Wanted: More Processing Speed!

Jody L Meyer
White Light Computing
42759 Flis Drive
Sterling Heights, MI 48314-2850
Voice: (616) 384-2151
Fax: (586) 254-2539
Email: JLMeyer@WhiteLightComputing.com

ParallelFox is the brain-child of VFPx Award winner, Joel Leach. Joel has made parallel processing easier and more approachable for all VFP developers by developing this powerful tool. Jody will walk you through the main components of ParallelFox and demonstrate the basics. She will then show you how you might be able to apply ParallelFox to real-world process such as error-handling, data dictionary rebuilds, and time-consuming reports. The possibilities are endless. ParallelFox is a must have in your developer toolkit to get more speed out of your Visual FoxPro application.

In the following pages to come, we will journey together to discover terms associated with parallel processing, what ParallelFox is, the basics of how to use ParallelFox and possible uses for such a powerful too.

Why would you want to look at ParallelFox?

Bottom-line…. Speed. It's hard to imagine squeezing anymore out of your already pristine processing desktop application code. You have optimized your queries. Your code is as efficient and concise as you can make it, but your customers are still complaining it's too slow! ParallelFox might be the answer you are looking for. It might help you squeeze more processing in less time for certain processes. Before we start looking at some code… let's talk about some basics.

Terms to Know

Parallel Processing

According to Wikipedia (http://en.wikipedia.org/wiki/Parallel_processing), parallel processing is defined as:

"The simultaneous use of more than one CPU or processor core to execute a program or multiple computational threads. Ideally, parallel processing makes programs run faster because there are more engines (CUPs or cores) running it. In practice, it is often difficult to divide a program in such a way that separate CPUs or cores can execute different portions without interfering with each other. Most computers have just one CPU, but some models have several, and multi-core processor chips are becoming the norm. There are even computers with thousands of CPUs."

The best way for me to describe this is to look at my home with me as the CPU. The different chores I have to perform are the processes. When my son, David, was little, he couldn't help me do household chores such as dishes, vacuuming, dusting and cooking dinner. I did these chores all by myself, one at a time. It didn't matter what order I did them in. They just had to get done. However, I couldn't do any of these chores or processes simultaneously. I was only one engine or CPU to do one chore or process at a time. But, when my son got older and I trained him how to vacuum. Viola, my home now has 2 CPUs. While David is vacuuming, I can do the dishes, dust or cook dinner. We can get the household chores done in less time because they can be done at the same time.

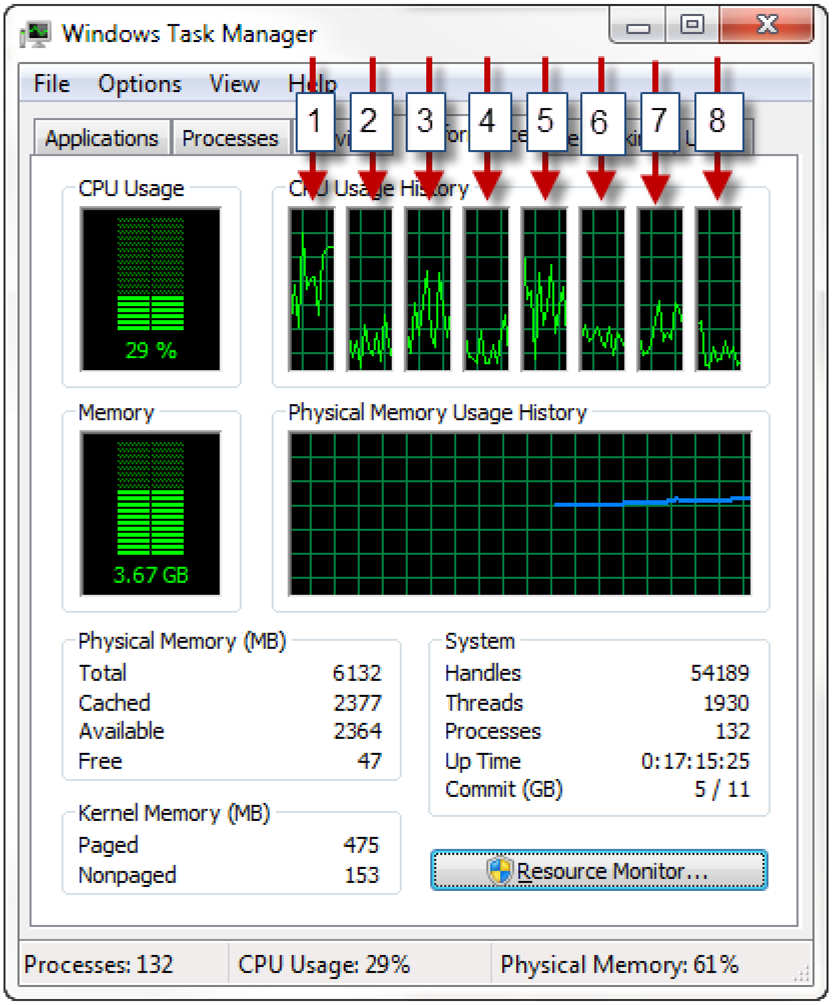
Hyper-Threading

According to Wikipedia (http://en.wikipedia.org/wiki/HyperTreading):

