OPEN CASCADE 学习笔记
——为什么布尔操作如此之慢

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 **OpenCascade** 的论坛上知道了 **Roman Lygin** 在他的博客上写了 **Open CASCADE　notes** 系列文章，但是却无法访问他的博客，幸而百度文库已经收录了 **Topology and Geometry** 和 **Surface Modeling** 两篇文章，拜读之后获益良多。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 **10** 月 **18** 日星期四

第 1 节　问题初探

经常有人提到 Open CASCADE 的布尔操作(Boolean Operations, BOPs) 非常的慢。是否有人知道这是为什么呢？

你可能还记得，我在另一个帖子(http://opencascade.blogspot.com/2008/11/open-cascade-inside-intel-or-intel.html)中提到，在 Intel 工作的时候，我们决定将 Open CASCADE 集成到我们

的测试数据库的应用程序中。我做了几个例子，用来测试 Intel Parallel Amplifier and Inspector (Intel 并行开发工作室(Parallel Studio Intel)新添加的部分应用程序)。

除了我最近导入IGES的这个测试例子(已经实现了多线程模式导入IGES文件的原型系统)，这次我还测试了布尔操作(BRepAlgoAPI)。在论坛上(http://www.opencascade.org/org/forum/thread_14933/)我要过几个模型，但令人惊奇的是回应的人并不多。不管怎么说，还是要谢谢 Evgeny L, Prasad G, Pawel K, Igor F，他们给我提供了模型。

下面说正经的。在比较复杂的模型中，所有案例中速度提高了从 4x(模型中有100+个面)倍速到 20x(几十个面)倍速都有。所有案例中使用的 CPU 时间都减少了，有从 80 秒到 20 秒的，也有 30 秒到 1.4 秒的。(声明：这篇文章在上周完成初稿之后，我又测试了一组由 Pawel Kowalski 提供的模型。经过测试发现了其他瓶颈，这将在后面讲到，因此先前做的提高性能的方法对它们无效。假如时间允许的话，我将会继续实验并将新的发现写出来。

正式开始
让我们按照下面的步骤做：

我主要关注 BopTools_DSFiller 类，它是布尔操作(Boolean Operations , BOP)的重要部分，它提供相交后模型，以便于之后根据相交结果进行并(fuse)、交(common)、差(cut)等操作，并重构这些操作结果。

在第一个例子中，使用了我在 OCC 时候的同事提供的两个模型，他也参与了 Intel 并行工作室 beta 版程序(Intel Parallel Studio beta program)的开发。两个模型都有 130+的面，BopTools_DSFiller::Perform()花了 67 秒的 CPU 时间。

我安装了最新版的 Intel 并行放大器(Intel Parallel Amplifier)(备注：公开的 Beta 版在 1 月初就有了，你可以现在在这个网站定购– www.intel.com/go/parallel)。唯一使用到的分析类型是“热点分析”(Hotspot Analysis)，它可以确定使用最多 CPU 时间的函数。Amplifier 也提供了“并发分析”和“等待锁定分析”('Concurrency Analysis' and 'Waits & Locks Analysis')，但是这些都没有用到，因为布尔操作当前还是以单线程方式运行，当然布尔操作也可以改进成多线程应用程序。

* 第一个发现 *

Amplifier 报告的使用 CPU 时间最多的函数位于 TKGeomAlgo.dll 中，与 IntPolyh 包有关。一点也不让人吃惊，布尔操作是基于网格交叉(meshes intersection)，而 IntPolyh 生成了这些网格。

前 3 位使用 CPU 时间最多的函数——IntPolyh_Triangle, _StartPoint, 和_Edge 的构造函数总共花了 20 秒，如图 1 所示。

