Parallel programming in an AutoCAD application

Gopinath Taget – Autodesk Inc.

CP2526

In this class, we will discuss the problems with implementing parallelism in an AutoCAD .NET and C++ application. You will learn which portions of an application can be safely parallelized. We will also cover what technologies the latest .NET and C++ libraries provide to build parallelism naturally and easily into your AutoCAD application. You will learn how to synchronize between threads in your application and AutoCAD functionality.

About the Speaker:
Gopinath is a member of the Autodesk Developer Technical Services Team. He has more than seven years of experience developing and supporting AutoCAD® APIs, including ObjectARX®, Microsoft® .NET, VBA and LISP. Gopinath also has several years of experience in software development on other CAD platforms, including MicroStation®, SolidWorks®, and CATIA® mainly using C++ and technologies such as MFC and COM. Gopinath was also involved in the development of Web-based applications for Autodesk® MapGuide® and AutoCAD Map. Gopinath has master’s degrees in Civil Engineering and Software Systems.
Parallel Programming in an AutoCAD® Application

Introduction

Multi-core CPUs and Multi-CPU hardware architectures are pervasive today and span from mobile devices to specialized server hardware. It allows you to create truly parallel applications that if done correctly:

1) Allows processes and threads to run truly concurrently
2) Drastically reduces time taken by compute intensive tasks
3) Create highly responsive applications
4) Naturally run autonomous regions of code
5) Enables real asynchronous programming
6) Makes your application very scalable
7) Makes you look cool

In view of these obvious advantages, you might think all applications would be implemented (or in the case of legacy applications, re-architected) to make use of this relatively cheap way (and in some cases the only way) to provide performance boosts to their application. So why don’t we see a surge in applications that are optimized for parallel computing?

The answer is that building an application optimized for parallel computing is hard. Again, this is fundamentally because thinking in terms of parallel tasks is harder than thinking linearly which comes naturally to most of us (Think back to the number of times you have been told you have a one track mind 😊).

Some of the problems that you might face when you try to build a parallel application (1) (2) (3):

1) Finding the parallelism
2) Deciding how concurrent your application should be and determining the critical path
3) Making the concurrent application faster than its sequential (linear) counterpart
4) Avoiding the bugs
5) Tuning performance
6) Future proofing
7) Using modern programming methods

Among these problems, possibly the trickiest problem is “Avoiding the bugs”. This is because the most popular (and possibly the oldest) model for concurrent programming is the “shared memory” model and this model makes it very difficult to maintain data integrity of data used between threads. Consider this: In a sequential application, if you allocate a certain amount of memory to a data object on the heap, you can be certain that only one instruction will access that data object and (possibly) modify it. This cannot be guaranteed for a multi-threaded application where multiple threads want to access the same data object. Please check out (1) for more discussion on this.

However for legacy applications, an equally daunting problem would be “Finding the parallelism”. Consider AutoCAD. This software has been in active development for close to 30
years. It has gone through more than twenty major revisions (not counting vertical products built on top of them). All these revisions were built without a lot of emphasis on parallel execution. Also, over the years the code base has increased by several orders of magnitude and stands at millions of lines of code today. Further, AutoCAD uses several languages/technologies (C/C++, COM/ActiveX, Lisp and .NET to name a few) for its components that communicate with each other. Finally, AutoCAD has recently gone back to being a multi-platform application (Windows and Mac).

All these factors add tremendous complexity to any attempt in identifying and transforming portions of code that can be safely parallelized. This is why it is nearly impossible to make the AutoCAD object model thread safe and this is why it is very unlikely that AutoCAD APIs will be supporting multi-threaded programming any time soon.

Having said that, all hope is not lost. If you’re clever, disciplined and conscientious and don’t ask AutoCAD APIs to do what they are not intended to do, you can still write applications that use parallelism to boost performance and responsiveness of the application. The rest of the sections in this handout discuss this and more.

