

多线程与多核编程

多线程与多核编程.....	1
1.1 进程与线程.....	2
1.1.1 进程与多任务.....	2
1.1.2 进程与线程.....	2
1.1.3 多线程编程的困难.....	3
1.2 MFC 的进程和线程编程.....	3
1.2.1 创建、管理和终止进程.....	3
1.2.2 创建、管理和终止线程.....	6
1.2.3 线程的同步*.....	9
1.3 .NET 下的进程和线程编程.....	13
1.3.1 进程编程.....	13
1.3.2 线程编程.....	18
1.4 Java 的进程和线程编程.....	19
1.5 超线程与多核处理器.....	19
1.5.1 SMT 与超线程.....	19
1.5.2 多核处理器.....	21
1.6 并行计算.....	34
1.6.1 为什么要做并行计算.....	35
1.6.2 什么是并行计算.....	35
1.6.3 并行计算机.....	37
1.6.4 并行计算机体系结构.....	38
1.6.5 并行计算模型.....	42
1.6.6 并行计算性能评测.....	44
1.6.7 并行计算的挑战.....	44
1.7 并行编程.....	44
1.7.1 并行编程环境.....	45
1.7.2 编程语言与编译器.....	45
1.7.3 并行软件程序员的工作.....	46
1.7.4 并行程序设计.....	46
1.7.5 并行编译器.....	48
1.8 Visual C++本地多核编程.....	49
1.8.1 OpenMP.....	49
1.8.2 PPL.....	78
1.9 Visual C# .NET 多核编程.....	79
1.9.1 TPL.....	80
1.9.2 PLINQ.....	85
1.9.3 并行编程数据结构.....	86
参考文献.....	88

多任务的并发执行会用到多线程（multithreading），而 CPU 的多核（mult-core）化又将原来只在巨型机中才使用的并行计算（parallel computing）带入普通 PC 应用的多核程序设计（multi-core programming）中。

1.1 进程与线程

进程（process）是执行中的程序，线程（thread）是一种轻量级的进程。

1.1.1 进程与多任务

现代的操作系统都是多任务（multitask）的，即可同时运行多个程序。进程（process）是位于内存中正被 CPU 运行的可执行程序。参见图 15-1。

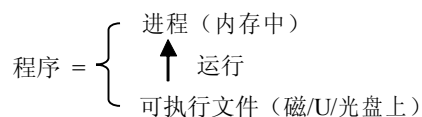


图 15-1 程序与进程

目前的主流计算机采用的都是冯·诺依曼（John von Neumann）体系结构——存储程序计算模型，程序（program）就是在内存中顺序存储并以线性模式在 CPU 中串行执行的指令序列。对于传统的单核 CPU 计算机，多任务操作系统的实现是通过 CPU 分时（time-sharing）和程序并发（concurrency）完成的。即在一个时间段内，操作系统将 CPU 分配给不同的程序，虽然每一时刻只有一个程序在 CPU 中运行，但是由于 CPU 的速度非常快，在很短的时间段中可在多个进程间进行多次切换，所以用户的感受就像多个程序在同时执行，我们称之为多任务的并发。

1.1.2 进程与线程

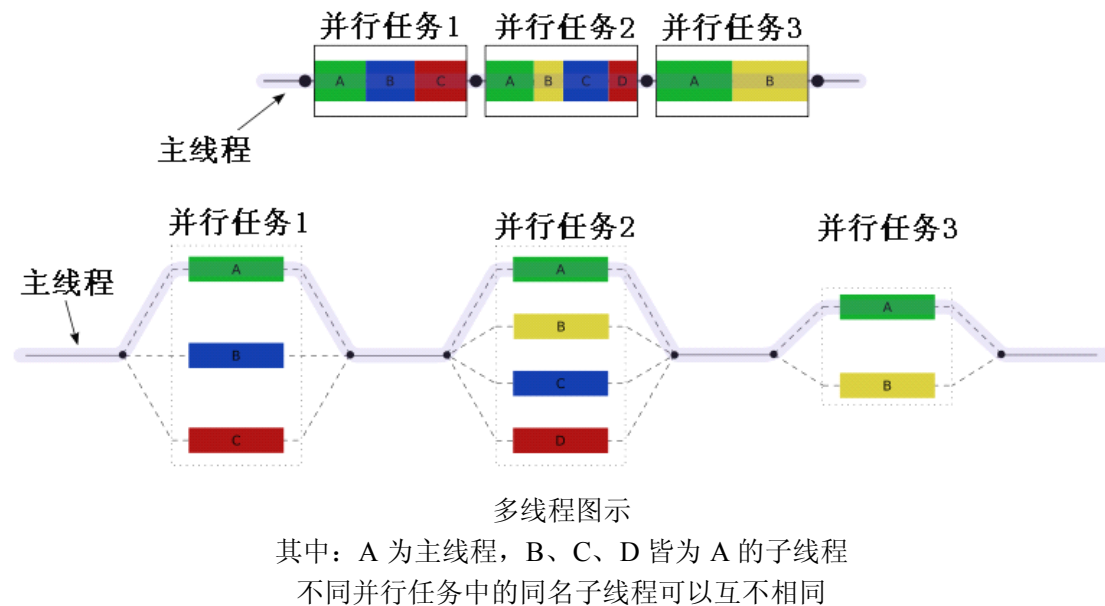
程序一般包括代码段、数据段和堆栈，对具有 GUI（Graphical User Interfaces，图形用户界面）的程序还包含资源段。进程（process）是应用程序的执行实例，即正在被执行的程序。每个进程都有自己的虚拟地址空间，并拥有操作系统分配给它的一组资源，包括堆栈、寄存器状态等。

线程（thread）是 CPU 的调度单位，是进程中的一个可执行单元，是一条独立的指令执行路径。线程只有一组 CPU 指令、一组寄存器和一个堆栈，它本身没有其他任何资源，而是与拥有它的进程共享几乎一切，包括进程的数据、资源和环境变量等。线程的创建、维护和管理给操作系统的负担比进程要轻得多，所以才叫轻量级的进程（lightweight process）。

一个进程可以拥有多个线程，而一个线程只能属于一个进程。每个进程至少包含一个线程——主线程，它负责程序的初始化工作，并执行程序的起始指令。随后，主线程可为执行各种不同的任务而分别创建多个子线程。

一个程序的多个运行，可以通过启动该程序的多个实例（即多个进程）来完成，也可以只运行该程序的一个实例（一个进程），而由该进程创建多个线程来做到。显然后者要比前

者更高效，更能节约系统的有限资源。这对需要在同一时刻响应成千上万个用户请求的 Web 服务器程序和网络数据库管理程序等来说是至关重要的。



有关进程和线程的进一步内容，大家会在将来的操作系统课程中学到。

1.1.3 多线程编程的困难

因为同一程序（进程）的多个线程共享同样的数据和资源，所以会出现同步、排队和竞争等问题，可能导致死锁、无限延迟和数据竞争等现象的发生，这些都需要我们在程序中添加解决。

MFC 虽然提供了一个线程类和若干同步类，但是仍然属于线程的低级编程，既困难又繁琐。利用 .NET 框架类库中的线程命名空间下的线程类，则可以简化线程编程。

1.2 MFC 的进程和线程编程

使用传统的 MFC/C++ 直接进行进程和线程编程异常复杂和繁琐，需要程序员自己处理线程间的同步、互斥、死锁等具体问题。

1.2.1 创建、管理和终止进程

MFC 中并没有提供处理进程的类，我们需要直接使用 Windows 的 API 函数来创建、管理和终止进程。

1. 创建进程

下面的 `CreateProcess` 函数用于在当前进程中创建一个新进程（和其主线程），以运行指定（路径/文件名或命令行）的应用程序：

`BOOL CreateProcess(// 成功返回非 0，失败返回 0（可用 GetLastError 函数返回出错代码）`

LPCTSTR lpApplicationName, // 可执行文件的全路径或文件名, 有命令行时可为 NULL
 LPTSTR lpCommandLine, // 命令行参数字符串, 有可应用名时可为 NULL
 LPSECURITY_ATTRIBUTES lpProcessAttributes, // 进程的安全属性, NULL 表默认安全
 LPSECURITY_ATTRIBUTES lpThreadAttributes, // 主线程的安全属性, NULL 表默认安全
 BOOL bInheritHandles, // 子进程是否继承新进程的句柄
 DWORD dwCreationFlags, // 创建标志, 用于设置进程的创建状态和优先级, 可为 0
 LPVOID lpEnvironment, // 环境变量, 为 NULL 时同当前进程的
 LPCTSTR lpCurrentDirectory, // 进程运行的当前目录, 为 NULL 时同当前进程的
 LPSTARTUPINFO lpStartupInfo, // 指向设置进程主窗口或控制条的各种属性的结构指针
 LPPROCESS_INFORMATION lpProcessInformation // 指向返回进程信息的结构指针

);

其中, 结构 STARTUPINFO 和 PROCESS_INFORMATION 的定义分别为:

```

typedef struct _STARTUPINFO {
    DWORD cb; // 结构的长度 (字节数)
    LPTSTR lpReserved; // 保留, 必须为 NULL
    LPTSTR lpDesktop; // 桌面-窗口站的名称
    LPTSTR lpTitle; // 控制台进程的标题
    DWORD dwX; // 窗口位置的横坐标
    DWORD dwY; // 窗口位置的纵坐标
    DWORD dwXSize; // 窗口的水平尺寸
    DWORD dwYSize; // 窗口的垂直尺寸
    DWORD dwXCountChars; // 控制台窗口的屏幕缓冲区宽度 (字符数)
    DWORD dwYCountChars; // 控制台窗口的屏幕缓冲区高度 (字符数)
    DWORD dwFillAttribute; // 控制台窗口的初始文本和背景色
    DWORD dwFlags; // 窗口的创建标志
    WORD wShowWindow; // 用作窗口显示函数 ShowWindow 的缺省参数
    WORD cbReserved2; // 保留, 必须为 0
    LPBYTE lpReserved2; // 保留, 必须为 NULL
    HANDLE hStdInput; // 标准输入的句柄
    HANDLE hStdOutput; // 标准输出的句柄
    HANDLE hStdError; // 标准错误的句柄
} STARTUPINFO, *LPSTARTUPINFO;
  
```

和

```

typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess; // 返回的进程句柄
    HANDLE hThread; // 返回的主线程句柄
    DWORD dwProcessId; // 返回的进程 ID
    DWORD dwThreadId; // 返回的主线程 ID
} PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
  
```

2. 管理进程

1) 获取进程的句柄和 ID

除了可从创建进程函数 CreateProcess 的最后一个 (返回) 参数——进程信息结构 PROCESS_INFORMATION 变量来获取所创建的新进程及其主线程的句柄和 ID 外, 还可利

用 API 函数 `GetCurrentProcess` 和 `GetCurrentProcessId` 来获取当前进程的句柄和 ID:

```
HANDLE GetCurrentProcess(void);
DWORD GetCurrentProcessId(void);
```

不过用 `GetCurrentProcess` 返回的是一个伪句柄，只能在当前进程中使用。可以调用 API 函数 `DuplicateHandle` 将此伪句柄转换为一个真正的句柄。

2) 获取和设置进程的优先级

在 Windows 操作系统中，进程有 6 种优先级别 (priority level/class)，从低到高分别为：空闲 (Idle)、低普通 (Below normal)、普通 (Normal)、高普通 (Above normal)、高 (High) 和实时 (Real time)，对应的符号常量为：

表 15-1 进程优先级符号常量

符号常量	对应数值
IDLE_PRIORITY_CLASS	0x00000040
BELOW_NORMAL_PRIORITY_CLASS	0x00004000
NORMAL_PRIORITY_CLASS	0x00000020
ABOVE_NORMAL_PRIORITY_CLASS	0x00008000
HIGH_PRIORITY_CLASS	0x00000080
REALTIME_PRIORITY_CLASS	0x00000100

可以在创建新进程时，利用其创建标志参数 `dwCreationFlags` 来设置。也可以用 API 函数 `GetPriorityClass` 和 `SetPriorityClass` 来获取和设置指定进程的优先级：

```
DWORD GetPriorityClass( HANDLE hProcess );
BOOL SetPriorityClass( HANDLE hProcess, DWORD dwPriorityClass );
```

例如：

```
DWORD p = GetPriorityClass( GetCurrentProcess() );
SetPriorityClass( GetCurrentProcess(), IDLE_PRIORITY_CLASS );
```

3) 等待进程返回

可以调用 API 函数 `WaitForSingleObject` 来等待指定进程（或线程）结束后返回：

```
DWORD WINAPI WaitForSingleObject( HANDLE hHandle, DWORD dwMilliseconds );
```

例如：`WaitForSingleObject(pi.hProcess, INFINITE);`

3. 结束进程

结束进程的方法有多种，可以调用 API 函数 `ExitProcessk` 来结束当前进程（及其所有线程）：（对控制台程序，在接收到 CTRL+C 或 CTRL+BREAK 信号后会调用此函数）

```
VOID ExitProcess( UINT uExitCode );
```

或调用 API 函数 `ExitProcess` 来结束指定进程（及其所有线程）：

```
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode );
```

进程的所有线程终止后进程也会自动终止。在用户关闭系统或注销推出系统时，也会导致进程终止。

在结束进程后，还需要调用 API 的 `CloseHandle` 函数来删除进程和线程对象：

```
BOOL CloseHandle(HANDLE hObject );
```

例如：`CloseHandle(pi.hProcess); CloseHandle(pi.hThread);`

4. 例子

下面是一个控制台程序，在该程序中，创建一个新进程来运行指定的另一个可执行程序。

为此，需要创建一个名为 Process 的“Visual C++/常规/空项目”，并将如下代码文件添加到此项目中：

```
// Process.cpp
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _tmain( ) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess(
        "E:\\CDPlay.exe", // Module name (须换成你自己磁盘上的某个可执行文件的路径)
        NULL,             // No Command line (use module name)
        NULL,             // Process handle not inheritable
        NULL,             // Thread handle not inheritable
        FALSE,            // Set handle inheritance to FALSE
        0,                // No creation flags
        NULL,             // Use parent's environment block
        NULL,             // Use parent's starting directory
        &si,               // Pointer to STARTUPINFO structure
        &pi )             // Pointer to PROCESS_INFORMATION structure
    ) {
        printf( "CreateProcess failed (%d)\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

1.2.2 创建、管理和终止线程

MFC 中提供了线程类 CWinThread，参见图 15-2。在 MFC 中区分两种类型的线程：用户界面线程（user-interface thread）

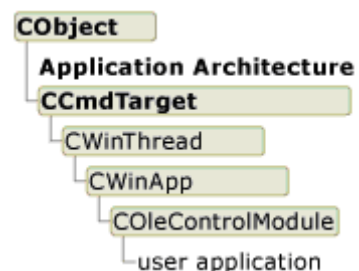


图 15-2 CWinThread 类
及其派生类

和辅助线程（worker thread）。用户界面线程通常用于处理用户输入及响应用户生成的事件和消息。辅助线程通常用于完成不需要用户输入的任务（如重新计算）。Win32 API 则不区分线程类型；它只需要了解线程的起始地址以开始执行线程。MFC 为用户界面中的事件提供消息泵（message pump），从而对用户界面线程进行专门处理。CWinApp 是用户界面线程对象的一个示例，因为它从 CWinThread 派生并对用户生成的事件和消息进行处理。

1. 创建线程

MFC 应用程序中的所有线程都由 CWinThread 对象表示。大多数情况下，甚至不必显式创建这些对象，而只需调用 MFC 框架的助手型全局函数 AfxBeginThread，该函数将自动创建 CWinThread 对象。也可以先创建自己的线程（C++）对象，再利用线程类的成员函数 CreateThread 来创建 Windows 线程。

1) 利用全局函数 AfxBeginThread 创建线程

AfxBeginThread 函数有两个版本，分别用于创建用户界面线程和辅助线程：

```
CWinThread* AfxBeginThread( // 创建用户界面线程
    CRuntimeClass* pThreadClass,
    int nPriority = THREAD_PRIORITY_NORMAL,
    UINT nStackSize = 0,
    DWORD dwCreateFlags = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);

CWinThread* AfxBeginThread( // 创建辅助线程
    AFX_THREADPROC pfnThreadProc,
    LPVOID pParam,
    int nPriority = THREAD_PRIORITY_NORMAL,
    UINT nStackSize = 0,
    DWORD dwCreateFlags = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
```

例如：

```
class CMyThread : public CWinThread {.....}
.....

CMyThread *pMyThread = (CMyThread*)AfxBeginThread(RUNTIME_CLASS(CSockThread));
if (pMyThread != NULL) {
    .....
    pMyThread->ResumeThread();
}
```

及

```
UINT WorkerThread(LPVOID pParam) {CWnd *pWin = (CWnd*) pParam; .....}
.....

AfxBeginThread(WorkerThread, this);
```

2) 利用 CWinThread 类的 CreateThread 函数创建线程

可以先从 CWinThread 类派生自己的线程类，再利用 CWinThread 的成员函数 CreateThread 来创建线程：

```

BOOL CreateThread(
    DWORD dwCreateFlags = 0,
    UINT nStackSize = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);

```

例如：

```

class CMyThread : public CWinThread {……}
……

CMyThread *pMyThread = new CMyThread;
pMyThread->CreateThread();
……

```

实际上，1) 中的 `AfxBeginThread` 函数也是通过先创建一个 `CWinThread` 对象，再调用其 `CreateThread` 函数来完成线程创建的。

2. 管理线程

1) 获取线程对象

可以利用 MFC 的全局函数 `AfxGetThread` 来获取当前的线程对象：

```
CWinThread* AfxGetThread( );
```

2) 获取和设置线程优先级

利用 `CWinThread` 类的成员函数 `GetThreadPriority` 和 `SetThreadPriority` 可以获取和设置线程的优先级：

```

int GetThreadPriority( );
BOOL SetThreadPriority( int nPriority );

```

线程的优先级是在其所属进程优先级的基础上的一种相对优先级，`nPriority` 的取值可为如下符号常量：

表 15-2 线程优先级符号常量

符号常量	相对数值
THREAD_PRIORITY_IDLE	-15
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_ABOVE_NORMAL	1
THREAD_PRIORITY_HIGHEST	2
THREAD_PRIORITY_TIME_CRITICAL	15

3) 检查线程是否活动

可以利用 API 函数 `GetExitCodeThread` 返回的退出代码是否为 `STILL_ACTIVE` (259) 来判断指定线程是否仍然在运行：

```
BOOL GetExitCodeThread( HANDLE hThread, LPDWORD lpExitCode );
```

例如：

```

DWORD ec;
GetExitCodeThread(pMyThread->m_hThread, &ec);
if (ec == STILL_ACTIVE) {……}

```


4) 挂起和恢复线程

可利用 CWinThread 类的成员函数 SuspendThread 和 ResumeThread 来挂起下车和恢复线程的运行:

```
DWORD SuspendThread( );  
DWORD ResumeThread( );
```

3. 终止线程

可以调用 MFC 的全局函数 AfxEndThread 来终止当前线程:

```
void AFXAPI AfxEndThread( UINT nExitCode, BOOL bDelete = TRUE );
```

为了正常终止一个辅助线程, 还可以使用 return 语句; 为了正常终止一个用户界面线程, 还可以在线程内调用 Windows SDK 中的 PostQuitMessage 函数:

```
void PostQuitMessage( int nExitCode );
```

还可以利用 API 函数 TerminateThread 强制终止指定线程:

```
BOOL WINAPI TerminateThread( HANDLE hThread, DWORD dwExitCode );
```

不过这样做是危险的。

1.2.3 线程的同步*

由于同一进程的多个线程共享同样的数据段, 具有同一地址空间。为了解决并行或并发的多个线程同时访问同一数据或资源所造成的冲突, 需要解决线程的同步 (synchronization) 问题。

Windows 操作系统提供了多种同步对象, 并可替我们管理同步对象的加锁和解锁操作。我们的任务只是对每个需要同步使用的数据和资源产生一个同步对象, 并在使用这些数据和资源前申请加锁, 且在使用完成后申请解锁。

Windows 设置了 4 种同步对象——临界区 (critical section)、互斥量 (mutex)、信号量 (semaphore) 和事件 (event), 其中, 除了临界区外, 其余三种同步对象都是操作系统的内核对象。MFC 封装了这 4 种同步对象, 对应的类分别为 CCriticalSection、CMutex、CSemaphore 和 CEvent, 它们都是同步对象类 CSyncObject 的派生类。另外, MFC 还提供了两个同步访问对象类 CMultiLock 和 CSingleLock, 它们俩都是没有基类的独立类, 参见图 15-3。

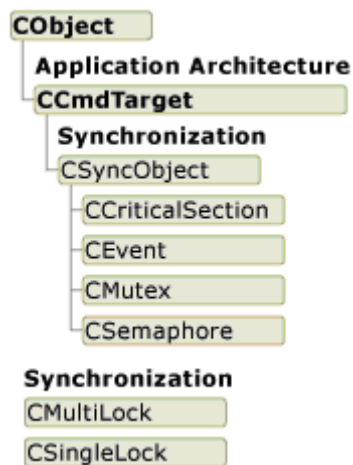


图 15-3 MFC 同步类

表 15-3 MFC 中的同步类

对象名	类名	描述	用于
临界区	CCriticalSection	只允许当前进程中的一个线程访问某个对象	只有一个应用程序使用此资源时
互斥量	CMutex	只允许系统中的一个进程内的一个线程访问某个对象	可有多多个应用程序使用此资源时
信号量	CSemaphore	只允许知道数目的线程同时访问某个对象	同一应用程序内多个线程可以同时访问此资源时
事件	CEvent	当某个事件发生时通知一个程序 (的线程)	必须等到发生某事才能访问资源时
多锁	CMultiLock	为多个访问对象加锁	在一特定时间需使用多个对象时

单锁	CSingleLock	为单个访问对象加锁	一次等待一个对象时
----	-------------	-----------	-----------

1. 临界区

临界区（critical section）是一个进程中的所有线程共享的某个受保护的资源或代码段，被锁定后每次只能被一个线程所使用。临界区也是最容易使用的同步对象，但是只能用于单个进程中的线程同步，而不能被其他进程共享。由于临界区对象不是 Windows 的内核对象，所以它存在于进程的内存空间中。

为了在 MFC 中使用临界区，必须先创建一个 CCriticalSection 对象；在线程进入临界区之前，调用该对象的成员函数 Lock 来锁定临界区；在线程离开临界区之后，调用该对象的成员函数 Unlock 来解锁临界区。如果在调用 Lock 函数时，没有其他线程锁定临界区，则 Lock 函数对临界区加锁后立即返回，使调用它的线程继续运行；如果在调用 Lock 函数时，已经有其他线程锁定了临界区，则 Lock 函数被挂起，直到其他线程解锁临界区后才能返回。

例子：

```
CCriticalSection g_cs; // 创建临界区对象
int g_iNum = 0; // 需锁定的全局变量（受保护的同步数据资源）
DWORD ThreadProc(LPVOID pParam) { // 自定义的线程过程处理函数
    g_cs.Lock(); // 加锁
    g_iNum++; // 临界区（受保护的代码段）
    g_cs.Unlock(); // 解锁
    return 0;
}
```

2. 互斥量

互斥量（mutex）的用途与临界区类似，但是它可以跨进程使用，能用来同步多个进程的数据共享访问，不过互斥量的操作速度比临界区的要慢近百倍。由于互斥量是 Windows 的内核对象，所以它存在于系统内存空间中，并具有引用计数。

MFC 的 CMutex 类封装了互斥量对象，其使用方法类似于临界区类的——先构造一个 CMutex 对象，再调用函数 Lock 来锁定互斥量；在线程离开互斥量之后，调用 Unlock 来解锁互斥量。

下面是 CMutex 类的构造函数及从其基类继承下来的加锁与解锁函数：

```
CMutex(
    BOOL bInitiallyOwn = FALSE, // 指定创建线程是否初始时有权访问被保护的资源
    LPCTSTR lpszName = NULL, // 跨进程使用时需指定相同的互斥量名
    LPSECURITY_ATTRIBUTES lpsaAttribute = NULL // 指向安全属性结构的指针
);
virtual BOOL Lock( DWORD dwTimeout = INFINITE );
virtual BOOL Unlock( ) = 0;
```

其中，构造函数中的 bInitiallyOwn 参数为 TRUE 时，则会直到（指定名称的）互斥量可用时，才会从对此构造函数的调用中返回，可用于在希望等待的时刻来创建一个 CMutex 对象。加锁函数中的 dwTimeout 参数，用于指定等待时间的毫秒数，缺省为无限长（INFINITE）。如果指定了该参数的有限值，则超时时会放弃加锁并返回 FALSE。

例子：

```
CMutex g_mx; // 创建互斥量对象
int g_iNum = 0; // 需锁定的全局变量（受保护的同步数据资源）
```

```

DWORD ThreadProc(LPVOID pParam) { // 自定义的线程过程处理函数
    if(g_mx.Lock(100)) { // 加锁
        g_iNum++; // 互斥量（受保护的代码段）
        g_mx.Unlock(); // 解锁
    }
    return 0;
}

```

3. 信号量

信号量（semaphore，旗语）也是一种内核对象，用于资源的计数（即可使用该资源的线程个数）。当线程用信号量对象的句柄作参数来调用等待函数 WaitForSingleObject 时，系统会检查该信号量所对应的资源数是否大于 0（即资源是否可用），若大于 0（称为有信号 signaled）则减少资源计数并唤醒线程，若等于 0（称为无信号 nonsignaled）则让线程进入睡眠状态直到（超出指定时间或当指定时间为无限时）占用该资源的其他线程释放资源并增加资源的计数（使信号量大于 0，即有信号或有资源可用）为止。

信号量与前面讲的临界区和互斥量不同，它不属于某个线程，而且它允许（不同进程中的）多个线程同时访问一个受保护的资源（资源的可访问线程数由信号量的最大计数值决定）。信号量对象可用来限制对共享资源（如串口）进行访问的线程数目（如须≤计算机的串口总数）。

MFC 中的信号量类为 CSemaphore，其构造函数为

```

CSemaphore(
    LONG lInitialCount = 1, // 初始计数值，须≥0 且≤lMaxCount
    LONG lMaxCount = 1, // 最大计数值，须≥lInitialCount
    LPCTSTR pstrName = NULL, // 信号量名串，用于多个进程中的线程
    LPSECURITY_ATTRIBUTES lpsaAttributes = NULL // 安全属性，NULL 表缺省
);

```

为了访问用信号量计数的资源，还需要创建一个 CSingleLock（或 CMultiLock）对象，来锁定和释放受保护的资源。CSingleLock 类的成员函数有：

```

explicit CSingleLock( // 构造函数
    CSyncObject* pObject, // 同步对象（如 CSemaphore 对象），不能为 NULL
    BOOL bInitialLock = FALSE // 初始时是否锁定资源
);
BOOL IsLocked( ); // 判断是否锁定
BOOL Lock( DWORD dwTimeOut = INFINITE ); // 锁定资源
BOOL Unlock( ); // 释放资源
BOOL Unlock( // 释放资源
    LONG lCount, // 释放的访问计数，须≥0
    LPLONG lPrevCount = NULL // （返回）原计数的指针，NULL 表不返回
);

```

例子：

```

CSemaphore m_smph(5, 5);
.....
WaitForSingleObject(m_smph.m_hObject, INFINITE );
.....

```