"*Hyper-threading works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources. This allows a hyper-threading processor to appear as two "logical" processors to the host operating system,* *allowing the operating system to schedule two threads or processes simultaneously. When execution resources would not be used by the current task in a processor without hyper-threading, and especially when the processor is stalled, a hyper-threading equipped processor can use those execution resources to execute another scheduled task. (The processor may stall due to a cache miss, branch misprediction, or data dependency.)"*

Let's go back to the example of my home. We live in an old house with old plumbing. We have learned that taking a shower needs to wait while the laundry is running otherwise we might get scalded. However, if the laundry process is in a state where it's just turning the clothes and not demanding more water, we can take a shower. Hyper-threading does much the same thing. If a process is in a wait state, another process can be executed.

My computer has a quad core or 4 CPUs. With hyper-threading turned on, my computer now has 8 logical processors. (See figure 1.)



Physical Memory (MB)	
Total	6132
Cached	2377
Available	2364
Free	47

Kernel Memory (MB)	
Paged	475
Nonpaged	153

System	
Handles	54189
Threads	1930
Processes	132
Up Time	0:17:15:25
Commit (GB)	5 / 11

Processes: 132 CPU Usage: 29% Physical Memory: 61%

Figure 1: My computer has 4 physical CPUs but with hyper-threading turned on, it has 8 logical processors.

Synchronous

Synchronous processing, according to Microsoft Technet (http://technet.microsoft.com/en-us/library/cc978253.aspx):

“Synchronous processes wait for one to complete before the next begins.”

It’s how we program in VFP today (unless of course you have already started to program something like or are already using ParallelFox). It is:

- One line of code at a time.
- Sequential: Step1 ...then... Step 2 ...then... Step 3 ...
- Control returns when called program or step is complete.

Asynchronous

According to Wikipedia (http://en.wikipedia.org/wiki/Asynchrony) asynchronous is defined as:

“In programming, **asynchronous** events are those occurring independently of the main program flow. **Asynchronous** actions are actions executed in a non-blocking scheme, allowing the main program flow to continue processing.”

ParallelFox is a parallel processing library (COM object) for Visual FoxPro. If we want to take advantage of the un-paralleled (*groan*) goodness, we need to start thinking asynchronously.

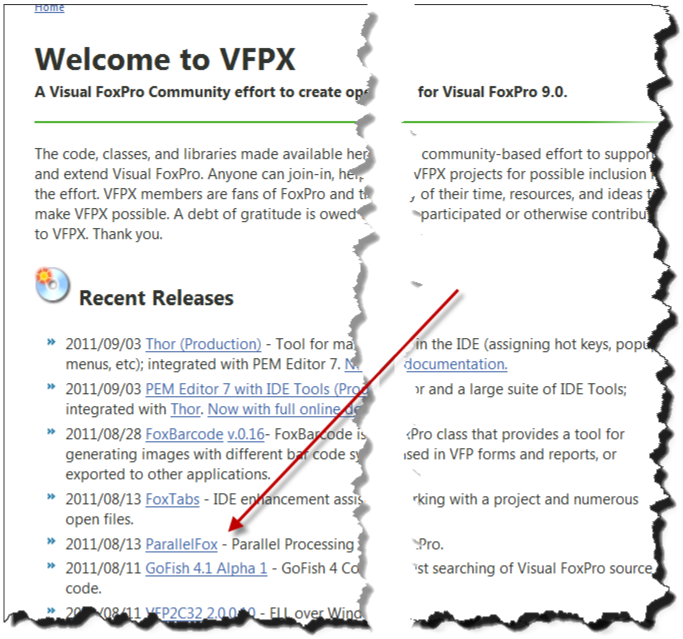
- Step 1, Step 2, Step 3.... Can run at the same time.
- Step 3 may finish before Step 1. Step 2 may complete after Step 5.
- Control returns immediately once the call to a step is made. No waiting involved.
- It’s definitely a ***different*** mindset.

ParallelFox Requirements

- Visual FoxPro 9.0 SP2
- CPUs – Even if you have one CPU, you can take advantage of hyper-threading (as long as you don’t have it turned off via the BIOS setup). (Every time I re-read this I chuckle. It’s something Joel said in his videos and caught me funny. Thanks Joel!)

