

Accelerating Excel 2010 with Windows HPC Server 2008 R2: MS_logo_KConverting Cluster-Safe UDFs for Offloading to a Windows HPC Cluster

Published: July 2010

By: Duncan Werner, Managing Member, Structured Data LLC

**Abstract**

Microsoft Excel 2010 extends the UDF model to the cluster by enabling Excel 2010 UDFs to run in a Windows HPC 2008 R2 cluster. In the cluster, UDFs work much like traditional UDFs, except that the calculation is performed by one or more servers. The key benefit is parallelization. If a workbook contains calls to long-running UDFs, multiple servers can be used to evaluate functions simultaneously.

In order to run on the cluster, the UDFs must be evaluated as appropriate for a cluster, marked as cluster-safe, and then the XLL file must be recompiled. This article walks through an example of how to convert an existing XLL file. The article also explains how UDF offloading works, performance considerations, and how to determine if a UDF is appropriate for cluster offloading.

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, Excel, SQL Server, Visual Basic, Visual Studio, Windows, and Windows Vista are registered trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

[Introduction 4](#_Toc261858140)

[Intended Audience 4](#_Toc261858141)

[Download Files 4](#_Toc261858142)

[Overview 5](#_Toc261858143)

[How it Works 6](#_Toc261858144)

[Before You Start 6](#_Toc261858145)

[Converting an XLL to use Cluster-Enabled UDFs 7](#_Toc261858146)

[Performance Testing 14](#_Toc261858147)

[Performance Considerations 15](#_Toc261858148)

[Restrictions & Limitations 16](#_Toc261858149)

[Additional Considerations 19](#_Toc261858150)

# Introduction

Excel has supported extension with User-Defined Functions (UDFs) since Excel version 4.0 in 1992. As each new version of Excel is released, more and more features are available to UDF developers. Excel 2007 added support for larger data types, larger spreadsheets, and multi-threaded calculation, and all of these were available to UDF developers through the Excel SDK.

In Office 2010, Excel adds support for calculating UDFs on a Windows HPC cluster. This new feature can potentially change the way we think about UDFs – opening the door to new kinds of Excel calculations and parallel processing.

In many ways this can be seen as the next logical step from multithreaded calculation – multithreading allowed UDFs to take advantage of all the processing cores on a single desktop machine, and cluster-enabled UDFs go a step further to take advantage of all the processing cores available on a Windows HPC cluster.

In terms of performance, it is a move in that direction. By running UDFs on a Windows HPC cluster, you can potentially perform tens, hundreds, even thousands of calculations in parallel. But moving to the cluster adds some new complexity, and the cluster infrastructure adds overhead to each calculation. So it’s not a solution to every problem, and it may require you to change the way you design and build your user-defined functions.

In this article we’ll walk through the process of converting an existing UDF library to take advantage of cluster computation. Then we’ll walk through deploying a cluster-enabled UDF to the cluster and using it in Excel 2010. We’ll discuss the performance you can expect in normal conditions, and the considerations you should take into account before adopting cluster-enabled UDFs in your environment.

Finally we’ll address the limitations and restrictions on cluster-enabled UDFs: what functions are available, what functions are not available, and how to work around some common issues converting desktop UDFs to cluster-enabled UDFs.

# Intended Audience

This article is intended for developers familiar with building and deploying UDFs in Excel Extension Libraries (XLLs). If you are not familiar with building XLLs, but you have experience building and compiling applications in C/C++, you should still be able to follow the examples below.

For more information on building XLLs in C/C++, see

<http://support.microsoft.com/kb/178474>

# Download Files

Included with this article are Visual Studio solutions for both VS 2008 and VS 2010. In the steps below we’ll walk through the process of converting a normal desktop UDF library (included in the Visual Studio solutions) to run on an HPC Cluster. If you’d prefer not to walk through the code development steps, final versions of the Add-in are provided for both 32-bit and 64-bit versions of Excel 2010.

The download files also include sample Excel workbooks which use the new UDF. We’ll describe these workbooks in the sections that follow.

# Overview

In this article we will not walk through the process of building a UDF library from scratch; that’s outside of the scope of this document. We will assume that you are familiar with building XLL libraries, or that you have a pre-existing library (including source code) which you’d like to convert to use an HPC cluster.

For the purposes of this article, we’ll use an example XLL that is included in the download files. The download includes complete source code for the XLL and a few test workbooks.

The basic XLL includes a single user-defined function, *LSMAmericanPut*, which calculates the value of a put option using the Longstaff-Schwartz least-squares Monte Carlo (LSM) method. It’s not important to understand too much about how the code works: it will be clear what it does when you look at the workbooks. For more information on the methodology and the underlying algorithm, see

<http://escholarship.org/uc/item/43n1k4jb>

The important thing for the purposes of this article is that the calculation is slow. It’s a complex, iterative calculation: depending on the parameters, it may run thousands of individual price calculations (using the Monte Carlo method), each of which can have hundreds of individual steps; and each of those individual steps can involve a number of matrix operations.