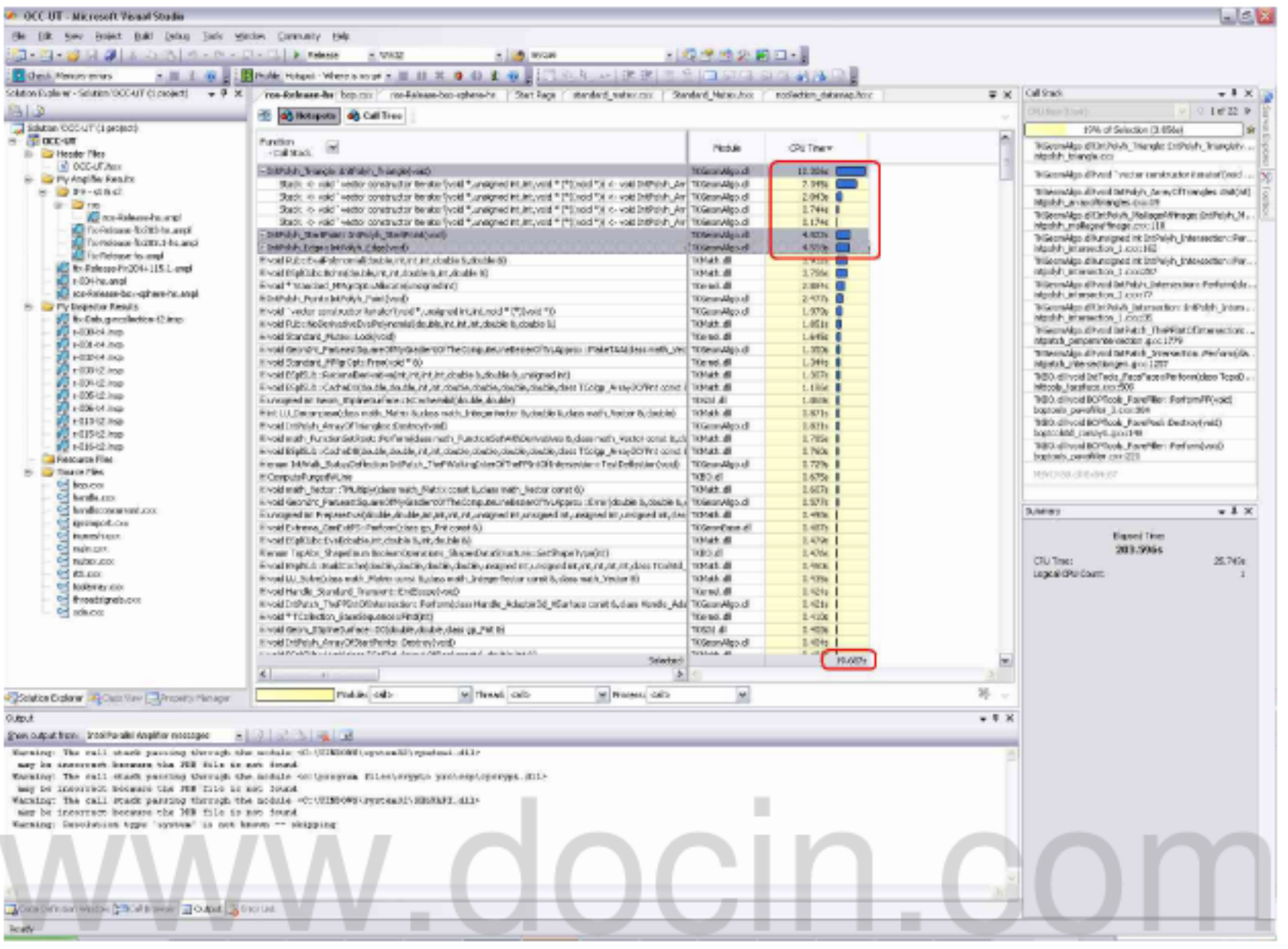


图 1 前 3 位使用 CPU 时间最多的函数总共花了 20 秒

待续... ...