```

CSingleLock singleLock(&m_smph);
singleLock.Lock(); // Attempt to lock the shared resource
if (singleLock.IsLocked()) { // Resource has been locked
    // Use the shared resource
    .....
    // Now that we are finished,
    // unlock the resource for others.
    singleLock.Unlock();
}

```

4. 事件

有时线程并不是要访问某个受保护的数据或资源，而是需要等待某一事件（event）的发生，这在 GUI 编程中十分常见。事件对象用于一个线程通知另一线程某一事件的发生（发信号表示某一操作已经完成），它是同步对象中形式最简单的一种，而且其同步的机制也是最具有弹性的。事件是一种内核对象，具有激发（有信号 signaled）和非激发（无信号 nonsignaled）两种状态，状态完全由程序来控制。

有两类事件对象——手工的（manual）和自动的（automatic）。手工事件对象会保持（由函数 SetEvent 或 ResetEvent 所设置的）状态不变，直到调用其他函数；而自动事件对象则在（至少一个）线程被释放后会自动返回无信号（不可用）状态。

MFC 的 CEvent 类封装了 Windows 的事件对象，其成员函数有：

```

CEvent( // 构造函数
    BOOL bInitiallyOwn = FALSE, // TRUE: 线程对单/多锁对象可用;
                                   // FALSE: 想访问资源的所有线程都必须等待
    BOOL bManualReset = FALSE, // TRUE: 手工事件对象; FALSE: 自动事件对象
    LPCTSTR lpszName = NULL, // 事件名串, 用于跨进程的线程
    LPSECURITY_ATTRIBUTES lpsaAttribute = NULL // 安全属性, NULL 表缺省
);
BOOL SetEvent( ); // 设置事件为有信号（激发），释放任意在等待的线程
BOOL ResetEvent( ); // 设置事件为无信号（未激发），直到调用 SetEvent 才能激发
BOOL PulseEvent( ); // 先激发事件，在释放等待的线程后，再自动非激发事件
BOOL Unlock( ); // 释放事件对象

```

使用 CEvent 对象的一般方法是，在适当需要的时候构造 CEvent 对象，在适当的时候调用其 SetEvent 函数激发事件（使事件对象处于有信号状态），在完成对所控资源的访问后，再调用其 Unlock 函数释放事件对象。

例子：

```

UINT __cdecl MyThreadProc(LPVOID lpParameter) {
    CEvent* pEvent = (CEvent*)(lpParameter);
    VERIFY(pEvent != NULL);

    // Wait for the event to be signaled
    ::WaitForSingleObject(pEvent->m_hObject, INFINITE);

    // Terminate the thread
    ::AfxEndThread(0, FALSE);
}

```

```

        return 0L;
    }

    void CEvent_Test() {
        // Create the CEvent object that will be passed to the thread routine
        CEvent* pEvent = new CEvent(FALSE, FALSE);

        // Create a thread that will wait on the event
        CWinThread* pThread;
        pThread = ::AfxBeginThread(&MyThreadProc, pEvent, 0, 0,
                                   CREATE_SUSPENDED, NULL);
        pThread->m_bAutoDelete = FALSE;
        pThread->ResumeThread();

        // Signal the thread to do the next work item
        pEvent->SetEvent();

        // Wait for the thread to consume the event and return
        ::WaitForSingleObject(pThread->m_hThread, INFINITE);
        delete pThread;
        delete pEvent;
    }

```

Windows API 还提供了多个互锁函数 `Interlocked*`，用于多个线程对一个共享变量的同步，能绝对保证改变变量的线程独占对该变量的访问。Windows API 还提供了一组使线程阻塞自身执行的等待函数，包括等待单个对象的 `SignalObjectAndWait`、`WaitForSingleObject` 和 `WaitForSingleObjectEx`；等待多个对象的 `WaitForMultipleObjects`、`WaitForMultipleObjectsEx`、`MsgWaitForMultipleObjects` 和 `MsgWaitForMultipleObjectsEx`；发出提示的 `MsgWaitForMultipleObjectsEx`、`SignalObjectAndWait`、`WaitForMultipleObjectsEx` 和 `WaitForSingleObjectEx`；注册登记的 `RegisterWaitForSingleObject` 和 `UnregisterWaitEx`。由于时间和篇幅的限制，这里就不详细介绍了。

1.3 .NET 下的进程和线程编程

1.3.1 进程编程

在 .NET 的框架类库中，与进程编程相关的类有 `Process`（进程）、`ProcessStartInfo`（进程启动信息）、`ProcessModule`（进程模块）等，它们都位于 `System.Diagnostics` 命名空间中（程序集为在 `System.dll` 中的 `System`）。`ProcessStartInfo` 为一个独立的类，`Process` 和 `ProcessModule` 的基类都为（`System.ComponentModel` 命名空间中的）`Component`。

`Process` 类（组件）提供对正在计算机上运行的进程的访问，可用来启动、停止、控制和监视应用程序等任务。使用 `Process` 组件，可以获取正在运行的进程的列表，也可以启动新的进程。

`Process` 类的 `ProcessorAffinity` 属性可用于获取或设置一些处理器，此进程中的线程可以按计划在这些处理器上运行。其属性值为 `System.IntPtr` 类型的位掩码，表示关联进程内的线程可以在其上运行的处理器。默认值为 $2^n - 1$ ，其中 n 是计算机上的处理器数。这一点可用于多核编程。

可使用 `ProcessStartInfo` 类来更好地控制启动的进程。至少必须以手动方式或使用构造函数来设置（应用程序或文档的）文件名属性 `FileName`。此处，将文档定义为具有与其关联的打开或默认操作的任何文件类型。使用操作系统提供的“文件夹选项”对话框，可以查看计算机中已注册的文件类型及其相关应用程序。单击“高级”按钮可打开一个对话框，其中显示了是否存在与特定注册文件类型相关联的打开操作。

另外，还可使用 `ProcessStartInfo` 类来设置定义要对该文件执行的操作的其他属性。可以为 `Verb` 属性指定特定于 `FileName` 属性的类型的值。例如，可以为文档类型指定“print”。另外，还可以指定 `Arguments` 属性值，这些值将成为传递给文件的打开过程的命令行参数。例如，如果在 `FileName` 属性中指定一个文本编辑器应用程序，则可以使用 `Arguments` 属性指定将用该编辑器打开的一个文本文件。

在进程启动前，可更改任何 `ProcessStartInfo` 属性的值。而启动进程后，更改这些值是没有效果的。

`ProcessModule` 类表示加载到特定进程中的 .dll 或 .exe 文件。每个进程包含一个或多个模块，可用该类来获取进程中模块的信息。

下面是几个相关的 C# 例子：

例 1：（`Process` 类）

```
using System;
using System.Diagnostics;
using System.ComponentModel;

namespace MyProcessSample {
    /// <summary>
    /// Shell for the sample.
    /// </summary>
    class MyProcess {
        // These are the Win32 error code for file not found or access denied.
        const int ERROR_FILE_NOT_FOUND = 2;
        const int ERROR_ACCESS_DENIED = 5;

        /// <summary>
        /// Prints a file with a .doc extension.
        /// </summary>
        void PrintDoc() {
            Process myProcess = new Process();

            try {
                // Get the path that stores user documents.
                string myDocumentsPath =
                    Environment.GetFolderPath(Environment.SpecialFolder.Personal);
```

```

        myProcess.StartInfo.FileName = myDocumentsPath + "\\MyFile.doc";
        myProcess.StartInfo.Verb = "Print";
        myProcess.StartInfo.CreateNoWindow = true;
        myProcess.Start();
    } catch (Win32Exception e) {
        if(e.NativeErrorCode == ERROR_FILE_NOT_FOUND) {
            Console.WriteLine(e.Message + ". Check the path.");
        }
        else if (e.NativeErrorCode == ERROR_ACCESS_DENIED) {
            // Note that if your word processor might generate exceptions
            // such as this, which are handled first.
            Console.WriteLine(e.Message +
                ". You do not have permission to print this file.");
        }
    }
}

public static void Main() {
    MyProcess myProcess = new MyProcess();
    myProcess.PrintDoc();
}
}
}

```

例 2 (ProcessStartInfo 类)

```

using System;
using System.Diagnostics;
using System.ComponentModel;

namespace MyProcessSample {
    /// <summary>
    /// Shell for the sample.
    /// </summary>
    class MyProcess {

        /// <summary>
        /// Opens the Internet Explorer application.
        /// </summary>
        void OpenApplication(string myFavoritesPath) {
            // Start Internet Explorer. Defaults to the home page.
            Process.Start("IExplore.exe");

            // Display the contents of the favorites folder in the browser.
            Process.Start(myFavoritesPath);
        }
    }
}

```

```

    }

    /// <summary>
    /// Opens urls and .html documents using Internet Explorer.
    /// </summary>
    void OpenWithArguments() {
        // url's are not considered documents. They can only be opened
        // by passing them as arguments.
        Process.Start("IExplore.exe", "www.northwindtraders.com");

        // Start a Web page using a browser associated with .html and .asp files.
        Process.Start("IExplore.exe", "C:\\myPath\\myFile.htm");
        Process.Start("IExplore.exe", "C:\\myPath\\myFile.asp");
    }

    /// <summary>
    /// Uses the ProcessStartInfo class to start new processes, both in a minimized
    /// mode.
    /// </summary>
    void OpenWithStartInfo() {

        ProcessStartInfo startInfo = new ProcessStartInfo("IExplore.exe");
        startInfo.WindowStyle = ProcessWindowStyle.Minimized;

        Process.Start(startInfo);

        startInfo.Arguments = "www.northwindtraders.com";

        Process.Start(startInfo);
    }

    static void Main() {
        // Get the path that stores favorite links.
        string myFavoritesPath = Environment.GetFolderPath(
Environment.SpecialFolder.Favorites);

        MyProcess myProcess = new MyProcess();

        myProcess.OpenApplication(myFavoritesPath);
        myProcess.OpenWithArguments();
        myProcess.OpenWithStartInfo();
    }

```



```
}  
}
```

例 3 (ProcessModule 类)

```
Process myProcess = new Process();  
// Get the process start information of notepad.  
ProcessStartInfo myProcessStartInfo = new ProcessStartInfo("notepad.exe");  
// Assign 'StartInfo' of notepad to 'StartInfo' of 'myProcess' object.  
myProcess.StartInfo = myProcessStartInfo;  
// Create a notepad.  
myProcess.Start();  
System.Threading.Thread.Sleep(1000);  
ProcessModule myProcessModule;  
// Get all the modules associated with 'myProcess'.  
ProcessModuleCollection myProcessModuleCollection = myProcess.Modules;  
Console.WriteLine("Properties of the modules associated "  
    + "with 'notepad' are:");  
// Display the properties of each of the modules.  
for( int i=0;i<myProcessModuleCollection.Count;i++)  
{  
    myProcessModule = myProcessModuleCollection[i];  
    Console.WriteLine("The moduleName is " + myProcessModule.ModuleName);  
    Console.WriteLine("The " + myProcessModule.ModuleName + "'s base address is: "  
        + myProcessModule.BaseAddress);  
    Console.WriteLine("The " + myProcessModule.ModuleName  
        + "'s Entry point address is: " + myProcessModule.EntryPointAddress);  
    Console.WriteLine("The " + myProcessModule.ModuleName + "'s File name is: "  
        + myProcessModule.FileName);  
}  
// Get the main module associated with 'myProcess'.  
myProcessModule = myProcess.MainModule;  
// Display the properties of the main module.  
Console.WriteLine("The process's main moduleName is: "  
    + myProcessModule.ModuleName);  
Console.WriteLine("The process's main module's base address is: "  
    + myProcessModule.BaseAddress);  
Console.WriteLine("The process's main module's Entry point address is: "  
    + myProcessModule.EntryPointAddress);  
Console.WriteLine("The process's main module's File name is: "  
    + myProcessModule.FileName);  
myProcess.CloseMainWindow();
```

1.3.2 线程编程

.NET 框架类库的（位于 mscorlib.dll 的程序集 mscorlib 中的）System.Threading 命名空间中的 Thread 类（基类为 CriticalFinalizerObject），用于创建并控制线程、设置其优先级并获取其状态。

一个进程可以创建一个或多个线程以执行与该进程关联的部分程序代码。使用 ThreadStart 委托或 ParameterizedThreadStart 委托指定由线程执行的程序代码。使用 ParameterizedThreadStart 委托可以将数据传递到线程过程。

在线程存在期间，它总是处于由 ThreadState 定义的一个或多个状态中。可以为线程请求由 ThreadPriority 定义的调度优先级，但不能保证操作系统会接受该优先级。

从 Object 类继承的 GetHashCode 方法，提供托管线程的标识。在线程的生存期内，无论获取该值的应用程序域如何，它都不会和任何来自其他线程的值冲突。

需要的说明是，操作系统 ThreadId 和托管线程没有固定关系，这是因为非托管宿主能控制托管与非托管线程之间的关系。特别是，复杂的宿主可以使用 CLR Hosting API 针对相同的操作系统线程调度很多托管线程，或者在不同的操作系统线程之间移动托管线程。

一旦启动线程，便不必保留对 Thread 对象的引用。线程将继续执行，直到该线程过程完成。

例子（C#）：

```
using System;
using System.Threading;

// Simple threading scenario: Start a static method running
// on a second thread.
public class ThreadExample {
    // The ThreadProc method is called when the thread starts.
    // It loops ten times, writing to the console and yielding
    // the rest of its time slice each time, and then ends.
    public static void ThreadProc() {
        for (int i = 0; i < 10; i++) {
            Console.WriteLine("ThreadProc: {0}", i);
            // Yield the rest of the time slice.
            Thread.Sleep(0);
        }
    }

    public static void Main() {
        Console.WriteLine("Main thread: Start a second thread.");
        // The constructor for the Thread class requires a ThreadStart
        // delegate that represents the method to be executed on the
        // thread. C# simplifies the creation of this delegate.
        Thread t = new Thread(new ThreadStart(ThreadProc));

        // Start ThreadProc. Note that on a uniprocessor, the new
        // thread does not get any processor time until the main thread
```

```

// is preempted or yields. Uncomment the Thread.Sleep that
// follows t.Start() to see the difference.
t.Start();
//Thread.Sleep(0);

for (int i = 0; i < 4; i++) {
    Console.WriteLine("Main thread: Do some work.");
    Thread.Sleep(0);
}