As with a lot of approximation functions, you can run more calculations in the algorithm to increase precision, but at the expense of performance. Fewer calculations can be faster, but then you add potential error to the generated value.

Slow is a relative term in computation, of course. Even with very high precision, calling the function one time may not take more than one second on a modern desktop workstation. In typical situations, however, we want to call the function many times – to generate prices for a large number of securities or to walk through different parameters. If the function takes one second, running 1,000 functions in a spreadsheet will take more than 15 minutes.

This is where the cluster can help. By running the user-defined functions on an HPC cluster, even a workbook containing 1,000 pricing functions can run faster – because the functions can run in parallel on multiple compute nodes. On a cluster with 64 compute cores, the same spreadsheet which takes 15 minutes on the desktop can calculate in 15 seconds.

*Note: it’s not quite as simple as a straight-line performance improvement. As we’ll discuss below, there are many considerations impacting performance: HPC session overhead, network latency, process startup time, and so on. Nevertheless performance can really improve when you use an HPC cluster, as you’ll see when you run the example.*

# How it Works

A typical user-defined function is built in an Excel Extension Library (XLL), which is a code library loaded by Excel at runtime.

When Excel encounters a UDF, it locates the library containing the UDF and makes a function call against that library. The library executes the calculation and returns a result; Excel then uses this result as the function value. All this happens within the Excel process running on the desktop.

HPC Services for Excel adds two new components: the *Cluster Connector* and the *XLL Container*. When Excel encounters a UDF that is marked as cluster-enabled, instead of loading the XLL library, it loads the Cluster Connector. The Cluster Connector is a library provided with HPC Server 2008 R2, which is installed when you install the HPC Server 2008 R2 client utilities.

Excel passes the function call – the name of the function and any arguments – to the Cluster Connector. The Cluster Connector includes code that can connect to an HPC cluster and send calculation requests. Every time Excel passes a function call to the Cluster Connector, it will send a request to the HPC cluster.

On the HPC cluster, the scheduler passes these calculation requests to the XLL Container service. The XLL Container locates the XLL library (installed on the compute node), and executes the function call. When the function call completes, the XLL Container passes the result back to the HPC scheduler; the HPC scheduler returns the result to the Cluster Connector on the desktop; and the Cluster Connector passes the result back to Excel.

All this happens automatically. To a user working with a spreadsheet, there is no difference in how UDFs are used whether they run on the desktop or on the cluster. Users can enable or disable cluster calculation at any time using an Excel option setting (as we’ll discuss below).

# Before You Start

In the sections that follow we’ll walk through the steps of building and running a cluster-enabled UDF. This article assumes that you have an HPC cluster installed and configured. The UDF calculation service is installed by default with HPC Server 2008 R2 (Enterprise edition only); in the HPC Cluster Manager application, you can use the diagnostic test **Microsoft** > **Excel** > **UDF Service Loading Test** to validate your configuration.

You’ll need Excel 2010 (either the 32-bit or the 64-bit version) running on your desktop workstation, and you’ll need Visual Studio to build and debug the code samples.

On the cluster, you’ll need the HPC Pack 2008 R2 Enterprise and HPC Pack 2008 R2 for Workstation edition installed on the head node, and on the nodes that you will use for UDF offloading. The Express edition of HPC Pack 2008 R2 does not include the cluster-side features of HPC Services for Excel. For more information, see [HPC Services for Excel](http://technet.microsoft.com/en-us/library/ff877820(WS.10).aspx).

# Converting an XLL to use Cluster-Enabled UDFs

Open the XLL project from the download files (ExampleClusterUDF) in Visual Studio. Solutions are provided for both Visual Studio 2010 and Visual Studio 2008.

**Select a Platform Architecture**

In Visual Studio, build the XLL for the 32-bit architecture or the 64-bit architecture, depending on the version of Excel 2010 you have on your desktop workstation. XLL files are dynamically-loadable libraries (DLLs), which are loaded into the process space of the running Excel application. Therefore they must match the code architecture of the Excel process, either 32-bit or 64-bit. If you need to support users of both architectures, you will need to build multiple versions of the XLL and deploy the appropriate XLL to each user’s machine.

The HPC cluster can support both 32-bit and 64-bit XLLs. When you deploy the XLL to the cluster, you should use the same architecture (32 or 64-bit) as you are using on the desktop.

As we described in the section “How it Works”, above, when Excel uses the cluster to calculate a user-defined function it uses the Cluster Connector to create a connection to the HPC cluster; and on the cluster the XLL Container loads the XLL library and executes the function call.

The Cluster Connector and the XLL Container are designed to match the architecture of Excel running on your desktop workstation. If you are running 32-bit Excel on the desktop, the service will look for 32-bit XLLs deployed on the compute nodes; conversely if you’re using 64-bit Excel on the desktop, the service will look for 64-bit XLLs deployed on the compute nodes.