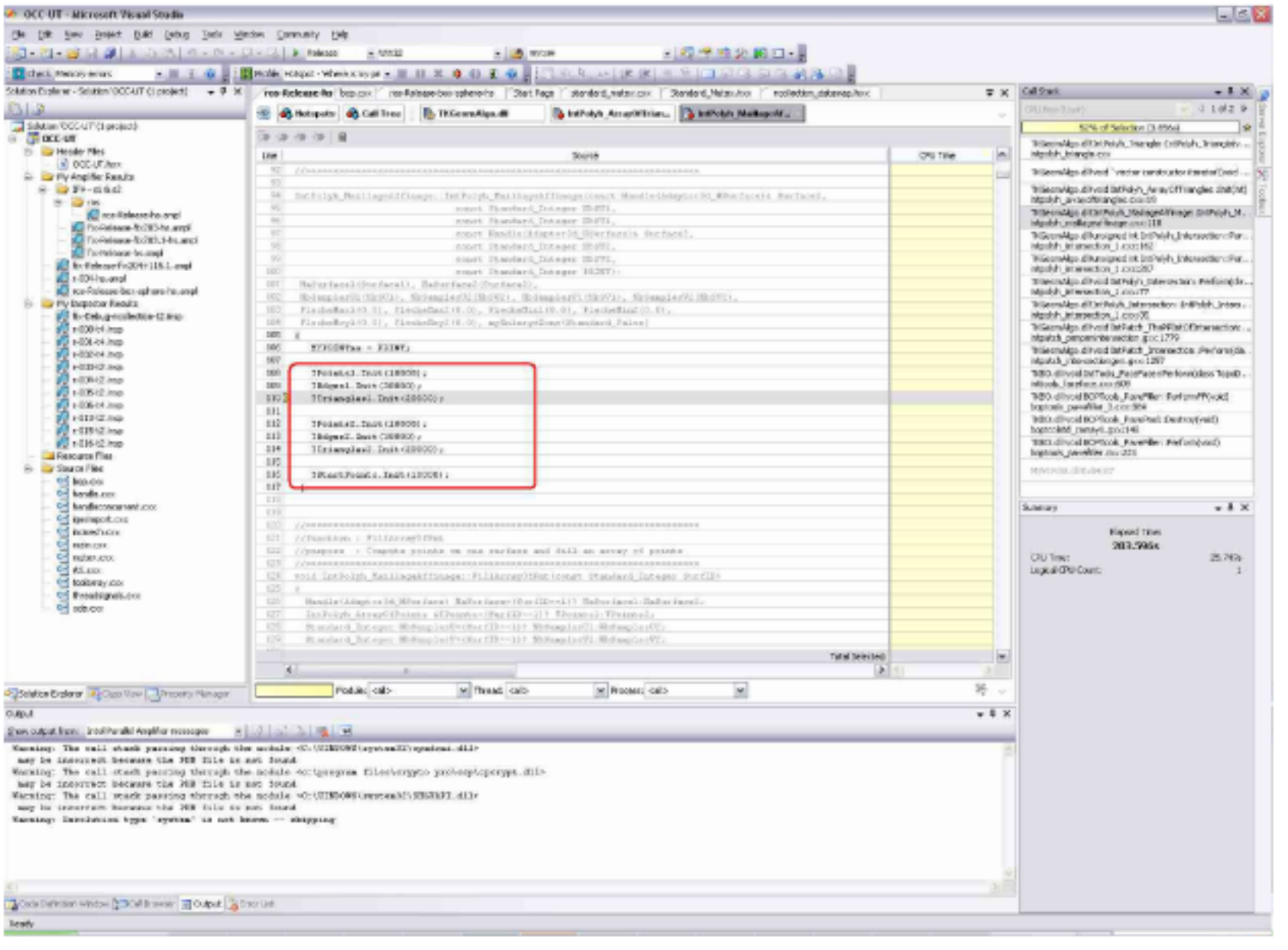
POSTED BY ROMAN LYGIN AT 15:47, 2008-12-06

第 2 节 避免调用 new 操作

接上节... ...

所以，我仔细研究了这些代码，发现构造函数只是简单的为内部类型分配空间(例如，double, integer, pointer)。

所有的三个构造函数看起来都一样。对堆栈解读发现了原因——对象都是用new[]操作生成的(例如，ptr = (void*) (new IntPolyh_Triangle [N]))，而且有大量的这种对象的拷贝。看下面：



每一个 IntPolyh_MaillagaAffinage 的构造函数为每一个面都生成了 10 000 个点、20 000 个三角形、30 000 条边的数组。这些分配的空间最终都使用到了吗？使用调试器我跟踪了所有的使用到这些数组的执行步骤，猜猜我发现了什么？通常它们只填充使用了不超过 100 个元素。只使用几十个元素，却分配了上万个元素空间？！真是难于想象！

