( j = 0; j < 4; j++ ){
         printf(“j = %d, ThreadId = %d\n”, j, omp_get_thread_num());
     }
```

运行结果：

程序二：

```
int j = 0;
#pragma omp parallel
{
#pragma omp for
     for ( j = 0; j < 4; j++ ){
         printf(“j = %d, ThreadId = %d\n”, j, omp_get_thread_num());
     }
}
```

1、运行结果：

2、分析 parallel 与 for 语句连写与分开写的区别

4、sections 和 section 指令的用法

section 语句是用在 sections 语句里用来将 sections 语句里的代码划分成几个不同的段，每段都并行执行。用法如下：

```
#pragma omp [parallel] sections [子句]
{
   #pragma omp section
   {
            代码块
   }
}
```

程序一：

```
void main(int argc, char *argv)
{
#pragma omp parallel sections {
#pragma omp section
    printf(“section 1 ThreadId = %d\n”, omp_get_thread_num());
#pragma omp section
    printf(“section 2 ThreadId = %d\n”, omp_get_thread_num());
#pragma omp section
    printf(“section 3 ThreadId = %d\n”, omp_get_thread_num());
#pragma omp section
    printf(“section 4 ThreadId = %d\n”, omp_get_thread_num());
}
```

1、运行结果：

2、分析结果产生“第 4 段代码执行比第 3 段代码早”的原因

5、master 和 barrier 制导语句的用法

```
int main( ) {
int a[5], i;
#pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 5; i++)     a[i] = i * i;
        #pragma omp master
        for (i = 0; i < 5; i++)
        {
            printf_s("The thread number is:%d",omp_get_thread_num());
            printf_s("a[%d] = %d\n", i, a[i]);
        }
        pragma omp barrier
        pragma omp for
            for (i = 0; i < 5; i++)  a[i] += i;
    }
}
```

1、运行结果

2、分析 master 与 barrier 语句在本程序中的作用

6、atom 制导语句的用法

```
int main()
{
    int count = 0;
    for (int j=0;j<16;j++)
    {
            count = 0;
            #pragma omp parallel num_threads(2)
                {
                    for (int i=0;i<100000;i++)
                        #pragma omp atomic
                        count++;
                }
                printf_s("Number of threads: %d\n" ,count);

    }
    return 0;
}
```

1、运行结果

2、分析 atom 语句所起到的作用

7、private 与 reduction 子句的用法

```
void   main()
{
   static   long   num_steps   =   1000000000;
   double   step;
   int   i;
   double   x,   pi,   sum   =   0.0;
   step   =   1.0/(double)   num_steps;
   // #pragma omp parallel num_threads(2)
    #pragma   omp   parallel   for  private(x)  reduction(+:sum)