To build a 64-bit XLL you will need x64 support available in Visual Studio. In VS 2008, this was not installed by default. If you have not previously installed x64 support, run the VS 2008 installer and select **Add or Remove Features**; then select **x64 Compilation Support** and add the feature. In VS 2010 x64 support is installed by default.

*Note: if you build cluster-enabled XLLs with Visual Studio 2010, it may be a good idea to change the platform toolset. In the project properties, change the option* ***Configuration Properties*** *>* ***General*** *>* ***Platform Toolset*** *to* ***v90****; this matches version 90 of the Visual Studio C runtime library. If you build against version 100 (the default in Visual Studio 2010), you’ll need to deploy the latest C runtime to the cluster compute nodes and any client machines that don’t already have it installed.*

**Build the Solution**

Build the solution for your target architecture. The solution should compile successfully as-is if you have the necessary compilation tools. If you receive any errors, check for missing tools or libraries and make changes as necessary.

**Test the XLL in Excel 2010**

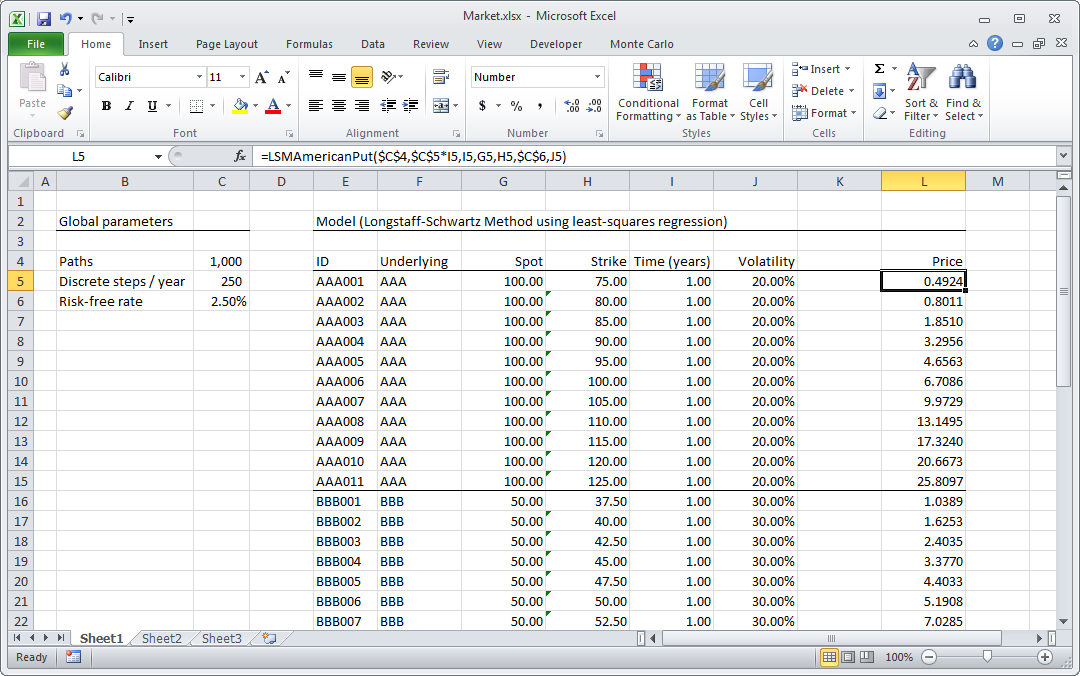
Start Excel 2010. You can run this from the Visual Studio debugger if you change the project options **Configuration Properties** > **Debugging** > **Command** to launch Excel 2010, or you can start Excel normally from the Windows Start menu.

In Excel, load the new XLL. From the main Excel 2010 window click the **File** tab on the ribbon. Click **Options** on the left. In the Excel options dialog, select **Add-ins** on the left. At the bottom, in the drop-down list, select **Excel Add-ins** and click **Go**.

In the Excel Add-ins dialog, click **Browse** and locate the Add-in you built in the last step. This will be in the project directory, either in the directory **Release** (if you built a 32-bit version), or in the directory **x64\Release** (if you built a 64-bit version).

If you receive a dialog message saying the file is not a valid Excel Add-in, you may have built the solution for the wrong platform architecture. Double-check that you are building the correct architecture for your version of Excel (either 32-bit or 64-bit). You can check which version of Excel you have installed from the **File** tab of the ribbon; click **Help** on the left and check the version under **About Microsoft Excel**.

After you have installed the XLL, open the example workbook “Market.xlsx”, included in the download files. This workbook uses the UDF function to generate prices for a number of securities.



In this spreadsheet, column L contains the UDF function *LSMAmericanPut*. Each row in the spreadsheet represents a single security price. On the left-hand side, we have some general parameters which are applied to every row: **Paths** and **Discrete Steps** affect the precision of the calculation, and **Risk-free Rate** is a general property used for each calculation. In the rows, columns E:J are values which are specific to each security.

