OPEN CASCADE 学习笔记
——并行程序开发

著: **Roman Lygin**

译：**George Feng**

这是一篇关于开源三维建模软件 OPEN CASCADE 内核的博文：ROMAN LYGIN 是 OPEN CASCADE 的前程序开发员和项目经理，曾经写过许多关于该开源软件开发包的深入文章，可以在他的博客(HTTP://OPENCASCADE.BLOGSPOT.COM)上面找到这些文章。

序

在 **Open Cascade** 的论坛上知道了 **Roman Lygin** 在他的博客上写了 **Open Cascade notes** 系列文章，考虑到 **Open Cascade** 的学习资料并不多，于是从他的博客上下载了其中绝大部分文章，将其翻译过来以方便大家学习交流。如果大家发现文中翻译有错误或不足之处，望不吝赐教，可以发到我的邮箱 **fenghongkui@sina.com.cn**，十分感谢。

2012 年 **11** 月 **22** 日星期四

第 1 节 并行程序开发综述

正如在之前的文章中提到的，我正在开发 CAD Exchanger 的 ACIS 导入部分，并且将其开发成并发执行的。到目前为止结果非常理想(除了 STL 的流解析，因为之前文章中提到的 Microsoft 的 bug，我安装了 VS2008SP1 正在检测这个错误是否得到了修正)。所以我准备在这篇文章中分享我的经验，希望能够对其他开发人员有帮助(有关并行性问题在论坛上讨论的也越来越多)。也希望 Open CASCADE 小组能够从我的发现中受益。

我之前已经简短介绍过几次并行应用程序开发了，再强调一下在多核时代并行应

用程序将在某个时期成为主流，你最好现在就准备好应对这个趋势。这对你的职业路线是非常有助的，这些能力将增强你的竞争力。最近发布的 Intel Parallel Studio(它已经成为我的工具箱的很重要的部分)可以调试多线程应用程序，简化开发人员的工作。关于并行程序开发这个方面有很多基础书籍。我现在在读的是 Timothy Mattson 等写的《Patterns for Parallel Programming》(我的一位这方面非常熟练的同事推荐的)，还有 Erich Gamma 写的《Design Patterns》也是类似的书(所有专业软件开发人员都需要读的书)。它帮助我设计 CAD Exchanger 中 ACIS 导入部件的构架。

回到 Open CASCADE，我可以明确的说 Open CASCADE 对于并行应用程序非常有用，但是正如其他软件库一样需要小心使用。

一般性评述

假如从较高的角度看问题，从问题领域看，而不是从一个特定的算法的角度看。看看什么能够并发进行，什么需要顺序执行。例如，在我的初始 IGES 转换(参考这里)实验中，我特别注意了 IGES 组的并行转换模块，这个模块运行的非常好。但是整个程序的性能提高非常少，因为不是这部分占用的时间最多，而是 Shape Healing 占用了整个时间的 50%-70%，而且该部分仍然是顺序执行的。所以在重新设计算法之前，性能不会提高太多。我不得不重构 ACIS 读取器，重新设计传统的瀑布模型结构(其中遍历模型从根节点到叶节点)，然后是 shape healing 模块。由此产生了很多有价值的东西。《Design Patterns》这本书具有很好的指导作用，可以帮助确定算法的模式。

句柄

从 6.2.x 开始 Open CASCADE 的句柄(Handle_Standard_Transient 的子类)已经对引用计数实现了线程安全。要利用这个优点，你必须定义 MMGT_REENTRANT 系统变量为非空，或者在线程之间使用句柄之前调用 Standard::SetReentrant (Standard_True)，这使引用计数依赖于原子态的增加/减小(例如 Windows 上的 InterlockedIncrement())，而不是++，这可能并不是原子态的。

```
void Handle(Standard_Transient)::BeginScope()
{
if (entity != UndefinedHandleAddress)
{
if ( Standard::IsReentrant() )
```

```
Standard_Atomic_Increment (&entity->count);
else
entity->count++;
}
}
```

内存管理

缺省情况下，Open CASCADE 将变量 MMGT_OPT 定义为 1。这意味着所有的对于 Standard::Allocate()和::Free()的调用都最终到达 Open CASCADE 的内存管理器(Standard_MMgrOpt)，其会优化内存分配，消除内存碎片 (可能这值得另外专门写一篇文章，讨论内存管理的内部实现机制)。

Standard_MMgrOpt 自身是线程安全的，并且安全的处理几个线程的并发内存分配/释放请求。然而它是基于 Standard_Mutex 的(将来会有更多基于这个类的实现)，其在当前的实现中具有比较多的额外花费，这使得在并行程序中对于内存管理的优化作用显得无足轻重(然而在单线程环境中它运行的很好)。

所以为了克服这个缺陷应该使用 MMGT_OPT=0，这就会激活 Standard_MMgrRaw，它会简化前向调用(forward calls)malloc/free...

(待续...)

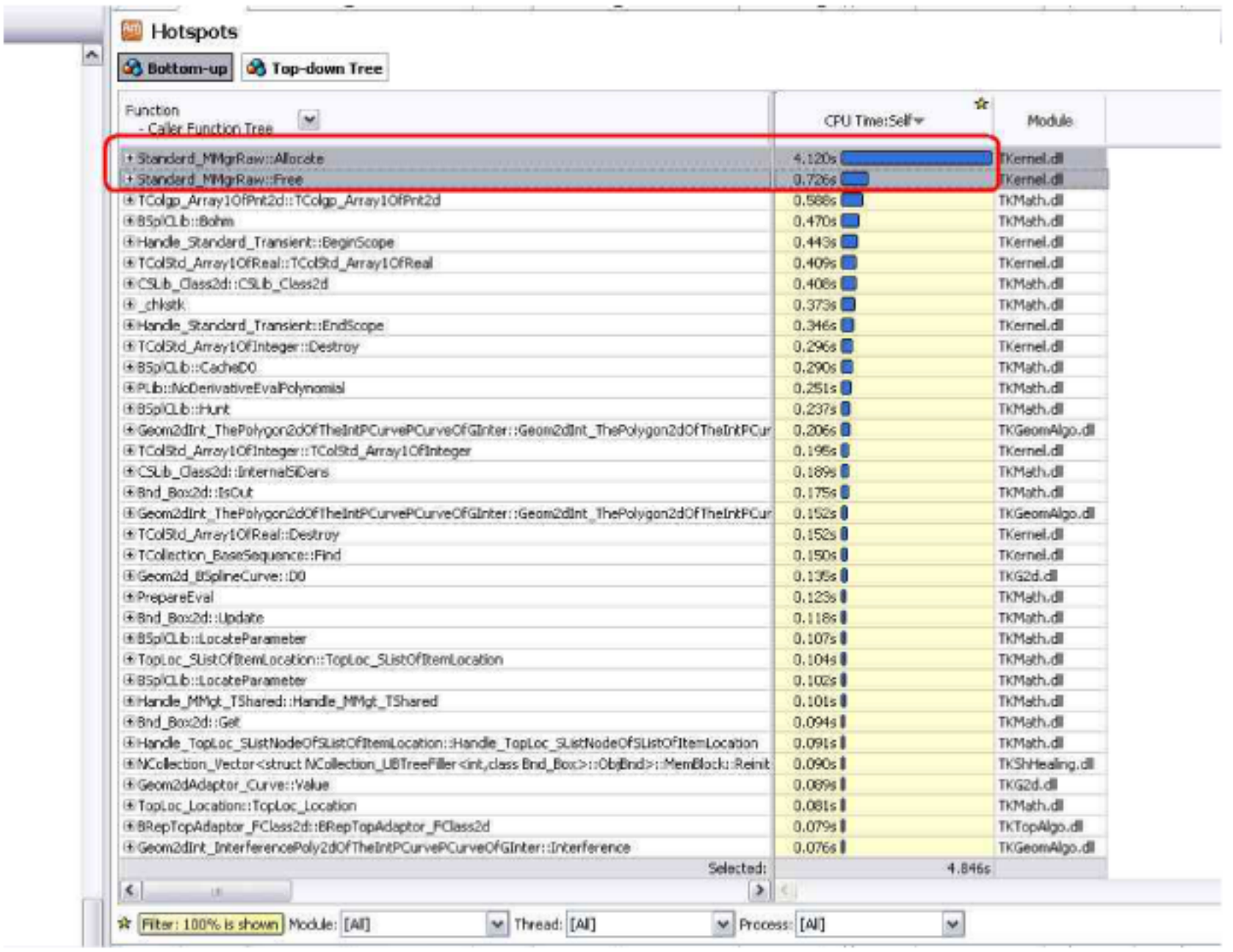
POSTED BY ROMAN LYGIN AT 19:50, 2009-06-09