   for   (i=0;i <   num_steps;   i++)
   {
      x = (i+0.5)*step;
      sum =  sum  + 4.0/(1.0+x*x);
   }
   pi   =   step   *   sum;
   printf( "Pi = %12.5f \n ", pi,);
}
```

1、 运行结果：

2、说明 private 和 reduction 子句在程序中的作用。

3、比较 OpenMP 并行前后 pi 值计算所耗费的时间，并说明为什么耗费时间不是：线程数：1？

内容二 算法及程序设计

1、求解积分 $\int_0^{\pi} \sin x \, dx$ 的近似值（计算步长≥10^8）。

2、遍历查找 8000x8000 矩阵中的最大最小数；

要求：

① 采用显式和 OpenMP 方法编程实现；

② 统计计算时间；

③ 注释程序，结果截图，提交 word 文档。

实验四 多核程序性能优化与分析

【实验目的】

1、掌握使用 Parallel Inspector 发现内存访问错误；
2、掌握使用 Parallel Amplifier 查找优化机会；
3、掌握使用 Parallel Composer 生成 OpenMP 代码；
4、掌握使用 Parallel Inspector 查找多线程错误
5、掌握使用 Parallel Amplifier 分析多线程并行性

【预备知识】

1、Intel Parallel Studio 性能优化工具

Intel Parallel Studio 包括以下 4 个组件：

Intel® Parallel Composer 将编译器、函数库和 Microsoft Visual Studio* 调试器的扩展融为一体。英特尔® C++ 完全兼容 Microsoft Visual C++，并且拥有支持最新的 OpenMP* 技术，可以提供语言扩展来支持并帮助简化为代码添加并行能力的工作。英特尔®Parallel Debugger Extension 可简化并行调试并确保线程的准确性。英特尔® 线程构建模块(TBB)和英特尔®集成性能库可提供已经线程化的通用和应用指定函数，从而使得开发人员可以迅速为应用添加并行能力。

Intel® Parallel Inspector 将线程和内存错误查找功能融入一个强大的错误检测工具中。此工具有助于提高 Microsoft Visual Studio* 内部 C/C++ 应用的可靠性、安全性和准确性。英特尔® Parallel Inspector 使用动态方法，不需要特殊的测试工具或编译器，因此通常更加易于对代码进行测试。

Intel® Parallel Amplifier 使迅速查找多线程性能瓶颈变得简单，不需要了解处理器架构或代码也可以查找。Parallel Amplifier 省略了推断过程，能够分析 Windows* 应用程序的性能行为，快速访问度量信息，从而加快并改进决策制定。

Parallel Amplifier 是一个并行程序性能分析和调试工具。其功能类似于原来 Intel® VTune 和 Intel® Thread Profiler 的集合。与 VTune 能够基于时间或者事件采样不同的是 Amplifier 只能基于时间采样。Amplifier 去掉了各种可能让部分程序员感到困惑，难于理解的事件，比如 L2 cache miss 或者 Branch miss-prediction。Amplifier 可以帮助开发人员找到他们程序执行的热点在源代码中的位置，这些热点就是需要性能优化的地方。除此以外，Amplifier 能够让程序员知道他们的多线程程序并行执行是否高效。

Intel® Parallel Advisor Lite 是一个预览版组件，能够洞悉并行化如何提升应用程序性能，帮助提高并行程序设计效率。Parallel Composer 集合了传统的 Intel® C++compiler, Intel®线程构建模块（Threading Building Blocks）,Intel®集成性能库（Integrated Performance Primitives）以及 Intel® Parallel Debugger Extension。Parallel Inspector 包括了 Intel® 线程检查器（Thread Checker）以及内存检查的功能。这两个工具集帮助开发人员提高生成代码和查找错误的效率。