Change one of the global parameters and recalculate the spreadsheet. Change the **Risk-free Rate**, for example. The workbook will recalculate and you will see values in column L update. This should be reasonably fast on most desktop workstations.

Now change the value of **Paths** in cell C4 to a larger value. If you set **Paths** to 10,000, prices will update very slowly – it may take up to a second to recalculate each row (this workbook includes about 40 rows so it won’t take more than about 40 seconds). While Excel is calculating, it may not repaint the screen so you may see blinking - you can tell when Excel is finished, because you’ll be able to select cells in the spreadsheet.

**Update the XLL for Cluster Calculation**

Next we’ll rebuild the XLL so it supports calculation on the HPC cluster. Close Excel (it’s not important to save the spreadsheet) and return to Visual Studio. There are three changes to make to any XLL to support cluster-enabled UDFs.

1. Update the XLL for the Excel 2010 SDK

If an XLL was built for an earlier version of Excel, you’ll need to update the SDK files to match the latest version. There are three important files: XLCALL.h, XLCALL.cpp, and XLCALL32.lib (the header, source file, and static library, respectively). The Excel2010 SDK includes updated versions of these files.

In Visual Studio, remove the files XLCALL.h and XLCALL.cpp from the XLL project. Add the new versions of these files (included in the download files in the **Excel2010** directory) to the project. Update the project properties to set the include directory to the Excel2010 directory; in the project properties dialog, update the value in **Configuration Properties** > **C/C++** > **General** > **Additional Include Directories** to point to the Excel2010 directory.

This XLL does not use the library XLCALL32.lib. If you are updating an XLL that links against XLCALL32.lib, change the linker settings to point to the updated version of this file. The Excel 2010 SDK includes versions of XLCALL32.lib built for both 32-bit and 64-bit architectures.

If you are updating an existing XLL library, you will need to link against XLCALL32.lib if you include any calls to the functions **Excel4** or **Excel4v**. Beginning with Excel 2007, Microsoft included support for the new XLOPER12 type as a replacement for the XLOPER type. The Excel 2007 SDK also included the functions **Excel12** and **Excel12v**, which replaced the older **Excel4** and **Excel4v** functions.

If you don’t need to support versions of Excel prior to Excel 2007, you can build XLLs entirely without any calls to **Excel4** or **Excel4v**. However if you are updating an existing library, including these functions won’t affect your ability to build cluster-enabled UDFs. Any function you mark as cluster-enabled must not use **Excel4** or **Excel4v** – it should only use XLOPER12s and the **Excel12** or **Excel12v** functions. But other functions in the same library can use the older functions and older value type.

The function we’ll update in this example only uses the XLOPER12 type, so that’s not an issue here.

1. Update the function registration signature

If you’re familiar with XLLs and UDFs, you’ll know that any function added to Excel must be registered. Registration is typically handled in the **xlAutoOpen** or **xlAutoRegister** callback functions. When a function is registered, you pass in a string representing the argument types and return value. In this example, the function registration is handed in **xlAutoOpen**. The actual function registration strings are listed in a table, in the file **ExampleClusterUDF.h**.

The function *LSMAmericanPut*, listed in ExampleClusterUDF.h, is registered with the type signature "QJJBBBBB". That means it returns an XLOPER12 type (Q), and takes a variety of parameters as Integer (J) and Double (B) types.

To mark a function as cluster-safe, the only change to the function registration signature is adding an ampersand (“&”) to the function registration. Change the function registration string to match the following:

" LSMAmericanPut", " QJJBBBBB&", " LSMAmericanPut",

so that the function registration string ends with an ampersand. That’s the only change that’s required to the code to mark the function as cluster-enabled.

However it’s simple in this case only because we know that the function is safe for running on the cluster. In the sections that follow, we’ll discuss the limitations of cluster-enabled UDFs, and describe what functions are and are not available when you run a function on an HPC cluster.

1. Rebuild the XLL

Rebuild the XLL with the changes you just made. If it compiles successfully, you’re ready to run the UDF on the cluster. If you receive any compilation errors, double-check the changes from the last section. In particular make sure that the path to the new XLCALL.h file is correct.

**Test the updated XLL on the desktop**

Before running on the cluster, open Excel and the “Market.xlsx” workbook again to make sure the XLL is functioning correctly. If you don’t enable cluster calculation, the UDF will run on the desktop as before. Change one of the global parameters – the **Risk-free Rate**, for example – and check that the calculations update as expected.

**Deploy the XLL to the cluster**

Now that the Add-in is updated, you can deploy it to the cluster. The XLL library must be installed on each compute node (or each compute node you will use to run Excel UDF calculations).