The AutoCAD Problem

Build an AutoCAD .NET application with the following piece of code, load it into AutoCAD and run the “AccessDwg” command:

```csharp
[CommandMethod("AccessDwg")]
public void AccessDwg()
{
    string dwgName = @"C:\ParallelTest\DrawingToFill.dwg";
    Thread t = new Thread(DrawLineInDwg);
    t.Start(dwgName);
}

static void DrawLineInDwg(object str)
{
    string dwgName = (string)str;
    using (Database db = new Database(false, true))
    {
        db.ReadDwgFile(dwgName, FileMode.OpenForReadAndWriteNoShare, false, null);
        using (Transaction trans = db.TransactionManager.StartTransaction())
        {
            BlockTable btl = (BlockTable)trans.GetObject(
                db.BlockTableId, OpenMode.ForRead);
            BlockTableRecord btr = (BlockTableRecord)trans.GetObject(
                btl[BlockTableRecord.ModelSpace], OpenMode.ForWrite);

            using (Line ln = new Line(new Point3d(0, 0, 0), new Point3d(10, 10, 0)))
            {
                btr.AppendEntity(ln);
                trans.AddNewlyCreatedDBObject(ln, true);
            }
        }
    }
}
```
The code above launches a .NET thread and in that thread, we open a DWG file, add a line to it and save it before we dispose off the drawing (Database) object.

However, as you might have guessed, this code will crash. Specifically it will crash at the call to `db.Save()`. This crash is pretty symptomatic of any attempt to use the AutoCAD API in any thread other than the “Main AutoCAD UI thread”. For a better understanding of what UI threads are and what other kinds of threads there are, please refer to the following sources. AutoCAD has only one UI thread like most MFC based applications. This brings us to the…..

**Important Rule of Concurrency with AutoCAD:** Never *ever* try to use the AutoCAD API in a non-UI thread.

This leads us to another, slightly more subtle problem with concurrent programming in AutoCAD. Let’s say you never use the AutoCAD API in any thread other than AutoCAD main UI thread. Let’s also say you do all your non-AutoCAD based, computationally intensive heavy lifting in worker threads. Everything is fine so far. However, there will come a time when your computationally intensive worker threads will want to communicate with the AutoCAD main UI thread (why else would you want to write an AutoCAD application?). For instance, after some intensive Finite Element Analysis (FEA) computations in a worker thread, your application might want to display the analysis results in the AutoCAD drawing. However, the drawing operation can only be performed by the main UI thread so the worker thread has to find a way to tell the main UI thread to draw its results. So what is the best way for a worker thread to communicate with the Main UI thread so the results can be displayed?

One useful technique that .NET developers use to communicate between main UI thread and the worker thread is to instantiate an object of type `System.Windows.Forms.Control` in the main UI thread and use `Invoke/BeginInvoke`\(^6\) in the case of WinForms and WPF dispatcher\(^7\) in the case of WPF applications. Here is how a simple AutoCAD application would look:

```csharp
public class BackgroundInvoke : IExtensionApplication
{
    static Control syncCtrl;
    public void Initialize()
    {
        // the control created to help with marshaling
        // needs to be created on the main thread
        syncCtrl = new Control();
        syncCtrl.CreateControl();
    }
}
```
public void Terminate()
{
}

void BackgroundProcess()
{
    // this is to represent the background process
    System.Threading.Thread.Sleep(5000);
    // now we need to marshal the call to the main thread
    if (syncCtrl.InvokeRequired) // I don't see how this could ever be false
        // in this context, but I check it anyway
        //syncCtrl.Invoke(new FinishedProcessingDelegate(FinishedProcessing));
    syncCtrl.BeginInvoke(new FinishedProcessingDelegate(FinishedProcessing));
    else
        FinishedProcessing();
}

delegate void FinishedProcessingDelegate();
void FinishedProcessing()
{
    // if we want to modify the database, then we need to lock the document
    // since we are in session/application context
    using (doc.LockDocument())
    {
        using (Transaction tr = doc.Database.TransactionManager.StartTransaction())
        {
            BlockTable bt = (BlockTable)tr.GetObject(doc.Database.BlockTableId,
                OpenMode.ForRead);
            BlockTableRecord ms = (BlockTableRecord)tr.GetObject(
                bt[BlockTableRecord.ModelSpace],
                OpenMode.ForWrite);