Console.WriteLine("Main thread: Call Join(), to wait until ThreadProc ends.");
t.Join();
Console.WriteLine("Main thread: ThreadProc.Join has returned. Press Enter to end
program.");
Console.ReadLine();
}
}

```

1.4 Java 的进程和线程编程

1.5 超线程与多核处理器

Intel 公司的超线程技术将一个物理处理器核模拟成两个逻辑核，可并行执行两个线程，从而能有效提高处理器的运行效率。

一直以来，处理器芯片厂商都是通过不断提高主频来提高处理器的性能（例如 Intel 于 2004 年 12 月 12 日推出的 Pentium 4 HT 570J 处理器的主频就达到了 3.8GHz）。但是随着芯片制程工艺的不断进步，单个芯片上集成的晶体管数已超过数亿，传统处理器体系结构技术面临瓶颈，很难单纯通过提高主频来提升性能。而且在主频的提高同时，带来功耗的迅速提高以及散热等问题非常严重，这些也是直接促使单核转向多核的深层次原因。从应用需求来看，日益复杂的多媒体、科学计算、虚拟化等多个应用领域都呼唤更为强大的计算能力。在这样的背景下，各主流处理器厂商将产品战略从提高芯片的时钟频率转向多线程和多内核方面。

总之，不能永远靠加快频率的方法来改善性能。频率高到一定程度以后，必然要转向多核技术。这是由芯片的先天气质决定的。

1.5.1 SMT 与超线程

SMT（Simultaneous MultiThreading，同时多线程）使用硬件多线程来改善超标量 CPU 的整体性能，它允许执行多个（如 2~16 个或更多）独立的线程来更好地利用现代处理器架构所提供的资源。超线程（HT，Hyper-Threading，早期曾叫 Super-Threading）是 Intel 公司

研发的一种在一个实体处理器核中提供两个逻辑线程的技术，是 SMT 技术的特例。

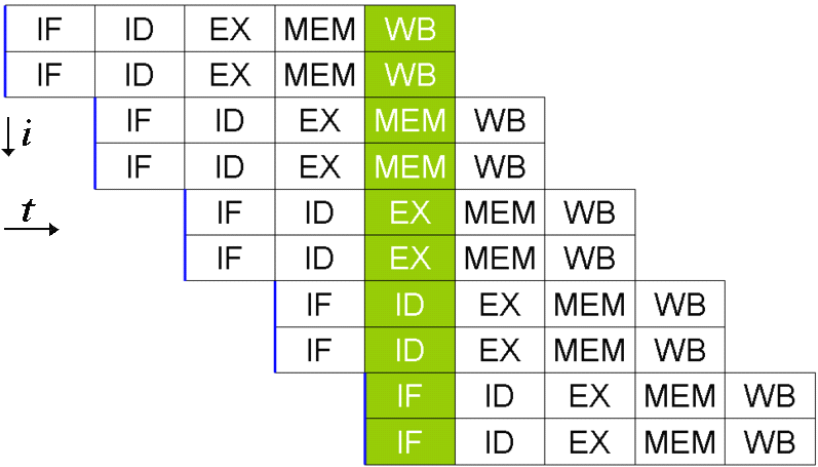
1. 超标量与流水线

超标量 (superscalar) CPU 架构，使用单颗核心来实现一种被称为指令集并行 (instruction-level parallelism) 的并行运算形式，它能够在相同的时钟频率下增加 CPU 的吞吐量。超标量处理器，通过同时分派多条指令给处理器上的冗余功能单元，可以在一个时钟周期内执行一条以上的指令。每个功能单元 (functional unit) 不是单独的 CPU 核，而是在单个 CPU 内的一个执行资源，例如一个 ALU (Arithmetic Logic Unit，算术逻辑单元)、一个位移器 (bit shifter) 或一个乘法器 (multiplier)。

在典型情况下，超标量 CPU 同时也是流水线的，它们是两种不同的性能增强技术。非流水线的超标量 CPU 或流水线的非超标量 CPU 在理论上是可能的。

流水线 (pipeline，管道/管线)，是一个串连在一起的数据处理元素的集合，其中的一个元素的输出是下一个元素的输入。流水线中的元素常常以并行或时间片方式执行，在这种情况下，在元素之间常常插入一些数量的缓冲存储器。

指令流水线 (instruction pipelines) 是与计算机相关的流水线 (其他流水线有图形流水线和软件流水线等) 中的一种，它被用于处理器中，允许在同一时钟周期 (circuitry) 内重叠执行多个指令。时钟周期通常分割成阶段，包括指令解码、算术和寄存器读取阶段，其中每个阶段每次处理一条指令。

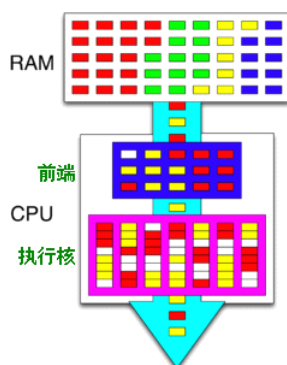


简单的超标量流水线

通过每次读取和分派两个指令，在每个指令周期可以完成最多两个指令

2. 超线程

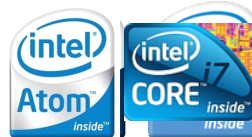
Intel 的超线程技术利用特殊的硬件指令，把两个逻辑内核模拟成两个物理芯片，让单个处理器核都能使用线程级并行计算，进而兼容多线程操作系统和软件，减少了 CPU 的闲置时间，提高的 CPU 的运行效率。参加下图：



采用超线程及时可在同一时间里，应用程序可以使用芯片的不同部分。虽然单线程芯片每秒钟能够处理成千上万条指令，但是在任一时刻只能对一条指令进行操作。而超线程技术可以使芯片同时进行多线程处理，使芯片性能得到提升。

Intel 表示，超线程技术让（P4）处理器在只增加 5% 的芯片面积的情况下，就可以换来 15%~30% 的效能提升。但实际上，在某些程序或未对多线程编译的程序而言，超线程反而会降低效能。除此之外，超线程 CPU 技术也需要主板芯片组和操作系统的配合，才能充分发挥超线程的效能。Intel 公司的 i865PE 和 i875P 及更新的芯片组。微软公司的 Windows XP、Windows Vista 和 Windows 7 等操作系统都能较好地支持 Intel 的超线程 CPU。

2002 年 2 月 Intel 公司在其推出的代号为 Prestonia 的 130nm 新款至强（Xeon）处理器中首次采用超线程技术。Intel 在其 2003 年 5 月 21 日推出的代号为 Northwood 的新款 Intel Pentium 4 HT CPU（及绝大多数更新推出的 P4 处理器）中也内含超线程技术。Intel 公司在其 2005 年 5 月 1 日推出的代号为 Smithfield 的 EE（Extreme Edition，极致版）系列和 2006 年 1 月 16 日代号为 Presler 的双核 Pentium D 处理器中也支持超线程技术。但是，Intel 公司于 2006 年 7 月 27 日开始推出的 Core 2（酷睿 2）处理器，包括 Solo（单核，只限笔记本电脑）、Duo（双核）、Quad（四核）及 Extreme（极致）等型号，都不支持超线程。不过，Intel 公司在其于 2008 年 4 月 2 日发布的用于笔记本和上网本的 Atom（凌动）单核处理器和 2008 年 11 月 17 日推出的 Core i7（酷睿 i7）桌面四核处理器又开始重新支持超线程技术。



1.5.2 多核处理器

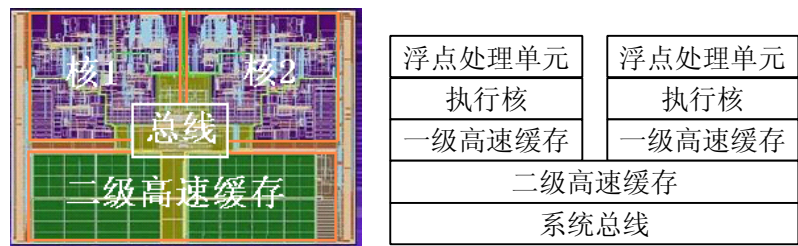
多核，即多微处理器核心，是将两个或更多的独立处理器核封装在一个集成电路（IC）芯片中的一种方案。一般说来，多核心微处理器允许一个计算设备，在不需要将多个处理器核心分别进行独立的物理封装情况下，可以执行某些形式的线程级并行处理（Thread-Level Parallelism, TLP）。这种形式的 TLP，通常被认为是芯片级别的多处理（Chip-level MultiProcessing, CMP）。

1. 多核构架

按硬件层次划分，多核的种类有：

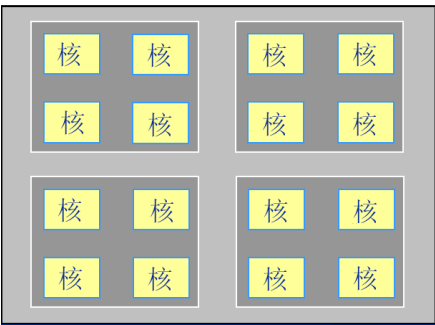
- 芯片级（多核芯片）：片上多核处理器（Chip Multi-Processor, CMP）就是将多个计算内核集成在一个处理器芯片中，从而提高计算能力。按计算内核的对等与否，CMP 可分为同构多核（如 Intel 和 Sun）和异构多核（如 IBM）。CPU 核心数据共享与同步，包括总线共享 Cache 结构（每个 CPU 内核拥有共享的二级或三级 Cache，用于保存比较常

用的数据，并通过连接核心的总线进行通信。例如 Intel 的 Core 2 Due 和 Core i7）和基于片上互连的结构（每个 CPU 核心具有独立的处理单元和 Cache，各个 CPU 核心通过交叉开关或片上网络等方式连接在一起。例如 Intel 的 Pentium D 和 Core 2 Quad）。参见下图：



Intel Core 2 Due 的平面和逻辑结构图

●板级：在一块主板上集成多个（多核）芯片。参见下图：



- 机架级：将多个含（多核）处理器的主板置于同一机箱内，主板之间利用专用芯片和线路进行通信。
- 网络级（网络）：将多个（多核）主机用（局域或互联）网连接在一起，构成分布式多核系统。

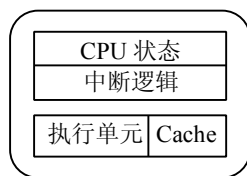
我们下面只讨论 CMP 级的多核，并且以 Intel 公司的 Core 系列微处理器为主。

2. 体系结构

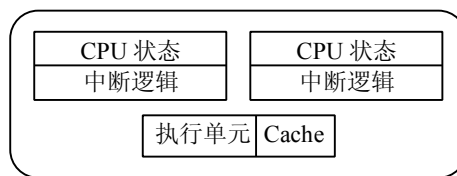
下面简单介绍一般的超线程与多核的体系结构，以及主流的单核和多核处理器——Intel 公司的 Pentium、Pentium D、Core 2 和 Core i7 的逻辑结构及其基础微架构。

●多核和超线程

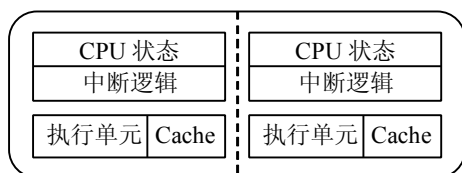
下面是单核、多核和超线程处理器的体系结构（architecture）示意图：



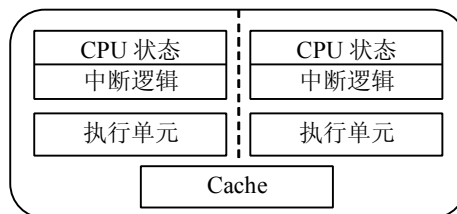
单核单线程 CPU



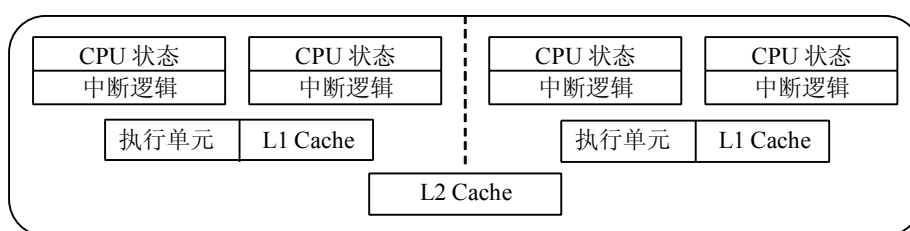
单核双线程 CPU



独立 cache 的双核双线程 CPU

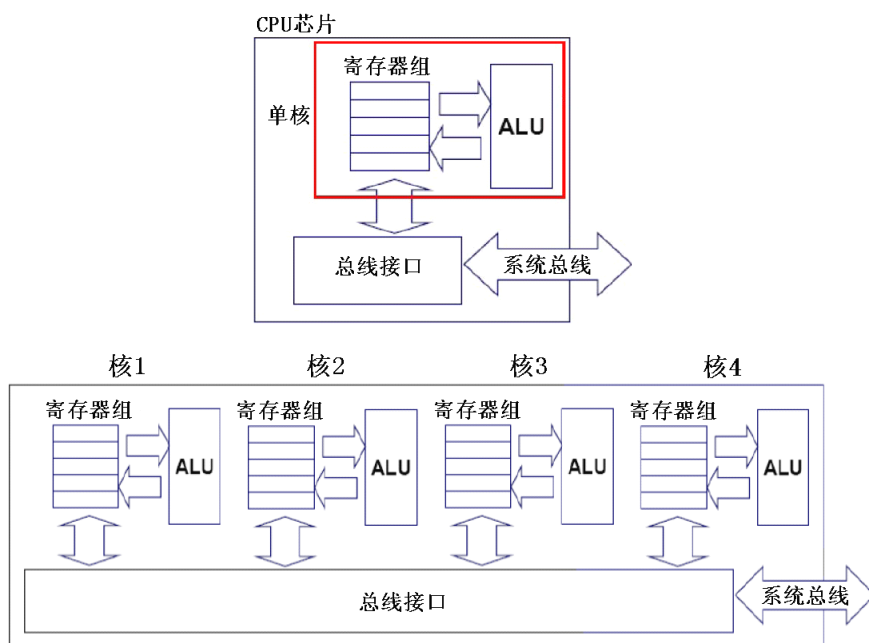


共享 cache 的双核双线程 CPU



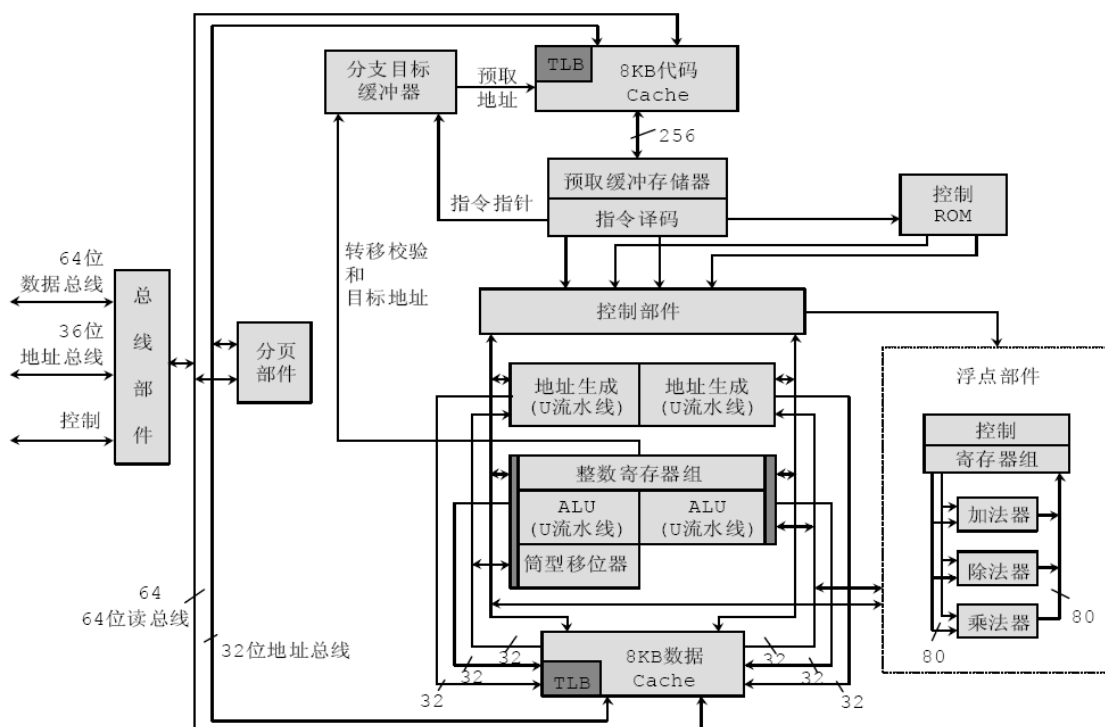
双核四线程 CPU

下面是分别单核与多核处理器的芯片结构图：



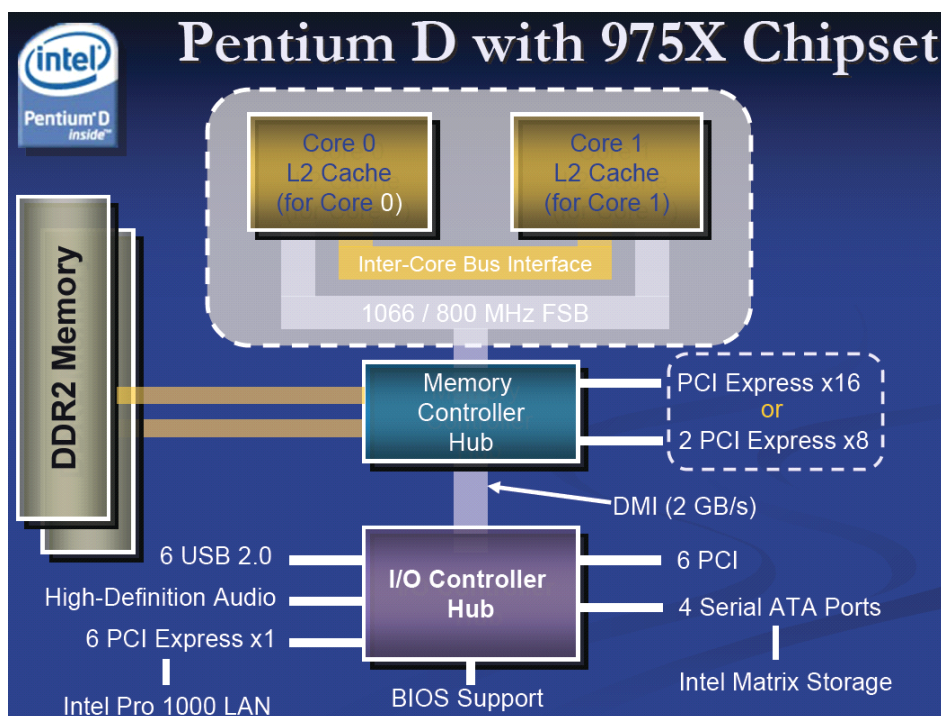
● 奔腾处理器

下面是 Intel Pentium（奔腾）微处理器的功能结构图：



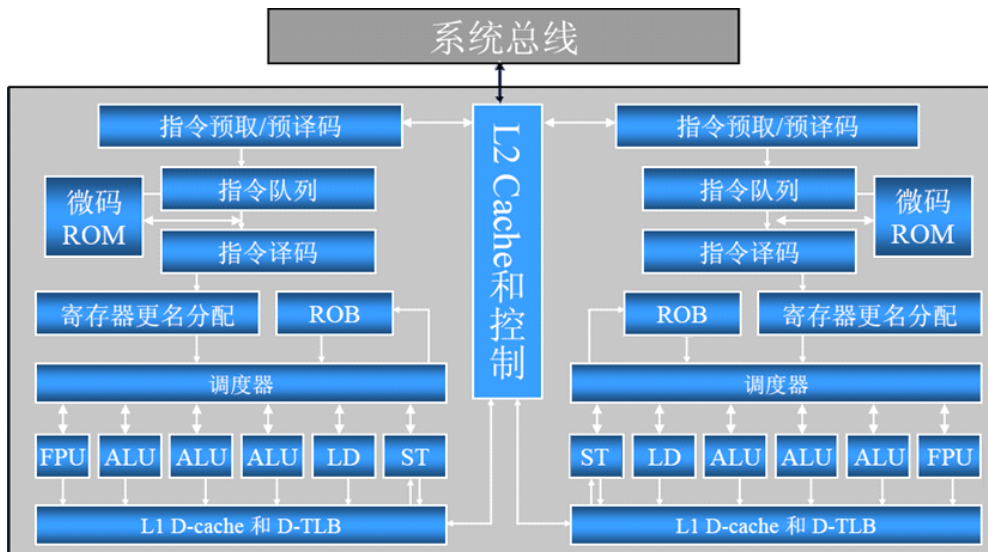
●奔腾 D 处理器

下面是 Intel Pentium D 微处理器及其配套芯片组的功能结构图：



●Intel Core 2 与 Intel Core 微架构

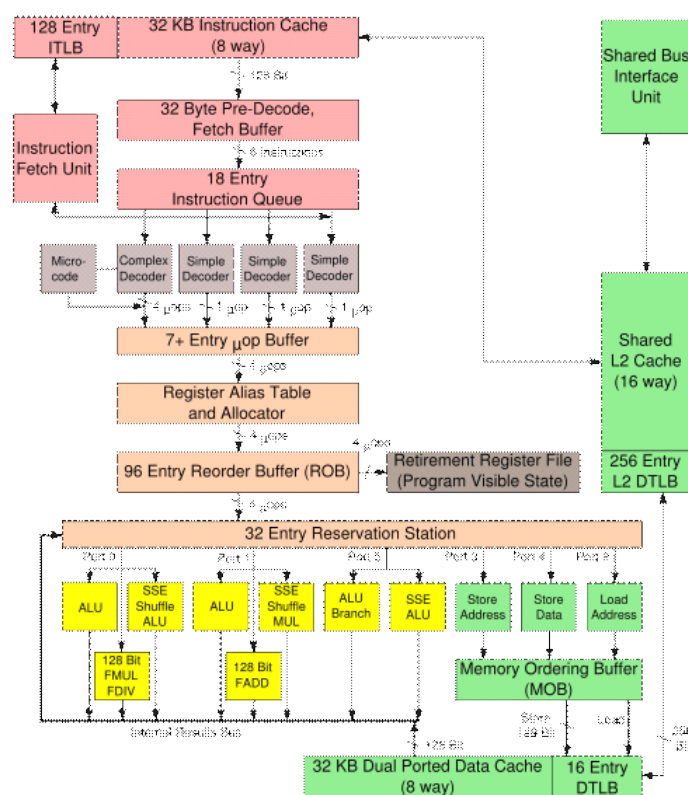
下面是酷睿 2 双核处理器的逻辑结构图：



Core 2 Duo 处理器逻辑结构图

其中：ROM = Read Only Memory 只读存储器、ROB = 、
 FPU = Float Point Unit 浮点运算单元、ALU= Arithmetic Logical Unit 算术逻辑部件、
 TLB = Translation Lookaside Buffer 转译后备缓冲器（转址旁路缓存/页表缓存）、
 LD = 、ST = 、D-TLB = Data-TLB 数据 TLB

酷睿 2 处理器是基于 Intel Core 微架构（microarchitecture）的，下面是其逻辑结构图：

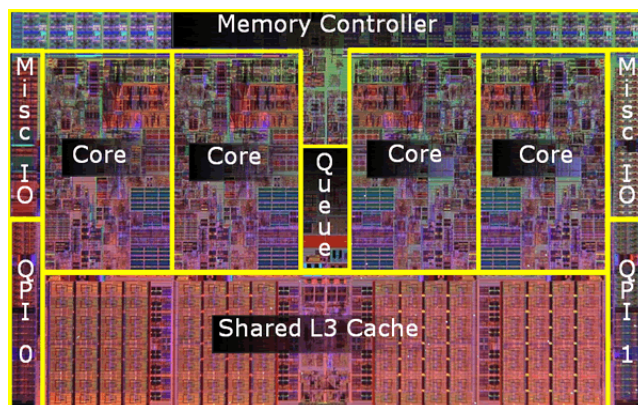


Intel Core 2 Architecture

Intel Core 微架构逻辑结构图

●Intel Core i7 与 Intel Nehalem 微架构

下面是酷睿 i7 四核处理器的平面结构图：



Core i7 处理器平面结构图

其中：Memory Controller = 内存控制器、Misc = 其他、Core = 核、IO = I/O = 输入/输出
Queue = 队列、QPI = QuickPath Interconnect = 快速通道互连、
Shared L3 Cache = 共享三级高速缓存

可见四核 Core i7 的基本构成：有超大容量的三级高速缓存、I/O 控制单元、内存控制器电路和两条 QPI 总线连接。不同级别的 Nehalem 处理器将会有不同条数的 QPI 连接，普通桌面处理器通常只有一条 QPI 连接，工作站以上级别的将会有多条 QPI 连接。

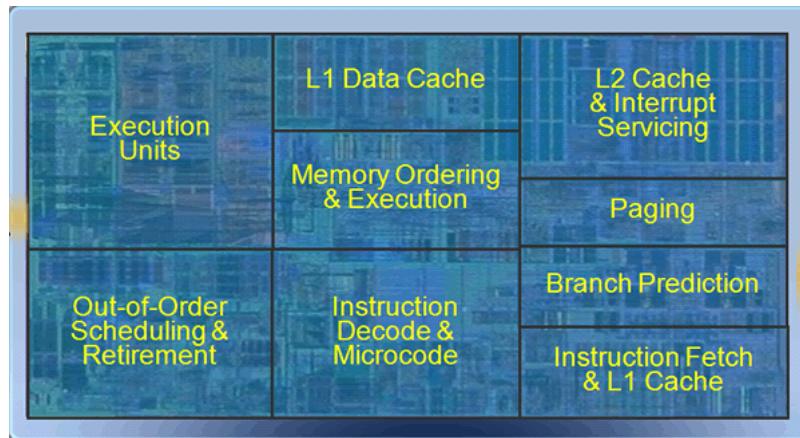
Core i7 处理器使用的是 Nehalem 微架构，而 Nehalem 采用了可扩展架构。主要是每个处理器单元均采用了组装模块化设计，组件包括：核心数量、SMT 功能、L3 缓存容量、QPI 连接数量、IMC 数量、内存类型、内存通道数量、整合 GPU、能耗和时钟频率等，这些组件均可自由组合，以满足多种性能需求，比如可以组合成双核心、四核心甚至八核心的处理器，而且组合多个 QPI（QuickPath Interconnect，快速通道互连）连接更可以满足多路服务器的需求。整合了 GPU 的 Nehalem 架构处理器 Havendale 可能在今年第四季度生产。



模块化设计的可伸缩 Nehalem 微架构

其中：IA = Intel Architecture 英特尔架构、
IGP = Integrated Graphics Processor 集成图形处理器、
QPI = QuickPath Interconnect 快速通道互连、

IMC = Integrated Memory Controller 集成内存控制



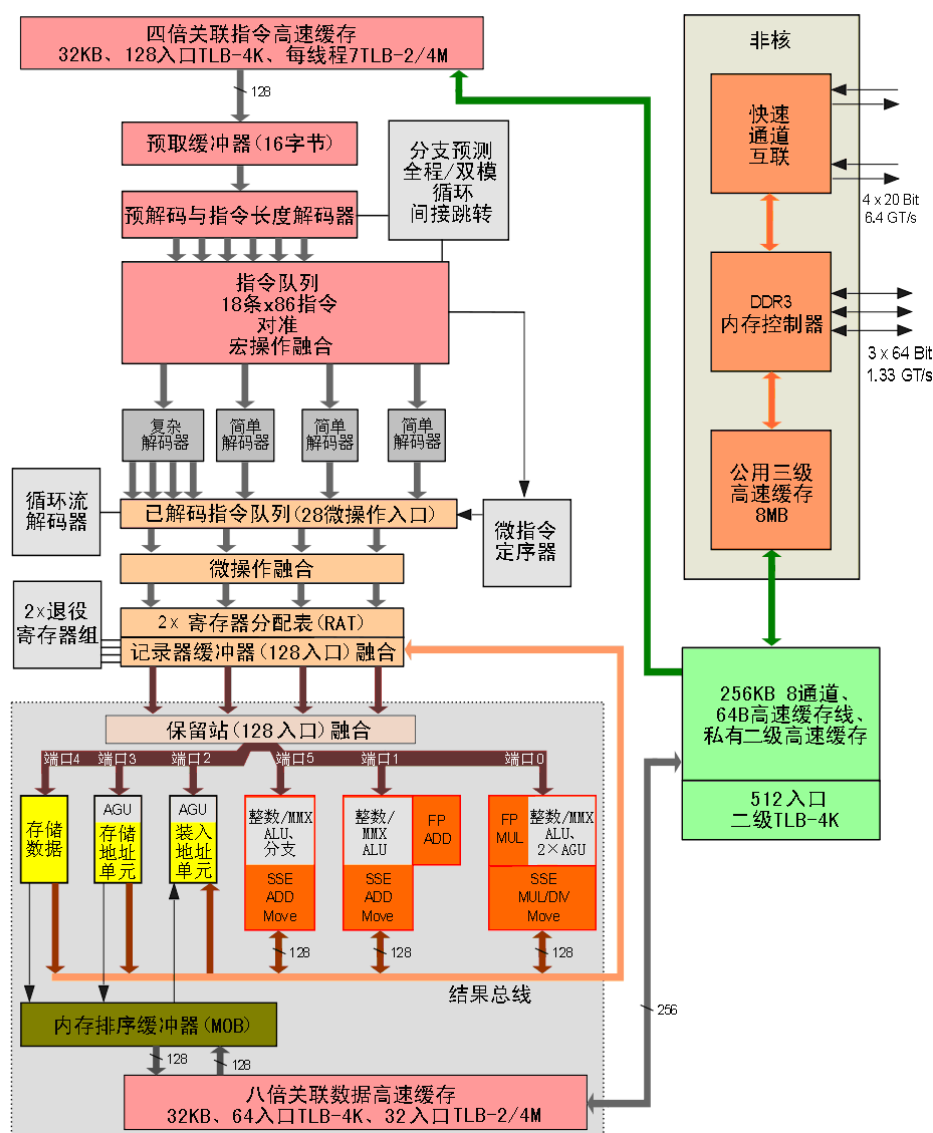
单个执行核心的基本构成

其中：Execution Units = 执行单元、L1 Data Cache = 一级数据高速缓存、
Memory Ordering & Execution = 内存排序与执行、
L2 Cache & Interrupt Servicing = 二级高速缓存与中断维护、Paging = 页面调度、
Out-of-Order Scheduling & Retirement = 乱序调度与退役、
Instruction Decode & Microcode = 指令解码与微码、
Branch Prediction = 分支预测、
Instruction Fetch & L1 Cache = 取指令与一级高速缓存

在每个执行核心中，包括乱序执行单元和完整的逻辑电路，有了这些才算是一个完整的高级处理核心，另外还有 L1、L2 缓存等电路，L1、L2 缓存的面积并不大，大概也就 1/4，像解码单元、分支预测逻辑判断单元、内存排序和页处理单元也占了不少面积。

Nehalem 的改进是全方位的，比如改善循环监测机制，Nehalem 的 LSD 能够缓冲 28 个微指令（Core 为 18），能处理更多的分支指令。Nehalem 中进一步添加了指令融合机制，支持目前所有 Core 中的宏指令技术，更具备有 Core 不支持的 64 位宏融合模式，在处理 64 位代码的时候，将会有明显的性能改善。Nehalem 还提升分支预测能力，搭载多级分支预测机制，提供了更高的性能表现。Nehalem 同时增强并行计算功能，在 Core 体系架构上，并行计算可以同时处理 96 个微指令，Nehalem 处理器将乱序窗口尺寸扩大了 33%，这样就能同时处理 128 个微指令。参见下图：

Intel Nehalem 微架构



Intel Nehalem 微架构逻辑结构图

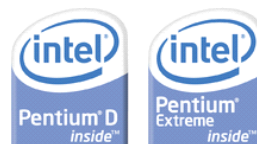
3. 发展历史

IBM 于 2001 年 10 月发布了世界上首款多核处理器——双核 RISC 处理器 Power 4，它将两个 64 位的 Power PC 处理器集成在一颗芯片上。Power 4 采用 180 纳米技术，主频为 1.1 和 1.3 GHz，两核心共享 1.41 MB 二级高速缓存（L2 cache）。

HP 于 2004 年 2 月也发布了其首款（64 位 RISC）双核处理器 PA-RISC8800，采用 130 纳米技术，主频为 0.8 和 1GHz，两核心共享 32 MB L2 cache。

Sun 于 2004 年 3 月发布了其首款（64 位 RISC）双核处理器 UltraSPARC IV，也采用 130 纳米技术，主频为 1.05 和 1.2GHz，两核心共享 16 MB L2 cache，支持超线程。2005 年 11 月 14 日 Sun 发布了代号为 Niagara（尼亚加拉瀑布）低能耗的 8 核 32 线程的 UltraSPARC T1 处理器，采用 90 纳米技术，主频为 1.2GHz。

Intel 于 2005 年 4 月 18 日（5 月 25 日）发布（推出）了世界上



首款基于 x86 架构的（64 位 CISC）双核处理器——Pentium D（奔腾 D）820、830 和 840，主频分别为 2.8、3.0 和 3.2 GHz。它将两颗 Pentium 4 Prescott 核心放在同一块芯片上（胶水 CPU），采用 90 纳米技术，两核各自拥有独立的 1 MB L2 cache。最初的 Pentium D 型号不支持超线程，后来推出的 EE（Extreme Edition，极致版）系列和 Presler 型号才支持超线程技术。

AMD 于 2005 年 4 月 21 日也发布了其首（64 位 CISC）多核处理器——双核版的 Opteron（皓龙）200 和 800 系列，采用 90 纳米技术，主频为 1.4 到 2.2GHz，两核各自拥有独立的 512 KB ~ 1 MB L2 cache。AMD 的 Opteron 处理器，是针对服务器的高档 CPU。



2006 年 1 月 5 日 Intel 推出用于笔记本的 Core Duo（酷睿双核）（32 位 CISC）处理器 T2300~T2600 和 L2300~T2500，主频分别为 1.666~2.166 GHz 和 1.5~1.833 GHz，共享 2 MB L2 cache。Core Duo 是 Intel 首次采用 65 纳米技术的处理器，也是全球首款低能耗的双核处理器。。



2006 年 7 月 23 日 Intel 推出用于服务器的代号为 Conroe XE 的 Core 2 Extreme（酷睿 2 极致）（64 位 CISC）处理器 X6800，也采用 65 纳米技术，主频为 2.933 GHz，共享 4 MB L2 cache。



2006 年 7 月 27 日 Intel 发布 Core 2 Duo（酷睿 2 双核）（64 位 CISC）处理器，采用 65 纳米技术，包括代号为 Allendale 的 E6300 和 E6400，主频分别为 1.866 和 2.133 GHz，共享 2 MB L2 cache；和代号为 Conroe 的 E6600 和 E6700，主频分别为 2.4 和 2.666 GHz，共享 4 MB L2 cache。



2007 年 1 月 7 日 Intel 发布了 Core 2 Quad（酷睿 2 四核）（64 位 CISC）处理器 Q6600，也采用 65 纳米技术，它将两颗 Core 2 Duo Conroe 核心放在同一块芯片上（胶水 CPU），主频为 2.4 GHz，两核各自拥有独立的 4 MB L2 cache。

2007 年 11 月 19 日 AMD 推出其代号为 Agena 的首款（64 位 CISC）四核（64 位 CISC）处理器 Phenom（羿龙）X4 9500 和 9600，采用 65 纳米技术，主频分别为 2.2 和 2.3 GHz，四个核各自拥有独立的 512 KB L2 cache，共享 2 MB L3 cache。与 Intel 的胶水四核 CPU Core 2 Quad 不同，AMD 的 Agena 为首款真正四核的 x86 处理器。



2008 年 3 月 27 日 AMD 推出代号为 Toliman 的三核处理器 AMD Phenom X3 8400 和 8600，采用 65 纳米技术，主频分别为 2.1 和 2.3 GHz，三个核各自拥有独立的 512 KB L2 cache，共享 2 MB L3 cache。Toliman 是屏蔽掉 AMD 四核处理器 Agena 的一个核心后的产物（废品再用）。

2008 年 9 月 15 日 Intel 推出了代号为 Dunnington 的基于 Intel Core 微架构的 64 位 6 核至强（Six-Core Xeon）处理器 L7455、X7460 和 E7450，采用 45 纳米技术，主频分别为 2.133、2.4 和 2.667 GHz，3 组双核每组各自拥有独立的 3MB L2 cache，共享 12 或 16 MB L3 cache。



2008 年 11 月 17 日 Intel 推出了面向高端应用的代号为 Bloomfield 的 64 位四核处理器 Core i7（酷睿 i7）920 和 965 Extreme Edition，基于 Intel Nehalem 微架构，采用 45 纳米技术，主频分别为 2.66 和 3.20 GHz，四个核各自拥有独立的 256 KB L2 cache，共享 8 MB L3 cache，支持超线程技术。Intel 计划于 2010 年推出代号为 Gulftown 的 32 纳米 Core i7，将拥有六个核。





Core i7-940 及其 LGA 1366 触点

Intel 计划于 2009 年 9 月推出代号为 Lynnfield 的 2~4 核的 64 位处理器 Core i5，也基于 Intel Nehalem 微架构，可视为 Core i7 的简化版，面向主流应用。

Intel 于 2009 年 6 月 18 日又宣布了 Core i3 处理器，也基于 Intel Nehalem 微架构，低能耗，面向笔记本和上网本等移动应用。

在 2009 年 2 月的 ISSCC 2009 国际固态电路会议上，Intel 首次宣布了 8 核心服务器处理器“Nehalem-EX”。2009 年 6 月 1 日，Intel 则正式公布了该处理器的详细资料，并将于 2009 年 6 月 26 日向公众详细介绍 Nehalem-EX。8 核 16 线程 Nehalem-EX 基于 45nm 工艺 Nehalem 架构，支持 QPI 总线互联，集成双芯片、四通道内存控制器，三级缓存容量 24MB，晶体管数量也达到了惊人的 23 亿个，热设计功耗 130W，接口为新的 LGA1567。该系列处理器预计将于今年年底或明年年初发布，针对多路服务器市场，替代 Penryn 微架构的六核“Dunnington”，成为 Intel 多路服务器领域的旗舰产品，目标甚至是成为 RISC 超级计算机的低成本替代者。

4. 并行性

多核中的并行性可以分成指令级并行和线程级并行两种：

●指令级并行(Instruction-Level Parallelism, ILP)

当指令之间不存在相关时，它们在流水线中是可以重叠起来并行执行的。这种指令序列中存在的潜在并行性称为指令级并行，是在机器指令级并行。通过指令级并行，处理器可以调整流水线指令重执行顺序，并将它们分解成微指令，能够处理某些在编译阶段无法知道的相关关系（如涉及内存引用时），并简化编译设计；能够允许一个流水线机器上编译的指令，在另一个流水线上也能有效运行。指令级并行能使处理器速度迅速提高。

●线程级并行 (Thread Level Parallelism, TLP)

线程级并行将处理器内部的并行由指令级上升到线程级，旨在通过线程级的并行来增加指令吞吐量，提高处理器的资源利用率。TLP 处理器的中心思想是：当某一个线程由于等待内存访问结构而空闲时，可以立刻导入其他的就绪线程来运行。处理器流水线就能够始终处于忙碌的状态，系统的处理能力提高了，吞吐量也相应提升。

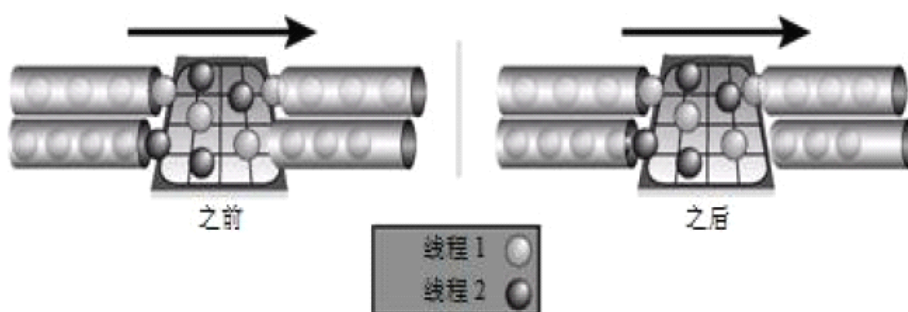
服务器可以通过每个单独的线程为某个客户服务（Web 服务器，数据库服务器）。单核超标量体系结构处理器不能完全实现 TLP，而多核架构则可以完全实现 TLP，解决了以上问题。现在业界普遍认为，TLP 将是下一代高性能处理器的主流体系结构技术。

5. 特点

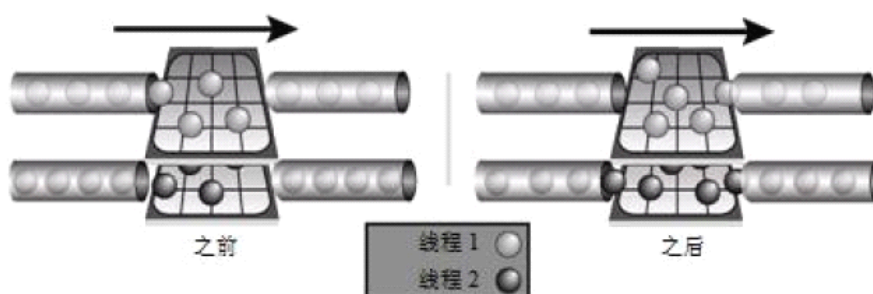
- **MIMD 架构**——多核处理器是一种特殊的多处理器，所有的处理器都在同一块芯片上，属于 MIMD 架构：不同的核执行不同的线程（多指令），在内存的不同部分操作（多数据）。多核是一个共享内存的多处理器：所有的核共享同一个内存。但可以有各自的一、二级高速缓存。
- **同步多线程（Simultaneously Multithreading, SMT）**——容许多个独立的线程在同一个核上同步执行，可以将多个线程组合到同一个核上。例如：如果一个线程正在等待一个浮点操作的结束，其他的线程可以使用整数单元。
- **实现多核架构难点**——内存共享（同步访问）、独立缓存（缓存一致性）、核之间的通信、与系统其他部分的通信。

6. 多核与超线程的比较

- **超线程技术与多核体系结构的区别：**
 - 超线程技术是通过延迟隐藏的方法提高了处理器的性能，本质上就是多个线程共享一个处理核。因此，采用超线程技术所获得的性能并不是真正意义上的并行，因此采用超线程技术多获得的性能提升，将会随着应用程序以及硬件平台的不同而参差不齐。
 - 多核处理器是将两个甚至更多的独立执行核嵌入到一个处理器内部。每个指令序列（线程），都具有一个完整的硬件执行环境，所以，各线程之间就实现了真正意义上的并行。



两个线程在支持超线程技术的单个处理器核上执行



两个线程在双核处理器上并行执行

- **超线程技术与多核体系结构的联系：**
 - 超线程技术：充分利用空闲 CPU 资源，在相同时间内完成更多工作。

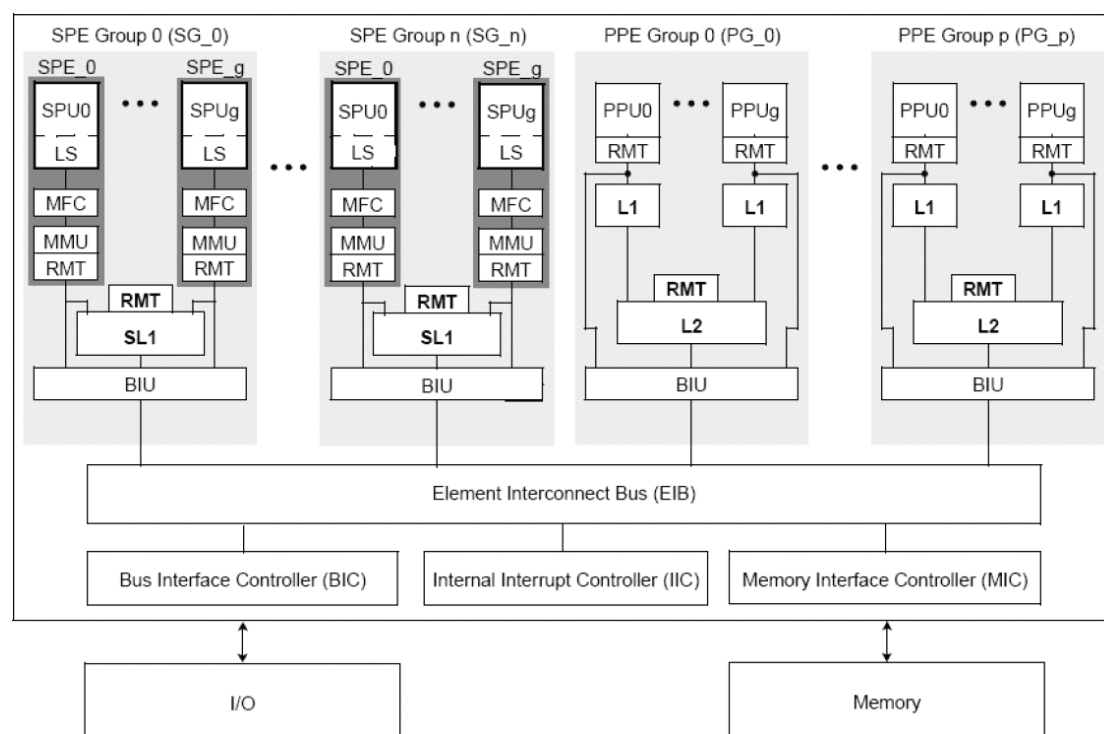
- 与多核技术相结合：给应用程序带来更大的优化空间，进而极大地提高系统的吞吐量。
- 单核与多核平台上的多线程技术对比：
 - 在面对多核体系结构开发应用程序的时候，只有有效地采用多线程技术，并仔细分配各线程的工作负载，才能达到最高性能。
 - 而单核平台上，多线程一般都当作是一种能够实现延迟隐藏的有效变程。
 - 单核与多核平台下的开发必须采用不同的设计思想：主要体现在存储缓存 (memory caching) 和线程优先级 (thread priority) 上。

7. 异构多处理器

除了 Intel、AMD 和 Sun 的对称多核处理器外，还有一类是异构多核处理器。典型代表是 2005 年由 IBM、索尼和东芝联合推出的 Cell 处理器。已经被应用到索尼 PS3 和微软 X360 游戏机中，是第一款投入实际商用的异构多核处理器。

Cell 处理器具有 1 个运行 Power 指令的主核 (PPE) 和 8 个 SIMD 辅助核 (SPE)，通过 1 条高速总线 (EIB) 进行连接。

PPE 包括 1 个 64 位、双发射、多线程、顺序执行的运算核心，可以同时提取 4 条指令和 2 个结果，负责运转操作系统和协同 SPE；SPE 是由 1 个辅助处理单元 (SPU) 和 1 个内存流量控制器 (MFC) 的标准设计组成。SPU 是 1 个带有 SIMD 支持和 256KB 局部存储器的 128 位计算引擎；MFC 有 1 个 DMA 联合 MMU 的控制器，从专属的局部存储器直接进行指令和数据操作，同时处理其他的 SPU 以及 PPU 同步运转，而且可以独立运行，当 SPU 运行的时候，并行的翻译地址和进行 DMA 传输。EIB 连接 PPE、SPE 和外部 I/O，由 1 个地址母线和 4 个 128 位的数据环线构成，2 个顺时针运转，另外 2 个逆时针运转，每个环线最大可以允许 3 个并行发生的数据传输。参见下图：



Cell 架构图

Cell 架构由于具有 8 个可以并行进行 SIMD 运算的 SPE, 并且通过 EIB 和高速内存提供足够的数据通路, 从而加速了浮点运算、矩阵运算、科学计算、多媒体处理等数据处理的能力。Cell 的 3.2GHz 单处理器的单精度浮点运算的峰值可以达到 201GFLOPS, 而 Intel 同频率的奔腾 4 处理器的峰值仅为 25.6GFLOPS。但是这种体系结构的巨大改变, 一方面要求操作系统必须提供足够的支持, 另外一个方面编译器和编程模式也发生了巨大的改变。Cell 的编程规范中要求程序员对每个核进行单独的编程, PPE 和 SPE 是不同的编程模式, 通过特殊的编译器和链接器得到二进制代码, 这给软件开发带来了新的压力和挑战。

8. 操作系统对多核处理器的支持

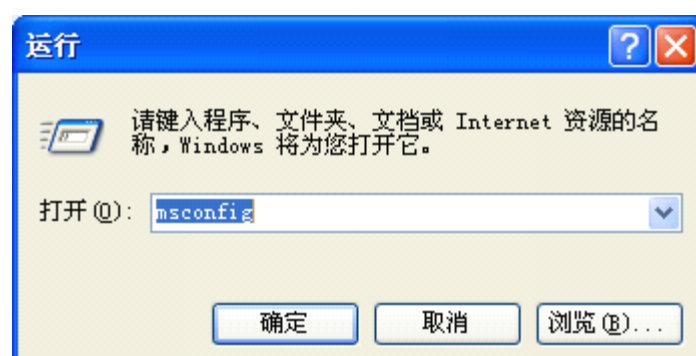
主要体现在调度与中断上:

- 对任务的分配进行优化——使同一应用程序的任务尽量在一个核上执行。
- 对任务的共享数据优化——由于 CMP 体系结构共享二级缓存, 可以考虑改变任务在内存中的数据分布, 使任务在执行时尽量增加二级缓存的命中率。
- 对任务的负载均衡优化——当任务在调度时, 出现了负载不均衡, 考虑将较忙处理器中与其他任务最不相关的任务迁移, 以达到数据的冲突量小。

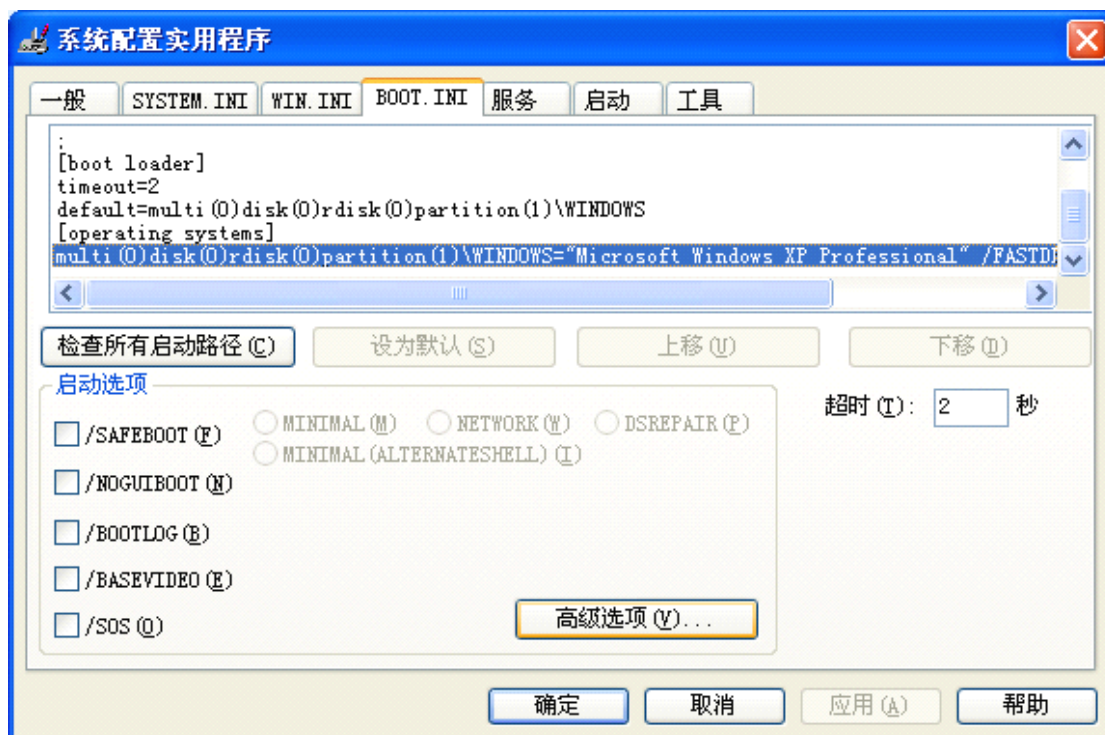
9. 多核设置

可以使用主板 BIOS 和操作系统来进行多核 CPU 的核心使用数的设置:

- BIOS 设置——一些主板 BIOS 提供了 CPU 核心设置项, 这样方便在 BIOS 中关闭其中若干核心或在多核与单核间切换。方法是进入 BIOS 设置, 找到“CPU Configuration”选项, 然后在下面的 CPU Core i 项中, 将其中的若干项设置为“Disabled”即可, 保存退出后, 这样就屏蔽了其中若干个核心, 或者将双核 CPU 变为一个同频率的单核产品。
- Windows 系统下设置——在 Windows XP/2003 中, 使用 MSCONFIG 工具: 选中“开始\运行”, 在弹出的“运行”对话框的“打开”组合框栏中键入 MSCONFIG (大小写均可) 后按“确定”钮, 参见下图:



在弹出的“系统配置实用程序”对话框中选择“BOOT.INI”选项卡, 再选择对应窗格中下部的“高级选项”按钮, 参见下图:



选中其中的处理器数量复选框“/NUMPROC”，在其右边的下拉式列表中，设置为希望使用的 CPU 核心数，参见下图。



比如使用如果是 Intel 四核处理器 Core 2 Quad，可以只开 2 个核心冒充 Intel 双核处理器 Core 2 Due。如果你使用的是双核处理器的话，也可以只开 1 个核心来冒充单核 CPU。

1.6 并行计算

计算科学是理论科学、实验科学之外的第三种研究手段。计算科学与传统的两种科学，即理论科学和实验科学，并立被认为是人类认识自然的三大支柱，他们彼此相辅相成地推动科学发展与社会进步。在许多情况下，或者是理论模型复杂甚至理论尚未建立，或者实验费用昂贵甚至无法进行时，计算就成了求解问题的唯一或主要的手段。

对未知世界的探索为计算技术带来了巨大的挑战，并行计算是解决计算挑战的必由之

路。随着通用集群技术和多核技术的发展，并行计算技术正逐步走向普及。

并行计算的研究涵盖并行计算机体系结构、并行算法和并行编程三个方面，其挑战在于软件和应用。

并行计算(Parallel Computing)是高性能计算(High-end Parallel Computing)又叫高端计算(Highend Computing)或超级计算(Super Computing)的核心技术。

1.6.1 为什么要做并行计算

并行计算的出现是为了适应各种应用对计算机性能和速度的不断增长要求。

1. 应用需求

人类对计算及性能的要求是无止境的，从系统的角度：集成系统资源，以满足不断增长的对性能和功能的要求；从应用的角度：适当分解应用，以实现更大规模或更细致的计算更细致的计算。

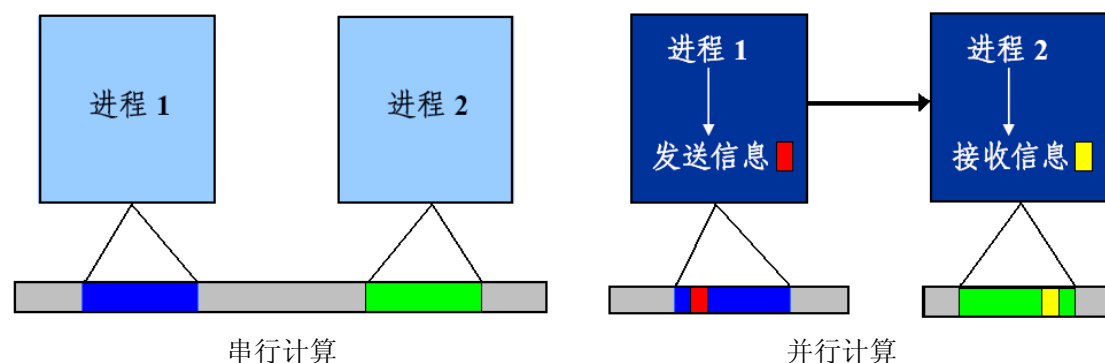
2. 计算速度要求

科学和工程问题的数值模拟与仿真，具有计算密集、数据密集、网络密集、及这三种混合的特点。要求在合理的时限内完成计算任务，如秒级的制造业、分钟级的短时天气预报(当天)、小时级的中期天气预报(3~10 日)、尽可能快的长期天气预报(气候)、可计算的湍流模拟。

1.6.2 什么是并行计算

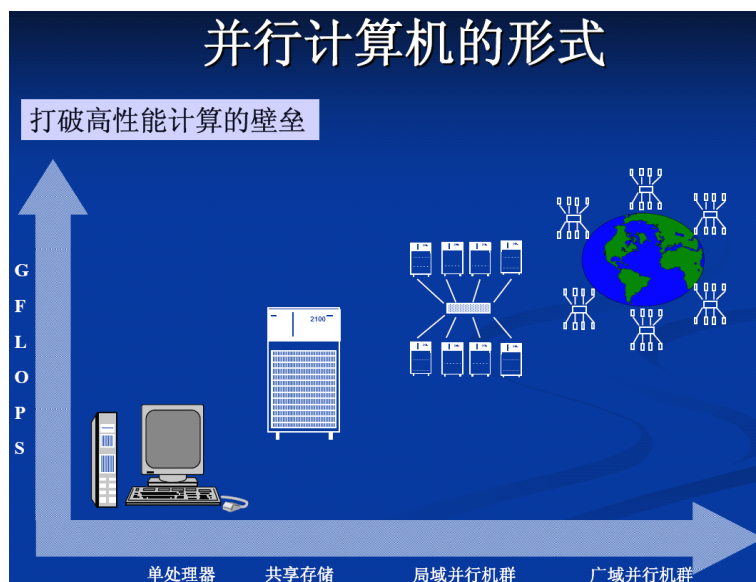
并行计算(parallel computing)是由运行在多个部件上的小任务合作，来求解一个规模很大的复杂计算问题的一种方法。

传统的串行计算，分为“指令”和“数据”两个部分，并在程序执行时“独立地申请和占有”内存空间，且所有计算均局限于该内存空间。并行计算将进程相对独立的分配于不同的节点上，由各自独立的操作系统调度；享有独立的 CPU 和内存资源（内存可以共享）；进程间相互信息交换通过消息传递。参见下图：

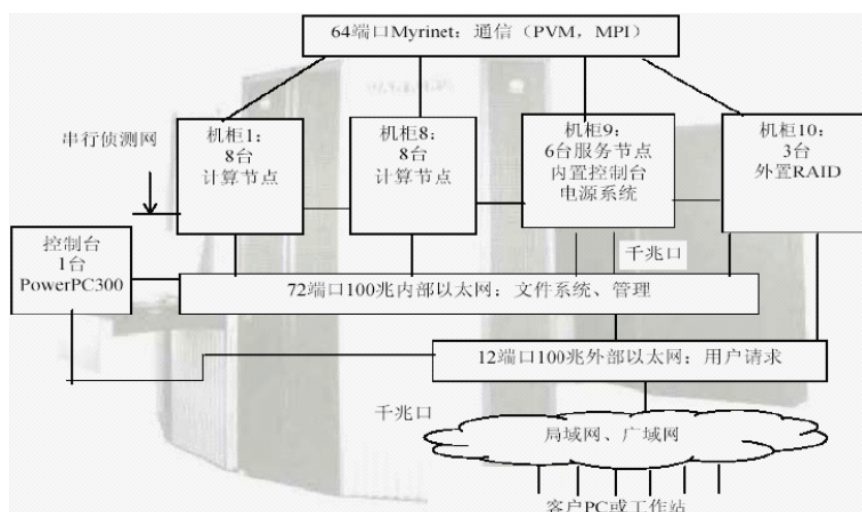


1. 并行计算机形式

并行计算机包括芯片级的多核单处理器、板级和机架式的共享存储多处理器、网络级的局域和广域并行机群（cluster）等形式，参见下图：



其中：GFLOPS = Giga Floating point Operations Per Second 每秒十亿次浮点运算
用于度量计算机的性能



多机柜局域网式并行计算机

2. 并行计算功能

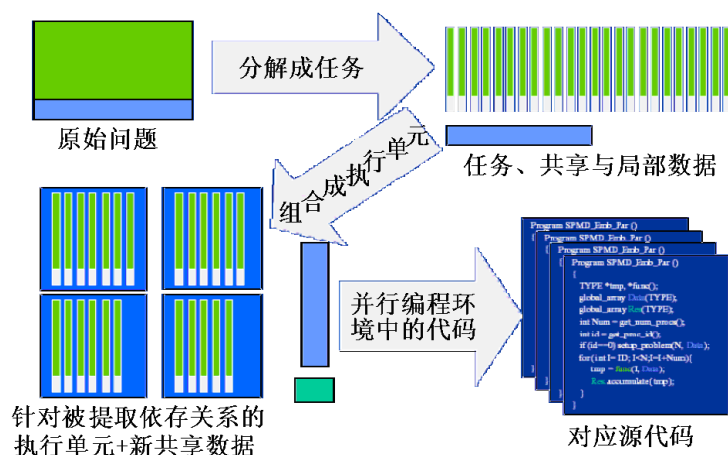
并行计算的功能主要有：

- 降低单个问题求解的时间。
- 增加问题求解规模、提高问题求解精度。