2、卷积运算

对于两个函数 f 和 g，它们的卷积就是一个作用于它们之上的数学操作并且产生了第三个函数。f 和 g 的卷积 H(x)可以定义为如下的积分形式：

$$H(x)=(f*g)(x)=\int f(r)*g(x-r)dr$$

数字图像处理或者一些其他领域使用的是离散卷积。假设 f 和 g 分别定义在整形数集合上，f 和 g 的离散卷积可以如下定义：

$$H(n)=(f*g)(n)=\sum_{m}(f[m])*g[n-m])$$

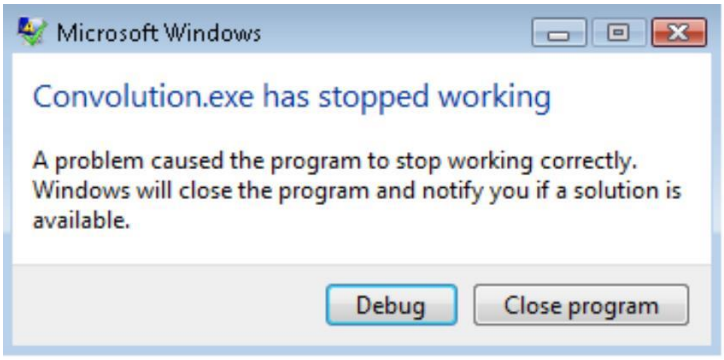
【实验内容】

内容一 基础性能工具的使用

1、使用 Intel® Parallel Inspector 发现内存访问错误

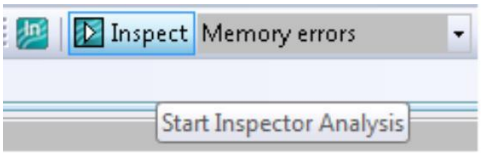
卷积经常使用在各种图像处理算法中。本实验的重点在于怎样利用 Intel ® Parallel Studio 来调试程序。

打开“Convolution - RuntimeErr”目录下的工程。编译工程时没有错误，但是当按下“Ctrl + F5”运行程序的时候，出现了运行时错误如下：

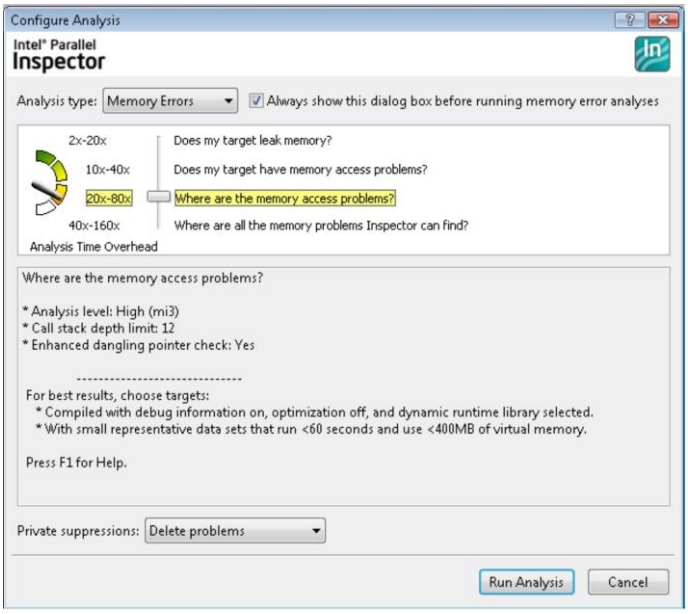


通常运行时错误是由于堆栈溢出或者无效内存地址访问。在 C/C 程序中通常有成千上万的内存访问，以及数组，指针，指针的指针等。人工查找哪一处的内存访问导致了运行时错误是非常困难的。现在，Intel® Parallel Inspector 可以帮助开发人员定位出错的语句和变量。

首先，找到工具栏中的“Inspect”按钮，并且确保选择的是“Memory errors”。然后点击“Inspect”。

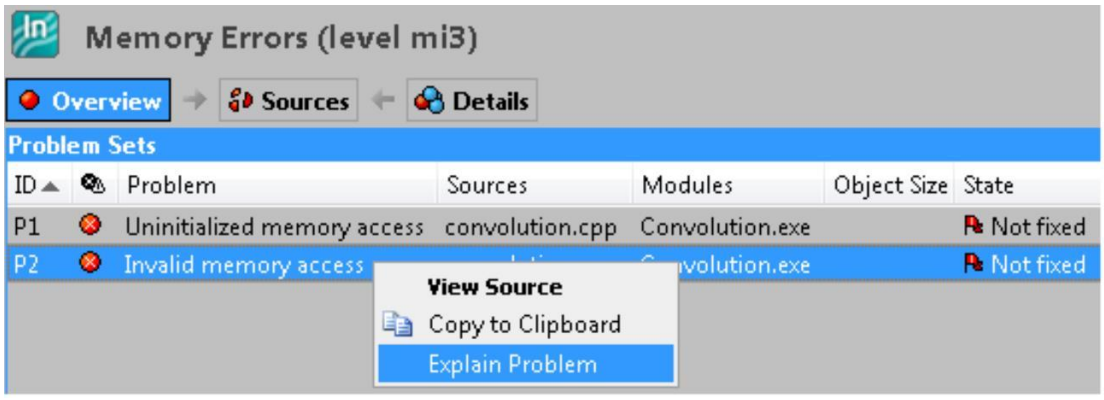


这时会有一个 Configure 对话框出现。使用者可以拖动滚动条去选择不同的级别。滚动条从上到下，分析过程的负载计算量逐级增加。这里，选择第三级“Where are the memory access problem”。然后点击“Run Analysis”按钮。

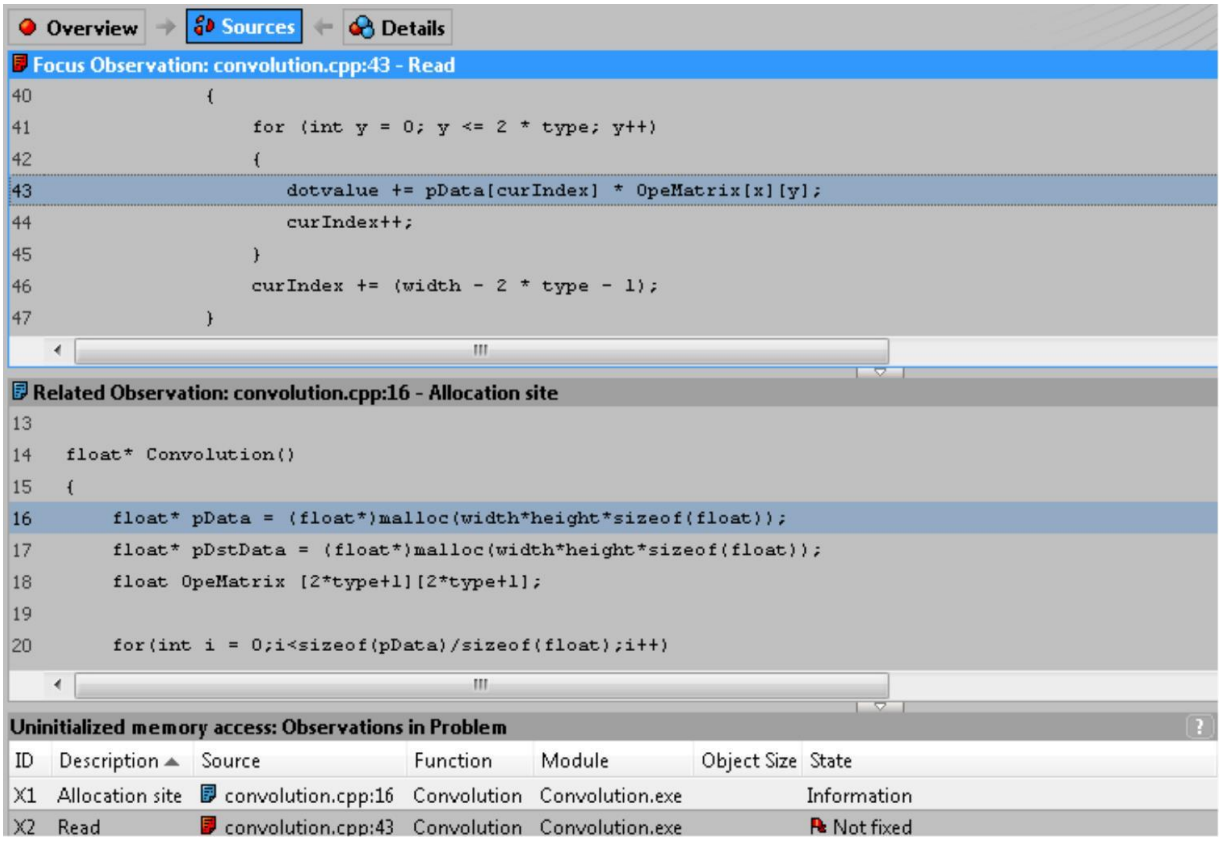


检查的过程需要花费一定的时间。运行完成后，前面提到的运行时错误的消息框再次出现。点击“Close program”，然后等待 Intel® Parallel Inspector 分析完成后，两个内存访问的错误被发现并显示出来。一个是“Uninitialized memory access”另一个是“Invalid memory access”。

接下来，用户可以点击“Interpret Result”去查看问题的详细信息。出现了一个“Overview”页面，两个问题显示出来。这里，用户可以右击任意一个问题然后选择“Explain Problem”以打开 MSDN 获取问题的详细解释。



用户也可以直接双击问题项去查看其对应的程序源代码行。在这里，请双击 P1:“Uninitialized memory access”。两行代码被高亮显示。一个是内存区域分配，另一个是读取内存区域里的数值。两处代码都访问了同一个数组：“pData”。其中一个对 pData 的操作导致了无效地址访问。由于内存分配那条语句看来应该不存在问题——仅仅是调用了 malloc()函数，那么读操作的语句应该是导致运行时出现异常的语句。确定了此原因后，我们可以假设读取 pData 值的索引值 curIndex 可能超出了 pData 数组最大索引值。通过阅读源代码，可以看到，curIndex 值随着每次循环而增加。因此可以怀疑对 curIndex 的增值被错误的计算了。



分析源码，发现 curIndex 在每次(j)循环开始时被以 iniIndex 的值初始化:

```
iniIndex += j;
int curIndex = iniIndex;
```

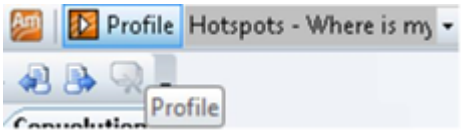
iniIndex 总是存储参与卷积计算矩阵的左上角那个元素的索引值。这个值每次循环应该只加 1。因此“iniIndex += j”这句计算有错误导致 iniIndex 每次多加，并且若干次循环后会超出索引值的上限。移去这句程序，并且在每次循环（j）的末尾加上“iniIndex++”。重新编译此工程，并且运行，运行时错误不再出现。

2、使用 Intel® Parallel Amplifier 查找优化机会

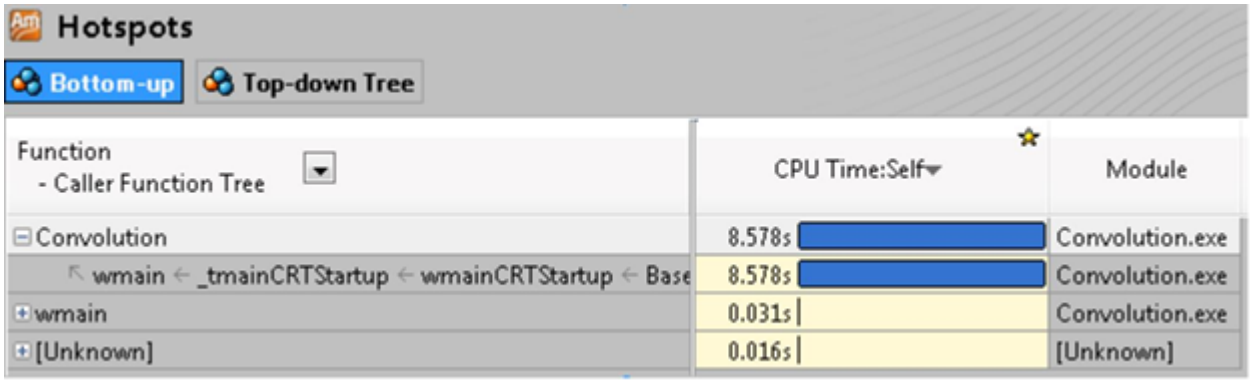
很多程序运行的效率不高需要进行优化。大多数程序员并不知道如果在庞大的代码中准确找到需要优化的地方（热点）。他们大多根据经验判断哪些函数是性能瓶颈，需要优化。但是这样做并不准确。Intel Parallel Amplifier 能帮助程序员准确找到程序执行的热点，这些热点就是可以被优化的地方，至于优化方法，可以选择多线程，SSE 或者其他方法。另外针对多线程程序，Amplifier 能够告诉程序员执行中多线程并行信息以判断并行度是否高。

打开“Convolution”目录下的工程。这个工程已经修改了上一节中出现的运行时错误。按“Ctrl + F5”运行程序。运行结束后，一个弹出框显示这次运行时间是 8563 毫秒（根据用户机器不同会有不同结果）。这个时间将用来同优化后的版本作比较。

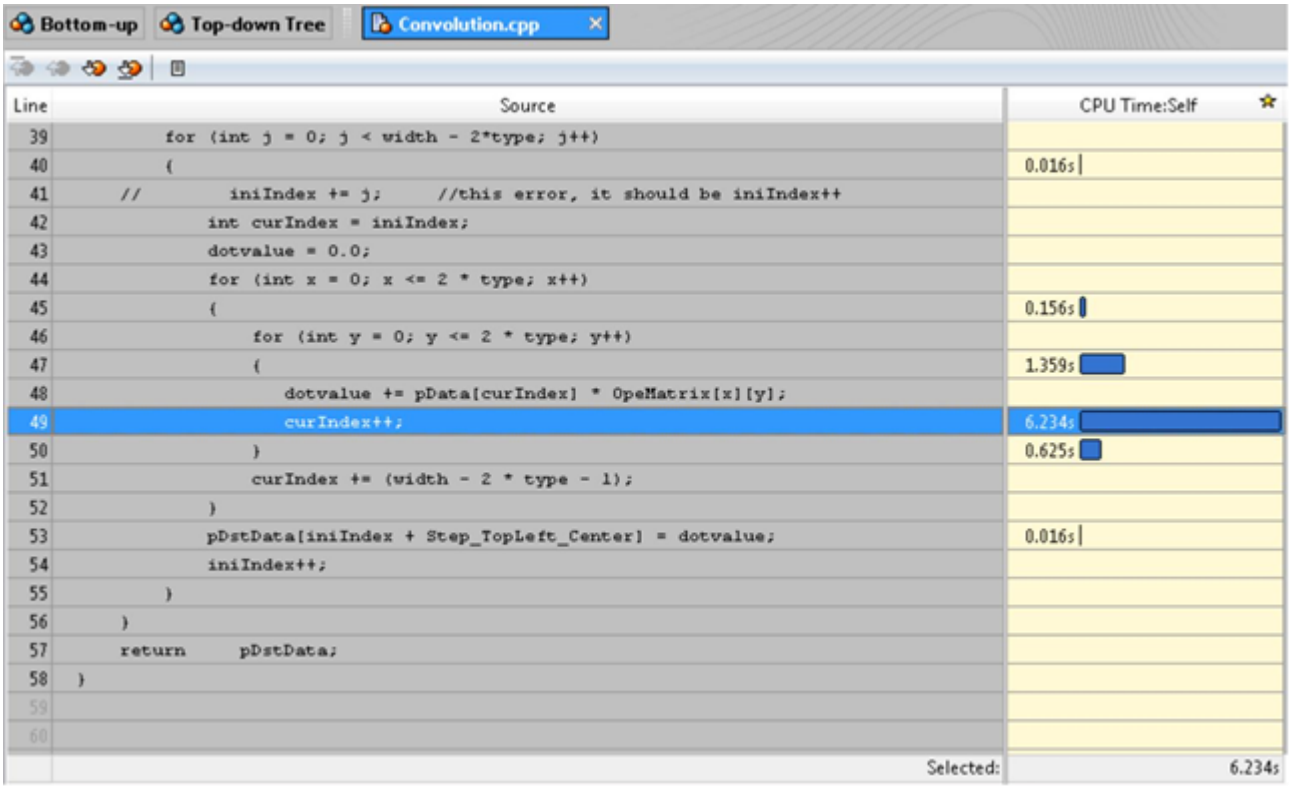
在工具栏“Profile”项中选择“Hotspots - Where is my program spending time?”然后点击“Profile”。



数据收集完毕后，Amplifier 显示每个函数执行时间信息：



结果显示函数“Convolution” 占用了运行过程中绝大多数的时间。这个函数就是程序中的执行热点或者瓶颈。双击 “Convolution” 以得到更详细的信息。

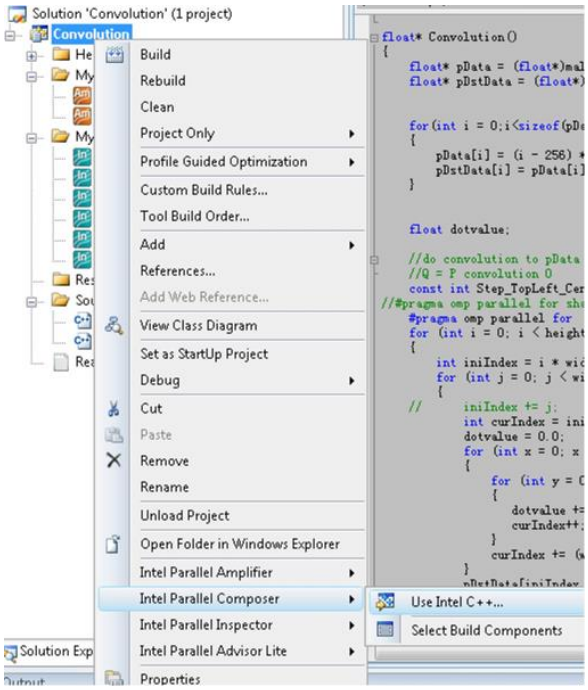


从这张图中可以看出在“Convolution” 函数中，绝大多数时间又是在循环中执行。如果循环能够被优化，那么整个程序的性能将得到相应的提升。由于越来越多的 PC 平台有多核 CPU，多线程优化是一个不错的选择。

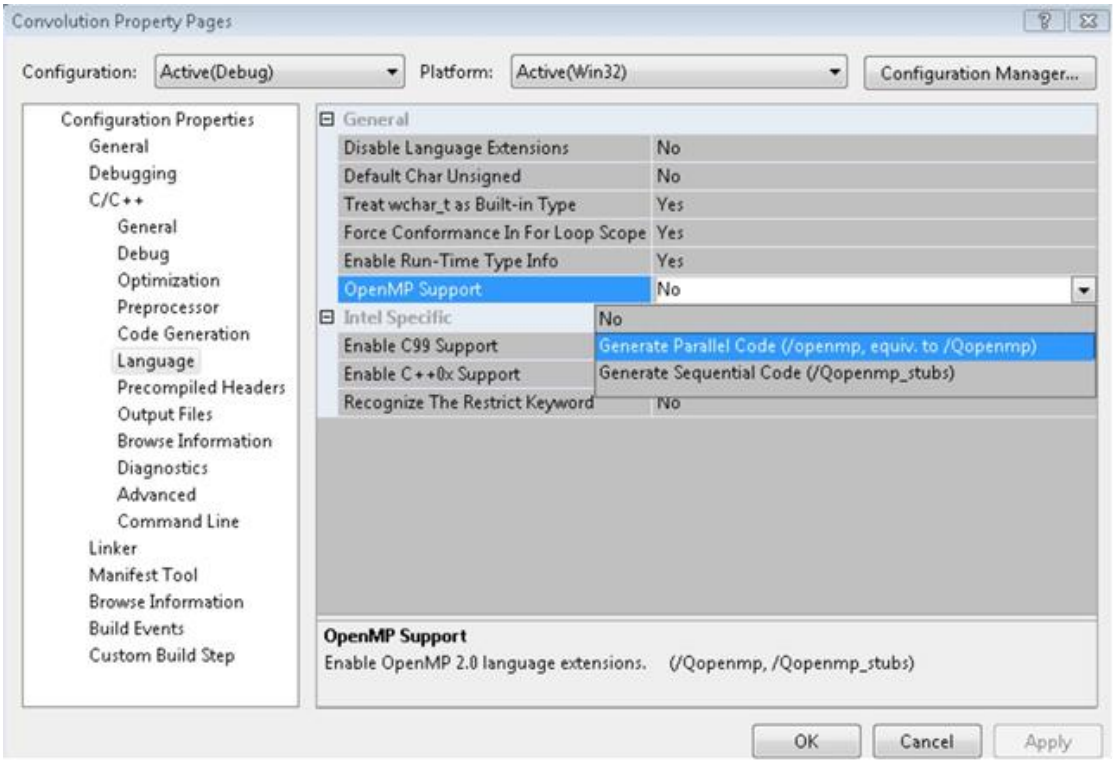
3、使用 Intel® Parallel Composer 生成 OpenMP 代码

OpenMP* 是一个简单的实现多线程的方法，它可以方便的将“for”循环多线程化。

去掉“#pragma omp parallel for”语句前的注释。这个“pragma”语句可以指导相关支持 OpenMP 的编译器生成多线程程序。集成在 Intel Parallel Composer 中的编译器支持 OpenMP 3.0 的规范。在 Visual Studio 中右键点击“Convolution”工程。然后选择“Intel Parallel Composer” 以及 “Using Intel C++”。