第 2 节 内存管理器

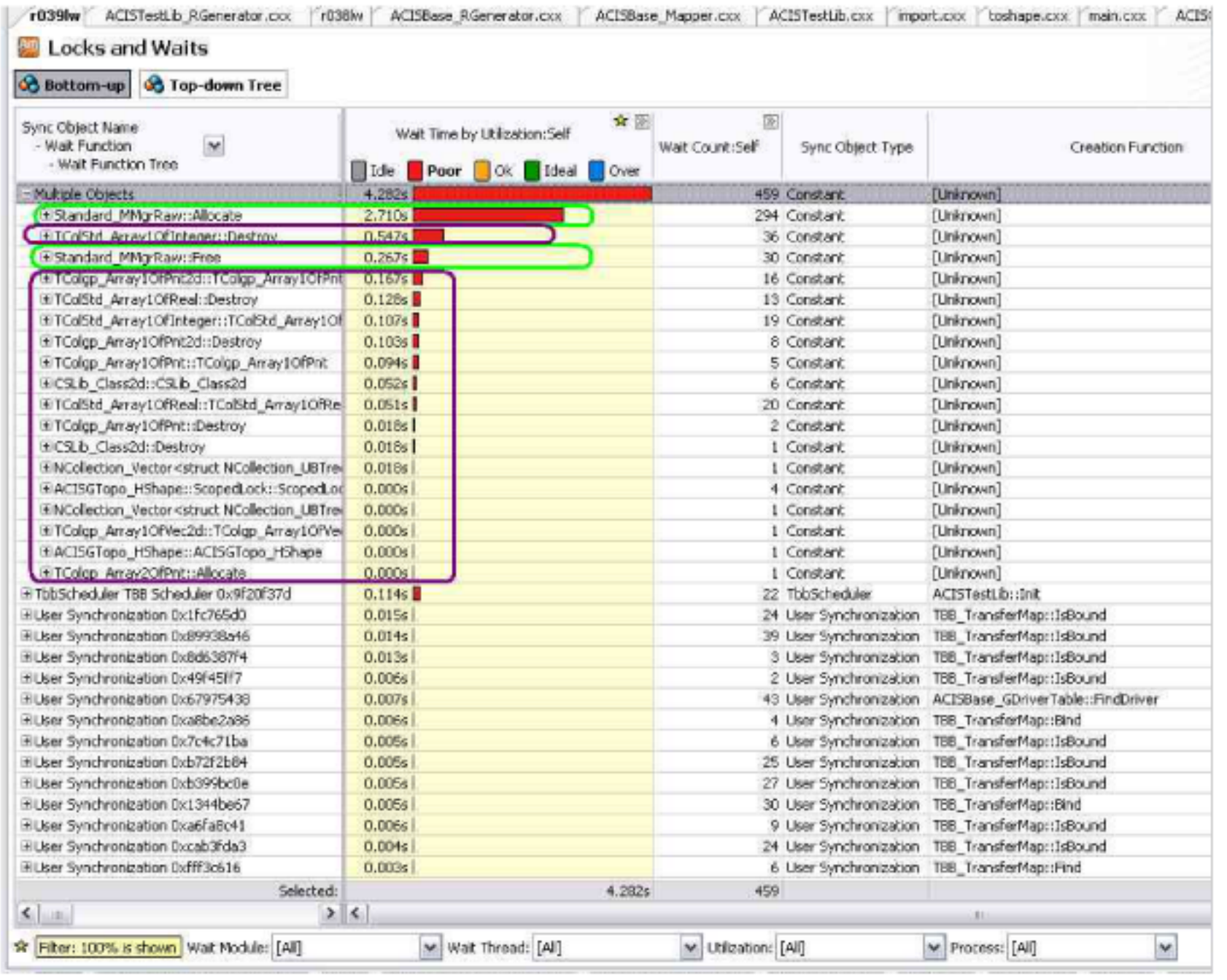
(接上节...)

顺便说一下，在应用程序启动之前必须设置环境变量(MMGT_OPT)，以及其他控制 Open CASCADE 的内存管理的环境变量。这非常重要，因为内存管理的选择是在加载 DLL 文件之前开始的，在运行时无法改变用户的内存管理策略，这也是一个非常不方便的限制。