- (多机同时执行多个串行程序)容错、更高的可用性、提高吞吐率。

3. 并行化方法——分而治之

并行化的主要方法是分而治之：

- 任务并行——根据问题的求解过程，把任务分成若干子任务(任务级并行或功能并行)。
- 数据并行——根据处理数据的方式，形成多个相对独立的数据区，由不同的处理器分别处理。



1.6.3 并行计算机

并行计算机 (parallel computer) 由一组处理单元组成，这组处理单元通过相互之间的通信与协作，以更快的速度共同完成一项大规模的计算任务。并行计算机的两个最主要的组成部分是计算节点和节点间的通信与协作机制。

1. 出现背景

1960 年代初期，晶体管以及磁芯存储器的出现，处理单元变得越来越小，存储器也更加小巧和廉价。出现规模不大的共享存储多处理器系统，即大型主机 (Mainframe)。

1960 年代末期，同一个处理器开始设置多个功能相同的功能单元，流水线技术也出现了，在处理器内部的应用大大提高了并行计算机系统的性能。

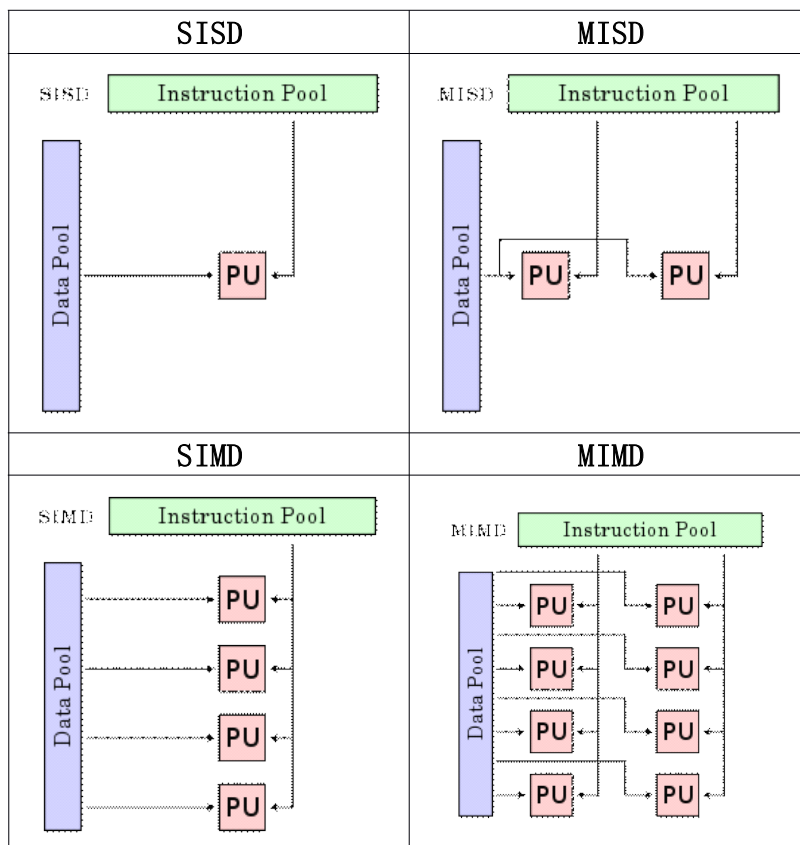
2. 弗林分类 (Flynn's taxonomy)

1966 年 Michael J. Flynn 根据指令流和数据流的不同组织方式，把计算机系统的结构分为以下四类：

- SISD (Single Instruction stream Single Data stream, 单指令流单数据流)
- SIMD (Single Instruction stream Multiple Data stream, 单指令流多数据流)
- MISD (Multiple Instruction stream Single Data stream, 多指令流单数据流)
- MIMD (Multiple Instruction stream Multiple Data stream, 多指令流多数据流)

弗林分类

	单指令	多指令
单数据	SISD	MISD
多数据	SIMD	MIMD



弗林分类的图示

其中：PU = Processing Unit 处理单元

SISD 是普通的顺序处理串行机（如 Intel 486）；SIMD 是一种特殊的并行机制（如专用的向量机和 Intel 的 MMX[MultiMedia eXtension，多媒体扩展]和 SSE[Streaming SIMD Extensions，流 SIMD 扩展]指令集）；MISD 型的计算机是根本不可能存在的，但也有人认为流水线可以视为 MISD 结构；而 MIMD 则是典型的并行计算机（如 Intel Celeron 2 和 Core i7、AMD 的 Opteron 和 Phenom 等多核处理器，各种巨型机，包括中国的巨型机：国防科技大学的银河、国家智能计算机研究开发中心的曙光、国家并行计算机工程技术研究中心等的神威、联想集团的深腾等）。

1.6.4 并行计算机体系结构

并行计算机与超级计算机技术，为多核计算机的出现奠定了基础，而集成电路技术是多核芯片得以实现的物理条件。

1. 分类

并行计算机系统结构可以分成如下五类：

1. 分布式存储器的 SIMD 处理机——含有多个同样结构的处理单元（PE，Processing Element），通过寻径网络以一定方式互相连接。每个 PE 有各自的本地存储器（LM，Local Memory）。在阵列控制部件的统一指挥下，实现并行操作。

程序和数据通过主机装入控制存储器。由于通过控制部件的是单指令流，所以指令的执行顺序还是和单处理机一样，基本上是串行处理。指令送到控制部件进行译码。如果是标量指令，则直接由标量处理机执行。如果是向量指令，则阵列控制部件通过广播总线将它广播到所有 PE 并行执行。划分后的数据集通过向量数据总线分布到所有 PE 的本地存储器 LM。

PE 通过数据寻径网络互连。数据寻径网络执行 PE 间的通信。控制部件通过执行程序来控制数据寻径网络。PE 的同步由控制部件的硬件实现。也就是说，所有 PE 在同一个周期执行同一条指令。但是可以用屏蔽逻辑来决定任何一个 PE 在给定的指令周期执行或不执行指令。

2. 向量超级计算机（共享式存储器 SIMD）——集中设置存储器，共享的多个并行存储器通过对准网络与各处理单元 PE 相连。在处理单元数目不太大的情况下很理想。这是集中设置存储器的一种方案。共享的多个并行存储器通过对准网络与各处理单元 PE 相连。存储模块的数目等于或略大于处理单元的数目。为了减少存储器访问冲突，存储器模块之间必须合理分配数据。通过灵活高速的对准网络，使存储器与处理单元之间的数据传送在大多数向量运算中都能以存储器的最高频率进行。这种共享存储器模型在处理单元数目不太大的情况下是很理想的。存储器模块数与 PE 数互质可以实现无冲突并行访问存储器。

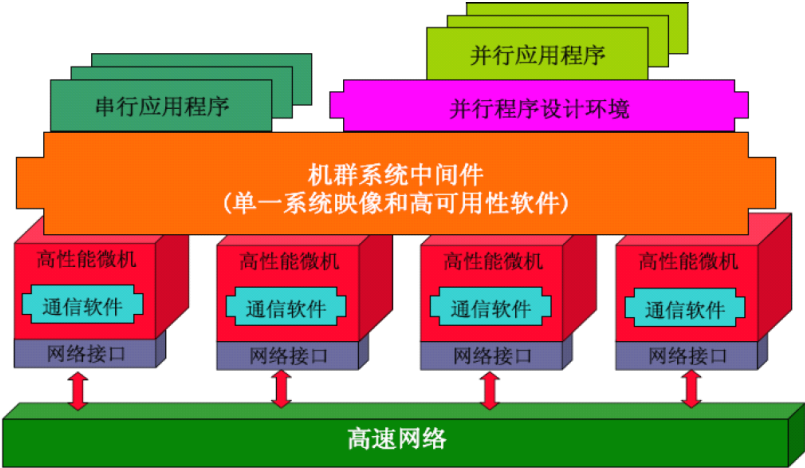
3. 对称多处理器（SMP，Symmetric Multiple Processor）——一个计算机上汇集了一组处理器，各处理器之间共享内存子系统以及总线结构。它是相对非对称多处理技术而言的、应用十分广泛的并行技术。

在这种架构中，一台电脑不再由单个 CPU 组成，而同时由多个处理器运行操作系统的单一复本，并共享内存和一台计算机的其他资源。虽然同时使用多个 CPU，但是从管理的角度来看，它们的表现就像一台单机一样。系统将任务队列对称地分布于多个 CPU 之上，从而极大地提高了整个系统的数据处理能力。所有的处理器都可以平等地访问内存、I/O 和外部中断。在对称多处理系统中，系统资源被系统中所有 CPU 共享，工作负载能够均匀地分配到所有可用处理器之上。

4. 并行向量处理机（PVP，Parallel Vector Processor）——在并行向量处理机中有少量专门定制的向量处理器，每个向量处理器有很高的处理能力。并行向量处理机，通过向量处理和多个向量处理器并行处理，两条途径来提高处理能力。并行向量处理机通常使用定制的高带宽网络将向量处理器连向共享存储器模块。存储器可以以很高的速度向处理器提供数据。这种机器通常不使用高速缓存，而是使用大量的向量寄存器和指令缓冲器。

5. 集群计算机（computers cluster）——集群计算机是随着微处理器和网络技术的进步而逐渐发展起来的，它主要用来解决大型计算问题。集群计算机是一种并行或分布式处理系统，由很多连接在一起的独立计算机组成，像一个单独集成的计算机资源一样协同工作。计算机节点可以是一个单处理器或多处理器的系统，拥有内存、IO 设备和操作系统。一个集群一般是指连接在一起的两个或多个计算机（节点）。节

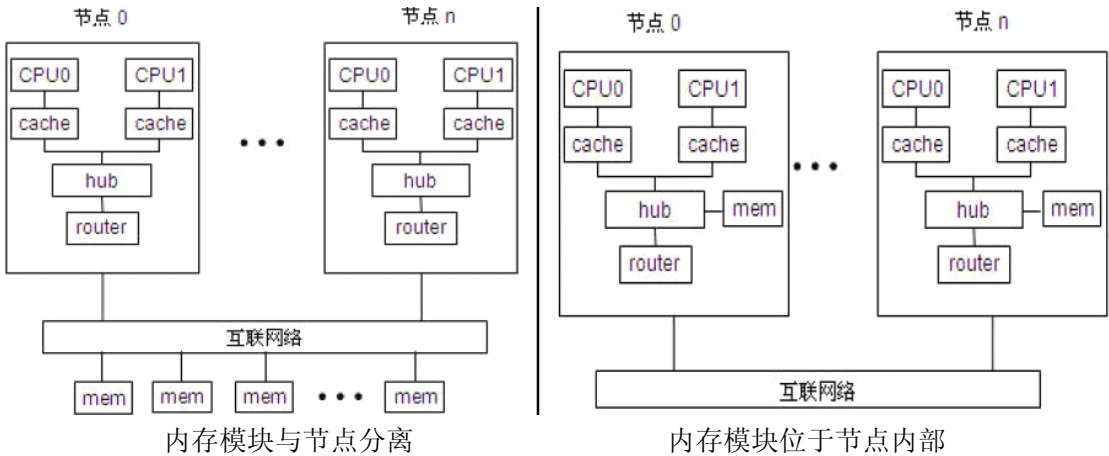
点可以是在一起的，也可以是物理上分散而通过网络连结在一起的（参见下图）。一个连接在一起的计算机集群对于用户和应用程序来说像一个单一的系统，这样的系统可以提供一种价格合理的且可获得所需性能和快速而可靠的服务的解决方案，而在以往只能通过更昂贵的专用共享内存系统来达到。集群计算机实际上就是典型的“云计算”设备。



集群系统的体系结构

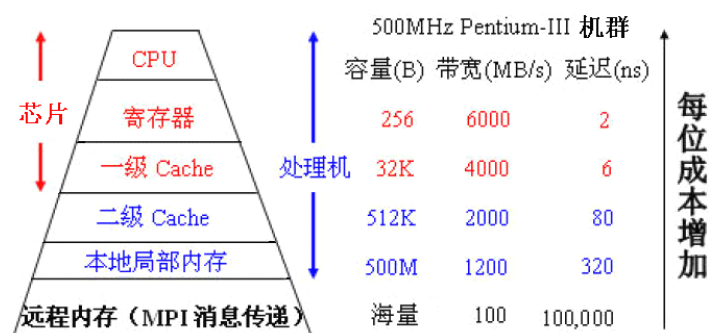
2. 并行计算机组成

组成并行计算机主要由节点（node）、互连网络（interconnect network）和内存（memory）等部分组成。参见下图：



3. 多级存储体系结构

为了解决内存墙（memory wall）性能瓶颈问题，现代计算机一般采用多级存储体系结构。其中最快的是 CPU 中的寄存器、其次是位于处理器内部的 cache（高速缓存）、然后是本地局部内存、最后是机群内远程内存，单位价格是逐级降低，容量是逐级增加。参见下图：



位于多核处理器内的高速缓存，一般分为两级：位于核内的小型一级高速缓存（L1 cache）和位于核外并由所有核共享的二级高速缓存（L2 cache）。在一些 4 核、8 核或更多核的处理器中，将核进行分组，组内共享二级高速缓存，组外共享三级高速缓存（L3 cache）。

L1 cache 连接 CPU 寄存器和 L2 cache，负责缓存 L2 cache 中的数据到寄存器中。类似地，L2 cache 连接 L1 cache 和 L3 cache，负责缓存 L3 cache 中的数据到 L2 cache 中。

4. 高速缓存映射策略

cache 映射策略（mapping strategy）是指内存块和 cache 线之间建立的相互映射关系，包括如下几种：

- 直接映射策略（direct mapping strategy）——每个内存块只能被唯一的映射到一条 cache 线中。
- K-路组关联映射策略（K-way set association mapping strategy）——Cache 被分解为 V 个组，每个组由 K 条 cache 线组成，内存块按直接映射策略映射到某个组，但在该组中，内存块可以被映射到任意一条 cache 线。
- 全关联映射策略（full association mapping strategy）——内存块可以被映射到 cache 中的任意一条 cache 线。

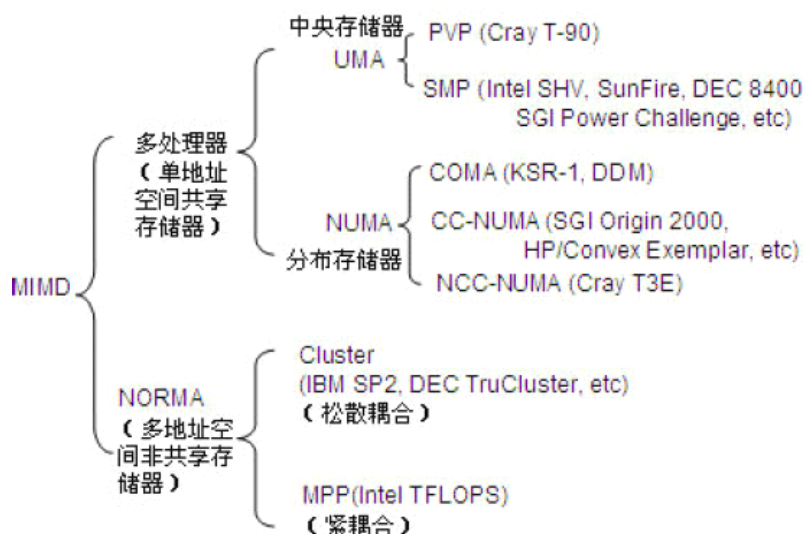
5. 并行计算机访存模型

并行计算机访存模型（Memory Access Model）有如下四种：

- UMA（Uniform Memory Access，统一访存）模型：
 - 物理存储器被所有节点共享；
 - 所有节点访问任意存储单元的时间相同；
 - 发生访存竞争时，仲裁策略平等对待每个节点，即每个节点机会均等；
 - 各节点的 CPU 可带有局部私有高速缓存；
 - 外围 I/O 设备也可以共享，且每个节点有平等的访问权利。
- NUMA（Non-Uniform Memory Access，非统一访存）模型：
 - 物理存储器被所有节点共享，任意节点可以直接访问任意内存模块；
 - 节点访问内存模块的速度不同，访问本地存储模块的速度一般是访问其他节点内存模块的 3 倍以上；
 - 发生访存竞争时，仲裁策略对节点可能是不等价的；
 - 各节点的 CPU 可带有局部私有高速缓存（cache）；
 - 外围 I/O 设备也可以共享，但对各节点是不等价的。

- COMA (Cache-Only Memory Access, 只高速缓存访存) 模型:
 - 各处理器节点中没有存储层次结构, 全部高速缓存组成了全局地址空间;
 - 利用分布的高速缓存目录 D 进行远程高速缓存的访问;
 - COMA 中的高速缓存容量一般都大于 2 级高速缓存容量;
 - 使用 COMA 时, 数据开始时可以任意分配, 因为在运行时它最终会被迁移到要用到它的地方;
- NORMA (No-Remote Memory Access, 非远程访存) 模型:
 - 所有存储器都是私有的;
 - 绝大多数 NORMA 都不支持远程存储器的访问;
 - 在 DSM 中, NORMA 就消失了。

下图是并行计算机系统的不同访存模型分类: (括号中为机器例)



1.6.5 并行计算模型

并行计算模型可以分成 SIMD 同步并行计算模型和 MIMD 异步并行计算模型两大类, 而且在这两种模型中主要是 PRAM (Parallel Random Access Machine, 并行随机存储器) 模型。

1. SIMD 同步并行计算模型

单指令流多数据流的同步并行计算模型可以细分成如下两类:

- 共享存储的 SIMD 模型 (PRAM 模型)
 - 又可以分成如下三个子类:
 - PRAM-EREW (Exclusive--Read and Exclusive--Write), 不允许同时读和同时写;
 - PRAM-CREW (Concurrent--Read and Exclusive--Write), 允许同时读但不允许同时写;
 - PRAM-CRCW (Concurrent--Read and Concurrent--Write), 允许同时读和同时写。

优点:

- 适合于并行算法的表达、分析和比较；
 - 使用简单，很多诸如处理器间通信、存储管理和进程同步等并行计算机的低级细节均隐含于模型中；
 - 易于设计算法和稍加修改便可运行在不同的并行计算机上；
 - 且有可能加入一些诸如同步和通信等需要考虑的方面。
- 分布存储的 SIMD 模型（SIMD 互连网络模型）
- 又可以分成如下九个子类：
- SIMD-LC——采用一维线性连接的 SIMD 模型；
 - SIMD-MC——采用网孔连接的 SIMD 模型；
 - SIMD-TC——采用树形连接的 SIMD 模型；
 - SIMD-MT——采用树网连接的 SIMD 模型；
 - SIMD-CC——用立方连接的 SIMDSIMD 模型；
 - SIMD-CCC——采用立方环连接的 SIMD 模型；
 - SIMD-SE——采用洗牌交换连接的 SIMD 模型；
 - SIMD-BF——采用蝶形连接的 SIMD 模型；
 - SIMD-MIN——采用多级互连网络连接的 SIMD 模型。

2. MIMD 异步并行计算模型

多指令流多数据流的并行计算模型都属于 APRAM（Asynchronism PRAM，异步 PRAM）模型。

- APRAM 特点：
- 每个处理器都有其本地存储器、局部时钟和局部程序；
 - 处理器间的通信经过共享全局存储器；
 - 无全局时钟，各处理器异步地独立执行各自的指令；
 - 处理器任何时间依赖关系需明确地在各处理器的程序中加入同步（路）障（Synchronization Barrier）；
 - 一条指令可在非确定但有限的时间内完成。
- APRAM 模型的四类指令：
- 全局读——将全局存储单元中的内容读入本地存储器单元中；
 - 局部操作——对本地存储器中的数执行操作，其结果存入本地存储器中；
 - 全局写——将本地存储器单元中的内容写入全本地存储器单元中；
 - 同步——同步是计算中的一个逻辑点，在该点各处理器均需等待特别的处理器到达后才能继续执行其局部程序。

异步 PRAM 模型可以进一步分成如下三种：

● BSP 模型

BSP（Bulk Synchronous Parallel，大块同步并行）模型作为计算机语言和体系结构之间的桥梁，由以下三个参数描述分布存储的并行计算机模型：

- 处理器/存储器模块（PMM）；
- PMM 模块之间点到点信息传递的路由器；
- 执行以时间间隔 L 为周期的路障同步器。

BSP 的特点：

- 将 PMM 和路由器分开，强调了计算任务和通信任务的分开，而路由器仅施行点到点的消息传递，不提供组合、复制或广播等功能，这样做既掩盖了具体的互联网络拓扑，又简化了通信协议；
- 采用路障方式的以硬件实现的全局同步是在可控的粗粒度级，从而提供了执行紧耦合同步式并行算法的有效方式，而程序员并无过分的负担；
- 在分析 BSP 模型的性能时，假定局部操作可在一个时间步内完成，而在每一超级步中，一个 PMM 至多发送或接受 h 条消息（ h -relation）。
- LogP 模型——一种分布存储的、点到点通信的多处理机模型，其中通信网络由一组参数来描述，但它并不涉及到具体的网络结构，也不假定算法一定要用显式的消息传递操作进行描述。LogP 机由 L 、 o 、 g 和 P 四个参数描述：（这也是该模型名称的来历）
 - L （Latency，等待/潜伏时间）——通信介质的等待时间；
 - o （overhead，开销）——发送和接收消息的开销；
 - g （gap，间隙）——发送/接收操作之间所需的间隙；
 - P （Processing units，处理模块）——处理模块的数量。每次在每个机器上的本地操作花费同样的（单位）时间，该时间被称为处理器周期。
 前三个参数都用处理器周期来度量。
- C^3 （Computation、Communication、Congestion，计算、通信、拥塞）模型——一个与体系结构无关的粗粒度的并行计算模型，旨在能反映计算复杂度、通信模式和通信期间潜在的拥挤等因素，对粗粒度网络算法的影响。

1.6.6 并行计算性能评测

并行计算性能一般通过并行程序执行的时间来评测。行程序执行时间等于从并行程序开始执行到所有进程执行完毕，墙上时钟走过的时间，也称为墙上时间（wall clock time）。对各个进程，墙上时间可进一步分解为计算 CPU 时间、通信 CPU 时间、同步开销时间、同步导致的进程空闲时间。

并行程序性能评价方法有：

- 浮点峰值性能与实际浮点性能；
- 数值效率和并行效率。

1.6.7 并行计算的挑战

并行计算的挑战面临众多的挑战，包括如何协调、如何控制、如何监视、并行编程、采用多线程解决同一个问题和在并行线程之间的通信与同步机制等。

1.7 并行编程

并行编程涉及并行软件程序员的工作、并行程序设计方法、并行程序设计模型、并行编程标准和并行算法描述等方面。

1.7.1 并行编程环境

比较流行的并行编程环境主要有三类：消息传递、共享存储和数据并行，参见下表：

特征	消息传递	共享存储	数据并行
典型代表	MPI、PVM	OpenMP	HPF
可移植性	所有主流并行计算机	SMP、DSM	SMP、DSM、MPP
并行粒度	进程级大粒度	线程级细粒度	进程级细粒度
并行操作方式	异步	异步	松散同步
数据存储模式	分布式存储	共享存储	共享存储
数据分配方式	显式	隐式	半隐式
学习入门难度	较难	容易	较易
可扩展性	好	较差	一般

其中：MPI = Message Passing Interface 消息传递接口、PVM = Parallel Virtual Machine 并行虚拟机、OpenMP = Open Multi-Processing 开放多处理、HPF = High Performance Fortran 高性能 Fortran、SMP = Symmetric Multiple Processor 对称多处理器、DSM = Distributed Shared Memory 分布式共享内存、MPP = Massively Parallel Processing 大规模并行处理。

1.7.2 编程语言与编译器

在科学计算领域已有三项成功的并行编程技术：自动并行化、数据并行语言（HPF）和共享存储并行编程接口（OpenMP）。

1. 自动并行化

自动并行化一直是人们的奋斗目标。1980 年代中期，基于依赖分析的自动向量化工具已经成熟，可以帮助程序员将 Fortran 语言代码移植到向量计算机上进行并行计算。后来的研究转向共享存储的 MIMD 和分布式存储结构的自动并行化，碰到很大的困难。现在，研究重点又逐步转向基于语言的策略研究，即从用户那里获得更多信息，同时利用自动化并行技术来减轻程序设计的负担。

2. HPF：数据并行编程

HPF（High Performance Fortran，高性能 Fortran）是 Fortran 90 的扩展版（在 Fortran 95 和 Fortran-2008 包含了对 HPF 的支持），提供了注释形式的指令来扩展变量类型的说明，能够对数组的数据布局进行相当详细的控制。

HPF 由 HPFF（HPF Forum，HPF 论坛，由美国 Rice University 的 Ken Kennedy 领导，网址为：<http://hpff.rice.edu/>）公布，1993 年春夏推出 1.0 版 HPF-1、1997 年 1 月推出 2.0 版 HPF-2。

由于具有 HPF 功能的 Fortran 编译器的实现和使用都遇到了不少困难，现在大多数厂商开始转向基于 OpenMP 的并行处理。

3. OpenMP: 共享存储并行编程

OpenMP (Open Multi-Processing, 开放多处理)是一种支持多平台共享内存多处理编程的 C、C++和 Fortran 语言 API。它支持许多体系结构, 包括 Unix 和 Microsoft Windows。它包含一组编译器指令、库程序、和影响运行时行为的环境变量。支持 OpenMP 的编译器包括 Sun Compiler、GNU Compiler、Intel Compiler 和 Microsoft Visual C++等。

OpenMP 的 API 规范由 OpenMP ARB (Architecture Review Board, 架构评审委员会, 网址为: <http://openmp.org/wp/>) 公布。1997 年 10 月推出 OpenMP for Fortran 1.0、1998 年 10 月推出 OpenMP for C/C++ 1.0、2000 年推出 OpenMP for Fortran 2.0、2002 年 10 月推出 OpenMP for C/C++ 2.0、2005 年推出 2.5 (for C/C++/Fortran)。2008 年 5 月推出 3.0。

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各种专用的 pragma 指令 (directive, 指示/命令) 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 (串行) 程序, 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。同时, 使用 OpenMP 也提供了更强的灵活性, 可以较容易的适应不同的并行系统配置。线程粒度和负载平衡等是传统多线程程序设计中的难题, 但在 OpenMP 中, OpenMP 库从程序员手中接管了部分这两方面的工作。

但是, 作为高层抽象, OpenMP 并不适合需要复杂的线程间同步和互斥的场合。OpenMP 的另一个缺点是不能在非共享内存系统 (如计算机集群) 上使用。在这样的系统上, MPI 使用较多。

1.7.3 并行软件程序员的工作

并行编程涉及不同的层次:

- 指令层: 非常细的粒度;
- 数据层: 细粒度;
- 控制层: 中粒度;
- 任务层: 大粒度。

前两层大都由硬件和编译器负责处理, 程序员通常处理后两层的并行。

1.7.4 并行程序设计

1. 方法

并行程序设计可以分成如下两类:

- 隐式并行程序设计: (即并行自动化)
 - 常用传统的语言编程成顺序源编码, 把“并行”交给编译器实现自动并行。
 - 程序的自动并行化是一个理想目标, 存在难以克服的困难。

- 语言容易，编译器难。
- 显式并行程序设计：
 - 在用户程序中出现“并行”的调度语句。
 - 显式的并行程序开发则是解决并行程序开发困难的切实可行的。
 - 语言难，编译器容易。

2. 模型

并行程序设计有如下四种模型：

- 隐式并行 (Implicit Parallel) ——程序员用熟悉的串行语言编程(未作明确的制定并行性)，编译器和运行支持系统自动转化为并行代码。具有如下特点：语义简单、可移植性好、单线程（易于调试和验证正确性）、细粒度并行、效率很低。
- 数据并行 (Data Parallel) ——是 SIMD 的自然模型，局部计算和数据选路操作。具有如下特点：多线程、并行操作于聚合数据结构（数组）、松散同步、单一地址空间、隐式交互作用、显式数据分布。数据并行的优点是编程相对简单且串并行程序一致；缺点有程序的性能在很大程度上依赖于所用的编译系统及用户对编译系统的了解、并行粒度局限于数据级并行、粒度较小。
- 共享变量 (Shared Variable) ——是 PVP、SMP 和 DSM 的自然模型。具有如下特点：多线程 (SPMD, MPMD)、异步、单一地址空间、显式同步、隐式数据分布、隐式通信。
- 消息传递 (Message Passing) ——是 MPP 和 COW 的自然模型。具有如下特点：多线程、异步、多地址空间、显式同步、显式数据映射和负载分配、显式通信。

其中：SPMD (Single Process/ Program, Multiple Data, 单进程/程序、多数据)和 MPMD (Multiple Process/ Program, Multiple Data, 多进程/程序、多数据)都是 MIMD 的子类。COW (Cluster of Workstations, 工作中机群)是集群计算机的一种。

3. 并行编程标准

并行编程标准有：

- 数据并行语言标准——HPF、显式数据分布描述、并行 DO 循环；
- 共享变量编程标准：
 - 线程库(Thread Library)——Win32 API、POSIX threads 线程模型；
 - 编译制导(Compiler Directives)——OpenMP (可移植共享内存并行性)；
- 消息传递库 (Message Passing Libraries) 标准——MPI 和 PVM。

可以将并行编程标准归为如下三类：

- 数据并行——HPF，用于 SMP 和 DSM；
- 共享编程——OpenMP，用于 SMP 和 DSM；
- 消息传递——MPI 和 PVM，用于所有并行计算机。

三者可混合使用，如对以 SMP 为节点的 Cluster 来说，可以在节点间进行消息传递，在节点内进行共享变量编程。

4. 基本并行化方法

并行化的基本方法有：

- 相并行（Phase Parallel）
- 流水线并行（Pipeline Parallel）
- 主从并行（Master-Slave Parallel）
- 分治并行（Divide and Conquer Parallel）
- 工作池并行（Work Pool Parallel）

5. 程序性能优化

● 串行程序性能优化

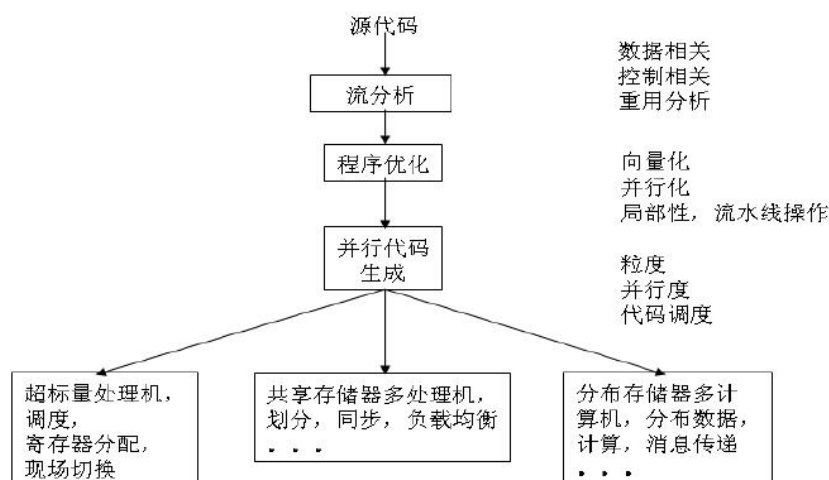
- 调用高性能库，比如优化的 BLAS（Basic Linear Algebra Subprograms，基本线性代数子程序），FFTW（Fastest Fourier Transform in the West，西方快速傅立叶变换，是最快的 FFT 自由软件库）等；
- 选择适当的编译器优化选项；
- 合理定义数组维数；
- 注意嵌套循环的顺序，尽量改善数据访问的局部性；
- 循环展开。

● 并行程序性能优化

- 减少通信量、提高通信粒度；
- 全局通信尽量利用高效集合通信算法；
- 挖掘算法的并行度，减少 CPU 空闲等待；
- 负载平衡；
- 通信、计算的重叠；
- 通过引入重复计算来减少通信，即以计算换通信。

1.7.5 并行编译器

并行编译过程如下图所示：



1. 流分析

流分析 (flow analysis) 主要是相关性分析 (dependency analysis), 包括流相关、反相关、输出相关和控制相关。还需要进行数据相关性测试, 以证明同一数组变量的下标引用对之间的相关性不存在。

2. 程序优化

程序优化就是代码优化, 主要包括代码向量化和代码并行化:

- 代码向量化 (Code Vectorization) —— 把标量程序中的由一种可向量化循环完成的操作变换成向量操作。
- 代码并行化 (Code Parallelization) —— 并行代码的优化是将一个程序展开成多线程以同时供多台处理机并行执行, 其目的是要减少总的执行时间。

3. 代码生成

并行代码生成 (Code Generation) 涉及到将优化后的中间形式的代码转换成可执行的具体的机器目标代码。包括执行次序、指令选择、寄存器分配、负载平衡、并行粒度、代码调度以及后优化 (Postoptimization) 等问题。

1.8 Visual C++本地多核编程

Visual C++从 2005 版开始支持 OpenMP 2.0 的多核编程 (2008 和 2010 版也只支持 2.0 版)。Visual C++ 2010 Beta 1 版支持本地 C++的 PPL (Parallel Pattern Library, 并行模式库) 编程。

1.8.1 OpenMP

本小节介绍 OpenMP 多核编程, 主要内容包括: OpenMP 简介、OpenMP 编程技术、OpenMP 应用程序设计的考虑因素和 Visual C++的 OpenMP 多核编程。

OpenMP 的 MSDN 帮助文档位于: 开发工具与语言\Visual Studio\Visual C++\参考信息\Libraries Reference\OpenMP\ (为英文版)。

1. OpenMP 简介

OpenMP (Open Multi-Processing, 开放多处理)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, 是一种能够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 (API), 包含一组编译器指令、库程序、和影响运行时行为的环境变量。OpenMP 具有良好的可移植性, 支持多种编程语言 C/C++ 和 Fortran 等。支持 OpenMP 的编译器包括 Sun Compiler、GNU Compiler、Intel Compiler 和 Microsoft Visual C++等。OpenMP 能够支持

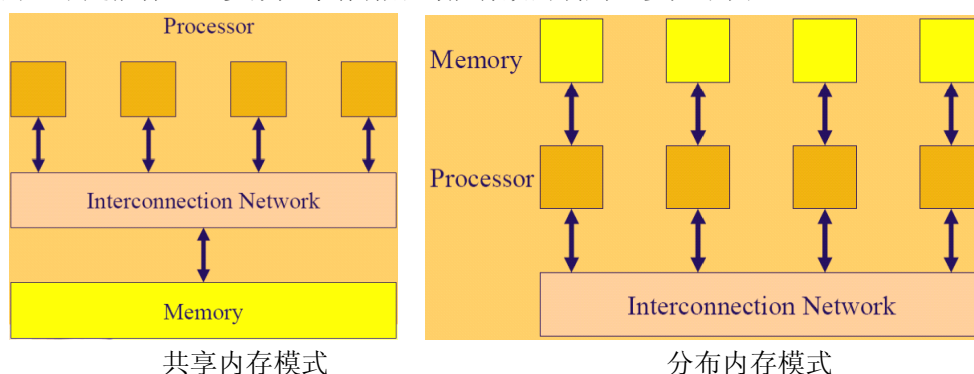
多种平台，包括大多数的类 UNIX 系统以及 Windows NT 系统（Windows 2000、Windows XP、Windows Vista、Windows 7 等）。

1) OpenMP 特点

OpenMP 的设计目标为：标准性、简洁实用、使用方便、可移植性。

OpenMP API（Application Programming Interface，应用编程接口）由三个基本部分（编译指令、运行部分和环境变量）构成，参见下图。是 C/C++ 和 Fortran 等的标准 API，已经被大多数计算机硬件和软件厂家所接受。

OpenMP 不包含的性质有：不是建立在分布式存储系统上的、不是在所有的环境下都是一样的、不是能保证让多数共享存储器均能有效的利用。参见下图：



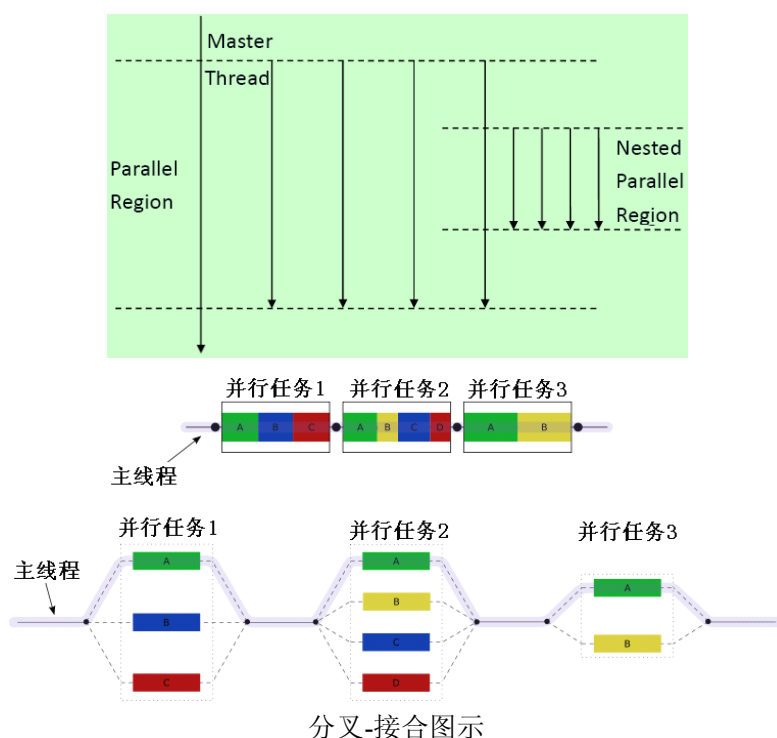
2) OpenMP 的历史

- 1994 年，第一个 ANSI X3H5 草案提出，被否决。
 - 1997 年，OpenMP 标准规范代替原先被否决的 ANSI X3H5，被人们认可。
 - 1997 年 10 月公布了与 Fortran 语言捆绑的第一个标准规范 FORTRAN 1.0。
 - 1998 年 11 月 9 日公布了支持 C 和 C++ 的标准规范 C/C++ 1.0。
 - 2000 年 11 月推出 FORTRAN 2.0。
 - 2002 年 3 月推出 C/C++ 2.0。
 - 2005 年 5 月推出的 OpenMP 2.5 将原来的 Fortran 和 C/C++ 标准规范结合在一起。
 - 2008 年 5 月推出 OpenMP 3.0。
 - 2008 年 11 月推出 OpenMP 3.0 的 C/C++ 语法摘要规范。
 - 2009 年 3 月推出 OpenMP 3.0 的 Fortran 语法摘要规范的修订版。
- 相关的规范可在 <http://openmp.org/wp/openmp-specifications/> 中下载。

3) OpenMP 多线程编程基础

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的控制。

采用 Fork-Join（分叉-接合）的形式，参见下列两图：



其中：A 为主线程（master thread），B、C、D 皆为 A 的子线程
不同并行任务（task）中的同名子线程可以互不相同

4) 编译指导语句

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 程序的一些语义。

#pragma omp <directive> [clause[[,] clause]...] newline

其中，红色部分为关键字，**#pragma**（编译指示/附注/注记/杂注）为编译指令，**omp** 表示 OpenMP；<directive>（指导/指令/指示/指向）部分就包含了具体的编译指导语句，包括：**parallel**、**for**、**parallel for**、**section**、**sections**、**single**、**master**、**critical**、**flush**、**ordered** 和 **atomic**；**clause**（子句）为可选的若干子句，子句间可以用逗号或空白符分隔；**newline** 为换行符，每个 OpenMP 语句必须以换行符结束。例如：

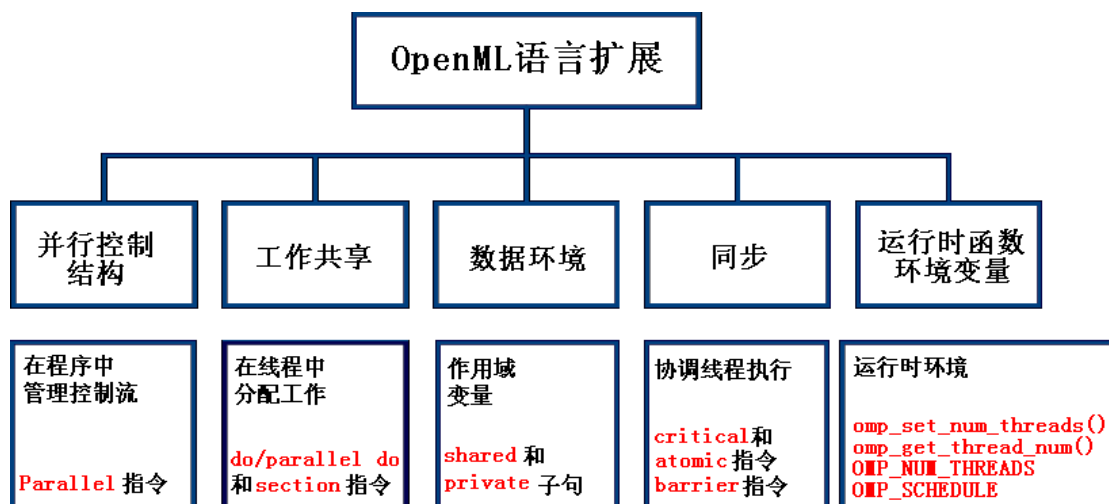
#pragma omp parallel private(var1, var2) shared(var3) {……}

编译指导语句的功能是将串行的程序逐步地改造成一个并程序，达到增量更新程序的目的，减少程序编写人员一定的负担。

4) 运行时库函数

OpenMP 运行时函数库原本用以设置和获取执行环境相关的信息，它们当中也包含一系列用以同步的 API。支持运行时对并行环境的改变和优化，给编程人员足够的灵活性来控制运行时的程序运行状况。

参见下图：



5) OpenMP 程序结构

基于 C/C++语言的 OpenMP 程序的结构为：