缺省情况下编译器并不支持 OpenMP。打开工程属性页，选择 C/C++, Language，在“OpenMP support” 下拉框中选择“Generate Parallel Code(/OpenMP, equiv to /QOpenMP)”。

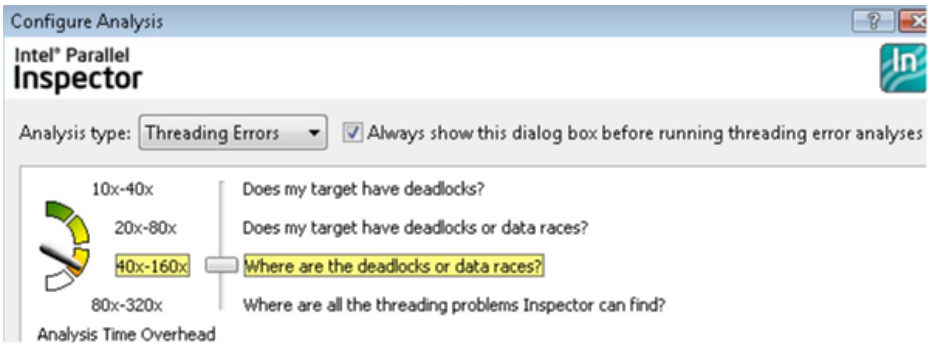


编译并运行新代码。但是结果并非想象的运行加速，反而这次运行耗时 50281 毫秒（根据用户机器不同会有不同结果）。同刚才串行程序相比，反而运行速度大为降低。新加入的 OpenMP 语句反而导致程序性能出现问题。

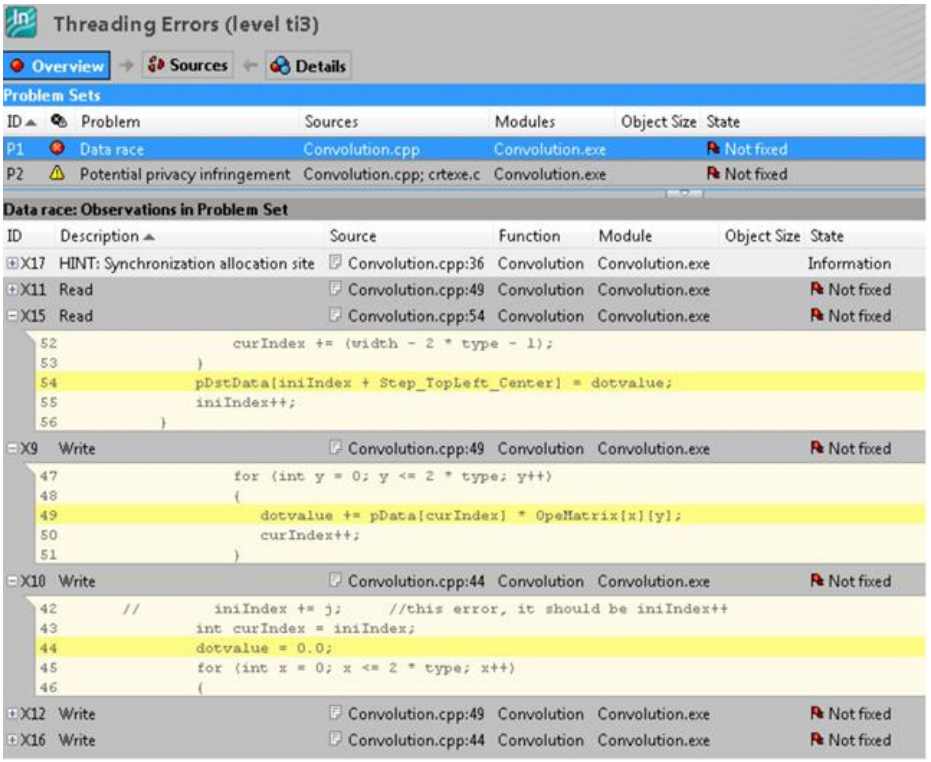
4、使用 Intel® Parallel Inspector 查找多线程错误

Parallel Inspector 可以帮助程序员查找多线程运行错误。在工具条面板 “Inspector”旁的下拉框中选择“Threading errors”，然后点击“Inspect”.

在接下来的弹出框中，选择第三选项：“Where are the deadlock or data races?”。来查找并定位死锁，数据访问等常见的多线程编程错误。



点击 “Run analysis” 运行 inspector。数据收集完成后，查到的错误或者疑似错误会被显示出来，包括 “Data race” 和 “Potential privacy infringement” 等。点击“Interpet Result”可以查看这些问题的详细信息。



首先看“Data race”错误，如上图，多个错误都同第 44，49，54 行相关。其中每一行都访问了 dotvalue，既有读也有写。这意味着多个线程同时访问 dotvalue，有的读它的值，有的则是写入新值，而程序对这个变量却没有相应的保护。进一步分析源代码，dotvalue 在每次循环中用来存储临时计算结果。这就意味着 dotvalue 是每个循环私有变量，也就应该是

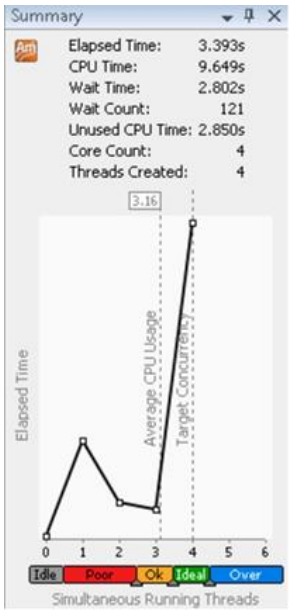
每个线程私有变量。因此，刚才加入的语句“#pragma omp parallel for”应该改为“#pragma omp parallel for private(dotvalue)”。重新编译此工程并且用 inspector 分析，没有发现线程错误。

再次运行程序没有运行时错误，整个运行时间为 3483 毫秒。同串行版本比较有大约 2.46x 倍性能提升(8563 VS. 3483)。

5、使用 Intel® Parallel Amplifier 分析多线程并行性

使用 Intel Parallel Amplifier 开发人员可以知道程序运行时多线程并行性的细节信息。在这里使用的例子工程中，有条 MessageBox 语句用来显示程序运行使用的时间信息。但是这一语句会导致程序暂停以等待用户点击 “Ok”。这样会影响 Amplifier 分析程序的并行性。在运行 Amplifier 之前，请将源码中的 MessageBox 语句注释掉。在工具栏中“Profile”旁边的下拉框中，选择 “Concurrency - Where is my concurrency poor?” 然后点击“Profile”。