Installation

- Go to VFPx: http://vfpx.codeplex.com/ (See figure 2)



Home

Welcome to VFPX

A Visual FoxPro Community effort to create op for Visual FoxPro 9.0.

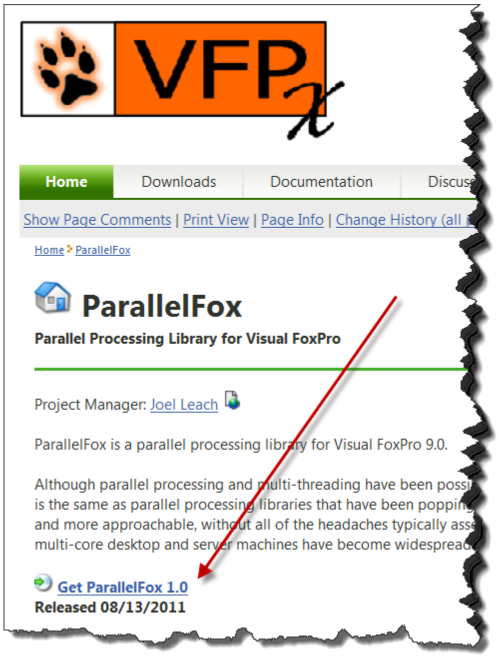
The code, classes, and libraries made available her community-based effort to suppor and extend Visual FoxPro. Anyone can join-in, he VFPX projects for possible inclusion the effort. VFPX members are fans of FoxPro and th of their time, resources, and ideas t make VFPX possible. A debt of gratitude is owed participated or otherwise contribu to VFPX. Thank you.

Recent Releases

- 2011/09/03 Thor (Production) - Tool for ma in the IDE (assigning hot keys, popu menus, etc); integrated with PEM Editor 7. N documentation.
- 2011/09/03 PEM Editor 7 with IDE Tools (Pro or and a large suite of IDE Tools; integrated with Thor. Now with full online d
- 2011/08/28 FoxBarcode v.0.16- FoxBarcode is Pro class that provides a tool for generating images with different bar code s sed in VFP forms and reports, or exported to other applications.
- 2011/08/13 FoxTabs - IDE enhancement assis king with a project and numerous open files.
- 2011/08/13 ParallelFox - Parallel Processing Pro.
- 2011/08/11 GoFish 4.1 Alpha 1 - GoFish 4 Co st searching of Visual FoxPro source code.
- 2011/08/11 VFP2C32 2.0.0.0 - FLL over Wind

Figure 2: Welcome to VFPX

- Under Recent Releases, click on **ParallelFox**. The most current release as of this writing is August 13, 2011 as of this writing.
- In the ParallelFox page, click on :**Get ParallelFox 1.0** (see figure 3) and follow the instructions:



Home | Downloads | Documentation | Discus

Show Page Comments | Print View | Page Info | Change History (all

Home > ParallelFox

ParallelFox

Parallel Processing Library for Visual FoxPro

Project Manager: Joel Leach

ParallelFox is a parallel processing library for Visual FoxPro 9.0.

Although parallel processing and multi-threading have been possi is the same as parallel processing libraries that have been poppin and more approachable, without all of the headaches typically ass multi-core desktop and server machines have become widespread

Get ParallelFox 1.0

Released 08/13/2011

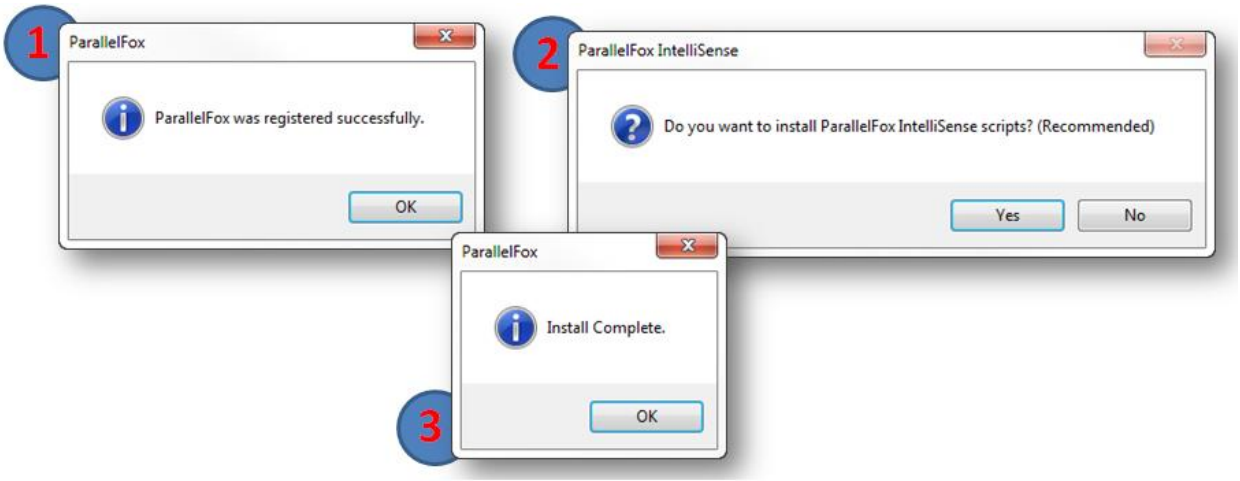
Figure 3: Download ParallelFox 1.0 from the ParallelFox page

- Unzip to a directory of your choice.