```
#include <omp.h>
main () {
    int var1, var2, var3;
    /* 串行代码*/
    ...
    /* 开始并行段。分叉一组线程、指定变量作用域 */
    #pragma omp parallel private(var1, var2) shared(var3) {
        /* 并行段被所有的线程执行 */
        ...
        /* 所有线程接合进主线程并解散 */
    }
    /* 恢复串行代码 */
    ...
}
```

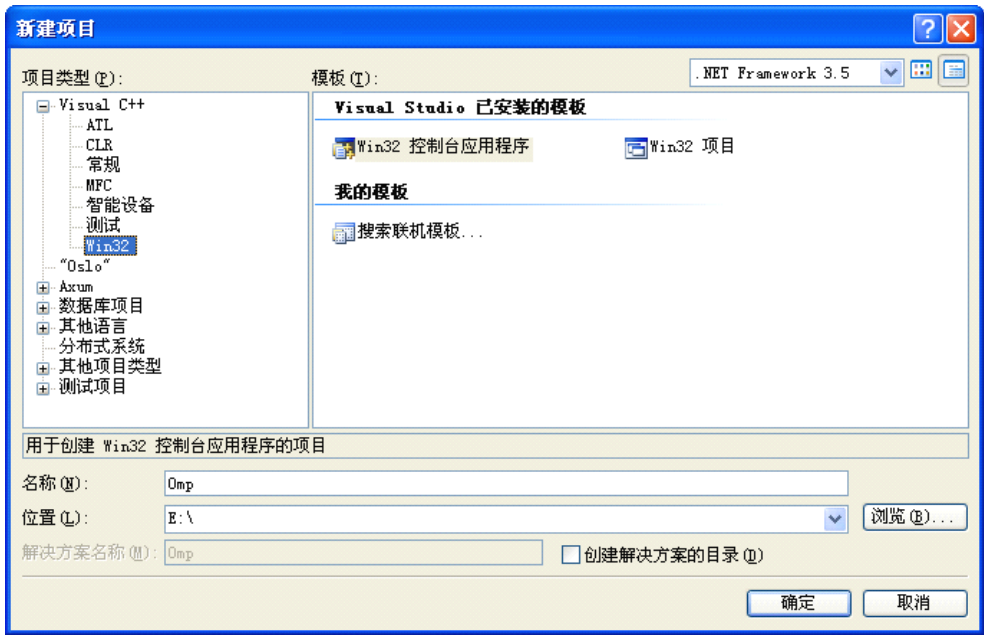
2. Visual C++的 OpenMP 编程步骤

在 CPU 为超线程和多核处理器的 PC 机上，可以使用 Visual Studio 2005/2008/2010 等版本中的 Visual C++进行 OpenMP 编程。为了简单，下面我们以一个没有图形界面的 Win32 控制台应用程序 Omp 为例，来说明编程的具体步骤。

1) 创建项目

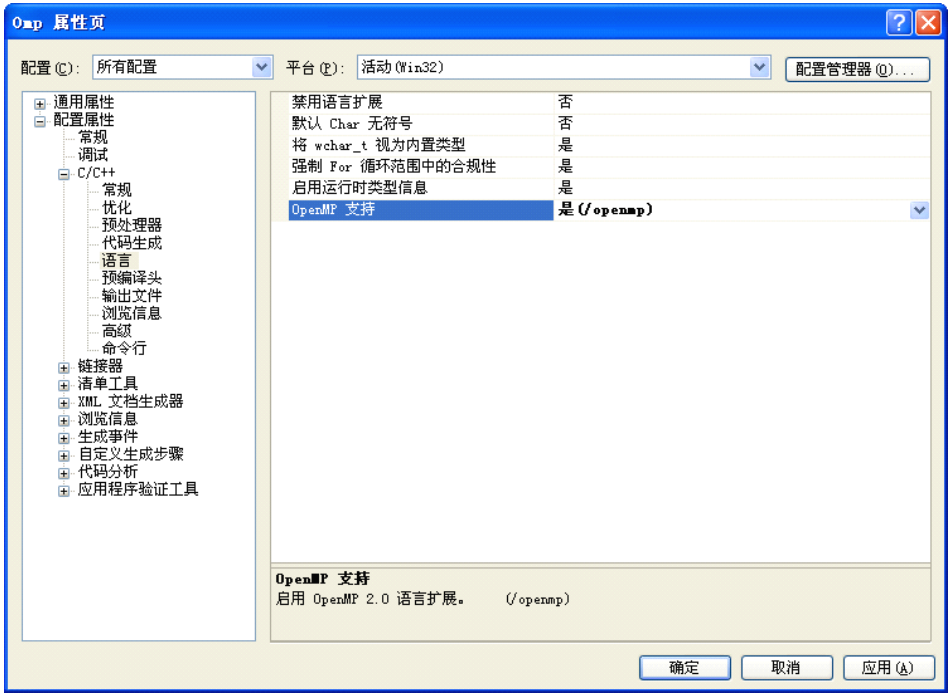
运行 VS08，用“起始页”窗格中的“创建：项目(P)...”栏、选中“文件\新建\项目”菜单项、或直接按“Ctrl+Shift+N”快捷键，打开“新建项目”对话框，参见下图。创建一

个名为 Omp 的“Visual C++\Win32”的“Win32 控制台应用程序”类型的项目（不创建解决方案目录），其余全选择缺省设置，按“完成”按钮创建新项目。



2) 设置项目属性

为了使我们的项目支持 OpenMP 编程，需要进行必要的项目属性设置。选中“项目\Omp 属性(P)”菜单项或按“Alt+F7”快捷键，打开“Omp 属性页”对话框。在左上角的“配置”栏中选择“所有配置”，展开“配置属性”的“C/C++”项，选中其“语言”子项。在右边列表栏的最后一项“OpenMP 支持”栏中，选择“是(/openmp)”项，参见下图：



按“应用”和“确定”按钮关闭对话框，完成设置。

3) 编写代码

在 OpenMP 程序中，包含 OpenMP 的头文件 `omp.h` 并加入各种 OpenMP 编译指导语句。例如，在 `wem` 项目的主程序文件 `Omp.cpp` 中，加入红色的代码部分：