在内容 1-2 中，Amplifier 查找出“Convolution”函数是线性版本程序运行时的性能瓶颈。而上图显示在大多数的时间里“Convolution”运行在“Ideal” 状态（绿色）“Ideal”意味着处于激活并且运行中状态的线程个数同系统中 CPU 的核数相等。多线程版本并行度很好。



这时，在 Visual Studio 集成开发环境的右下角有一个总结信息窗口。从总结信息可以知道系统中 CPU 有四个核心，并且程序运行中产生了四个线程。CPU 的平均使用度是 3.16。大概是 79% CPU 占用率。

通过多线程，程序中的热点函数通过产生多线程充分利用了系统中的 CPU 资源。Amplifier 报告显示通过优化，程序执行效率得到了极大地提升。

内容二 性能工具的使用与分析

1、使用 Intel 性能优化工具对 PI 值计算程序进行分析，分析几种线程分配方法间的差异。

2、使用 Intel 性能优化工具对基于 OpenMP 的 PI 值计算程序进行分析，分析与单线程程序、多线程程序之间对比的性能差异。

实验五 IPP 程序设计

【实验目的】

1、掌握 Intel® Integrated Performance Primitives (Intel® IPP)的功能、特点和数据类型；

2、掌握 Intel® IPP 基本分析和使用方法；

【预备知识】

1、熟练掌握 C++语言；

2、掌握 Visual Studio* .NET*集成开发环境的使用；

3、Intel IPP 的基本知识；

4、音视频编码与格式的基本知识。

【实验原理】

Intel IPP 简介

Intel IPP 是一套跨平台的软件函数库，它提供了广泛的多媒体功能：音频解码器（例如： H263 、 MPEG-4 ）、图像处理 (JPEG) 、信号处理、语音压缩（例如： G723 、 GSM 、 AMR ）和加密机制。“Intel IPP” 包含各种的函数，用于进行向量与图像处理、颜色转换、过滤、分屏、设置域值、变换，以及算术、统计、几何与图形运算。对于每个函数， “Intel IPP” 均支持多种数据类型和分布，同时保持了数据结构的最小化，它提供了丰富的选项供用户在设计与优化应用程序时选用，不必再去编写特定代码。Intel IPP 针对大量的 Intel Microprocessor（微处理器）进行，包括多核处理器系列。

英特尔® IPP 的函数涵盖了以下领域中一些重要的基础算法：

视频解码/编码	图像处理
音频解码/编码	图像颜色转换
JPEG/JPEG2000	计算机视觉
数据压缩	信号处理
加密 - CAVP 验证	矢量/矩阵数学

语音编码 语音识别	字符串处理 射线跟踪/渲染

Intel IPP 功能优化多媒体函数，其主要特点体现在：

① 不再需要以 Hard-Coding 方式针对特定的处理器进行优化；

② 提高应用程序的可靠性；

③ 节省时间：内置调度器选择优化处理，并根据处理器自动调度特定的运行时代码；

④ 在基于信号和图像的数据结构的特定约束方面，改善了应用程序的灵活性。

【实验内容】

内容一 IPP 基础函数的使用

调试并运行以下程序，分析 IPP 函数的用途

程序一：