            Line line = new Line(new Point3d(0, 0, 0), new Point3d(10, 10, 0));
            ms.AppendEntity(line);
            tr.AddNewlyCreatedDBObject(line, true);
            tr.Commit();
        }
    }
    // also write some message to the command line
    // Note: using AutoCAD notification bubbles would be a nicer solution :)
    // TrayItem/TrayItemBubbleWindow
    ed.WriteMessage("Finished the background process!\n");
}

[CommandMethod("ProcessInBackground")]
public void ProcessBackground()
{
    // let's say we got some data from the drawing and
    // now we want to process it in a background thread
Another way to communicate between UI thread and worker thread is to use the .NET System.ComponentModel.BackgroundWorker type. This type was specifically designed to make communication between UI threads and background threads simple. Here is a sample of how this type could be used with AutoCAD .NET:

```csharp
public class BkWorkerClass
{
    // BackgroundWorker is a class provided by .NET 2.0 that makes
    // it easy to perform asynchronous operations in a background thread
    BackgroundWorker _bw;

    [CommandMethod("bkworkerlaunch")]
    public void BkWorkerLaunch()
    {
        // We are going to use asynchronous method to do work.
        // For this we use built in .NET class called
        // BackgroundWorker. This class makes it easy to launch a thread,
        // do our stuff and synchronize back with
        // the main UI thread, in this case the modeless custom Palette.
        if (null == _bw)
        {
            // Instantiate the backgroundworker class
            _bw = new BackgroundWorker
            {
                WorkerReportsProgress = true,
                WorkerSupportsCancellation = true
            };
            // This is the event handler that actually does the work
            _bw.DoWork += new DoWorkEventHandler(_bw_DoWork);
            // This event handler gets called when there is a change in state
            _bw.ProgressChanged += new ProgressChangedEventHandler(_bw_ProgressChanged);
            // This event handler gets called when the BackgroundWorker is done.
            _bw.RunWorkerCompleted += new RunWorkerCompletedEventHandler(_bw_RunWorkerCompleted);
            _bw.RunWorkerAsync(); // Start the background worker
        }
        ed.WriteMessage("Started background processing.
        You can keep working as usual.\n");
    }
}
```
#region "The BackgroundWorker DoWork event handler implementation"
// This event handler is where the action is. We call the Service Method here
void _bw_DoWork(object sender, DoWorkEventArgs e)
{
    // this is to represent the background process
    System.Threading.Thread.Sleep(5000);
}
#endregion

#region "The BackgroundWorker ProgressChanged event handler implementation"
// This event handler gets called when the BackgroundWorker.ReportProgress method
// gets called in the DoWork event. This event is special because it is run in the
// Main thread. That is the magic of the BackgroundWorker class
void _bw_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
}
#endregion

#region "The BackgroundWorker RunWorkerCompleted event handler implementation"
// This event handler gets called when the BackgroundWorker is done
// with what it is doing.
// In our implementation we don't do much but just access the exception
// if any when the background worker was running.
void _bw_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    System.Exception exp = e.Error; // Get the error if any
    // if we want to modify the database, then we need to lock the document
    // since we are in session/application context
    using (doc.LockDocument())
    {
        using (Transaction tr = doc.Database.TransactionManager.StartTransaction())
        {
            BlockTable bt = (BlockTable)tr.GetObject(
                doc.Database.BlockTableId, OpenMode.ForRead);
            BlockTableRecord ms = (BlockTableRecord)tr.GetObject(
                bt[BlockTableRecord.ModelSpace], OpenMode.ForWrite);