```
// Omp.cpp：定义控制台应用程序的入口点。
//

#include "stdafx.h"
#include <omp.h>
// #include <conio.h>

int _tmain(int argc, _TCHAR* argv[])
{
    printf("你好！——串行。\\n");
    printf("线程号 = %d。\\n", omp_get_thread_num());
    #pragma omp parallel num_threads(4)
    {
        printf("你好！——并行，线程号 = %d。\\n", omp_get_thread_num());
    }
    printf("你好！——又回到串行。\\n");
    // getch();
    return 0;
}
```

其中，头文件 `omp.h` 为 OpenMP 2.0 的标准头文件，一般位于如下目录中：

C:\Program Files\Microsoft Visual Studio 9.0\VC\include\omp.h

绿色部分用于离开 IDE 单独运行可执行程序时，能够保留控制台窗口，等用户按下任何键盘后才关闭。

4) 编译运行

下面是编译运行后的输出结果：（因为系统启动线程的顺序是任意的，所以每次输出的结果可能都不一样）



3. 编译指导语句

OpenMP 的 #pragma 语句的格式为：

#pragma omp 指令名 [子句,]...↵（换行符）

OpenMP 编译指导语句各部分的含义

#pragma omp	指令名 directive-name	[子句 clause, ...]	换行符 newline
指导指令前缀。对所有的 OpenMP 语句都需要这样的前缀。	指导指令。在指导指令前缀和子句之间必须有一个正确的 OpenMP 指导指令。	子句。在没有其它约束条件下，子句可以无序，也可以任意的选择。这一部分也可以没有。	换行符。表明这条指导语句的终止。

1) 作用域

OpenMP 编译指导语句的作用域有静态范围、孤立语句和动态扩展范围三种类型：（参见下表）

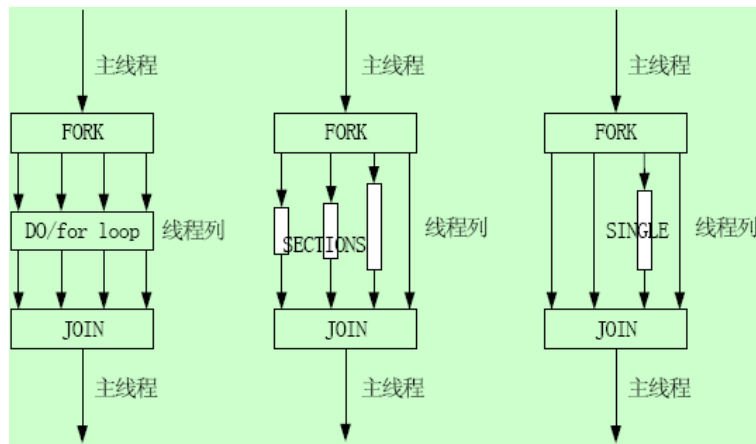
- 静态范围——文本代码在一个编译指导语句之后，被封装到一个结构块中。
- 孤立语句——一个 OpenMP 的编译指导语句不依赖于其它的语句。
- 动态扩展——包括静态范围和孤立语句。

作用域

动态扩展范围	
静态范围	孤立语句
for 语句出现在一个封闭的并行域中	critical 和 sections 语句出现在封闭的并行域之外
<pre>#pragma omp parallel { ... #pragma omp for for(...) { ... sub1(); ... } ... sub2(); ... }</pre>	<pre>void sub1() { ... #pragma omp critical ... } void sub2() { ... #pragma omp sections ... }</pre>

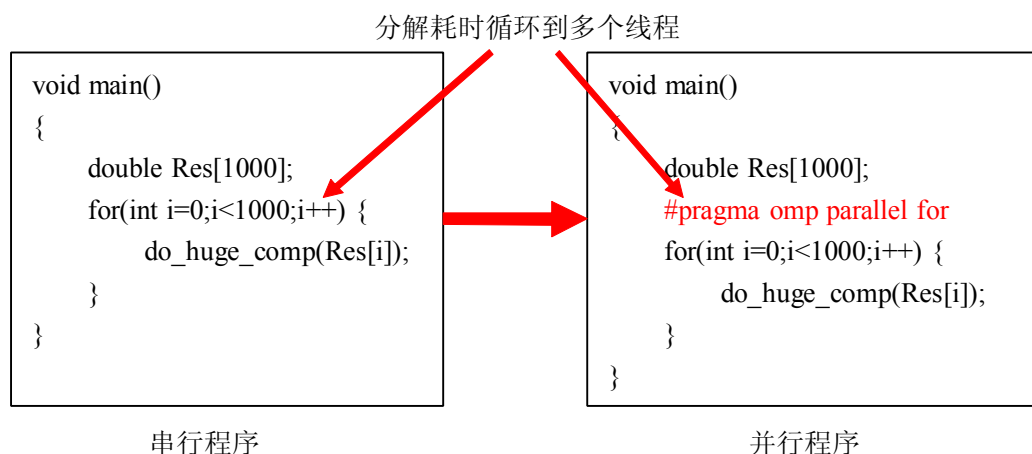
2) 共享任务结构

共享任务结构将它所包含的代码划分给线程组的各成员来执行，包括：并行 for 循环、并行 sections 和串行执行。参见下图：



3) OpenMP 典型应用

OpenMP 常用于循环并行化，通过循环并行化编译指导语句使得一段代码能够在多个线程内部同时执行。这需要寻找程序代码中最耗时的循环，并将其分解到多个线程中去。参见下图：



4. OpenMP 编程技术

下面讨论循环并行化、并行区域、工作分区、单一线程、线程同步等方面的 OpenMP 编程技术。

1) 循环并行化

循环并行化（loop parallelize）是指使用 OpenMP 的 `parallel for` 指导语句将 C/C++ 的 `for` 循环并行化，即将循环中的迭代（平均）分配给线程组中的各个线程分别执行后再汇总。

（1）格式

循环并行化编译指导语句的格式为：

`#pragma omp parallel for [子句 [子句 ...]]` (换行符)

```

    for( 索引 = 初值; 测试表达式; 增量表达式 ){
        循环体
    }

```

for 语句指定紧随它的循环语句，必须由线程组并行执行。

(2) 限制

对循环并行化语句有很多限制：

- 必须是 for 循环、且其必须具有规范的格式、能够推测出循环次数；
- 索引必须为整数类型；
- 测试表达式必须具有如下形式：

索引 比较运算符 终值

其中的比较运算符可以是 <、<=、>=、>；

- 增量表达式必须为：索引++、++索引、索引--、--索引、索引+=增量、索引-=增量、索引=索引+增量、索引=增量+索引、索引=索引-增量；
- 初值、增量和终值都可以是任意数值表达式，但是都必须在循环过程中保持值不变，以保证在循环前就能计算出循环的次数；
- 循环语句块应该是单出口与单入口的。因此，不能使用 break 语句，也不能用 goto、return 等语句从循环中跳出。可以使用 continue 语句（因为它不会影响循环次数的计算），也可以使用 goto 语句跳到循环内。C++ 不能在循环内部抛出异常，因为这样会导致从循环中退出。但是可以在循环体内使用 exit() 函数退出整个程序。当某一个线程调用此函数后，会同步其他所有线程来退出程序，不过退出时的状态是不确定的。

(3) 嵌套

循环并行化编译指导语句可以加在任意一个循环（包括嵌套的循环）之前，则对应的最近的循环语句（不是外嵌套循环）被并行化。

(4) 子句

循环并行化编译指导语句中的子句可以是：

- Schedule(type [,chunk])
- ordered
- private (list)
- firstprivate (list)
- lastprivate (list)
- shared (list)
- reduction (operator: list)
- nowait

其中的 schedule 子句描述如何将循环的迭代划分给线程组中的线程；如果没有指定块（chunk）大小，迭代会尽可能的平均分配给每个线程；type 为 static 时，循环被分成大小为 chunk 的块，静态分配给线程；type 为 dynamic 时，循环被动态划分为大小为 chunk 的块，动态分配给线程。

(5) 示例

```

#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main () {
    int i, chunk;

```

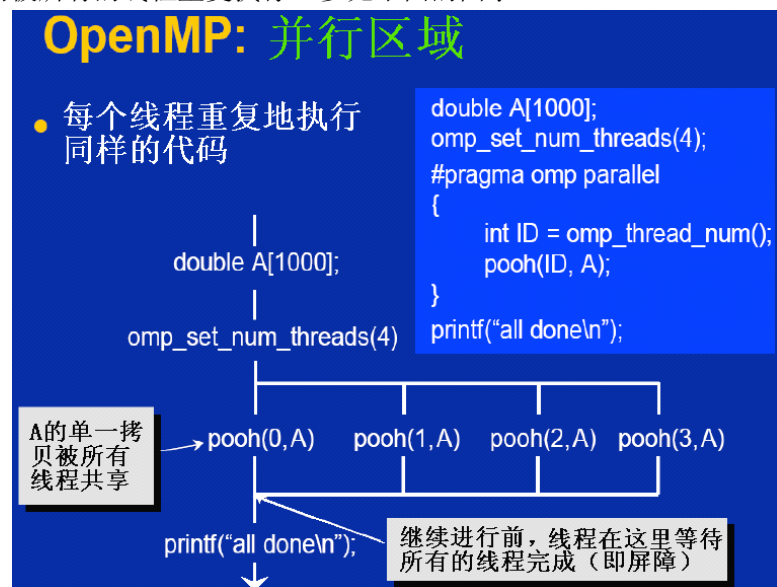
```

float a[N], b[N], c[N];
for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
// 每个线程执行 100 个循环
#pragma omp parallel for shared(a,b,c,chunk) private(i) schedule(static,chunk)
for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
}

```

2) 并行区域

并行区域（parallel region）指由 OpenMP 的 parallel 语句定义的并行控制语句块，并行区域中的代码被所有的线程重复执行。参见下面的图示：



（1）格式

并行区域编译指导语句的格式

```

#pragma omp parallel [子句 [子句 ]...] 换行符
语句[块]

```

其中，子句 =

- if(scalar-expression)
- private(list)
- firstprivate(list)
- default(shared | none)
- shared(list)
- copyin(list)
- reduction(operator: list)
- num_threads(integer-expression)

（2）使用限制

对并行区域的使用限制与 parallel for 语句的部分相似，即语句块必须是单入口与单

出口的。不允许从外面转入块内部，也不允许块内部有多个出口转到块外。

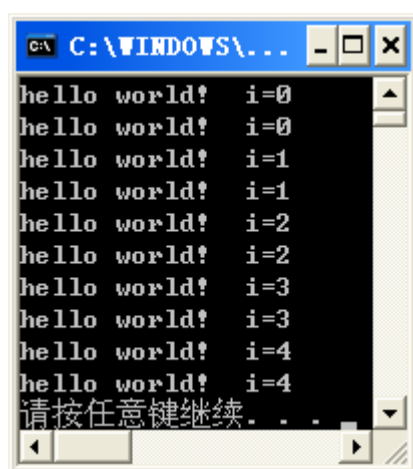
(3) 与 `parallel for` 语句的区别

并行区域和循环并行化的 `OpenMP` 语句相比，只是指令名少一个 `for` 关键字。但是他们的行为则相差很大：并行区域采用了复制执行方式，将代码在所有的线程内各执行一次；二循环并行化则是采用工作分配执行方式，将循环需做的所有工作量，按一定的方式分配给各个执行线程，全部线程执行工作的总合等于原先串行执行所完成的工作量。

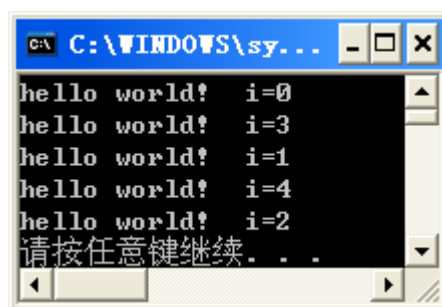
我们通过下面的两个例子来演示 `parallel` 和 `parallel for` 编译指导语句的执行过程：

<pre>#pragma omp parallel num_threads(2) for (int i = 0; i < 5; i++) printf("hello world! i=%d\n",i);</pre>	<pre>#pragma omp parallel for num_threads(2) for (int i = 0; i < 5; i++) printf("hello world! i=%d\n",i);</pre>
---	--

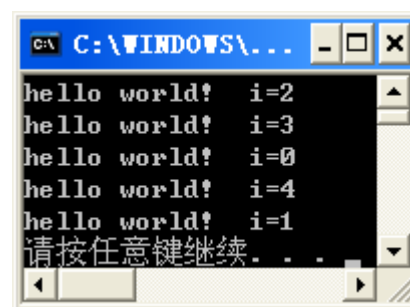
下面是程序运行的输出结果：(`parallel` 语句的每次输出都一样，而 `parallel for` 的则可能不同)



并行区域



循环并行化 1



循环并行化 2

3) 工作分区

工作分区（`sections`）是指利用 `OpenMP` 的 `sections` 编译指导语句，将用 `section` 语句指定的内部代码，划分成多个工作区分配给线程组中的各个线程，不同的 `section` 由不同的线程执行。各线程工作量的总合等于原来的工作量。这显然比并行区域所做的重复性劳动更有意义。

(1) 格式：

```
#pragma omp sections [子句 [[,]子句]...]换行符
{
    #pragma omp section newline
    ...
    #pragma omp section newline
    ...
}
```

其中的子句 =

- `private (list)`
- `firstprivate (list)`
- `lastprivate (list)`

- reduction (operator: list)
- nowait

在 sections 语句结束处有一个隐含的屏障，使用了 nowait 子句除外。

(2) 示例

工作分区 (sections) 编码示例 1:

```
#pragma omp parallel sections
{
    #pragma omp section
    printf("section 1 thread=%d\n",omp_get_thread_num());
    #pragma omp section
    printf("section 2 thread=%d\n",omp_get_thread_num());
    #pragma omp section
    printf("section 3 thread=%d\n",omp_get_thread_num());
}
```

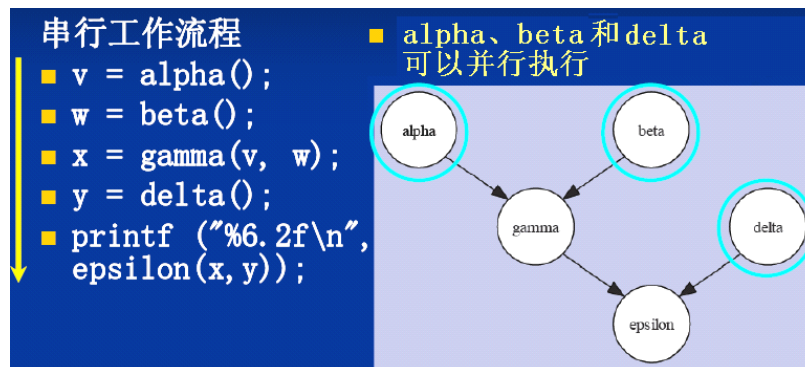
编译运行后的输出结果为:



sections 示例 2:

```
#pragma omp parallel sections
{
    #pragma omp section /* Optional */
    v = alpha();
    #pragma omp section
    w = beta();
    #pragma omp section
    y = delta();
}
x = gamma(v, w);
printf ("%6.2f\n", epsilon(x,y));
```

参见下图:



Section 示例三

```
#include "stdafx.h"
#include <omp.h>
#define N 1000
int main (){
    int i;
    float a[N], b[N], c[N];
    // 初始化
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    // 开始并行区域
    #pragma omp parallel shared(a,b,c) private(i)
    {
        // 开始工作分区
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N/2; i++)
                c[i] = a[i] + b[i];
            #pragma omp section
            for (i=N/2; i < N; i++)
                c[i] = a[i] + b[i];
        } // 结束工作分区
    } // 结束并行区域
}
```

4) 单一线程

`single` 编译指导语句指定内部代码只有线程组中的一个线程执行。线程组中没有执行 `single` 语句的线程会一直等待代码块的结束，使用 `nowait` 子句除外。

(1) 格式

```
#pragma omp single [clause[.,]clause]... newline
```

clause=

- `private(list)`
- `firstprivate(list)`

(2) 示例

```
void work1() {printf("work1 在运行。 \n");}
void work2() {printf("work2 在运行。 \n");}
int _tmain(int argc, _TCHAR* argv[])
{
    #pragma omp parallel num_threads(2)
    {
        #pragma omp single
```

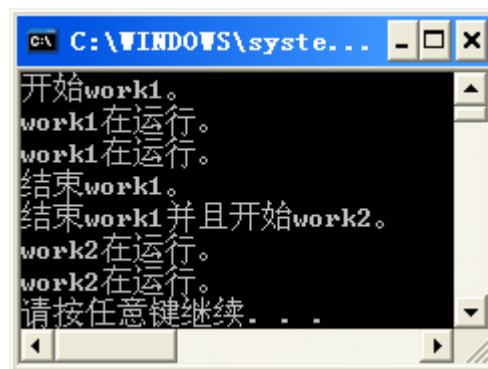


```

printf("开始 work1。 \n");
work1();
#pragma omp single
printf("结束 work1。 \n");
#pragma omp single nowait
printf("结束 work1 并且开始 work2。 \n");
work2();
}
return 0;
}

```

编译运行的输出结果如下：



5) 并行区域之间的工作共享方法

(1) 工作队列

工作队列的基本工作过程即为维持一个工作的队列，线程在并行执行的时候，不断从这个队列中取出相应的工作完成，直到队列为空为止。

(2) 根据线程号分配任务

由于每一个线程在执行的过程中的线程标识号是不同的，可以根据这个线程标识号来分配不同的任务。例如：

```

#pragma omp parallel private(myid)
{
    int nthreads = omp_get_num_threads();
    int myid = omp_get_thread_num();
    work_done(myid, nthreads); // 分配任务函数
}

```

(3) 使用循环语句分配任务

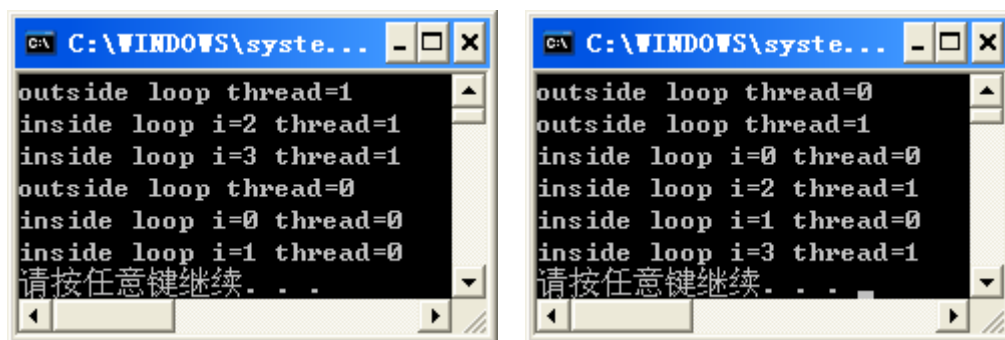
例如：

```

#pragma omp parallel num_threads(2)
{
    printf("outside loop thread=%d\n", omp_get_thread_num());
    #pragma omp for
    for(int i = 0; i < 4; i++)
        printf("inside loop i=%d thread=%d\n", i, omp_get_thread_num());
}

```

编译运行后的结果输出为：



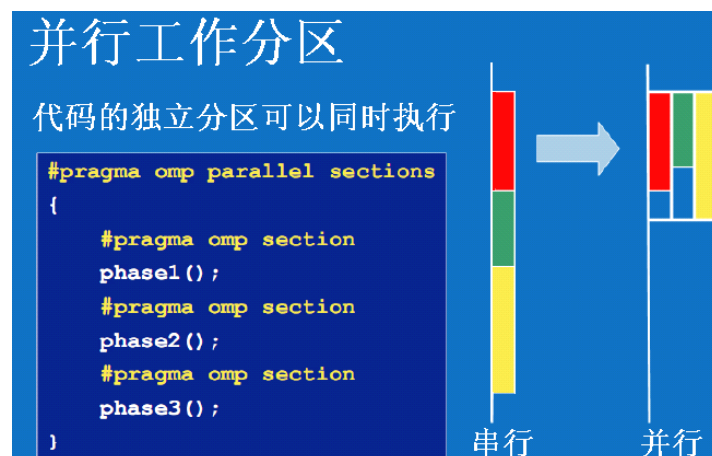
(4) 工作分区编码

```
#pragma omp parallel sections
{
    #pragma omp section
    printf("section 1 thread=%d\n",omp_get_thread_num());
    #pragma omp section
    printf("section 2 thread=%d\n",omp_get_thread_num());
    #pragma omp section
    printf("sectino 3 thread=%d\n",omp_get_thread_num());
}
```

程序运行结果为：



参见下图：



6) OpenMP 线程同步

为了保证在多线程执行的程序中，出现数据竞争时能够得到正确结果，OpenMP 提供了

两种不同类型的线程同步机制：互斥锁和事件通知机制。

（1）数据竞争

我们先看一个简单的寻找数组中最大整数值例子：

```
int max_num = -1;
#pragma omp parallel for
for ( int i = 0; i < N; i++ )
    if ( a[i] > max_num )
        max_num = a[i];
```

由于 max_num 是共享变量，多线程同时执行循环时，可能会出现错误的结果。例如 N=2, a[0]=5、a[1]=2，由两个线程各执行一个 if 语句，如果执行 i=1 的线程在完成比较操作后被挂起，此时执行 i=0 的线程完成比较和赋值（max_num = 5），接着执行 i=1 的线程被唤醒，因为比较操作已经完成，所以继续进行赋值操作（max_num = a[i];），最后的结果是 max_num = 2。

（2）互斥锁机制

在 OpenMP 中，提供了三种不同的互斥锁机制用来对一块内存进行保护，它们分别是临界区（critical）、原子操作（atomic）、及由库函数来提供的若干同步操作函数 omp_*_lock。

OpenMP 提供的线程同步语句有：master、critical、barrier、atomic、flush 和 ordered 等指导语句。

- Master 指导语句——master 指导语句指定代码段只有主线程执行，语句的格式为：

```
#pragma omp master newline
```

- Critical 指导语句——critical 指导语句表明域中的代码一次只能由一个线程执行，而其他线程被阻塞在临界区。语句格式为：（name 为需加锁的变量名）

```
#pragma omp critical [name] newline
```

在程序需要访问可能产生竞争的内存数据时，都需要插入相应的临界区代码。例如：

```
int max_num = -1;
#pragma omp parallel for
for ( int i = 0; i < N; i++ )
    #pragma omp critical (max_num)
    if ( a[i] > max_num )
        max_num = a[i];
```

critical 指导语句常与具有（指定共享变量的）share 子句的 OpenMP 语句配套使用，用于保护该共享变量（此时就不再需指明加锁变量的参数了）。例如：

```
int max_num = -1;
#pragma omp parallel for share (max_num) // 设置共享变量
for ( int i = 0; i < N; i++ )
    #pragma omp critical // 临界指导语句
    if ( a[i] > max_num )
        max_num = a[i];
```

- Barrier 指导语句——barrier 指导语句用来同步一个线程组中所有的线程，先到达的线程在此阻塞，等待其他线程。barrier 语句最小代码必须是一个结构化的块，而不能只是一个单行的指导语句。barrier 语句的格式为：

```
#pragma omp barrier newline
```

barrier 正确与错误使用比较

错误	正确
<pre>if (x != 0) #pragma omp barrier f(x);</pre>	<pre>if (x != 0) { #pragma omp barrier f(x); }</pre>

例如：

#pragma omp parallel for share (A, B, C) // 设置共享变量

```
{
    f(A, B);
    printf("处理过的 A 进入 B\n");
    #pragma omp barrier // 等待修改过的 B
    f(B, C);
    printf("处理过的 B 进入 C\n");
}
```

- Atomic 指导语句——atomic 指导语句指定特定的存储单元将被原子更新。语句格式为：

#pragma omp atomic newline

原子操作是 OpenMP 编程方式给同步编程带来的特殊的编程功能，通过编译指导语句的方式直接获取了现在多处理器计算机体系结构的功能。通过 #pragma omp atomic 编译指导语句提供。只能作用在语言内建的基本数据结构。如：

```
#pragma omp atomic
x <binop>=expr
```

或者

```
#pragma omp atomic
x++//or x--, --x, ++x
```

其中，x 是一个标量；expr 是一个不含对 x 引用的标量表达式，且不被重载；binop（二进制操作）是 +, *, -, /, &, ^, |, >>, or << 之一，且不被重载。

例如：

```
int counter=0;
#pragma omp parallel
{
    for(int i=0;i<10000;i++)
        #pragma omp atomic //atomic operation
        counter++;
}
printf("counter = %d\n", counter);
```

- flush 指导语句——flush 指导语句用以标识一个同步点，用以确保所有的线程看到一致的存储器视图。语句格式为：

#pragma omp flush (list) newline

flush 将在下面几种情形下隐含运行，nowait 子句除外：

- barrier
- critical: 进入与退出部分
- ordered: 进入与退出部分

- parallel:退出部分
- for:退出部分
- sections:退出部分
- single:退出部分

例子:

```
int iam, neighbor;
int sync[NUNBER_OF_THREADS];
float work[NUNBER_OF_THREADS];
#pragma omp parallel private(iam, neighbor) shared(work, sync)
{
    iam = omp_get_thread_num();
    sync[iam] = [0];
    #pragma omp barrier
    // 计算我的 work 数组部分
    work[iam] = .....;
    // 宣告我 (I am) 在干我的活 work, 第一个 flush 确保我的工作 在 sync
    // 之前成为可见, 第二个 flush 确保 sync 成为可见。
    #pragma omp flush(work)
    sync[iam] = 1;
    #pragma omp flush(sync)
    // 等待 neighbor
    neighbor = (iam > 0 ? iam : omp_get_num_threads()) - 1;
    while (sync[neighbor] == 0) {
        #pragma omp flush(sync)
    }
    // 读入 neighbor 的 work 数组值
    ..... = work[neighbor];
}
```

- **ordered** 指导语句——**ordered** 指导语句指出其所包含循环的执行在任何时候只能有一个线程执行被 **ordered** 所限定的部分, 只能出现在 **for** 或者 **parallel for** 语句的动态范围中。语句格式为:

#pragma omp ordered newline

例如:

```
void work(int k)
{
    #pragma omp ordered
    printf(" %d", k);
}
.....
#pragma omp for ordered schedule(dynamic)
for (i = lb; i < ub; i += st)
    work(i);
```

- **threadprivate** 编译指导语句——**threadprivate** 语句使一个全局文件作用域的变量在并行域内变成每个线程私有, 每个线程对该变量复制一份私有拷贝。语句格式为:

#pragma omp threadprivate (list) newline

例如:

```
int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)
int main ()
{
    // 第一个并行区域
    #pragma omp parallel private(i,beta)
    for (i=0; i < 10; i++)
        alpha[i] = beta[i] = i;
    // 第二个并行区域
    #pragma omp parallel
    printf("alpha[3]= %d and beta[3]=%d\n", alpha[3], beta[3]);
}
```

- 线程私有数据——线程私有数据与 threadprivate 和 copyin 子句: 使用 threadprivate 子句用来标明某一个变量是线程私有数据, 在程序运行的过程中, 不能够被其他线程访问到。使用 copyin 子句对线程私有的全局变量进行初始化。例如:

```
int counter = 0;
// 使用 threadprivate
#pragma omp threadprivate (counter)
void inc_counter() { counter++; }
int _tmain(int argc, TCHAR *argv[])
{
    #pragma omp parallel
    for(int i = 0; i < 10000; i++)
        inc_counter();
    printf("counter=%d\n", counter);
}
```

及

```
int global = 0;
#pragma omp threadprivate(global)
int _tmain(int argc, TCHAR * argv[])
{
    global = 1000;
    #pragma omp parallel copyin(global)
    {
        printf("global=%d\n", global);
        global = omp_get_thread_num();
    }
    printf("global=%d\n", global);
    printf("parallel again\n");
    #pragma omp parallel
    printf("global=%d\n", global);
}
```

7) 数据域属性子句

OpenMP 的数据域属性子句用于指定变量的作用域范围。数据域属性子句有：**private** 子句、**shared** 子句、**default** 子句、**firstprivate** 子句、**lastprivate** 子句、**copyin** 子句和 **reduction** 子句。

- **private** 子句——**private** 子句表示它列出的变量（可能局部）对于每个线程是局部的。
语句格式为：**private(list)**。

private 和 threadprivate 区别

	private	threadprivate
数据类型	变量	变量
位置	在域的开始或共享任务单元	在块或整个文件区域的例程定义上
持久性	否	是
扩充性	只是词法的，除非作为子程序的参数而传递	动态的
初始化	使用 firstprivate	使用 copyin

- **shared** 子句——**shared** 子句表示它所列出的变量被线程组中所有的线程共享
所有线程都能对它进行读写访问。语句格式为：**shared (list)**
- **default** 子句——**default** 子句让用户自行规定在一个并行域的静态范围中所定义的变量的缺省作用范围。语句格式为：**default (shared | none)**。例如：

```
int x, y, z[1000];
#pragma omp threadprivate(x)
void fun(int a) {
    const int c = 1;
    int i = 0;
    #pragma omp parallel default(none) private(a) shared(z)
    {
        int j = omp_get_num_thread(); // 正确！因为 j 声明在并行区域内
        a = z[j]; // 正确！因为 a 和 z 分别被列在 private 和 shared 子句中
        x = c; // 正确！因为 x 是 threadprivate，c 是常量限定类型
        z[i] = y; // 错误！不能在这里引用 i 或 y
    }
}
```

- **firstprivate** 子句——**firstprivate** 子句是 **private** 子句的超集，用于对变量做原子初始化。
语句格式为：**firstprivate (list)**。例如：

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for ( i = 0; i < MAX; i++) {
    if ( (i%2) == 0) incr++;
    a[i] = incr;
}
```

- **lastprivate** 子句——**lastprivate** 子句也是 **private** 子句的超集，用于将变量从最后的循环迭代或段复制给原始的变量。语句格式为：**lastprivate (list)**。例如：


```

void sq2(int n, double *last)
{
    double x;
    #pragma omp parallel
    #pragma omp for lastprivate(x)
    for (int i = 0; i < n; i++) {
        x = a[i] * a[i] + b[i] * b[i];
        b[i] = sqrt(x);
    }
    last = x;
}

```

- **copyin** 子句——**copyin** 子句用来为线程组中所有线程的 **threadprivate** 变量赋相同的值，主线程该变量的值作为初始值。语句格式为：**copyin(list)**。
- **reduction** 子句——**reduction** 子句使用指定的操作对其列表中出现的变量进行归约。初始时，每个线程都保留一份私有拷贝，在结构尾部根据指定的操作对线程中的相应变量进行归约，并更新该变量的全局值。语句格式为：**reduction (operator: list)**。

reduction 子句的格式：（与 **atomic** 指导语句的类似）

```

x = x op expr
x = expr op x (except subtraction)
x binop = expr
x++, ++x, x--, --x

```

其中：**x** 是一个标量；**expr** 是一个不含对 **x** 引用的标量表达式，且不被重载；**binop** 是 $+$ 、 $*$ 、 $-$ 、 $/$ 、 $\&$ 、 \wedge 、 $|$ 、 $\&\&$ 、 or 之一，且不被重载；**op** 是 $+$ 、 $*$ 、 $-$ 、 $/$ 、 $\&$ 、 \wedge 、 $|$ 、 $\&\&$ 、 or 之一，且不被重载。

操作数	+	*	-	\wedge	$\&$		$\&\&$	$ $
初始值	0	1	0	0	~ 0	0	1	0

8) 归约操作

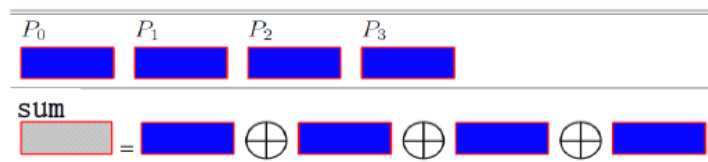
归约操作是 OpenMP 编程方式给同步编程带来的特殊的编程功能，该操作会反复将一个二元运算符应用在一个变量和另一个值上，并把结果保存在原变量中。该操作通过 **reduction** 语句提供。

- 累积求和例：（参见右图）

```

#pragma omp parallel for private (x) reduction (+: sum)
for(i = 0; i < 4; i++)
{
    x = 2*i;
    sum += x;
}

```



- 归约操作例：

```

#include <omp.h>
#define NUM_THREADS 2
void main ()
{

```

```

int i;
double ZZ, func(), sum=0.0;
omp_set_num_threads(NUM_THREADS) //设置线程数
#pragma omp parallel for reduction(+:sum) private(ZZ)
for (i=0; i< 1000; i++){
    ZZ = 2*i;
    sum = sum + ZZ;
}
}

```

●reduction 子句例:

```

#include <omp.h>
int main ()
{
    int i, n, chunk;
    float a[100], b[100], result;
    /* Some initializations */
    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }
    #pragma omp parallel for reduction(+:result)
    for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
    printf("Final result= %f\n",result);
}

```

子句/编译指导语句总结

编译指导 子句	parallel	DO/for	sections	single	parallel DO/for	parallel sections
if	✓				✓	✓
private	✓	✓	✓	✓	✓	✓
shared	✓	✓			✓	✓
default	✓				✓	✓
firstprivate	✓	✓	✓	✓	✓	✓
lastprivate		✓	✓		✓	✓
reduction	✓	✓	✓		✓	✓
copyin	✓				✓	✓
schedule		✓			✓	
ordered		✓			✓	
nowait		✓	✓	✓		

9) 语句绑定和嵌套规则

(1) 语句绑定规则

- 语句 DO/for、SECTIONS、SINGLE、MASTER 和 BARRIER 绑定到动态的封装 PARALLEL 中，如果没有并行域执行，这些语句是无效的；
- 语句 ORDERED 指令绑定到动态 DO/for 封装中；
- 语句 ATOMIC 使得 ATOMIC 语句在所有的线程中独立存取，而并不只是当前的线程；
- 语句 CRITICAL 在所有线程有关 CRITICAL 指令中独立存取，而不是只对当前的线程；
- 在 PARALLEL 封装外，一个语句并不绑定到其它的语句中。

(2) 语句嵌套规则

- PARALLEL 语句动态地嵌套到其它地语句中，从而逻辑地建立了一个新队列，但这个队列若没有嵌套地并行域执行，则只包含当前的线程；
- DO/for、SECTION 和 SINGLE 语句绑定到同一个 PARALLEL 中，则它们是不允许互相嵌套的；
- DO/for、SECTION 和 SINGLE 语句不允许在动态的扩展 CRITICAL、ORDERED 和 MASTER 域中；
- CRITICAL 语句不允许互相嵌套；
- BARRIER 语句不允许在动态的扩展 DO/for、ORDERED、SECTIONS、SINGLE、MASTER 和 CRITICAL 域中；
- MASTER 语句不允许在动态的扩展 DO/for、SECTIONS 和 SINGLE 语句中；
- ORDERED 语句不允许在动态的扩展 CRITICAL 域中；
- 任何能允许执行到 PARALLEL 域中的指令，在并行域外执行也是合法的。当执行到用户指定的并行域外时，语句执行只与主线程有关。

5. 运行库例程与环境变量

1) 运行库例程

OpenMP 标准定义了一个应用编程接口来调用库中的多种函数。对于 C/C++，在程序开头需要引用文件 “omp.h”。

(1) OpenMP 常用的 API

```
OMP_GET_NUM_THREADS()
OMP_GET_MAX_THREADS()
OMP_GET_THREAD_NUM()
OMP_GET_NUM_PROCS()
OMP_IN_PARALLEL()
OMP_SET_DYNAMIC(dynamic_threads)
OMP_GET_DYNAMIC()
OMP_SET_NESTED(nested)
OMP_GET_NESTED()

omp_init_lock(omp_lock_t *lock)
```

```

omp_init_nest_lock(omp_nest_lock_t *lock)
omp_destroy_lock(omp_lock_t *lock)
omp_destroy_nest_lock(omp_nest_lock_t *lock)
omp_set_lock(omp_lock_t *lock)
omp_set_nest_lock(omp_nest_lock_t *lock)
omp_unset_lock(omp_lock_t *lock)
omp_unset_nest_lock(omp_nest_lock_t *lock)
omp_test_lock(omp_lock_t *lock)
omp_test_nest_lock(omp_nest_lock_t *lock)
omp_get_wtime()
omp_get_wtick()

```

(2) OpenMP 运行时库函数的互斥锁支持

OpenMP 通过一系列的库函数支持更加细致的互斥锁操作。编译指导语句进行的互斥锁支持只能放置在一段代码之前，作用在这段代码之上。程序员必须自己保证在调用相应锁操作之后释放相应的锁，否则就会造成多线程程序的死锁。

函数名称	描述
<code>void omp_init_lock(omp_lock_t *)</code>	初始化一个互斥锁
<code>void omp_destroy_lock(omp_lock_t*)</code>	结束一个互斥锁的使用并释放内存
<code>void omp_set_lock(omp_lock_t *)</code>	获得一个互斥锁
<code>void omp_unset_lock(omp_lock_t *)</code>	释放一个互斥锁
<code>int omp_test_lock(omp_lock_t *)</code>	试图获得一个互斥锁，并在成功是返回真（true），失败是返回假（false）

互斥锁示例：

```

omp_lock_t lck;
int id;
omp_init_lock(&lck);
#pragma omp parallel shared(lck) private(id)
{
    id = omp_get_thread_num();
    omp_set_lock(&lck);
    printf("我的线程 ID 为%d。", id); // 该 printf 每次只能被一个线程执行
    omp_unset_lock(&lck);
    while(!omp_test_lock(&lck)) {
        skip(id); // 我们不拥有互斥锁，所以得干点其他事
    }
    work(id); // 我们拥有了互斥锁，所以可以干活
    omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);

```

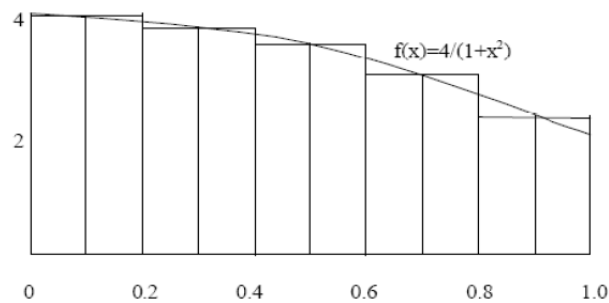
2) 环境变量

- OMP_SCHEDULE: 只能用到 for,parallel for 中。它的值就是处理器中循环的次数;
- OMP_NUM_THREADS: 定义执行中最大的线程数;
- OMP_DYNAMIC: 通过设定变量值 TRUE 或 FALSE,来确定是否动态设定并行域执行的线程数;
- OMP_NESTED: 确定是否可以并行嵌套。

6. 计算 π 实例

下面我们用矩形法则的数值积分方法来估算 π 的值:

$$\pi = \int_0^1 f(x)dx \approx \frac{1}{N} \sum_{i=1}^N f\left(\frac{i-0.5}{N}\right), \quad f(x) = \frac{4}{1+x^2}$$



1) 串行程序

```
static long num_steps = 100000;
double x, pi, sum = 0.0, step = 1.0/(double) num_steps;
for (long i = 1; i <= num_steps; i++){
    x = (i + 0.5)*step;
    sum += 4.0/(1.0 + x*x);
}
```

输出结果为:



2) 并行程序 1(并行区域)

```
#define NUM_THREADS 2
```

```

static long num_steps = 100000;
double x, pi = 0.0, sum[NUM_THREADS], step = 1.0/(double)num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    sum[id] = 0.0;
    for (long i = id; i < num_steps; i += NUM_THREADS) {
        x = (i + 0.5)*step;
        sum[id] += 4.0/(1.0 + x*x); // 根据 ID 号分配任务
    }
}
for(int i = 0; i < NUM_THREADS; i++) pi += sum[i] * step;
printf("Pi = %f\n", pi);
输出结果为:

```



3) 并程序 2(循环并行化)

```

#define NUM_THREADS 2
static long num_steps = 100000;
double x, pi = 0.0, sum[NUM_THREADS], step = 1.0/(double)num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    sum[id] = 0.0;
    #pragma omp for
    for (long i = 0; i < num_steps; i++)
    {
        x = (i + 0.5)*step;
        sum[id] += 4.0/(1.0 + x*x); // 存储每次的结果
    }
}
for(int i = 0; i < NUM_THREADS; i++) pi += sum[i] * step;
printf("Pi = %f\n", pi);
输出结果为:

```



4) 并行程序 3(私有化和临界区)

```
#define NUM_THREADS 2
static long num_steps = 100000;
double x, pi = 0.0, sum, step = 1.0/(double)num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(x, sum)
{
    int id = omp_get_thread_num();
    sum = 0.0;
    for (long i = id; i < num_steps; i += NUM_THREADS)
    {
        x = (i + 0.5)*step;
        sum += 4.0/(1.0 + x*x);
    }
    #pragma omp critical
    pi += sum*step;
}
printf("Pi = %f\n", pi);
```

输出结果为：



5) 并行程序 4(归约操作)

```
#define NUM_THREADS 2
static long num_steps = 100000;
double x, pi = 0.0, sum = 0.0, step = 1.0/(double)num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
for (long i = 0; i < num_steps; i++)
{
    x = (i + 0.5)*step;
    sum += 4.0/(1.0 + x*x);
}
```



```

}
pi += sum*step;
printf("Pi = %f\n", pi);

```

输出结果为：



7. OpenMP 应用程序设计的考虑因素

影响性能的主要因素：

- 根据 Amdahl 定律，我们应当努力提高并行化代码在应用程序中的比率，这是通用的提高效率的方法。
- OpenMP 本身的开销——OpenMP 获得应用程序多线程并行化的能力，需要一定的程序库支持。在这些库程序对程序并行加速的同时也需要运行库本身，这必然会带来一定的开销。
- 负载均衡——如果各个线程之间的负载不均衡，就有可能造成某些线程在执行过程中无事可干，经常处于空闲状态；而另外一些线程则负担沉重，需要很长时间才能够完成任务。
- 局部性——需要考虑高速缓存的局部性假设，以提高其利用效率。
- 线程同步带来的开销——应该考虑同步的必要性，消除不必要的同步，或者调整同步的顺序，以带来性能上的提升。

8. 程序的性能分析

我们可以调用 Window SDK 中的函数（位于 kernel32.dll 中，对应于 Kernel32.lib）：

```
BOOL QueryPerformanceFrequency( LARGE_INTEGER *lpFrequency );
```

```
BOOL QueryPerformanceCounter( LARGE_INTEGER *lpPerformanceCount );
```