    - For me, I put all my VFPx tools in the c:\VFPx folder. So for ParallelFox, it goes in c:\VFPx\ParallelFox.
- Start Visual FoxPro as Administrator so we can register ParallelFox. ParallelFox is a COM object and needs to be registered.
    - ***Note***: A great read is Doug Hennig's Southwest Fox 2011 session "*Developing VFP Applications for Windows 7*"
- In the command window, you will want to set your default to where you have copied ParallelFox. For me the command line would be:

```
SET DEFAULT TO C:\VFPx\ParallelFox
```

- To register ParallelFox, in the command window type in:
    - DO Install.prg
    - Press the Enter key
- You will see three Messageboxes appear (see figure 4). I highly recommend you select "Yes" to the ParallelFox IntelliSense scripts. It saves a lot of time typing.



ParallelFox was registered successfully.

Do you want to install ParallelFox IntelliSense scripts? (Recommended)

Install Complete.

Figure 4: After running ParallelFox\Install.prg, the following message boxes and dialog windows will appear.

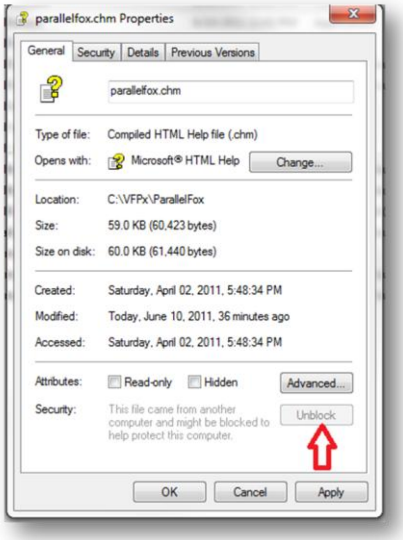
What comes with ParallelFox

- ParallelFox.Exe (COM object)
- ParallelFox project and source code
- Examples (21 programs)
- Help - ParallelFox.chm
    - ParallelFox Overview
    - Class Parallel

  - Class Worker

- IntelliSense
- Install.prg
- Uninstall.prg
- Training Videos! Found on VFPx

The ParallelFox.chm help file is jammed packed with everything you might want to know about ParallelFox but it might not be easy to open. If you run into difficulties opening the help file, you will want to right click on the file and in the General page tab, select **Unblock**.



parallelfox.chm Properties

parallelfox.chm

Type of file: Compiled HTML Help file (.chm)
Opens with: Microsoft® HTML Help

Location: C:\VFPx\ParallelFox
Size: 59.0 KB (60,423 bytes)
Size on disk: 60.0 KB (61,440 bytes)

Created: Saturday, April 02, 2011, 5:48:34 PM
Modified: Today, June 10, 2011, 36 minutes ago
Accessed: Saturday, April 02, 2011, 5:48:34 PM

Attributes: □ Read-only □ Hidden
Security: This file came from another computer and might be blocked to help protect this computer.

Note: If you would like to know more about this problem with opening up the help files, please check out Rick Strahl's blog: *http://www.west-wind.com/weblog/posts/2008/Dec/16/Problems-with-opening-CHM-Help-files-from-Network-or-Internet.*

Closer look at ParallelFox

Parallel Object

This object is used in the main process. It provides functionality for sending commands and managing the parallel processing. Joel recommends you instantiating this object as a local variable object:

```
Local Parallel as Parallel
Parallel = NewObject("Parallel", "ParallelFox.vcx")
```

Worker Object

This object provides features for use inside of workers to perform the work and communicate with the main process. Use the Parallel object to start the workers. Joel recommends you start the workers in your main application program. For example:

```
Parallel.StartWorkers(“MyApplication.exe")
```

Worker Processes

Workers are separate instances of VFP (one for each logical processor) that assists the main process and runs code in parallel. The workers are separate processes and not threads within the main process. (Note: You might want to check out Kevin Ragsdale's Southwest Fox session "Easy Multi-Threading with Visual FoxPro")

A Main Process

This is the place that you will want to instantiate the parallel object. What do I mean by Main Process? So glad you asked! This is a process where work is being done. It's calling to do Step 1 … then … Step 2 … and … then … Step 3, etc. This main process is where ParallelFox will fit in nicely and queue up work for the worker objects to do if the steps can be processes asynchronously.