You have a few options in deploying the XLL to the cluster compute nodes. The suggested directories for XLLs are

C:\Program Files\Microsoft HPC Pack 2008 R2\Bin\XLL64

for 64-bit XLLs, and

C:\Program Files\Microsoft HPC Pack 2008 R2\Bin\XLL32

for 32-bit XLLs. However, when a compute node attempts to load an XLL library it will search the PATH, including your user home directory. If you have Administrator privileges on the cluster, use the suggested directories above. If you don’t have Administrator privileges, you can use your home directory (e.g. C:\Users\MyUserName).

The easiest way to deploy the XLL to the cluster compute nodes is to use clusrun, an application provided with the HPC Client Utilities (we’ll run clusrun from HPC Powershell). You will need a share directory that is visible to the compute nodes. Create a share directory on the cluster head node, for example.

If you can create a share directory on the cluster head node, you can use the following steps to deploy the XLL to the cluster compute nodes:

1. Create a share directory on the head node. Name this directory “HPCTemp” or something similar.
2. Copy the XLL library from your project build directory to the share directory. Open an HPC Powershell window (from the Windows Start menu, **Programs** > **Microsoft HPC Pack 2008 R2** > **HPC Powershell**) and navigate to the project build directory. Run the command

> copy ExampleClusterUDF.xll \\Path\to\share\directory

1. Use clusrun in HPC Powershell to copy the file from the share directory to the target directory on the cluster compute nodes. In this example we’ll use the default directory for 64-bit XLLs; adjust the path to use the directory appropriate for your installation.

> clusrun /scheduler:HeadNodeName /all copy \\Path\to\share\directory\ExampleClusterUDF.xll

'C:\Program Files\Microsoft HPC Pack 2008 R2\Bin\XLL64'

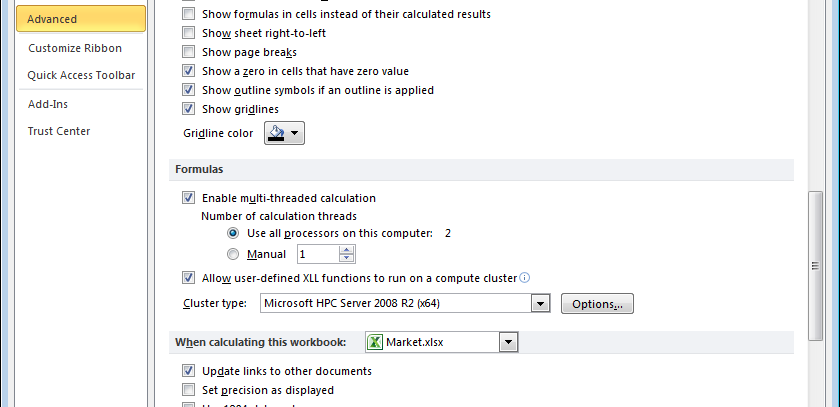
(enter the command on one line). The clusrun command tells the cluster head node (identified by the /scheduler parameter) to run the command on all nodes (with the /all parameter). It will copy the file from the share directory to the target directory on all cluster nodes.

**Run the UDF on the HPC cluster**

Now that the UDF is deployed, you’re ready to run it on the cluster. Go back to Excel and open the “Market.xlsx” spreadsheet.

Click the **File** tab in the ribbon and click **Options** on the left. In the Excel options dialog, click **Advanced** on the left.

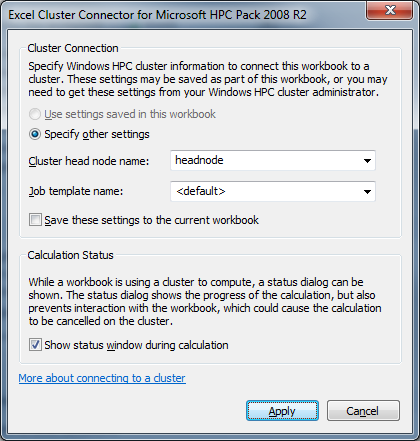
Scroll down to the **Formulas** section:



Check the box marked **Allow user-defined XLL functions to run on a compute cluster**. In the **Cluster type** drop-down box, select the entry for x64 or x32, matching your version of Excel.

If you don’t see any settings for **Cluster type**, or the drop-down box is disabled, make sure you have installed the HPC Server 2008 R2 client utilities. Run the HPC Server 2008 R2 installer on your desktop, and install the client utilities.

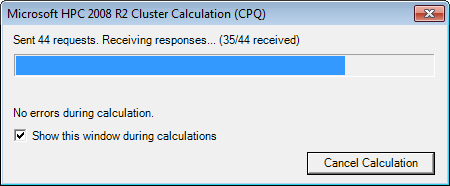
Next click the **Options** button to open the cluster options dialog:



In the options dialog, change the setting for **Cluster head node name** to match your installation. Check the box **Show status window during calculation**. Leave the rest of the settings at their default values, and click **Apply**.

Now that you have enabled cluster calculation, any time Excel finds a function marked cluster-enabled (with an ampersand in the function signature, as you saw in the last section), it will contact the cluster head node and send a calculation request.