```
Ipp64u start, end;
start=ippGetCpuClocks();
end=ippGetCpuClocks();
printf("Clocks to do nothing: %d\n",(Ipp32s)(end - start));
start = ippGetCpuClocks();
printf("Hello World\n");
end = ippGetCpuClocks();
printf("Clocks to print 'hello world': %d\n",(Ipp32s)(end - start));
```

程序二：

```
const  int  SIZE = 256;
Ipp8u  pSrc[SIZE], pDst[SIZE];
int i;
for (i=0; i<SIZE; i++)
     pSrc[i] = (Ipp8u)i;
ippsCopy_8u(pSrc, pDst, SIZE);
printf("pDst[%d] = %d\n", SIZE-1, pDst[SIZE-1]);
```

内容二 综合程序设计

1、运行、调试：Code\IPP\Lab1\Other\Digital Filtering\DFT 程序，分析程序的主要功能。

2、运行、调试：Code\IPP\Lab1\Other\Digital Filtering\FFTFilter 程序，分析程序的主要功能。

实验六 综合程序设计与分析

【实验目的】

1、掌握综合多核应用程序设计的方法；
2、掌握 IPP 程序设计；
3、掌握多核程序调试及性能分析与优化。

【实验内容】

结合算法，设计程序，实现以下功能。

设计一：已知二维方阵：A[N] [N]、B[N] [N]、C[N] [N]，N=8000，C=A×B，采用随机函数 srand 产生[0－100]的随机整数，给矩阵 A，B 赋初始值，并计算 C 的值。

要求： ① 采用 OpenMP 进行并行处理；
② 统计 C 矩阵生成所耗费的时间；
③ 使用工具进行性能优化，比较优化前后时间差；

设计二：求函数 $y=|\sin(e^x)|$ 在区间（0—1000）做单位采样后的 DFT 变换结果。

要求： ① 统计转换时间，并绘图显示 C 曲线；
② 将 DFT 结果作 DFT 反变换，比较与原函数的差异；
③ 回答是否采用了多线程方法，从什么地方可以得出这一结论？

设计三：采用多核优化工具分析以上程序的热点问题及多线程性能情况。

实验七 OpenCV 综合程序设计

【实验目的】

1、掌握 OpenCV 应用程序设计的方法；
2、掌握多核 OpenCV 程序调试、性能分析与优化。

【实验内容】

内容一 OpenCV 程序设计基础

调试及运行以下程序，说明程序的主要功能

程序一：