来获取 CPU 的主频和当前计数，通过程序运行前后的计数差，可以获得程序所用的 CPU 周期数，再除以主频值就得到耗时的秒数。

这两个函数的参数都是用于输出的指针类型，LARGE_INTEGER 是 SDK 中定义的一种联合类型，用于表示 64 位有符号整数（对应的基本类型为 __int64 或 long long）

```

typedef union _LARGE_INTEGER {
    struct {DWORD LowPart; LONG HighPart;};
    struct {DWORD LowPart; LONG HighPart; } u;
    LONGLONG QuadPart;
} LARGE_INTEGER;

```

如果使用 64 位整数变量来调用这两个函数，则必须进行强制类型转换。例如：

```
long long freq, countBegin, countEnd;
```

```
QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
```

```
QueryPerformanceCounter((LARGE_INTEGER *)&countBegin);
```

.....

```
QueryPerformanceCounter((LARGE_INTEGER *)&countEnd);  
printf("CPU 频率: %lld, 计数: %lld, 耗时: %f 秒\n", freq, countEnd - countBegin,  
       (countEnd - countBegin) / (double)freq);
```

注意，为了打印输出 64 位整数，需要在 printf 的格式字符串的整数类型指示符 d 之前，加上两个 ll（只有一个 l 代表 long 类型），如 "%lld"。

在我的 PC 机（Intel Core 2 E6600 双核、主频 2.4GHz、4GB 内存，Windows XP SP3 中文专业版）上，使用 Visual C++ 2008 SP1 编译运行。

在 NUM_THREADS = 2、num_steps = 100,000（十万）时，求 π 各程序的耗时结果：（对并行版本，每次的运行结果可能有少许差别）



```
C:\WINDOWS\system32\cmd.exe  
Pi = 3.141573  
CPU频率: 2397630000, 计数: 7900659, 耗时: 0.003295秒  
请按任意键继续. . .
```

串行




```
C:\WINDOWS\system32\cmd.exe  
Pi = 3.141593  
CPU频率: 2397630000, 计数: 13879080, 耗时: 0.005789秒  
请按任意键继续. . .
```

并行 1



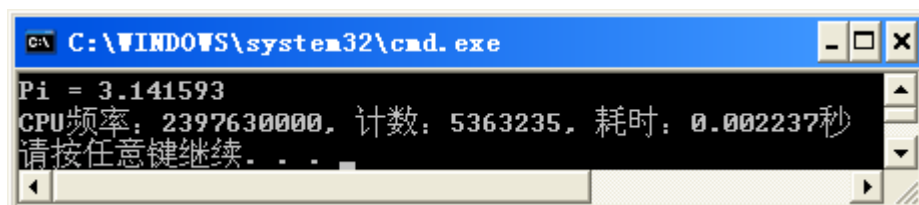
```
C:\WINDOWS\system32\cmd.exe  
Pi = 3.141593  
CPU频率: 2397630000, 计数: 6543027, 耗时: 0.002729秒  
请按任意键继续. . .
```

并行 2



```
C:\WINDOWS\system32\cmd.exe  
Pi = 3.141593  
CPU频率: 2397630000, 计数: 5467905, 耗时: 0.002281秒  
请按任意键继续. . .
```

并行 3



```
C:\WINDOWS\system32\cmd.exe  
Pi = 3.141593  
CPU频率: 2397630000, 计数: 5363235, 耗时: 0.002237秒  
请按任意键继续. . .
```

并行 4

可见，并行 1（并行区域）最慢（5.8ms）、串行第二慢（3.3ms）、其余差不多都在 2~3ms。

若将 num_steps 改为 1,000,000（百万），则结果为串行：33ms、并行 1：51ms、并行 2：50ms、并行 3：21ms、并行 4：21ms，可见，并行 1 和 2 都很慢、串行次之、并行 3 和 4 快。

1.8.2 PPL

VS10 中本地 C++ 的 PPL（Parallel Pattern Library，并行模式库）通过方法调用提供并行构建。下面的内容取自微软的 MSDN 在线帮助。

在线（英文）帮助文档的位置：MSDN\MSDN Library\Development Tools and Languages\Visual Studio 2010 Beta 1\Visual Studio\Visual C++\Visual C++ Reference\Concurrency Runtime\Parallel Patterns Library (PPL)\，对应网址：

[http://msdn.microsoft.com/en-us/library/dd492418\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd492418(VS.100).aspx)

1. 概述

PPL 提供了一种命令式的编程模型，提升了开发并行（concurrent，并发/同时）应用程序的可伸缩性和易用性。PPL 建立在并行运行时（Concurrency Runtime）的调度和资源管理之上。PPL 通过提供作用于并行数据的泛型、类型安全算法和容器，提高了你的应用程序与底层线程机制之间的抽象级别。PPL 还通过提供对共享状态的选择性，使你能够开发不通规模的应用程序。

PPL 提供下列特性：

- 任务并行性：一种并行执行若干工作项（任务）的机制。
- 并行算法：可用于并行数据集合的泛型算法。
- 并行容器和对象：可对其元素提供安全访问的泛型容器类型。

PPL 提供了一种类似于标准模板库（STL）的编程模型。例如，考虑下面对数组对象中的每个元素执行操作的程序，该例首先使用 STL 的 for_each 算法遍历数组对象中的元素，然后使用 PPL 的 parallel_for_each 函数来并行完成同样的任务。

```
// print-array.cpp
// compile with: /EHsc
#include <ppl.h>
#include <array>
#include <algorithm>

using namespace Concurrency;
using namespace std;
using namespace std::tr1;

int main() {
    // 创建包含一些元素的数组对象
    array<int, 3> a = {13, 26, 39};
    // 使用 for_each 算法串行打印数组的每个元素
```

```

printf_s("Using for_each:\n");
for_each(a.begin(), a.end(), [&](int n) {
    printf_s("%d\n", n);
});
// 使用 parallel_for_each 来完成同样的任务
printf_s("Using parallel_for_each:\n");
parallel_for_each(a.begin(), a.end(), [&](int n) {
    printf_s("%d\n", n);
});
}

```

下面是该例的输出：

```

Using for_each:
13
26
39
Using parallel_for_each:
13
39
26

```

由于诸如 `parallel_for_each` 的并行算法同时行动，此例的并行版本可能产生与串行版不同的输出。

2. 任务并行性

描述如何使用并行任务机制，如有结构和无结构的任务组。

3. 并行算法

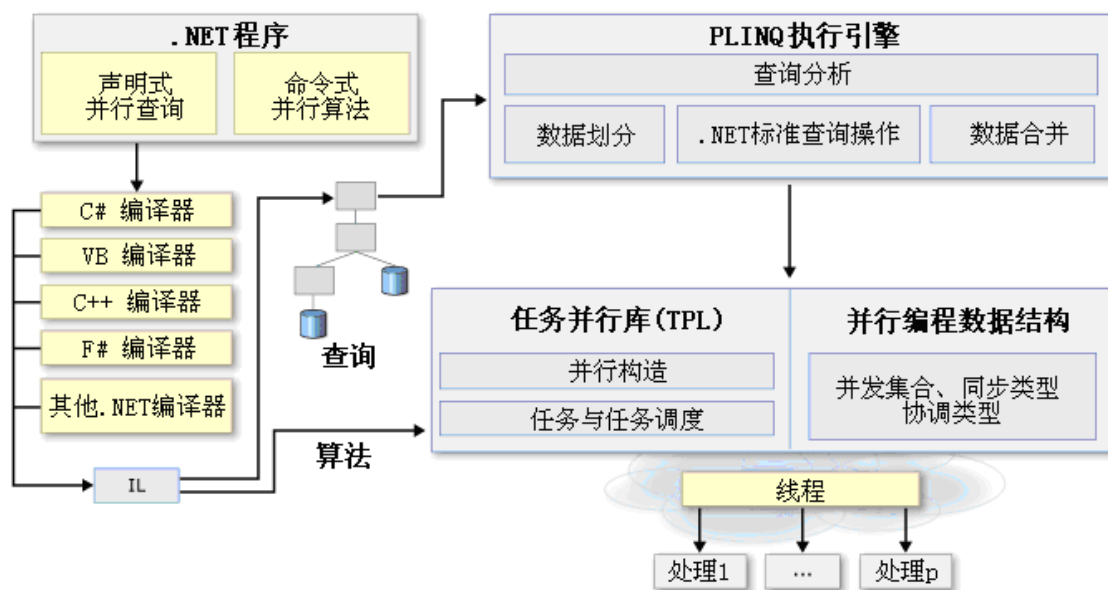
描述如何使用并行算法，如 `parallel_for` 和 `parallel_for_each`。

4. 并行容器和对象

描述 PPL 提供的各种并行容器和对象。

1.9 Visual C# .NET 多核编程

.NET 4.0 引入了若干并行编程新技术，主要包括 TPL（Task Parallel Library，任务并行库）、PLINQ（Parallel LINQ，并行 LINQ）和用于并行编程的若干数据结构。TPL 是 PLINQ 的基础。参见下图：



.NET 4.0 的并行编程架构

本节将简介 TPL 的基本内容和编程方法，主要取自微软的 .NET 框架 4.0 Beta1 和 Visual Studio 2010 Beta1 的 MSDN 在线（英文）帮助文档。

TPL 帮助的位置：MSDN\MSDN Library\ .NET Development\Beta Versions and Previews\ .NET Framework Advanced Development\Parallel Programming\Task Parallel Library Overview，对应网络地址为：

[http://msdn.microsoft.com/en-us/library/dd460717\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd460717(VS.100).aspx)

Visual Studio 2010 Beta 1 的帮助位置：MSDN\MSDN Library\Development Tools and Languages\Visual Studio 2010 Beta 1\，对应网络地址为：

[http://msdn.microsoft.com/en-us/library/dd410112\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd410112(VS.100).aspx)

另外，我的个人网页上有 Visual Studio 2010 Beta1 的英文专业版（含 .NET Framework 4.0 Beta1 版，但不含 MSDN）可供下载(1.13 GB)。

PLINQ 的内容将在第 14 章“数据库编程”中再介绍。

1.9.1 TPL

1. 概述

TPL（Task Parallel Library，任务并行库）是对 .NET 框架 4.0 引进的一组位于 System.Threading 和 System.Threading.Tasks 命名空间的共用类型和 API 的称呼。这些类型依赖于一种与 .NET ThreadPool（线程池）集成在一起的新任务调度程序。TPL 的目标是，通过简化添加并行性和并发性到应用程序的过程，来提高开发人员的生产力。

TPL 可动态调整并发度，以最有效地利用所有可用的处理器。基于 TPL 的并行代码不仅可以在今天的双核和四核计算机上运行，而且不需要重新编译，它就可以自动扩展到即将成为桌面标准的更多核计算机。

你使用 TPL 编写多线程循环，非常类似于编写串行循环。下面的代码会基于机器中的处理器数目将一项工作自动划分给多个任务：（C# 代码）

```
Parallel.For(startIndex, endIndex, (currentIndex) => DoSomeWork(currentIndex));
```

利用 `Parallel.Invoke()`，你可以用单一的方法调用，来请求任意数量的异步并行操作：

```
Parallel.Invoke(  
    () => MethodA(),  
    () => MethodB(),  
    () => MethodC() );
```

利用 TPL 来并行化你的代码，你能够聚焦于你设计的程序所要完成的工作本身。然而，并不是所有的串行循环都是并行化的好候选。重要的是理解，任何多线程代码，并行循环可能在你的程序执行中引入复杂性。虽然 TPL 简化了多线程的方案，但是为了有效地利用它，你仍然应该对诸如锁、死锁和竞争条件等线程概念有一个基本的了解。

在 TPL 中，基本的抽象是任务（Task），而不是线程。任务是 Task 类的实例，可以被取消和等待、可以返回值、可以在其完成时调用另一个任务。当你使用 `Parallel.For` 和 `Parallel.ForEach` 时，甚至 Task 对象本身也是隐式的。在你的代码中，你只需简单地提供完成所期望工作的委托，其余的都由 TPL 来负责处理。在默认情况下，TPL 使用它自己的任务调度程序，该程序是与 .NET ThreadPool 集成在一起的。然而，你也可以提供自己定制的任务调度程序，使用另外的线程调度机制。在任务和线程之间不存在固定的关系，一个线程可以在任何给定的并行代码块中接连运行几个任务，一个任务也可以定义一个在同一线程或不同线程上运行的子任务。一个任务也可以调用在别处被定义的另一任务。

你还可以通过并行 LINQ（PLINQ），用说明性语法来访问并行功能。在内部，PLINQ 是与 TPL 紧密集成在一起的。

1) 数据并行性：Parallel 类

一个通用编程模型使用循环来对数据源中的每一个元素完成同样的一个或一组动作。在数据的并行操作中，源集合被划分，以便多个线程可以同时操作不同部分段。TPL 用 `System.Threading.Parallel` 类来支持数据并行性，该类提供基于方法的 `for` 和 `foreach` 循环的并行实现。你编写一个针对 `ParallelFor()` 或 `ParallelForEach()` 循环的循环逻辑，非常像你编写一个串行的循环。你不必创建线程或队列工作项，在基本的循环中也不需要获取锁，TPL 为你处理所有的底层工作。下面的代码例子展示了一个简单的 `foreach` 循环及其并行的等价代码：

```
// 串行版  
foreach (var item in sourceCollection) { Process(item); }  
  
// 并行等价版  
Parallel.ForEach(sourceCollection, item => Process(item));
```

在一个并行循环运行时，TPL 划分数据源使循环能够同时操作多个部分。在背后，Task Scheduler 根据系统资源和工作量来划分任务。在可能情况下，调度程序会在工作量变得不平衡时，重新分配工作给多个线程和处理器。

注意，你也可以在必要时，提供自己定制的划分程序或调度程序。

`ParallelFor()` 和 `ParallelForEach()` 方法都有几个重载，使你能够停止或中断循环的执行、监控其它线程上的循环状态、维护线程局部（thread-local）状态、终结线程局部对象、控制并行度、等等。提供此功能的助手（helper）类型包括 `ParallelLoopState`、`ParallelOptions` 和 `ParallelLoopResult`、`CancellationToken` 和 `CancellationTokenSource`。

2) 任务并行性: **Parallel.Invoke** 与 **Task**

In task parallelism, multiple distinct operations are performed concurrently on the same or different sources. For example, one might search a chunk of text for the most frequently occurring word, while at the same time searching for the longest word, and also analyzing the content with some proprietary algorithm. In traditional .NET Framework applications, you performed this concurrent work by using the ThreadPool or by explicitly creating and starting threads. With the TPL, you think in terms of tasks, not threads.

The `Parallel.Invoke()` method provides a convenient way to run any number of arbitrary statements concurrently. Simply pass in an Action delegate for each item of work. The easiest way to create these delegates is to use lambda expressions. The lambda expression can either call a named method, or provide the code inline. The following example shows a simple `Invoke()` call that creates and starts two tasks that run concurrently:

C#Copy Code

```
Parallel.Invoke(() => DoSomeWork(), () => DoSomeOtherWork());
```

Note

The number of tasks that are created behind the scenes by `Parallel.Invoke` is not necessarily equal to the number of delegates that are provided. TPL may employ various optimizations, especially with large numbers of delegates.

For greater control over task execution, such as canceling, waiting on, and continuing from a task, you have to work with task objects more explicitly.

(1) 任务介绍

Introduction to Tasks

The Task Parallel Library, as its name implies, is based on the notion of the task. A task is represented by the `System.Threading.Tasks.Task` class. A task that returns a value is represented by the `System.Threading.Tasks.Task<Of <(TResult)>>` class, which inherits from `Task`. The `Task` class has methods that enable you to start, cancel and wait on a task instance. The static `WaitAny` and `WaitAll` methods enable you to wait on multiple tasks. Both `Task` and `Task<Of <(TResult)>>` have overloads of a `ContinueWith` method to specify a task to be invoked when the current task completes. In the following example, the tasks are created by using the static `StartNew` method that creates the task and starts it in a single operation. This is the preferred way to create and start tasks if creation and scheduling do not need to be separated. Because the tasks are of type `Task<double>` they each have a public `Result` property that contains the result of the computation. The tasks run asynchronously and may complete in any order. In the for loop, the `Result` property blocks until the task completes.

C#Copy Code

```
Task<double>[] taskArray = new Task<double>[]
{
    Task<double>.Factory.StartNew(() => DoComputation1()),
    Task<double>.Factory.StartNew(() => DoComputation2()),
    Task<double>.Factory.StartNew(() => DoComputation3())
};
```



```
double[] results = new double[taskArray.Length];
for(int i = 0; i < taskArray.Length; i++)
    results[i] = taskArray[i].Result;
```

(2) 任务继续

Task Continuations

The `TaskContinueWith()` method enables you to specify a task to be started when the current task (the antecedent task) completes. When `ContinueWith()` returns, the continuation task has been created and initialized. It is started only when the antecedent task completes. At that time, the continuation task receives as input a reference to the antecedent task, so that it can examine its properties, for example `IsFaulted()` to determine whether the task threw an unhandled exception. In addition, a user-defined value can be passed from the antecedent to its continuation task in the `Result` property, so that the output of the antecedent task can serve as input for the continuation. In the following example, `getData` is started by the program code, then `analyzeData` is started automatically when `getData` completes, and `reportData` is started when `analyzeData` completes. `getData` produces as its result a byte array, which is passed into `analyzeData`. `analyzeData` processes that array and returns a result whose type is inferred from the return type of the `Analyze` method. `reportData` takes the input from `analyzeData`, and produces a result whose type is inferred in a similar manner and which is made available to the program in the `Result` property.

C#Copy Code

```
Task<byte[]> getData = new Task<byte[]>(() => GetFileData());
Task<double[]> analyzeData = getData.ContinueWith(x => Analyze(x.Result));
Task<string> reportData = analyzeData.ContinueWith(y => Summarize(y.Result));
getData.Start();
System.IO.File.WriteAllText(@"C:\reportFolder\report.txt", reportData.Result);
```

Note

When dealing with tasks, you have to distinguish between the antecedant-continuation relationship and the parent-child relationship. If a task creates another task, it is that task's parent. An antecedent task differs from a parent task.

The `ContinueWhenAll()` and `ContinueWhenAny()` methods enable you to continue from multiple tasks.

(3) 继续选项

Continuation Options

Several overloads of the `ContinueWith` method takes a `System.Threading.Tasks.TaskContinuationOptions` enumeration. This enumeration enables you to specify one or more conditions to the continuation behavior. For example, you can use the `OnlyOnFaulted` value to specify that the continuation only run if the antecedent task threw an unhandled exception. The `DetachedFromParent` value is used to prevent the child task from holding a reference to the parent task, which is useful when the lifetime of the child is expected to extend beyond that of the parent. Several other options are provided. For more information, see `System.Threading.Tasks.TaskContinuationOptions`.

(4) 等待任务

Waiting on Tasks

The `System.Threading.Tasks.Task` and `System.Threading.Tasks.Task<Of <(TResult)>>` types each provide several overloads of a `Wait` method that enable you to wait for the task to complete. Some overloads enable you to specify a timeout, and others take an additional `CancellationToken` as an input parameter.

In addition, overloads of the static `Task.WaitAll()` and `Task.WaitAny()` methods enable you to wait for any or all of an array of tasks to complete.

C#Copy Code

```
Task[] tasks = new Task[3]
{
    Task.Factory.StartNew(() => MethodA()),
    Task.Factory.StartNew(() => MethodB()),
    Task.Factory.StartNew(() => MethodC())
};
Task.WaitAll(tasks);
```

When you wait on a task, you implicitly wait on all children of that task, unless those children were created by using the `DetachedFromParent` option. Any detached children must be waited on separately. `Wait` returns immediately if the task has already completed. Any exceptions raised by a task will be thrown by a `Wait` method, even if the `Wait` method was called after the task completed.

The `Wait` methods provide the means by which you handle exceptions that are thrown by tasks, as discussed in the following section.

(5) 在任务中处理异常

Handling Exceptions in Tasks

When a task or group of tasks throws one or more exceptions, the exceptions are wrapped in a `AggregateException` and propagated back to the thread that invoked the task. The calling code can handle the exceptions by using `Wait`, `[M:System.Threading.Tasks.Task.WaitAll, or M:System.Threading.Tasks.Task.WaitAny]` on the task or group of tasks, and enclosing the `Wait` method in a try-catch block. The calling code can iterate over the inner exceptions and handle each individually, or use the `Handle()` method. The `Flatten()` method returns a single `AggregateException` that directly contains all the inner exceptions that were contained by it and by any nested `AggregateExceptions`.

Calling code can also handle exceptions by accessing the `Exception` property before the task is garbage-collected. By accessing this property, you prevent the unhandled exception from being rethrown when the object is disposed.

(6) 取消任务

Cancelling Tasks

Note

The following section applies to Visual Studio 2010 Beta 1. Task cancellation in later releases is expected to change to conform more closely to the new unified cancellation model in the .NET Framework 4.0.

In Visual Studio 2010 Beta 1, you cancel a running task by calling the `Cancel` method. When you call `Cancel`, the Task's `IsCancellationRequested` property is set to true, but the task does not necessarily stop or transition to the `Canceled` state immediately.

In Beta 1, the recommended approach is to use a `System.Threading.CancellationTokenSource` to create a `System.Threading.CancellationToken`, and use the `Register` method to provide a delegate that calls `Cancel` on the task instance when the token itself is cancelled. In the code that initiates the cancellation, call `Cancel`, which will set the `IsCancellationRequested` property on the token to true. In your task delegate, you can monitor the property at regular intervals and respond to the cancellation request as appropriate for your application. This approach is closer to how cancellation is expected to work in later releases. For more information, see [How to: Cancel a Task](#).

(7) 定制

Customization

As an application or library developer, you generally do not care which processor the task runs on, or how it synchronizes its work with other tasks or is scheduled onto the `System.Threading.ThreadPool`. You only require that it execute with the greatest degree of concurrency possible on the host computer. For those cases where you require more fine-grained control over the scheduling details, the TPL enables you to configure some settings on the default task scheduler, and even enables you to supply a custom scheduler. For more information, see [TaskScheduler](#).

3) 有关数据结构

Related Data Structures

The TPL uses several new public types that are useful in both parallel and sequential scenarios. These include several thread-safe and fast collection classes in the `System.Collections.Concurrent` namespace, and several new synchronization types such as `SemaphoreLock` and `[T:System.Threading.ManualResetEventSlim, which]` are more efficient than their predecessors for specific kinds of workloads. Other new types in the .NET Framework 4.0, such as `System.Threading.Barrier` and `System.Threading.SpinLock`, provide new functionality that was not available in previous releases. For more information, see [Data Structures for Parallel Programming](#).

1.9.2 PLINQ

PLINQ (Parallel LINQ, 并行 LINQ) ——LINQ 引擎的并行实现, 放在第 14 章 “数据库编程” 中介绍。

1.9.3 并行编程数据结构

Data Structures for Parallel Programming

The .NET Framework 4.0 introduces several new types that are useful in parallel

programming, including a set of concurrent collection classes, lightweight synchronization primitives, and types for lazy initialization. You can use these types with any multithreaded application code, including the Task Parallel Library and PLINQ.

1. 并行集合类

Concurrent Collection Classes

The collection classes in the `System.Collections.Concurrent` namespace provide thread-safe add and remove operations that avoid locks wherever possible and use fine-grained locking where locks are necessary. Unlike collections that were introduced in the .NET Framework versions 1.0 and 2.0, a concurrent collection class does not require user code to take any locks when it accesses items. The concurrent collection classes can significantly improve performance over types such as `System.Collections.ArrayList` and `System.Collections.Generic.List<T>` (with user-implemented locking) in scenarios where multiple threads add and remove items from a collection.

The following table lists the new concurrent collection classes:

Type	Description
<code>System.Collections.Concurrent.BlockingCollection<T></code>	Provides blocking and bounding capabilities for thread-safe collections that implement <code>System.Collections.Concurrent.IProducerConsumerCollection<T></code> . Producer threads block if no slots are available or if the collection is full. Consumer threads block if the collection is empty. This type also supports non-blocking access by consumers and producers. <code>[T: System.Collections.Concurrent.BlockingCollection<T>]</code> can be used as a base class or backing store to provide blocking and bounding for any collection class that supports <code>IEnumerable<T></code> .
<code>System.Collections.Concurrent.ConcurrentBag<T></code>	A thread-safe bag implementation that provides scalable add and get operations.
<code>System.Collections.Concurrent.ConcurrentDictionary<TKey, TValue></code>	A concurrent and scalable dictionary type.
<code>System.Collections.Concurrent.ConcurrentQueue<T></code>	A concurrent and scalable FIFO queue.
<code>System.Collections.Concurrent.ConcurrentStack<T></code>	A concurrent and scalable LIFO stack.

For more information about how to use the concurrent collection classes, see [How to: Implement a Parallel Producer-Consumer Pattern](#).

2. 同步原语

Synchronization Primitives

The new synchronization primitives in the System.Threading namespace enable fine-grained concurrency and faster performance by avoiding expensive locking mechanisms found in legacy multithreading code. Some of the new types, such as Barrier and CountdownEvent have no counterparts in earlier releases of the .NET Framework.

The following table lists the new synchronization types:

Type	Description
System.Threading.Barrier	Enables multiple threads to work on an algorithm in parallel by providing a point at which each task can signal its arrival and then block until some or all tasks have arrived.
System.Threading.CountdownEvent	Simplifies fork and join scenarios by providing an easy rendezvous mechanism.
System.Threading.ManualResetEventSlim	A synchronization primitive similar to System.Threading.ManualResetEvent. ManualResetEventSlim is lighter-weight but can only be used for intra-process communication.
System.Threading.SemaphoreSlim	A synchronization primitive that limits the number of threads that can concurrently access a resource or a pool of resources.
System.Threading.SpinLock	A mutual exclusion lock primitive that causes the thread that is trying to acquire the lock to wait in a loop, or spin, for a period of time before yielding its quantum. In scenarios where the wait for the lock is expected to be short, SpinLock offers better performance than other forms of locking.
System.Threading.SpinWait	A small, lightweight type that will spin for a specified time and eventually put the thread into a wait state if the spin count is exceeded.

3. 迟缓初始化类

Lazy Initialization Classes

With lazy initialization, the memory for an object is not allocated until it is needed. Lazy initialization can improve performance by spreading object allocations evenly across the lifetime of a program. You can enable lazy initialization for any custom type by wrapping the type Lazy<T>.

The following table lists the lazy initialization types:

Type	Description
System.Lazy<T>	Provides lightweight, thread-safe lazy-initialization.
System.Threading.ThreadLocal<T>	Provides a lazily-initialized value on a per-thread basis, with each thread lazily-invoking the initialization function.
System.Threading.LazyInitializer	Provides static methods that avoid the need to allocate a dedicated, lazy-initialization instance. Instead, they use r

	ferences to ensure targets have been initialized as they are accessed.
--	--

参考文献

- 多核系列教材编写组（陈天洲等）. 多核程序设计. 清华大学出版社, 2007 年 9 月. 16 开/283 页/36 元。
- Shameem Akhter & Jason Roberts 著（李宝峰等译）. 多核程序设计技术——通过软件多线程提升性能. 电子工业出版社, 2007 年 3 月. 16 开/351 页/49 元（英文版原书: Multi-Core Programming : Increasing Performance through Software Multi-threading. Copyright © 2006 Intel Corporation）。
- 郑宏. 多核架构及编程技术. 武汉大学电子信息学院（武汉大学智能计算与智能系统联合实验室）课件, 2008 年 9 月。网址: <http://icis.whu.edu.cn/03.html>（底部, 有第 1~4 和 6~7 章）
- MSDN 本地和在线帮助文档。