Make a change to the spreadsheet – change the **Risk-free Rate**, for example – and you’ll see the cluster calculation run. When the calculation executes, you’ll see a dialog box with calculation status:



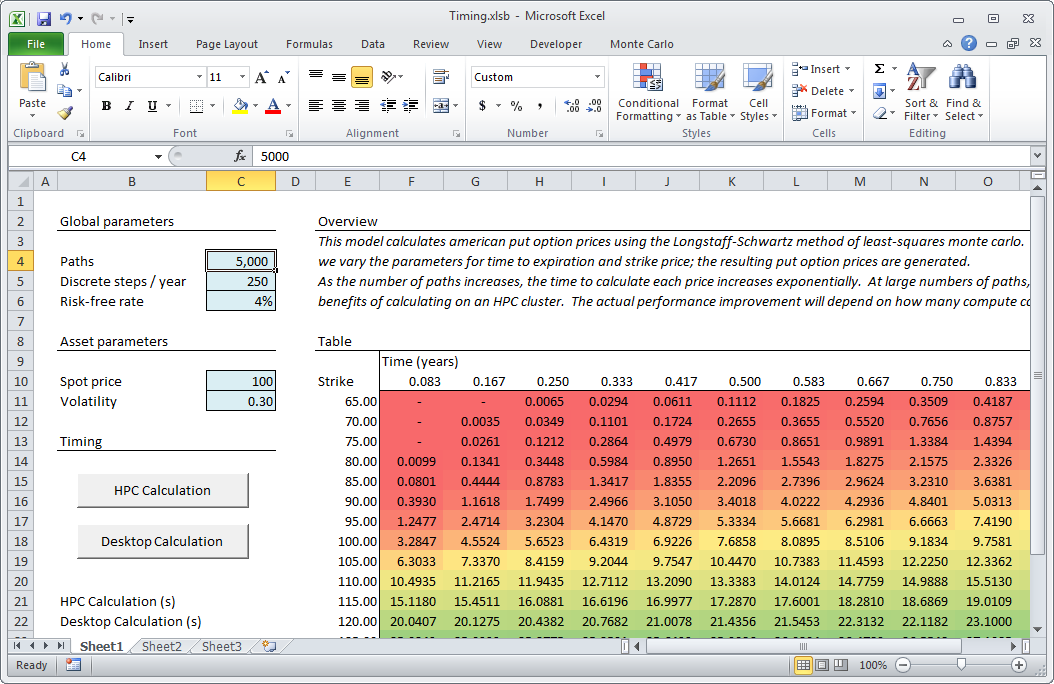
If you see any errors during the calculation, click the **Show Errors** button (not pictured here) for more information. Some errors are not fatal; if you see errors of the type “Session Terminated”, you can safely ignore them. The “Session Terminated” error means that the cluster session was closed unexpectedly – in that event Excel will automatically open a new session and re-submit the calculation requests as necessary.

The most common problem at this point will be an error of the type “UDF”; this usually means that the UDF was not deployed correctly. If you receive an error of type “UDF”, check that the XLL is correctly installed on the cluster compute nodes. Double-check your settings against the description in the previous section. If possible, open a remote desktop connection to one of your cluster compute nodes and ensure that the file is installed in the correct directory. If necessary, re-deploy the file and try again.

If you receive any other errors, check the event log for more information. In the event viewer, you’ll find these events logged under **Applications and Services Logs** > **Microsoft** > **HPC** > **Excel**. There are additional categories for administrative, debug, and operational logs.

# Performance Testing

We’ve included a second spreadsheet, “Timing.xlsb”, which you can use to check the timing of both cluster and desktop calculation. This spreadsheet uses macros, so if you open it you may be prompted with a warning. Click **Enable Macros** to enable this content.



The Timing.xlsb spreadsheet includes a large table in which we vary some of the securities parameters. Each cell in the table represents a single price, with a call to the *LSMAmericanPut*function. The global settings for **Paths** and **Discrete Steps** are set at moderate values.

Click one of the two buttons on the spreadsheet to launch a calculation. These buttons use VBA to enable or disable cluster calculation, and then recalculate the spreadsheet.

Run the two calculation types to compare the total calculation time. In most cases, you should see a significant improvement when using the cluster calculation. The exact performance will be dependent on your desktop workstation (processor speed, &c.) and your cluster configuration (the number of available cores).

*Note: to enable or disable cluster calculation in VBA, use the property Application.UserClusterConnector. See the VBA source code for the Timing.xlsb spreadsheet for an example.*

# Performance Considerations

There are a number of factors which impact performance of cluster-enabled UDFs. Not all UDFs are good candidates for running on the cluster; only those which take a significant time to calculate should be considered.

**Workbook considerations**

When a UDF calculates in parallel, you can potentially run hundreds or even thousands at the same time (depending on your cluster configuration, or how many compute cores you have available). Therefore even for a long-running UDF, you will only get performance benefits if you have a large number of individual function calls. If you have just a single UDF call in your spreadsheet, running on the cluster can’t improve performance – even if the single call takes a long time to complete.