- A Main process queues command
- A Main process sends the command to the Worker
- The Worker executes the command
- The Worker notifies the main process when the command is completed
- While the Worker is processing, the Main process can continue on to do other commands.

Getting Started

ParallelFox enables parallel and background processing by running your code in "worker" processes. Before you can run any code with ParallelFox, there are a few things you might want to consider:

- At the start of your application,
- Before a process,
- During a process,
- After a process,
- And on shutdown of your application

Setting up the Workers via the Parallel Object

In the beginning, at the start of your application, before you start up your parallel processing, there are a number of things to consider:

- Include the ParallelFox class libraries in your application,
- _Vfp.AutoYield,
- Hyper-threading,
- CPU count,
- Number of workers you might want to run,
- And whether you want to run them in debug mode or not.

You will want to include the ParallelFox class libraries in your application:

```
* Add the ParallelFox class libraries to the application
SET CLASSLIB TO ParallelFox ADDITIVE
SET CLASSLIB TO WorkerMgr ADDITIVE
```

It is ***highly*** recommended to have ***_Vfp.AutoYield*** set to ***.T.*** (true) which is Visual FoxPro's default. There might be reasons why you would not want to do this, such as the use of ActiveX controls, but if _Vfp.AutoYield is set to .F. (false), this could block the code running on your workers. ***Note:*** See ParallelFox example program: AutoYield.prg which is demoed in video Worker Events: Part 2.

You will want to know if hyper-threading is enabled. You can determine this by using ***Parallel.DetectHyperThreading()***. If Parallel.DetectHyperThreading() returns a .T. then hyper-threading is enabled. If it returns a .F. then hyper-threading is disabled. This will help you to determine how many workers you want to use for parallel processing. Remember, you can set up a worker for each logical processor. For my computer, I could set up 8 workers, one (1) for each logical CPU.

To determine the number of CPUs that are available on the computer in use, ParallelFox has a property called ***CPUCount***. This will contain the number of logical processors. If hyper-threading is enabled, there will be two (2) logical processors per one (1) physical CPU.

Once you have determined how many logical processors you have available, you will use the **Parallel.SetWorkerCount()** to set the number of workers. The default is *Parallel.CPUCount* or all the logical processors on the computer. But you might not want to do that. There are a couple of reasons that come to mind:

- You don't want your Visual FoxPro application using all the CPUs available so that the user can still run Outlook, Word, Excel, etc. will ease.
- You want to test your code.

Here are a number of ways to set the worker count:

```
Local Parallel as Parallel
Parallel = NewObject("Parallel", "ParallelFox.vcx")

* Set the worker count to 1 for Debug Mode
Parallel.SetWorkerCount(1)

…  OR …

* Set the worker count to physical CPU count
IF Parallel.DetectHyperThreading()
   m.lnWorkerCount = Parallel.CPUCount / 2
ELSE
```

```
   m.lnWorkerCount = Parallel.CPUCount
ENDIF

Parallel.SetWorkerCount(m.lnWorkerCount)

…  OR …

* Set the worker count to all the logical CPUs available
Parallel.SetWorkerCount(Parallel.CPUCount)
```

After all that, you are now ready to start the workers. The parallel object can be instantiated as a global object. However, Joel recommends that it is best to run as a local object for each program/process that will be taking advantage of ParallelFox. Parallel.StartWorkers() starts the workers. For example:

```
* Starting Parallel main application program
LOCAL Parallel AS Parallel
Parallel = NewObject(‘Parallel’, ‘ParallelFox.vcx’)
Parallel.StartWorkers(‵MyApp.exe′,FULLPATH(‵MyApp.exe′),m.llDebugMode)
```

NOTE: **m.llDebugMode**. If m.llDebugMode is ***.T.*** (true), the workers are in debug mode. You will see a separate visible instance of Visual FoxPro for each worker. This comes in handy when debugging the worker code. If m.llDebugMode is ***.F.*** (false), the workers are run in the background with no visible Visual FoxPro instance being displayed.

In a Main Process Within the Application

Somewhere in your application, you will have a process that can be made parallel by using ParallelFox. Remember, this process needs to be able to run asynchronously. In other words, the steps in the process are independent of each other. For example, step 3 could finishes before step 1 and step 2 might finishes after step 5.