            Line line = new Line(new Point3d(0, 0, 0), new Point3d(10, 10, 0));
            ms.AppendEntity(line);
            tr.AddNewlyCreatedDBObject(line, true);

            tr.Commit();
        }
    }

    // also write some message to the command line
    // Note: using AutoCAD notification bubbles would be a nicer solution :)
    // TrayItem/TrayItemBubbleWindow
    ed.WriteMessage("Finished the background process!\n");
}
Both the techniques above are legitimate techniques for communicating between UI thread and worker thread. They both depend on one fundamental premise - that there is a delegate or event method in the Main UI thread that can be called synchronously or asynchronously from a worker thread to do its bidding. And it works. Most UI based applications both unmanaged and managed use this technique to update UI controls from worker threads. It even works with AutoCAD in some situations. However, there is a problem. Consider the following workflow:

1) Build a .NET application with the code above (the BackgroundWorker technique)
2) Set a breakpoint in the _bw_RunWorkerCompleted method
3) Load the application in AutoCAD 2012 and run the command “bkworkerlaunch”
4) Immediately launch LINE command. Do not perform any action here.

You will observe that the breakpoint in _bw_RunWorkerCompleted is hit even though the LINE command is active. And this is the fatal flaw. None of the thread communication or synchronization techniques are aware of AutoCAD Quiescent state. i.e., they are not aware if another operation is being performed in AutoCAD. The delegate or the event handler invoked by the worker thread is executed irrespective of any other active command or event handler in the AutoCAD main UI. This is the surest way of corrupting a drawing and completely unacceptable.

So is there a way out? The short answer is “yes”. The solution lies in the way the AutoCAD UI thread behaves. Since there is only one UI thread in AutoCAD, there is only one windows message handler and windows messages are handled one after another in a queue. i.e., only one windows message gets handled at any time. So if a worker thread could post a windows message, the AutoCAD main thread could handle the message and perform an action on behalf of the main thread.

The first step though is to make sure when a windows message from the worker thread is being handled, no other operation is in progress. This can be accomplished by a series of checks (C++ based ARX code below):

```cpp
bool isQuiescent(AcApDocument* pDoc = curDoc())
{
    return (pDoc != NULL)
        && pDoc->isQuiescent()
        && (!pDoc->lockMode() == AcAp::kNotLocked) ||
            (pDoc->lockMode() == AcAp::kNone))
        && (acedCommandActive() == 0)   // no script or lisp is active
        && (acDocManager != NULL)
        && (acDocManager->inputPending(pDoc) == 0)
        && (::GetInputState() == 0)
};
```
If the above function returns true, you can be sure that no other operation is being performed by AutoCAD.

The next step is to wire in the code necessary to register a windows message associated with the worker thread. For this we can use the Win32/MFC function `RegisterWindowMessage`:

```c
UINT AU2011_Adsk_ThreadDrawMessage = 0;

AU2011_Adsk_ThreadDrawMessage = ::RegisterWindowMessage(L"WM_AU2011AutoCADThreadingDraw");
```

The `RegisterWindowMessage` Win32 function allows you to register a custom windows message with the Win32 messaging system given a unique string and returns a unique message ID. If a message already exists for the string, the message ID of the existing message is returned.