两个附加的发现：

1. 数组中初始化的元素从来都没有读取过，而且还重复写入。
2. 使用的元素的可以很容易提前计算出来(例如，n * m)。

所以我看了看所有的 IntPolyh 类，从而确认这是都采用了相同的对象生成方法，这样我就可以使用延期初始化特定元素的方式(with a deferred initialization with a particular number of elements)很容易修正这个错误。正如通常看到的一样，生活并不总是跟和它看起来一样简单。一些类(例如 IntPolyh_ArrayOfEdges)表明使用的元素会随着时间的延长而增加，这个特征在网格优化时确实用到了。而且我发现，许多类实现了同样的数组使用模式，它们都有一些分配了很多空间的元素和经常用到的元素。但是我们已经看到这种策略多么的低效。

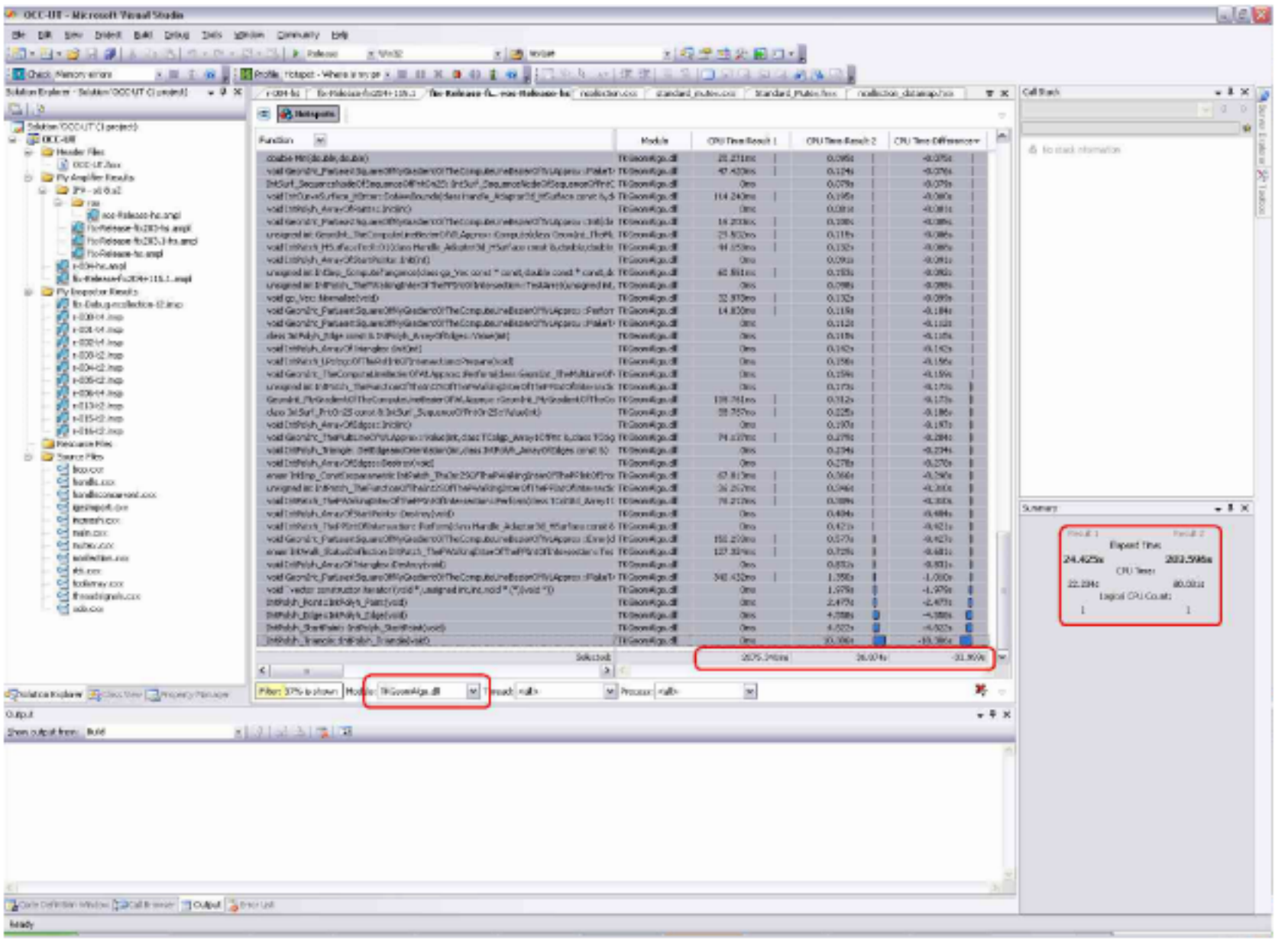
IntPolyh 包含的 7 个类 IntPolyh_Array*也是按照这种模式实现的，而且实现方式是通过复制代码。