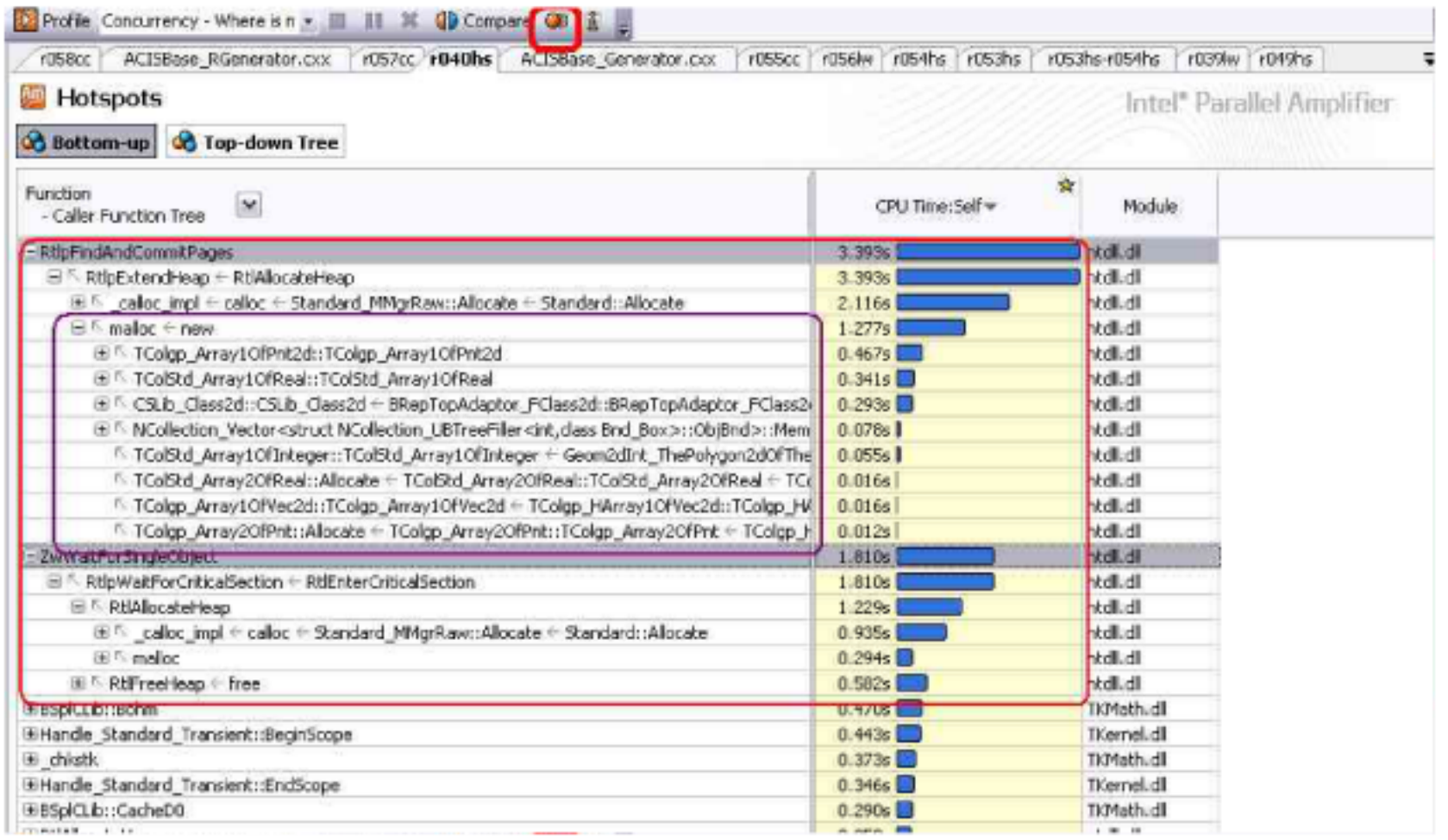
在应用程序的声明周期中，由于扩展内存的分配和释放可能会成为一个热点区域。例如，下图是 CAD Exhcanger ACIS 转换的热点分析截图。



直接使用操作系统的内存管理时(正如前面分析的不能使用 Open CASCADE 的内存管理)，测量并发和等待&死锁 (另外两项分析是由 Amplifier 完成的)，可以看到也引发了最大等待时间。



一方面，这是个好的信号，就算说代码的其他部分都运行的非常快。但是另一方面，它表明内存管理确实成为一个瓶颈。随着深入分析这个问题，我切换到直接看操作系统函数的模式(按下 Amplifier 工具条的按钮) (toggling off the button on Amplifier toolbar)，下图是我看到的：



这说明了什么？两个系统函数——RtlpFindAndCommitPages() 和 ZwWaitForSingleObject() ——是热点区域，而且追溯堆栈调用结果显示都是从内存分配/释放函数中调用的它们！

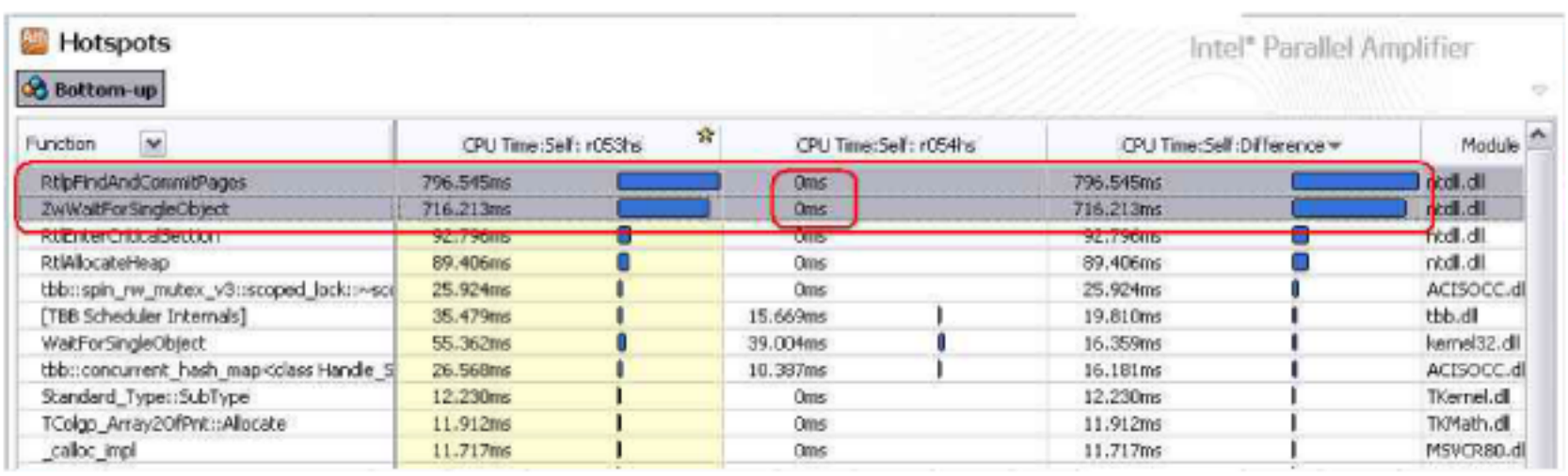
第一个热点是由于 ACIS 转换器生成了大量的小对象，这些对象保存了转换的结果(在这个特有的测试例子中是 22000+个)。这引起了大量的内存碎片，强迫系统不断的寻找新的内存块。

第二个热点(运行通过了临界区)是由于 Windows 系统缺省的内存管理机制引发的。你可能直到，Windows 上的堆内存分配是顺序完成，使用了互斥量(mutex)(邻接区) (critical section)，要完成内存分配需要花费大约 4000 个调用请求分配内存的时间，例如一个线程请求内存分配，而同时另一个线程也在做同样的事，后者并没有马上执行，而是进入睡眠模式，但是循环 4000 次才能让前者调用完成。