In the beginning of that process, you will create a local instance of the Parallel object. Remember the Parallel object manages the parallel process and sends commands to the workers:

```
* Starting Parallel main process
LOCAL Parallel AS Parallel
Parallel = NewObject(‘Parallel’, ‘ParallelFox.vcx’)
```

After the Parallel object has been instantiated, you will want to queue the workers filling them with work to do. There are a number of Parallel methods that can be called:

Parallel.Do(): Executes a program or procedure on a worker. ***Note:*** See VFPx ParallelFox example programs which are demoed in the video “*Running Code in Parallel*”:

- Steps_Before.prg

- Steps_After.prg

Parallel.Call(): Executes/Calls a function on worker. ***Note:*** See VFP ParallelFox example programs which are demoed in the video "*Running Code in Parallel*":

- Call_Before.prg
- Call_After.prg

Parallel.CallMethod(): Executes/Calls a class method on the worker. ***Note:*** See VFP ParallelFox example programs which are demoed in the video "*Running Code in Parallel*":

- CallMethod_Before.prg
- CallMethod_After.prg

Parallel.DoCmd(): Executes a single command on the worker. ***Note:*** See VFP ParallelFox example programs which are demoed in the video "*Running Code in Parallel*":

- DoCmdScript_Before.prg
- DoCmdScript _After.prg

Parallel.ExecScript():Executes a script on the worker. ***Note:*** See VFP ParallelFox example programs which are demoed in the video "*Running Code in Parallel*":

- DoCmdScript _Before.prg
- DoCmdScript _After.prg

All the above methods contain a ***lAllWorkers*** parameter which tells ParallelFox to run the command on all workers. This is a very useful when setting up the environment in the worker. For example, your startup program probably instantiates an application object and sets up the system's environment. You will probably want to do the same type of things in your workers. By passing ***.T.*** as lAllWorkers, the setup for the environment will be done on all the workers. Why do you want to do this? Because the Workers are running in their own environment separate from the application.

ParallelFox adds commands to a queue and executes those commands as workers become available. You can queue as many commands as you like, but the number of workers determines how many commands will run simultaneously or in parallel.

At the End of a Main Process Within the Application

All queued worker processes run asynchronously, which means that you may need to tell the main process to wait until all the processes are completed before continuing. ***Parallel.Wait()*** is used for that purpose. In the code listing below, you see the work being queued up by the ***ParallelDo()***. This queuing will happen very fast, but the workers need time to process the queued work. The ***Parallel.Wait()*** puts the program in wait mode until all the workers have completed the queued work and now the Messagebox() can now be displayed.

```
LOCAL Parallel AS Parallel
Parallel = NewObject(‘Parallel’, ‘ParallelFox.vcx’)
Parallel.StartWorkers(FULLPATH(‘YourProgram.prg’),,m.llDebugMode)

FOR m.lnx = 1 TO 100

   Parallel.Do(‘YourProgram.prg’,,,m.lnx)  && Queueing up the workers
ENDFOR

* Wait until commands are complete
Parallel.Wait()
= MESSAGEBOX(‘Process is complete.’,64,‘YourProgram’)
* Continue the rest of program
```

At the End of the Application

At the end of your application you will want to stop all the workers. By default, ParallelFox will wait until all pending commands are completed before stopping the workers. In your ON SHUTDOWN code, you will want to include the ***Parallel.StopWorkers()*** to stop the workers from further processing. An example of shutdown is listed below:

```
LOCAL Parallel AS Parallel
Parallel = NEWOBJECT('Parallel', 'ParallelFox.vcx')


ON SHUTDOWN

Parallel.StopWorkers()

CLOSE ALL

*-- Wait until worker shutdown is complete
WAIT ‘Shutting Down Workers...’ WINDOW NOWAIT
Parallel.Wait()

WAIT CLEAR

* Exit app
SET PROCEDURE TO
CLEAR EVENTS
CLOSE ALL
RETURN
```

Class Worker

The workers as you will recall are separate instances of VFP (with a maximum of 1 for each logical processor) that assists in the main process and runs code in parallel. The Worker class provides an avenue to raise events and communicate with the main process.

Worker.ReturnCursor(): Returns a cursor to the main process.

Worker.ReturnData (): Returns data to the main process

Worker.UpdateProgress(): Sends progress update to the main process

Worker.CPUCount: Number of logical cores on the computer

Worker.ProgressInterval: Minimum number of seconds between progress updates

Worker Events

ParallelFox sends commands to the workers to run programs in parallel or in the background. Messages can be sent back to the main progress from the workers by means of worker events. To do this, the main process must bind to the worker events via ***Parallel.BindEvent()***.

There are 4 available worker events:

- **Complete**: Fires when a program has finished running on a worker. The event includes the return value of the program.
- **ReturnData**: Fires when ***Worker.ReturnData()*** is called. Used to send data (up to 26 parameters, include objects) from the worker to the main process.
- **UpdateProgress**: Fires when ***Worker.UpdateProgress()*** is called. Used to send the main process progress updates when a long process is running on the worker. ***Worker.ProgressInterval*** controls how often the updates are sent.
- **ReturnCursor**: Fires when ***Worker.ReturnCursor()*** is called. Used to send a cursor from the worker to the main process.
- **ReturnError**: Fires when ***Worker.ReturnError()*** is called. Used in worker error handler to notify main process an error has occurred.