所以我生成了一个通用类 IntPolyh_DynamicArray，该类使用 Standard::Allocate()分配内存空间(从而避免调用 new[]和构造函数)。而且如果之前分配的内存不够了，能够随着时间推移而自动增长。所有的 7 个类都成为这个模板的实例，从而大大的降低了需要维护的代码量。

然后，当需要填充其中的元素时(例如在函数 IntPolyh_MaillageAffinage::FillArrayOfPnt()中)我延迟初始化这些数组。(Next, I made deferred initialization of these arrays when the number of elements to fill them in with is known (e.g. IntPolyh_MaillageAffinage::FillArrayOfPnt()).)

也有其他可能的简单的提高性能的方式，例如将所有的关于_Point, _Edge, 等的函数都变成内联函数(inlining)。我没有这样做，我想将这个工作留给 OCC 团队。

经过这样的修改，花费在 TKGeomAlgo.dll 中的时间(使用软件 Amplifier 测量)降低了(从 36.07 秒到 2~7.28 秒)，速度提高了 5x-18x 倍速!!! 整体速度提高了 3.5x 倍速(看下面的屏幕截图)。



这是可以解决的问题中相对比较简单的问题，正如“挂在树上的比较低位置上的果子”，很容易就摘到了。而且，效果也非常明显。我想应该还有一些可以改进的地方，我将鼓励 OCC 的人继续做更多的工作，包括多线程。我不知道 BOP 的内部实现方法，但是我将会检测面面相交能否在并行线程中运行。

我的研究还没有结束，下面将接着讲。

待续... ...