所有的这些问题都是由于直接使用 calloc/malloc/free 和 new/delete 引发的，要消除这个问题，我已经尝试了由 Intel Threading Building Blocks 2.2 提供的一种技术，使用该计数就可以将所有的 C/C++内存管理的调用替换成用 TBB 分配调用。这很容易通过包含下面的声明就可以做到了。

```
#include "tbb/tbbmalloc_proxy.h"
```

TBB 分配器并行运行(而不在内部锁定)，而且与 Open CASCADE 的工作方式相似——只重用一次，分配块但是不将它们返回到操作系统。这解决了热点问题，而且具有更高的速度！下面是对比结果：



另一方面，需要注意在之前的图中只有 Standard_MMgrRaw::Allocate()和::Free()的调用(它是通过设置 MMGT_OPT=0 调用 Standard::Allocate()和::Free())。也有一些生成数组时的直接调用 (例如 TColgp_Array1OfPnt2d 等)，还有其他的调用(用紫罗兰色高亮标出)，其对应于 new[]数组的调用操作，在 Open CASCADE 中没有对该类操作进行重新定义，从而绕过了 Open CASCADE 的内存管理器。Andrey Betenev 曾经给我指出过这个问题，下面是存在这个问题的证据。所以，OCC 团队应该采取措施修正这个被他们忽略的错误。

所以，总结如下：

- 缺省的(优化)Open CASCADE 内存管理器(设置 MMGT_OPT=1 时)是不能用于并行程序的；
- 原始的内存管理器(设置 MMGT_OP=0 时)直接调用操作系统的 malloc/free 会导致内存碎片并且在内存负载较大时存在瓶颈；
- 为了克服以上内存管理器的缺点，需要开发自己特有的内存管理器，例如 Intel TBB 从而代替操作系统的内存管理函数。

(待续... ...)

POSTED BY ROMAN LYGIN AT 20:38, 2009-06-23

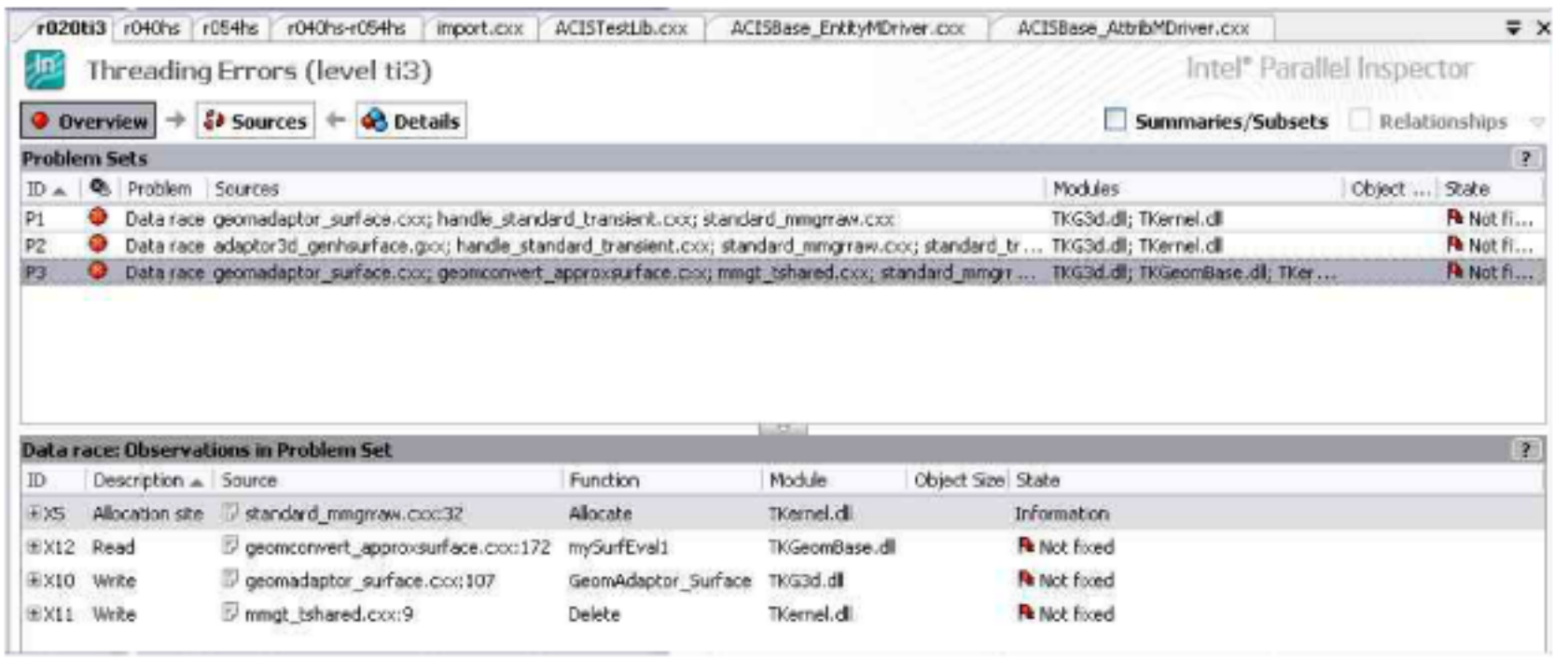
第 3 节 数据竞争

(接上节... ...)

数据竞争

当顺序执行程序变成并行执行程序时，数据竞争(Data races)是出现频率最高的问题。当你开发顺序执行的代码时，代码的执行通常是建立在各种不同的假定上的。可能需要一定的想象力才能理解代码在多线程环境中的工作方法，从而预测可能发生什么问题。

数据竞争可能导致不可预知的应用程序行为——有时候能得到正确的结果，有时候会出现不可重现的错误结果，甚至有时候会自己崩溃。假如你遇到了这种情况，很可能是在你的代码中或者是在第三方库中出现了数据竞争。要捕获该错误，可以使用 Intel Parallel Inspector，它能够确定不同的内存和线程错误。下图是 Inspector 检测器捕获到 Open CASCADE 错误时的样子：



假如 Open CASCADE 直接使用在并行应用程序中，会存在有许多会出现数据竞争的地方。我第一次提到 OpenCASCADE 的数据竞争是在去年实验 IGES 转换器的时候(可以在论坛上搜索到这个帖子)。在开发 ACIS 导入器时又发现了几个这样的错误。最经常出现数据竞争的例子如下：

a). 为函数中的静态变量返回 const&。例如：

```
const gp_Dir& gp::DX()
{
static gp_Dir gp_DX(1,0,0);
return gp_DX;
}
```