Debugging

Debugging in ParallelFox can be a little tricky. To debug ParallelFox, you will need to run the workers in debug mode by simply passing .T. as the third parameter to the ParallelFox.StartWorkers() method:

```
LOCAL oParallel AS Parallel OF ParallelFox.vcx
oParallel.SetWorkerCount(1)
```

```
m.llDebugMode = .T.
oParallel = NewObject(‘Parallel’,‘ParallelFox.vcx’)
oParallel.StartWorkers(FULLPATH(‘MyProc.prg’),,m.llDebugMode)
```

The Parallel.StartWorkers() third parameter passed as .T., tells ParallelFox to start the workers in full instances of VFP, rather than as standard COM objects. You can now simply add SET STEP ON in your worker code and the debugger window will open in the worker instance. I highly recommend you set your WorkerCount to 1. It’s much easier to step through one worker’s code instead of 8 workers.

You will need to set the pathing so the main process knows where ParallelFox.vcx and WorkerMgr.vcx. The Worker object needs to know where the ParallelFox.vcx is as well.

Running the code… is it really faster?

Joel has a number of examples that come with ParallelFox. The two examples that intrigued me the most were the CallMethod_before.prg and CallMethod_after.prg. The CallMethod_before.prg doesn’t run the code in parallel. The CallMethod_after.prg runs the same process but in parallel. I also changed the CallMethod_after.prg code around a bit to peg my laptop’s CPU a couple of different ways. I first tried just processing with the physical cores. Since I have a quad-core laptop, I set the worker count to 4. The second time, I used all logical cores which is 8. Listed below is the code:

```
* Example calling method using just the physical cores.
Local ;
   Lni ;
,lnTimer ;
,loMyObject

Set Path To "..;examples" Additive

Local Parallel as Parallel of ParallelFox.vcx
Parallel = NewObject("Parallel", "ParallelFox.vcx")

*-- Since I have a Quad-Core laptop and HyperThreading is enabled, I set the Worker
Count to 4.
Parallel.SetWorkerCount(4)

Parallel.StartWorkers(FullPath("CallMethod_After.Prg"),,.t.)

m.lnTimer = Seconds()

For m.lni = 1 to 16
       ? "Running Units of Work", i, 50
       Parallel.CallMethod("Test", "MyObject", FullPath("CallMethod_After.prg"),,,
50)
EndFor

Parallel.Wait()
? "Total Time", Seconds() - lnTimer
```

```
Return

*-----------------------------------
DEFINE CLASS MyObject AS Custom

*-----------------------------------
Procedure SimulateWork
*-----------------------------------
       Local ;
          lni
       For m.lni = 1 to 1000000
             * Peg CPU
       EndFor
EndProc
*-----------------------------------
Procedure Test
*-----------------------------------
       Lparameters tnUnits

       Local ;
          Lni

       ? Program(), tnUnits
       For m.lni = 1 to tnUnits
             This.SimulateWork()
       EndFor

EndProc

ENDDEFINE
```

When I ran the above code, my Windows Task Manager pegged the CPU as 50%. But, if I changed the worker count to include the logical cores made available to me via hyper-threading, I could peg the CPU to 100%. Table 1 lists the results of the 3 runs. Each time I ran the code, I would quit out of VFP and then come back in new.

Synchronous (not Parallel)	4 Workers	8 Workers
CPU: Pegged at 12%	CPU: Pegged at 50%	CPU: Pegged at 100%
20.717 seconds	12.824 seconds	8.467 seconds
20.826 seconds	12.152 seconds	8.569 seconds
20.623 seconds	10.860 seconds	8.449 seconds
20.576 seconds	12.855 seconds	8.418 seconds

Table 1: Differences between synchronous, parallel processing with 4 physical cores and parallel processing with 8 logical cores.

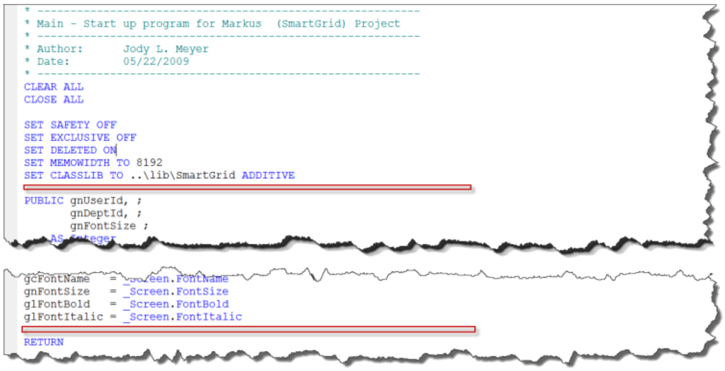
Conclusion: I think I'll try to figure out where I can use ParallelFox in all my lengthy and not so lengthy processes.