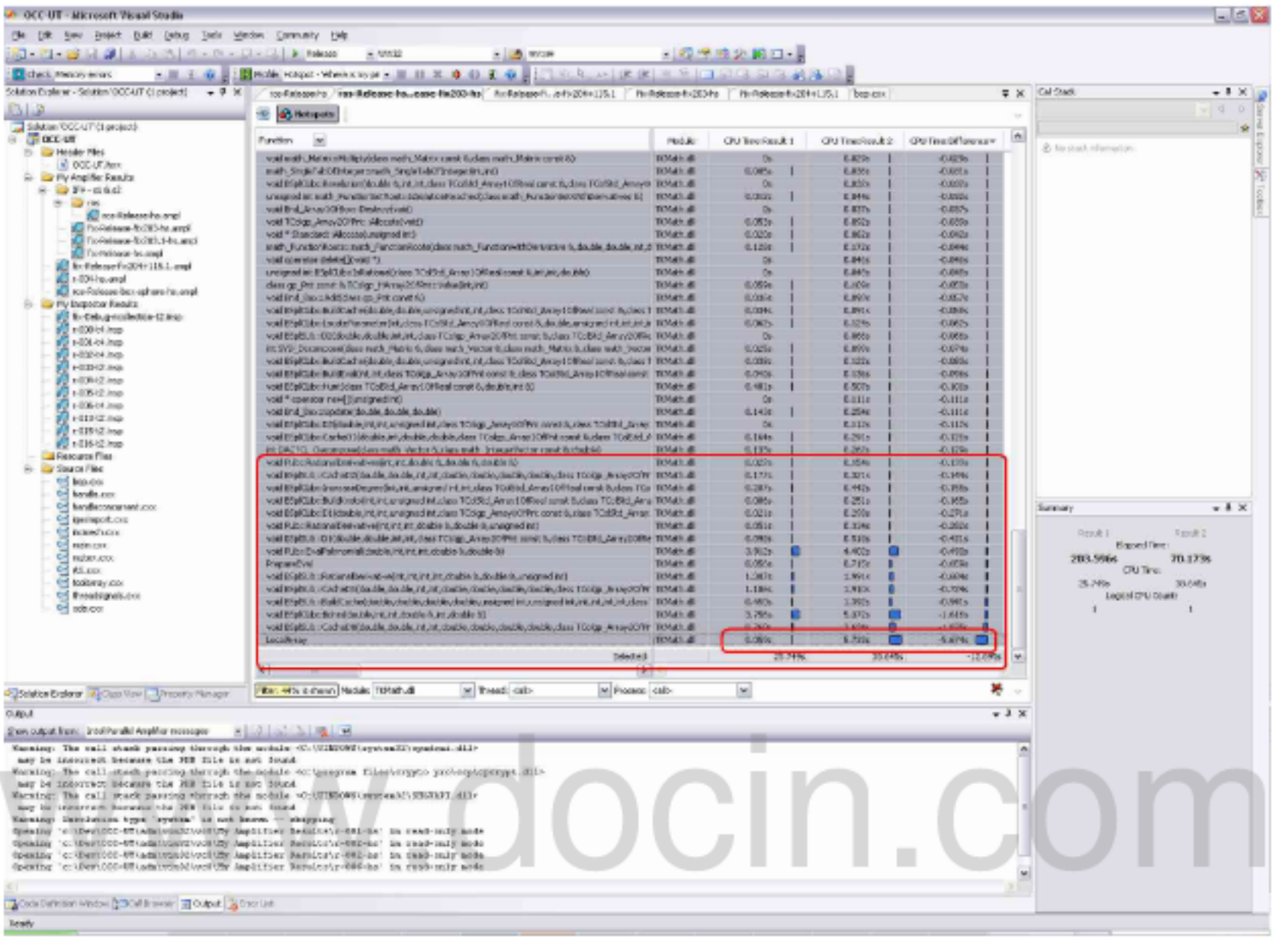
POSTED BY ROMAN LYGIN AT 17:19, 2008-12-06

第 3 节 多线程提升布尔操作性能

接上节... ...

根据我的经验，我使用我自己修改过的 Open CASCADE 6.3.0，这个版本包含有

所有为了实现多线程原型做的修改，我曾经在论坛上提到过这个事情。做的一些修改是 BSplSlib, BSplCLib 和 PLib 的线程安全版本。原始的 6.3.0 使用静态数组，在调用::Do(), D1()等时又重新使用到了，这显然不是线程安全的，甚至不是读取安全的(or even non-reentrant)，所以几个月前我将它们改成了局部变量，从而可以根据每一个相关的 Lib 函数调用分配和释放内存空间。这对于导入 IGES 文件的场合是适用的，但是 BOPs 中我发现我的代码性能倒退到了 6.3.0。如下图所示：



所以，很容易就可以发现最好不要使用通常的构造和析构函数。我计算了一下在 BOP 的测试例子中重新分配空间的次数，大概是 1170 万次，分配的空间从 2 到 264 个 doubles 长。我不得不回到先前的方法(当新请求的空间超过前面分配的空间时我就重新分配空间)。但是怎么确保线程读安全呢？答案是——线程局部存储(TLS, Thread-Local Storage)。也就说，每一个线程都有自己的空间，仅仅能被该线程使用，所以我写了一个类(暂时称它为 TLSManager)，其中包含一个类似于{Standard_ThreadId, set of buffers}的图结构(hash-table)，根据线程 id 返回要求的缓冲区。