在单线程程序中上面的代码工作非常完美，但是在并行程序中可能产生问题。当两个线程同时第一次调用 gp::DX()时，就会出现同时写入 gp_DX 变量，结果就

不可预测了。要修正这个错误就把定义写到函数的外面，从而在程序加载时就初始化变量。

```
static gp_Dir gp_DX(1,0,0);
const gp_Dir& gp::DX()
{
return gp_DX;
}
```

b). 使用在文件中定义的静态变量。

这些情况从强制使用(forced use)到错误使用(ugly usages)都有可能出现，当变量不能作为参数列表或者类成员的一部分时就需要强制使用静态变量，当对 C++ 理解不够透彻时就会出现错误使用静态变量。可以按下面的方法修改这些错误，即在不可避免使用静态变量时使用 Standard_Mutex，或者在栈上生成 auto 变量。

第一种情况的例子是 GeomConvert_ApproxCurve.cxx，其定义了 C 语言类型的函数，必须在该函数之外使用数据(例如，作为类的成员变量调用该函数)。这个函数的定义是在其他地方，而其自身又可以作为其他类的参数。

```
static Handle(Adaptor3d_HCurve) fonct = NULL;
static Standard_Real StartEndSav[2];
```



```
extern "C" void myEval3d(Standard_Integer * Dimension,
// Dimension
Standard_Real * StartEnd,
// StartEnd[2]
Standard_Real * Param,
// Parameter at which evaluation
Standard_Integer * Order,
// Derivative Request
Standard_Real * Result,
// Result[Dimension]
Standard_Integer * ErrorCode)
// Error Code
{
```

...
StartEndSav[0]=StartEnd[0];
StartEndSav[1]=StartEnd[1];
...
}

在这种情况下，我不得不在调用类中使用 Standard_Mutex 保护 fonct 和 StartEndSav 从而避免出现数据竞争。阅读 Open CASCADE 6.3.1 的发布报告(Release Notes)，可以知道这个问题已经解决掉了，OCC 团队做了大量的修改，引入了一个类(不再是 C 函数)，该类在数据传递时不再涉及到同步的问题。这非常好，我可以将我的修改删除掉。

c). 特殊例子 BSplCLib、BSplSLib 和 PLib。

在去年 12 月份，我已经在我的博客中讨论过这个问题了，文章的内容是关于加速布尔操作的，名字是为什么布尔操作如此之慢(Why are Boolean Operations so slooo...ooow)。下面我简要介绍一下这个问题。对于 B-Splines 的计算，版本 6.3.0 (以及之前的版本)使用一个辅助的在堆上分配的双精度数组；如果新申请的内存比之前分配的内存大，就需要重新为这个数组分配内存，如果新申请的内存比之前分配的内存小或者相等，就直接使用之前分配的内存。

那么讨论到现在，Andrey Betenev 建议因为 B-样条有最大维数的限制，所以可以使用 auto 变量(在栈上分配内存)。进一步深入的了解代码，我认识到虽然 99% 的情况下，这种方法能够很好的工作，但是在某些情况下，可能会超过最大阈值。因为 B-样条的节点(poles)的数量并没有限制(确实存在有 8192 个节点的曲线!)。所以，我利用另外一种方法改进算法，假如请求的内存数量小于阈值就使用自动缓冲区(atuo-buffer)，否则就使用堆内存。

```
template class BSplCLib_DataBuffer
{
public:

//8K * sizeof (double) = 64K
static const size_t MAX_ARRAY_SIZE = 8192;

BSplCLib_DataBuffer (const size_t theSize) : myHeap (0), myP (myAuto)
```

```
{
Allocate(theSize);
}

BSplCLib_DataBuffer () : myHeap (0), myP (myAuto) {}

virtual ~BSplCLib_DataBuffer()
{ Deallocate(); }

void Allocate (const size_t theSize)
{
Deallocate();
if (theSize > MAX_ARRAY_SIZE)
myP = myHeap = new T [theSize];
else
myP = myAuto;
}

operator T* () const
{ return myP; }
```



```
protected:

BSplCLib_DataBuffer (const BSplCLib_DataBuffer&) : myHeap (0), myP
(myAuto) {}
BSplCLib_DataBuffer& operator= (const BSplCLib_DataBuffer&) {}

void Deallocate()
{
if (myHeap) {
delete myHeap;
myHeap = myP = 0;
}
}
```

```
T myAuto [MAX_ARRAY_SIZE];
T* myHeap;
T* myP;
};
```

(待续... ...)

POSTED BY ROMAN LYGIN AT 20:39, 2009-06-29

第 4 节 线程与同步对象

(接上节... ...)

Open CASCADE 线程

Open CASCADE 提供了线程的抽象类 OSD_Thread，该类封装了针对各个操作系统的线程。该类接受一个类型为 OSD_ThreadFunction 的函数指针，该函数指针的定义如下：

```
typedef Standard_Address (*OSD_ThreadFunction) (Standard_Address data);
```

下面是使用 Open CASCADE 线程的例子：


```
static Standard_Mutex aMutex;

Standard_Address Test_ThreadFunction (Standard_Address /*theData*/)
{
Standard_Mutex::Sentry aSentry (aMutex);
std::cout << "Running in worker thread id:" << OSD_Thread::Current() <<
std::endl;
}

int main (int argc, char *argv[])
{
const int MAX_THREAD = 5;
OSD_Thread aThreadArr[MAX_THREAD];
```

```
std::cout << "Running in master thread id: " << OSD_Thread::Current() <<
std::endl;

OSD_ThreadFunction aFunc = Test_ThreadFunction;
int i;
for (i = 0; i < MAX_THREAD; i++) {
aThreadArr[i].SetFunction (aFunc);
}
for (i = 0; i < MAX_THREAD; i++) {
if (!aThreadArr[i].Run (NULL))
std::cerr << "Error: Cannot start thread " << i << std::endl;
}

for (i = 0; i < MAX_THREAD; i++) {
Standard_Address aRes;
if (!aThreadArr[i].Wait (aRes))
std::cerr << "Error: Cannot get result of thread " << i << std::endl;
}
return 0;
}
```


在另外一个线程中通过函数 OSD_Thread::SetFunction()设置函数指针。运行上面的例子，你可以看到具有不同的 id 的主要线程和工作线程(master and worker threads)。

OSD_Thread::Run()实例接受一个类型为 void*的参数，该参数可以是指向一些数据对象的指针，也可以是从兼容类型(例如整形)转换过来的类型。同样的转换也可以应用于 OSD_Thread::Wait()的输出参数。

当然，使用 OSD_Threads 不是必须的，可以使用其他实现(系统线程、Qt、Intel Threading Building Blocks、Open MP 等)。我已经变成 Intel TBB 的忠实用户了，并将其用在了 CAD Exchanger 中。