You can enable or disable cluster calculation at any time, using the Excel options dialog or using VBA macros (as described above). So the decision of whether to convert a UDF need not necessarily consider its use in typical workbooks. However it’s not usually necessary to convert UDFs which are used just a few times.

**UDF considerations**

Calculating a UDF on an HPC cluster includes a number of steps, as described in the section “How it Works”, above. Each of these steps takes some small amount of time. Taken together we describe this as *overhead*. Overhead includes a few key components:

|  |  |  |
| --- | --- | --- |
| Component | Order of time | Description |
| Creating the HPC Session | Seconds | The first time Excel sends a function call to the cluster, it must establish a session with the HPC scheduler. This typically takes a few seconds, but if the cluster is busy running other jobs, establishing a session may take longer.  You can see the effect of session startup in the Timing.xlsb spreadsheet. If you start Excel and run the HPC test, it will include the session startup time. If you run the calculation a second time, it will use the existing session, and it should run a few seconds faster. |
| Sending the UDF call to the cluster | Milliseconds | Each function call is sent over the network to the cluster for calculation, and takes some number of milliseconds |
| Loading the XLL | Milliseconds | The first time a function is called, the XLL Container must locate and load the XLL library |
| Receiving results | Milliseconds | As with sending function calls, receiving results requires a transfer over the network which takes a few milliseconds |

Generally speaking, you should only convert slow or long-running UDFs to run on the cluster. If the time to run a single UDF call is less than the overhead of sending the call to the cluster, you may not receive any performance benefits.

The actual consideration is somewhat complicated, because even if the UDF is relatively fast, if you have the ability to run hundreds of calculations simultaneously, you may still see better performance on the cluster than you would on the desktop.

There’s no fixed rule defining when it’s appropriate to execute a UDF on an HPC cluster. If you use a function only once in your spreadsheet, even if the function is very slow you won’t see any performance improvement when you execute it on the cluster – because the benefits of cluster calculation only accrue when you can run many tasks in parallel.

Similarly if you have a function that executes quickly, but you use the function thousands of times in your spreadsheet, you may see a benefit from cluster calculation even though you are adding overhead to each individual function call. This is because even if one individual calculation is slower on the cluster, if you can run hundreds of calculations in parallel the overall execution will be faster.

How many calculations you can run in parallel is dependent on your HPC cluster installation. As a general rule, the maximum number of calculations you can run in parallel is the number of available compute cores (the total number of individual processors available on the cluster compute nodes). In practice, the actual number may be less than that. If other jobs are running on the cluster, you may not be able to utilize all the available cores. You (or your administrator) may define job templates which limit Excel calculations to a subset of available cores.

Therefore the decision of whether and when to execute UDFs on the cluster is very dependent on the particular characteristics of each function, your spreadsheet, and available cluster resources. Nevertheless because Excel 2010 offers users the ability to enable or disable cluster calculation at any time, if you build a cluster-enabled UDF you (or your users) will always have the option of using cluster calculation when necessary or appropriate.

# Restrictions & Limitations

In the above section “How it Works”, we describe the difference in execution between desktop UDFs and cluster-enabled UDFs. Briefly, desktop UDFs execute within the Excel process space; and cluster-enabled UDFs execute within an XLL Container, on one or more compute nodes.

This may require some changes to how your UDFs work, and it means that some UDFs may not be appropriate for running on the cluster.

UDFs, and XLL libraries, are written in C/C++ and usually include some basic calculations. Any basic C/C++ code will run normally on the cluster. The limitations on what you can do in cluster-enabled UDFs fall into three general categories: (1) Excel function calls; (2) data types; and (3) the process environment.

**Excel function calls**

The Excel API includes functions for calling into the running Excel process (**Excel12** and **Excel12v** in Excel 2007 and later), and with these function calls you can run any number of Excel functions – such as looking up data in a spreadsheet, checking the name of the file or worksheet, even getting Excel’s window handle. These functions work because the XLL runs within Excel’s process space. Any information that is available to Excel on the desktop can be queried and interrogated with the Excel API functions.

Similarly, you can make changes to Excel as the function runs. You can update cells or change the window title, for example.

When a UDF runs on the cluster in the XLL container, it is not running in the Excel process and therefore these function calls won’t work. You cannot get a pointer to the Excel window handle, for example, or change the Excel window title, because the UDF is running on a completely different machine and has no access to the Excel process.

More importantly, when a UDF runs in the XLL container, the XLL container does not have a reference to the spreadsheet being calculated. That means that you cannot look up or change values in the spreadsheet.

Some of the Excel API functions are available to UDFs running on an HPC cluster. Typically these are functions which do not require access to the spreadsheet or the running Excel process. For example, you can use the Excel memory allocation and deallocation routines.