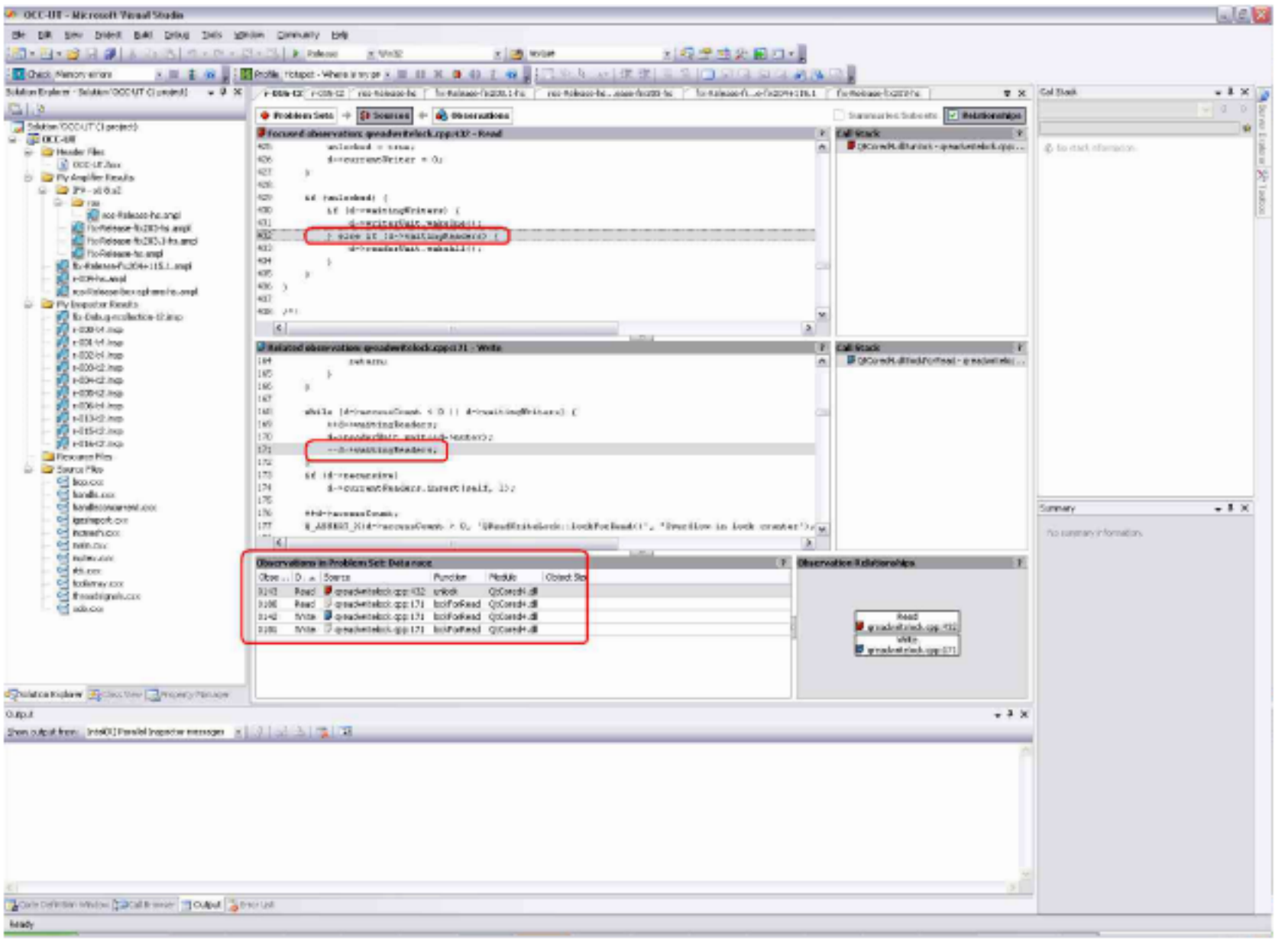
另外一个显然的问题——TLSManger 必须是线程安全的，但是使用 Standard_Mutex 来保护它也没有必要。有一个解决这个问题的方法，就是读写

锁，例如一个实例允许多个并发读取，从而共享资源(在我们这里是具有缓冲区集的图结构)，当一个新的线程生成一个新的缓冲区时，则避免写入。我自己构造了个轮子(re-invented a wheel)，添加了类 OSD_ReadWriteLock(RWL)。就像前不久在 OSD 中添加其他线程类一样，我曾经想过 OCC 应该引入 Boost(www.boost.org)，而不是重新设计自己的轮子。Salome 确实使用了 Boost，所以 OCC 显然也可以使用 Boost。例如，Boost 也提供了一个 TLS 模板 -http://www.boost.org/doc/libs/1_37_0/doc/html/thread/thread_local_storage.html。