同步对象

当前的 Open CASCADE 只提供了 Standard_Mutex 同步对象，其与 Windows

系统上的临界区。你应该多学习一些这方面的只是，这样才能开发并行程序。

例如，为了尽可能的减少冲突的区域，你应该使用同步对象保护尽可能少的代码段。使用 Standard_Mutex::Sentry 还可以为受保护的代码生成附加的嵌套域。

```
void MyClass::MyMethodThatCanRunConcurrently()
{
//线程安全的代码
...
{
//需要序列化执行的代码
Standard_Mutex::Sentry aSentry (myMutex);
MyNotConcurrentMethod();
}

//线程安全的代码
...
}
```

最方便和有效的使用 Standard_Mutex 的方法是通过 Standard_Mutex::Sentry，其实现了对域对象的封装，域对象通过互斥量获取和释放锁。Sentry 对象会在其构造时锁定互斥量，而在析构时释放互斥量。即使在执行过程中碰到了异常，也可以确保互斥量会解锁。否则，这种情况很容易产生死锁——其他线程继续等待互斥量锁，然而没有办法能够解开这个锁。

可以注意到当前 Standard_Mutex 的实现是非常低效的，因此不推荐直接使用它。它的方法 Standard_Mutex::Lock() 重复不断的调用函数 TryEnterCriticalSection()和 sleep()。这可能造成对处理器资源的浪费，而不是将这些资源分给其他任务。似乎程序员正在尝试实现轮询锁(spin lock)，其对于短的临界区非常有用(在这种情况下采用轮询的方式要比立即进入等待模式效率更高)。然而这必须以不同的方法完成——即使用具有轮询数量(spin count)的临界区。我必须对 Standard_Mutex 重新封装才能消除 TryEnter...()行为。

在早期的实验中，我实现了其他对象——等待条件和信号量——但是由于它们不是 Open CASCADE 的一部分，所以在此不讨论它们。

(待续... ...)

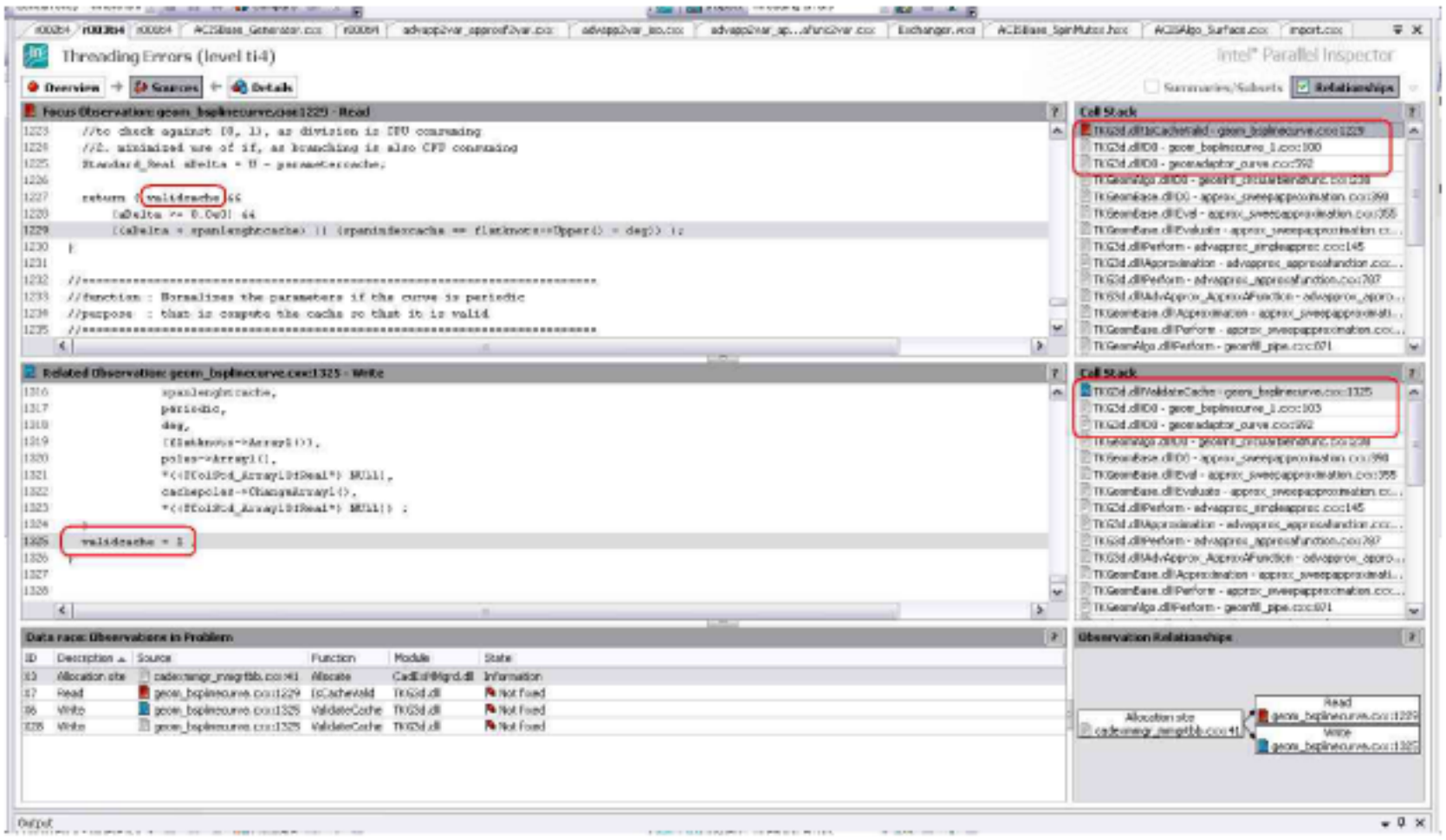
POSTED BY ROMAN LYGIN AT 16:13, 2009-07-06

B-样条中的多线程

这是去年开始写的并行性和 Open CASCAD 一文的延续。这篇文章是关于一个类的，但是该类是基础类，而且非常重要，所以值得分开来单独讨论一下。Geom_BSplineCurve, Geom_BSplineSurface, Geom2d_BSplineCurve, Geom_BezierCurve, Geom_BezierSurface, Geom2d_BezierCurve. 这些类实现了 B 样条和 Bezier 曲线和曲面。

假定要优化点和斜率的计算，它们使用缓存保存了最近一次计算的 B 样条曲线段的计算信息(they use a cache that stores information used in calculation of the lastest B-Spline segment)。在 2008 年，我在提高 Extrema 性能时，做了一些缓存提高性能的实验。试验了几种不同的负载——布尔运算、曲线投影、IGES 和 STEP 导入、还有其他几种功能——但是缓存猜测错误的频率大概是 ~50%-75%。也就是说，在超过一半的情况下，缓存不得不刷新，重新计算。这促使我想混存技术能不能总是能够猜对需要的数据，是否高强度计算的算法不能利用这个技术。当然，要下这样的结论我还需要进行更多更深入的实验，直到现在我还没有时间做这个事。所以我就一直没有考虑这个问题。(但是，假如有谁对这个问题有更深入的见解我非常乐意倾听)