A UDF which executes on the cluster should make no calls to the Excel API that require access to the spreadsheet or to the running Excel process.

**Data Types**

For the purposes of building UDFs, the Excel has a large number of data types which can be used as function parameters and function return values. Most of these are available to cluster-enabled UDFs. Primitive types (Integer, Double, Boolean) can always be used – these are the J, B, and L types, respectively, in function signatures.

You can also use the XLOPER12 (Q) type for function arguments and return values. XLOPER12 is a C union, which can itself contain a variety of different value types. You can use the Q type to pass or return any primitive type, the xlError enumeration, or even arrays of values.

*Note: when using the Q type for function arguments or return values, the maximum data size for any argument or return value is 64K.*

You cannot use the XLOPER12 reference (U) type. The reason you can use the Q type but not the U type is that the U type is designed to include references, rather than the values themselves. Any data which you would normally pass with the U type can be passed with the Q type, but it will be converted from a reference to a value or set of values.

You can think of this as passing by value, as opposed to passing by reference. If one of the arguments to your function is a range of 10 cells in the spreadsheet, if you use the U type the function will receive a reference to that range. In a desktop UDF, you can use the Excel API to retrieve the values from that range. However this Excel API function is unavailable when the UDF is running on the cluster – because it requires reading values from the spreadsheet.

If you pass the same argument as the Q type, when Excel makes the function call it will read the range of 10 cells, convert them to values, and then send the values as the argument to the UDF. Because the function is now dealing with values instead of references, the function will work both on the desktop and on the cluster.

In designing cluster-enabled UDFs, use primitive types and the Q type as arguments and as the return type; do not use the U type.

**The Process Environment**

A last set of restrictions on cluster-enabled UDFs has to do with the process environment: where the UDFs execute, and the life cycle of XLL libraries.

On the desktop, XLL libraries containing UDFs are typically loaded once by the Excel process – when Excel starts, or whenever the user loads the XLL in the Add-ins dialog.

After an XLL has been loaded, it will remain in memory (in the Excel process space) until Excel exits or the XLL is specifically unloaded by the user. Many XLLs take advantage of this by creating objects or other data structures in memory, for use by later function calls.

For example, imagine a UDF that includes an incrementing counter representing the number of times a workbook has been recalculated. This UDF would typically include a static variable – representing the counter – and each time the function was called, it would increment the static variable and return the value. That works on the desktop because the XLL library containing the UDF will be in memory as long as Excel is running.

On the cluster, on the other hand, UDFs are loaded only as long as necessary. Typically XLLs will be loaded by the XLL Container service (on cluster compute nodes) when Excel sends the first calculation request, and they will be unloaded when the cluster session ends.

In some cases the session might stay open as long as you have Excel running on the desktop, but you cannot rely on this behavior. In many cases the session will close if it is inactive or if another job is running on the cluster, and when the session closes, XLLs running on the cluster will be unloaded.

In addition, because the cluster is designed to spread work over multiple compute nodes, there might be multiple instances of the XLL library running at the same time, on different physical machines. (In fact because the XLL Container typically runs per-core, in a cluster configuration, there may be many instances of the XLL library running on any single compute node). In that environment, shared memory and static variables won’t be available, or may return inconsistent values.

UDFs which run on the cluster must not rely on shared memory or static variables. A UDF designed for execution on an HPC cluster should be atomic, meaning it refers only to local variables during the calculation.

# Additional Considerations

Even if your UDF meets the requirements described above, and even if you feel you would get performance benefits from running on the cluster, there are still a few things to take into consideration when deciding whether to convert a UDF.

**Cluster-enabled UDFs are only supported in Excel 2010**

In Excel 2010, the user can elect to run calculations on the desktop or on the cluster using the configuration option. However when a cluster-enabled UDF is registered, it includes the ampersand (“&”) in the parameter declaration. Earlier versions of Excel will not recognize this character, and as a result they will not register the function. That means that a function which is marked as cluster-enabled will not work in Excel 2007 or any previous version.

If you need to support both Excel 2010 and previous versions of Excel, consider building multiple versions of the XLL library: one with the cluster-enabled function for Excel 2010, and one without the ampersand in the declaration to support Excel 2007 (or earlier versions).

Another option is to build multiple functions in your XLL, one with cluster support and one without. Users can then make the determination which function to use based on their environment.

**You must rebuild the XLL to create cluster-enabled UDFs**

There is no way to “automatically” treat existing UDFs as cluster-enabled UDFs. To run on the cluster, the UDF must include the ampersand symbol in its function registration. That means you must have access to the source code and the ability to recompile the UDF in order to take advantage of cluster calculation.

Because there are significant limitations and restrictions on cluster-enabled UDFs, it’s important that code be specifically compiled to support execution on the cluster. In addition to changing the function signature, rebuilding the XLL gives you the opportunity to verify that you are not using any unsupported data types or making any calls to the Excel API which are not supported on the cluster.