The final step after this is to implement a windows message handler that will handle the custom message and registering it with AutoCAD:

```c
acedRegisterWatchWinMsg(MessageProc);
```

Registering the windows message and registering the handler can be conveniently performed during the load event of an ARX application. Here is the implementation of the message handler itself along with the helper functions it uses:

```c
void MessageProc(const MSG *pMsg) {
    // WM_AutoloaderOnProfileSwitchMsg
    if (pMsg->message == AU2011_Adsk_ThreadDrawMessage) {
        // make sure AutoCAD is quiescent, otherwise wait and try
        // again later
        if (!isQuiescent()) {
            static MSG msg;
            msg.hwnd = pMsg->hwnd;
            msg.message = pMsg->message;
            msg.wParam = pMsg->wParam;
            msg.lParam = pMsg->lParam;

            DWORD threadId;
            HANDLE hThread = ::CreateThread(NULL, 0, Sleeper,
                (LPVOID)&msg, 0, &threadId);
            return;
        }

    //bool bFoundApps =
    gAppUtil.checkAppFoldersForNewApps(AcAppUtil::kModeProfileSwitch);
    AddCircle();
    aecedPostCommandPrompt();
    return;
}
```
// else if (pMsg->message == gnAppNotificationMessageId) {
    // AppNotificationManager::GetInstance()->pumpQueue();
// }

DWORD WINAPI Sleeper(LPVOID pParam)
{
    if (!pParam)
        return 0;

    MSG* pMsg = (MSG*)pParam;
    HWND hWnd = pMsg->hwnd;
    UINT uMsg = pMsg->message;
    WPARAM wParam = pMsg->wParam;
    LPARAM lParam = pMsg->lParam;

    Sleep(5000);

    // post to the main thread we need a review of the loaded apps
    ::PostMessage(hWnd, uMsg, wParam, lParam);
    return 0;
}

static void AddCircle()
{
    static AcGePoint3d ctrPt=AcGePoint3d(0,0,0);
    acDocManager->lockDocument(curDoc(), AcAp::kWrite);
    AcDbDatabase *pDb = curDoc()->database();

    AcDbObjectPointer<AcDbCircle> pCir;

    { AcDbBlockTableRecordPointer
      pMdlSpc(ACDB_MODEL_SPACE, pDb, AcDb::kForWrite);
        if(Acad::eOk==pMdlSpc.openStatus())
        {
          pCir.create();
          pCir->setRadius(10);
          pCir->setCenter(ctrPt);
          AcDbObjectId objId;
          pMdlSpc->appendAcDbEntity(objId, pCir);
        }
    }
    acDocManager->unlockDocument(curDoc());
    ctrPt.setToSum(ctrPt, AcGeVector3d(15,15,0));
}

In the code above, the message handler checks if AutoCAD is Quiescent and if it is not, it creates another thread that sleeps for 5 seconds before resending the original windows. 
message again. Once AutoCAD is ready, a circle is drawn at specific locations within the model space.

This way, the mechanism we built maintains the integrity of the AutoCAD UI thread while also executing the instructions of the worker thread. Here is a simple .NET piece of code that uses the above registered windows message to send messages to AutoCAD from worker threads:

```csharp
[CommandMethod("TrdsInAcad")]
public void TrdsInAcad()
{
    if (0 == AU2011_Adsk_ThreadDrawMessage)
    {
        AU2011_Adsk_ThreadDrawMessage = RegisterWindowMessage(msgString);
    }
    for (int threadCount = 0; threadCount < 5; threadCount++)
    {
        Thread t = new Thread(PostMessageToMainThread);
        t.Start();
    }
}
static void PostMessageToMainThread()
{
    Thread.Sleep(5000);
}
```

As you can see, once we have a robust message handling infrastructure in place, we can build a highly performant and responsive AutoCAD application that uses a multi-core or multi-CPU architecture efficiently.

**Conclusion**

Though having a multi-threaded application has many benefits, creating a robust multi-threaded application that actually realizes performance and responsiveness goals is hard. It is much harder with AutoCAD applications which additionally have to deal with restrictions on the API that need to be used only in the main UI thread and also make sure that AutoCAD works on worker thread instructions only when it is quiescent. However, once we have a robust mechanism in place to deal with these problems, creating multi-threaded AutoCAD applications can be very rewarding.

In the AU class, I will talk about and demonstrate all these issues as well as some cool threading models that could make designing parallel applications a fun thing to do.
Bibliography