有一年(2009 年 12 月或者是 2010 年 1 月)这个问题又出现了，当时我正在准备发布 CAD Exchanger 的 ACIS 转换器。直到那时，它还是并行的，并且使用多线程利用多核 CPU 进行转换。然而，在一些模式下，时不时的发现程序崩溃了，这使我思考到底是什么原因导致的。因为症状看起来像是数据竞争(data races)，我启动了 Intel Parallel Inspector 来查看是不是这个原因。



当时令我感到奇怪的是，我看到的出错的地方是 Geom_BSplineCurve::Do() ！我的第一印象是 Inspector 监测器可能对于 OCC 太敏感了，可能这里并没有错，因为 Do()只是计算曲线上的一点，而且它是 const 函数。数据竞争是如何出现在两个 const 函数之间的？但是更为深入的查看栈空间、源代码，思考了一下，我知道了出错的原因。耶！Do()首先将 const 指针转化为非 cosnt 指针，然后调用了一个非 cosnt 方法：

```
Geom_BSplineCurve * MyCurve = (Geom_BSplineCurve *) this ;
MyCurve->ValidateCache(NewU) ;
```

那么这意味着什么呢？这意味着当两个或者更多线程使用同一个 B 样条曲线时，这些线程异步读写缓存会引发数据竞争。显然，由 Do() 返回的值可能完全是错的，可能导致之后的调用 Do()的算法崩溃(例如，将三维曲线投影到 B 样条曲面上)。由于这种情况(在不同的线程中同时使用相同的曲面对象)比较少见，所以崩溃并不是总是发生。

在那次发布软件之前我没有时间完全解决这个问题，所以就在 ACIS 的转换器中把多线程功能给去掉了。现在我又碰到这个问题了，在使用多线程之前我准备先介绍对象复制。(复制对象将会增加一些额外的花销，而且这将是一条很冷清的路，这样的情况比较少见)

所以在 Open CASCADE 中使用多线程必须注意这个潜在的问题。一定要将读写

数据保护好(例如，复制对象、同步访问等)或者单独复制一个 B 样条曲线、曲面。

POSTED BY ROMAN LYGIN AT 12:42, 2010-05-02

C++ 流的并行读写

这篇文章不是与 Open CASCADE 直接相关的，但是它对于那些需要并行化程序的人比较有用。我相信应用软件要想取得长远的成功多线程是非常重要的，每个人都必须准备朝着这个领域前进(提早进入比晚进入要好)。免费获得高性能计算的时代结束了，多线程是将来提升性能唯一的选择。

所以，我现在正在开发 CAD Exchanger 中 ACIS 转换器，正在设计它的构架，使其尽可能使用多线程。为了获得有效、流畅、轻量负荷的应用程序，需要思考并且重构应用程序好多次，但是所做的一切都是值得的。

我正在做的集成多线程的一个例子，其中将 ACIS 文件中永久表示对象转换(只是处理直接从文件里面读取 ascii 字符串容器)为临时表示对象(这些 C++对象表示 ACIS 类型，具有数据域和并在对象之间交叉引用)。在该方法中，首先生成空的临时对象，然后从永久对象和中获取它们的数据，并使用已经生成的占位符(placeholder)生成引用。这个方法能够处理各种对象，而且各对象之间相互独立，从而可以用于表示多线程的情况(and thus represents an excellent case for multi-threading)。首先我做了顺序转换，文件大小为 15Mb，包含有 110,000+ 个实体，转化花费了~3 秒。依次来与其他转化方法做对比。

为了简化解析永久实体字符串，我使用了 C++流，C++流封装了 char *缓存以及操作符>>，从而能够识别 doubles, integers, characters,等类型。这使得代码变得简洁而且易懂。为了能够并行运行，我使用了 Intel Threading Building Blocks，它是一个软件库(可以在商业化和开源授权下使用)，可以帮助解决许多开发多线程软件中经常遇到的问题，包括死锁、并发数据容器(concurrent data containers)、同步对象等。已经有好几个软件的开发使用了它，并且在世界范围获得了广大程序员的认可。这个软件库是由与 Parallel Amplifier、Inspector (现在叫 Advisor)一样的团队开发的，我也在这个 Intel 团队中。