```
IplImage* img=0;
int height,width,step,channels;
UCHAR* data;
int i,j,k;

img=cvLoadImage(***);
if(!img)
{
   printf("Could not load image file:%s\n",argv[1]);
   exit(0);
}
height=img->height;
width=img->width;
step=img->widthStep;
channels=img->nChannels;
data=(UCHAR*)img->imageData;
printf("Processing a%d*%d image with %d channels\n",height,width,channels);

cvNamedWindow("mainWin",CV_WINDOW_AUTOSIZE);
cvMoveWindow("mainWin",100,100);
for(i=0;i<height;i++)
   for(j=0;j<width;j++)
    for(k=0;k<channels;k++)
     data[i*step+j*channels+k]=255-data[i*step+j*channels+k];

cvShowImage(***,***);
cvWaitKey(0);
```

```
    cvReleaseImage(&img);
    return 0;
```

程序二：

```
    char wndname[] = "Edge";
    char tbarname[] = "Threshold";
    int edge_thresh = 1;
    IplImage *image = 0, *cedge = 0, *gray = 0, *edge = 0;
    // 定义跟踪条的 callback 函数
    void on_trackbar(int h)
    {
        cvSmooth( gray, edge, CV_BLUR, 3, 3, 0 );
        cvNot( gray, edge );
        // 对灰度图像进行边缘检测
        cvCanny(***, ***, (float)***, (float)edge_thresh*3, 3);
        cvZero( cedge );
        // copy edge points
       cvCopy( image, cedge, edge );

        cvShowImage(wndname, cedge);
    }
    int main( int argc, char** argv )
    {
        char* filename = argc == 2 ? argv[1] : (char*)"jpg";

        if( (image = cvLoadImage( filename, 1)) == 0 )
            return -1;
        // Create the output image
        cedge = cvCreateImage(cvSize(image->width,image->height), IPL_DEPTH_8U,
    3);
        // 将彩色图像转换为灰度图像
        gray = cvCreateImage(cvSize(***,***), IPL_DEPTH_8U, 1);
        edge = cvCreateImage(cvSize(***,***), IPL_DEPTH_8U, 1);
        cvCvtColor(***, ***, CV_BGR2GRAY);
        // Create a window
        cvNamedWindow(wndname, 1);
        // create a toolbar
        cvCreateTrackbar(tbarname, wndname, &edge_thresh, 100, on_trackbar);
        // Show the image
```

```
    on_trackbar(1);
    // Wait for a key stroke; the same function arranges events processing
    cvWaitKey(0);
    cvReleaseImage(&image);
    cvReleaseImage(&gray);
    cvReleaseImage(&edge);
    cvDestroyWindow(wndname);
    return 0;
}
```

程序三：

```
int main( int argc, char** argv )
{
***;
/* the first command line parameter must be image file name */
if( argc==2 && (src = cvLoadImage(argv[1], -1))!=0)
{
   IplImage* dst = cvCloneImage( src );
   int delta = 1;
   int angle = 0;
   int opt = 1;   // 1： 旋转加缩放
                  // 0: 仅仅旋转
   double factor;
   cvNamedWindow( "src", 1 );
   cvShowImage( "src", src );
   for(;;)
   {
    float m[6];
    CvMat M = cvMat( 2, 3, CV_32F, m );
    int w = src->width;
    int h = src->height;
    if(opt) // 旋转加缩放
                factor = (cos(angle*CV_PI/180.) + 1.05)*2;
    else // 仅仅旋转
                factor = 1;
    m[0] = (float)(factor*cos(-angle*2*CV_PI/180.));
    m[1] = (float)(factor*sin(-angle*2*CV_PI/180.));
    m[3] = -m[1];
    m[4] = m[0];
```

```
    // 将旋转中心移至图像中间
    m[2] = w*0.5f;
    m[5] = h*0.5f;
    cvGetQuadrangleSubPix( ***, ***, &M);//提取象素四边形，使用子象素精度
    cvNamedWindow( "dst", 1 );
    ***;
    if( cvWaitKey(5) == 27 )
     break;
    angle =(int) (angle + delta) % 360;
   } // for-loop
}
return 0;
}
```

内容二 综合程序设计

1、设计程序，对一张图片进行 DFT 变换。
2、设计程序，根据已训练好的分类器对人脸图像进行检测。
3、针对 2 中检测出的人脸，采用不同颜色的圆形框或矩形框标记出检测出的五官。