RWL 类已经写好了，然后我生成了简单的测试案例来检测数组元素是否被写入过，并使用 Intel 并行线程检测器(Intel Parallel Inspector)来检测。线程检测器可以确定内存问题(泄漏，使用非初始化内存，等问题)和线程错误(数据竞争 data races，死锁 deadlocks 等)。线程检测器是一类软件中的一个(据我所知有其他同类软件)，但是它花费的 CPU 时间比较高。

然而，在使用线程检测器时发现问题了！它报告数据竞争(data races)，好像我对 RWL 在同时读写(实例成员的)同一块内存。

我花费了好几个小时查看我的一页代码希望找到出错的原因，焦虑的用脑袋撞键盘以及周围所有的东西，在家里度过了一个疯狂的星期五晚上。当我要放弃时，我使用 QT 的 QReadWriteLock 重新编写了我的单元测试，然而同样的数据读写竞争(data races)报警出现了！



这使我更加疑惑了，我在纸上写下所有的行为场景，进一步确认不可能发生数据竞争(data races)，所有的东西都被保护起来了。最终我认为它仅仅只是一个假错误报警(例如报告一个根本不存在的问题)！我知道线程检测器有假报警问题，但是我不能想象它能这样折磨我。所以，我希望与我 Intel 的同事讨论这个问题。(注意，你要是下载了线程检测器并且碰到了同样的问题，一定要回想下我的例子，它可能是一些没有修正的假报警)。

好了，经过对我对 RWL 确认，我继续用 TLSManager 在 BSplSLib 中测试 RWL。速度退化终于消失了！使用 RWL(而不是共享静态缓冲区)的开销很小几乎可以忽略不计。非常好！

经过这样的修改，在 Open CASCADE 的测试例子中相对于 6.3.0 整体速度提高了 4x 倍速。我试了一下论坛板友发给我的简单模型，它提高了 20x 倍速。更小的测试案例只要不到一秒就完成了，速度提升并不显著。所以，速度提升的范围平均在 3x-10x 倍速。

还可以接着做其他事情，例如在 CDL 中设计 TLSManager，并将 PLib 和 BSplCLib 迁移到其上。我将会这么做，假如时间允许的话，希望不久之后我仍然记的清楚。

另外，顺便说一下，假如你想让我试验更新版本的模型，请通过 email 或者链接发给我。如果你想快点知道程序改进后的结果，也请告诉我。

回过头来看看，我想在这方面花费的时间是值得的。我希望这些发现对 OCC 团队有所启发，从而找到更进一步提高软件性能的方向，而不仅仅是在 BOP 方面。我也希望我在 Intel 的同事感谢我这些天发现的 14 个 bug 和软件改进需求，从而在商业发布时这些工具会比现在更好用。我能够在 OCC 建模算法的更深入层次上学到新东西，这就足够了。希望在新版本的 OCC 中包含有我的改动，这样板友们也会受益于新的 OCC 版本。

我将继续使用新的 OCC 测试案例用来测试应用程序。假如有什么有趣的东西，我会与你分享的。

关于性能问题我再说几句。在 Intel 工作之后，我对于性能的看法跟我在软件开发公司的时相比发生了改变。伙计们，人们总是随着时间在变(或者只要你原意，你已经随着时间发生改变)。以前每次发布新的处理器之后应用程序就会跑的更快，这种免费的午餐现在不存在了。成数量级提高速度(Megahertz)的时代结束了。要使你的程序跑的更快，你必须使它变成多线程的，多阶段可执行的(scalable)。性能并不仅仅是你的应用程序跑的更快。高性能也意味着可以添加更多的功能特征。看看在 MS Word 中的拼写检查，它在你键入文字的时候进行拼写检查。仅仅是因为 Word 能够跑的够快，而且因为它以平行线程的方式运行，所以才能够运行该功能。
假如你想在市场中保持竞争力，你必须平行多线程运行程序。没有其他路子。这个问题比较有挑战性，但是幸运的是有工具来帮助我们。我非常乐意讲述到了这些问题。可以试试 Intel tools(www.intel.com/go/parallel)。

保证书？好吧，也许这样说更合适，我绝对真诚的向你保证，我也是按照我说的做的。

祝你幸运！

(结束)

POSTED BY ROMAN LYGIN AT 17:28, 2008-12-06