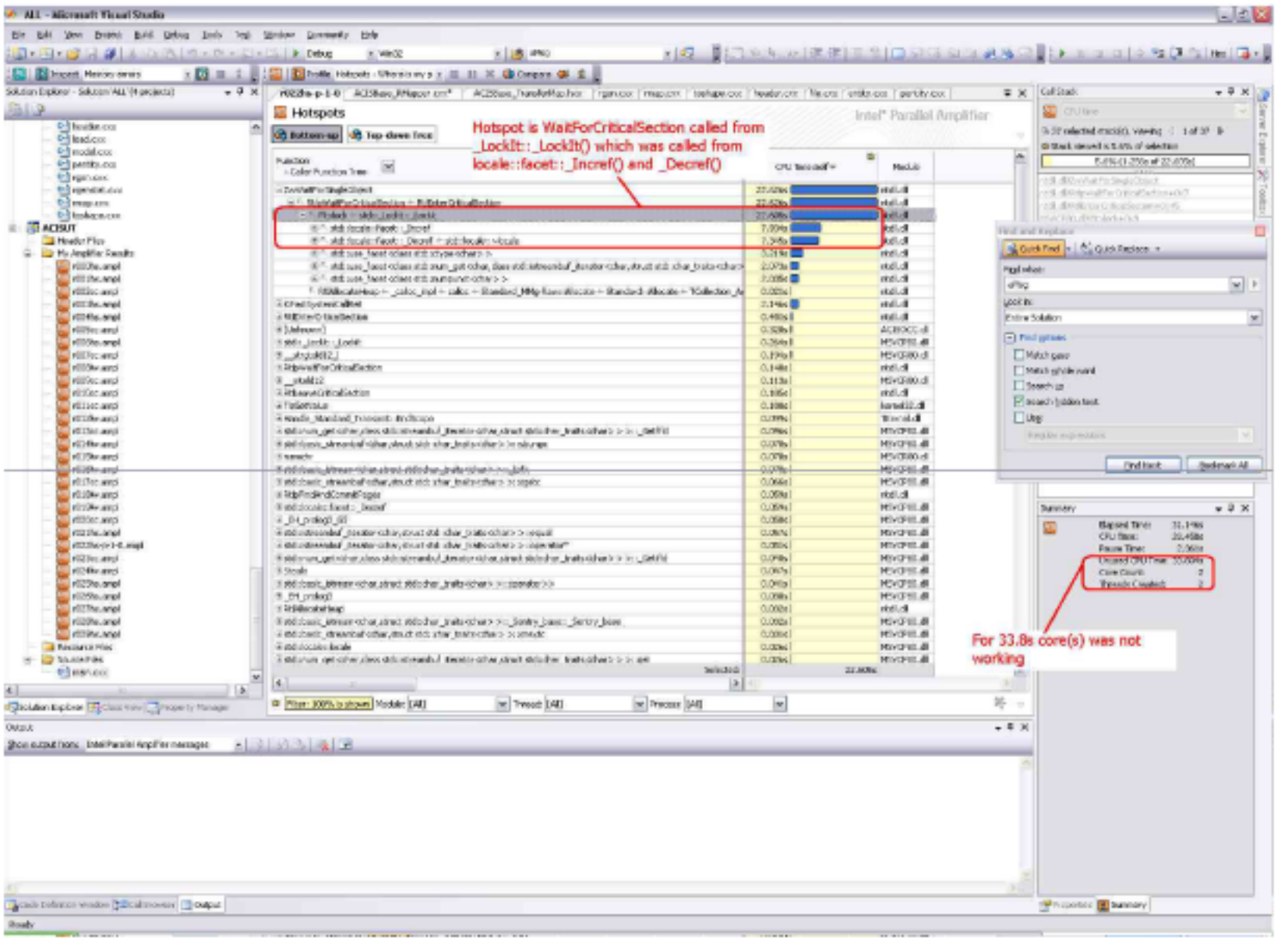
程序非常直白(续行的地方是注释)：

```
/*! \class ApplyPaste
\brief The ApplyPaste class is used for concurrent mapping of persistent representation into transient.

This is a functor class supplied to Intel TBB tbb::parallel_for(). It
uses ACISBase_RMapper to perform conversion on a range of entities that TBB
will feed to it.
*/
class ApplyPaste {
public:

//! Creates an object.
/*! Stores \a theMapper and model entites for faster access.*/
ApplyPaste (const Handle(ACISBase_RMapper)& theMapper) : myMapper (theMapper),
myMap (theMapper->File()->Entities())
{
}

//! Performs mapping on a range of entities
/*! Uses ACISBase_RMapper::Paste() to perform mapping.
The range \a r is determined by TBB.
*/
void operator() (const tbb::blocked_range& r) const
{
const ACISBase_File::EntMap& aPEntMap = myMap;
const Handle(ACISBase_RMapper)& aMapper = myMapper;
Handle (ACISBase_ACISObject) aTarget;
for (size_t i = r.begin(); i != r.end(); ++i) {
const Handle(ACISBase_PEntity)& aSource = aPEntMap(i);
aMapper->Paste (aSource, aTarget);
}
}
private:
```

```
const ACISBase_File::EntMap& myMap;
const Handle(ACISBase_RMapper)& myMapper;
};

/*! Uses either consequential or parallel implementation.
*/
void ACISBase_RMapper::Paste()
{
boost::timer aTimer;
Standard_Integer aNbEntities = myFile->NbEntities();

//parallel
tbb::parallel_for(tbb::blocked_range(0,aNbEntities), ApplyPaste(this),
tbb::auto_partitioner());

//sequential
//const ACISBase_File::EntMap& aPEntMap = myFile->Entities();
//Handle (ACISBase_ACISObject) aTarget;
//for (Standard_Integer i = 0; i < aNbEntities; i++) {
// const Handle(ACISBase_PEntity)& aSource = aPEntMap(i);
// Paste (aSource, aTarget);
//}
Standard_Real aSecElapsed = aTimer.elapsed();
cout << "ACISBase_RMapper::Paste() execution elapsed time: " <<
aSecElapsed << " s" << endl;
}
```


非常令人失望，在我的 Core 2 Duo 笔记本上跑了 17 秒(而不是顺序执行的 3 秒)！怎么回事?!显然这不是 tbb 引起的花费，另外没有用到它。那么到底是什么原因？

我马上启动 Intel Parallel Amplifier 看看什么地方有问题。下面是我看到的：



没有使用的 CPU 时间(例如，当一个或者更多的核没有在工作)是 33.8s，例如，至少一个核没有工作。热点树显示，构造函数 std::_Lockit::_Lockit()中有一些临界区(一个同步对象管理独占访问共享资源)，该构造函数又由 std::locale::facet::_Incref()或者_Decref()调用。第一眼一看比较神秘，所以我重新在 Debug 模式下编译我的应用程序，然后启动 Debugger，然后一切都变的都很清楚了。

原因是临界区用于保护公共局部对象。操作符>>()内部在栈上生成了 basic_istream::sentry 对象，它的构造函数调用(通过另外一个方法) 函数 ios_base::locale()，函数 ios_base::locale()通过拷贝构造函数返回 std::locale 对象(语法如下)。拷贝构造函数调用 Incref()增加引用记数。增加引用记数的部分在临界区内。

```
locale __CLR_OR_THIS_CALL getloc() const
{ // get locale
return (*_Ploc);
}
```

所以，所有的 streams 流都竞争同样的局部对象！而且，当轮询计数器(spin count)

= 0 时才能够进入临界区。也就是说，假如一个线程尝试然后没有解锁进入临界区，此时另外一个线程正在使用它，它会马上进入睡眠模式。当解锁之后，线程会被唤醒。但是所有的这些操作都非常的昂贵，因此花费了许多资源！那么轮询计数器(spin count)为非 0 时能不能进入临界区呢？这样程序就会跑的快些——轮询计数器(spin count)决定了进入睡眠模式之前成功解锁尝试的次数。例如内存管理函数(e.g. malloc())使用的轮询计数器(spin count)允许的最大值为大约 4000，这使得多线程应用程序跑的非常快，甚至并发分配内存。流 streams 是否也可以设置允许的最大值呢？

好了，我尽力解决这个问题，四处找同事讨论。其中一位同事给了我一个链接 http://msdn.microsoft.com/en-us/library/ms235505.aspx，在这个帖子里面其他部分都是在讨论特定线程的 locale。这看起来非常有用。但是经过实验并阅读 http://msdn.microsoft.com/en-us/library/ms235302(VS.80).aspx 的文章，我发现这还是没有什么用处。事实是只有 C 运行库的 locale 对线程来说是局部变量，然而 C++流使用全局的 locale::global。微软！为什么要这样！

所以，这就是我目前的工作。我将会继续研究，但是假如你做过在多线程程序中使用 STL 流或者听说过这方面的东西，请不吝赐教。我会非常感激你的指导的。当然也可以实现一个自定义字符串解析器(从而避开 STL 流)，但是也是我最不情愿的做的，不过作为一种备选方案并不能完全

尽管如此，通过 TBB 检测运行的非常好。我已经将它集成到转换部分，它能够将临时对象表示转换成 OpenCASCADE 的 shapes 对象。并行执行和使用 tbb::concurrent_hash_map 用于存储要比顺序存储好，使用的 CPU 内核越多性能就成倍的增加。我将会及时更新我的最新研究进展。

保重，

Roman

POSTED BY ROMAN LYGIN AT 09:25 3 COMMENTS, 2009-05-25