Using ParallelFox inside an application: How easy is it to do?

A few years back, I presented the SmartGrid to demonstrate how easy it is to use a grid for rapid development. Inside the SmartGrid processing was a setup form that did data dictionary processing: (Re)build and/or (Re)indexed tables.

I first had to think about the process. It made perfect sense to use ParallelFox here. Building, packing and indexing tables is not dependent on anything else and it doesn't matter what order the tables are processed. It was an easy target. I moved most of the rebuild and reindex processes to a procedure program. The workers will do building, packing and reindexing of the tables that are to be processes.

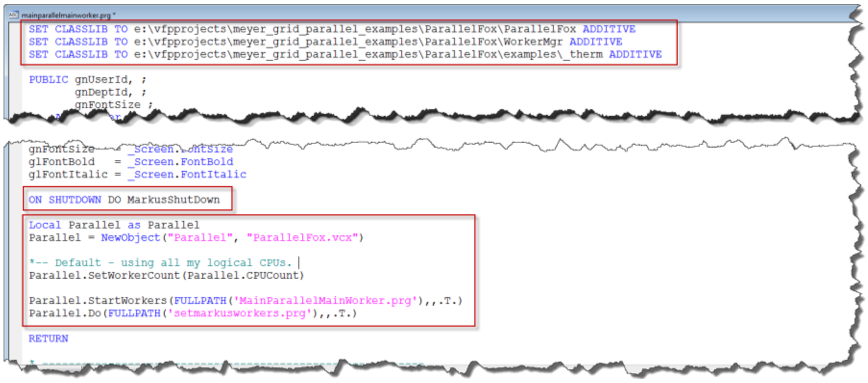
But before I changed the process for the SmartGrid's data dictionary, I changed my startup application program first. The code use to look like figure 5.



```
* -----------------------------------------------------
* Main - Start up program for Markus  (SmartGrid) Project
* -----------------------------------------------------
* Author:      Jody L. Meyer
* Date:        05/22/2009
* -----------------------------------------------------
CLEAR ALL
CLOSE ALL

SET SAFETY OFF
SET EXCLUSIVE OFF
SET DELETED ON
SET MEMOWIDTH TO 8192
SET CLASSLIB TO ..\lib\SmartGrid ADDITIVE

PUBLIC gnUserId, ;
       gnDeptId, ;
       gnFontSize ;
    AS Integer

gcFontName   = _Screen.FontName
gnFontSize   = _Screen.FontSize
glFontBold   = _Screen.FontBold
glFontItalic = _Screen.FontItalic

RETURN
```

Figure 5 Main application program before ParallelFox

The new code now looks like figure 6. As you can see, I added code to point to the ParallelFox class library and the WorkerMgr class library. I also added the FFC _therm class library for a process bar.

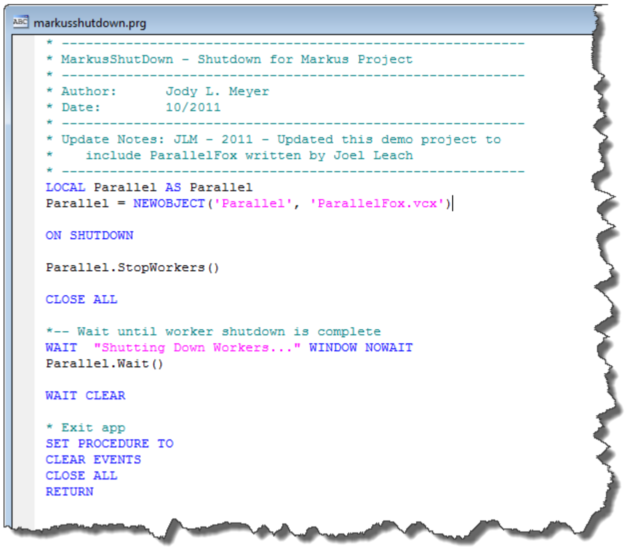


mainparallelmainworker.prg *

```
SET CLASSLIB TO e:\vfpprojects\meyer_grid_parallel_examples\ParallelFox\ParallelFox ADDITIVE
SET CLASSLIB TO e:\vfpprojects\meyer_grid_parallel_examples\ParallelFox\WorkerMgr ADDITIVE
SET CLASSLIB TO e:\vfpprojects\meyer_grid_parallel_examples\ParallelFox\examples\_therm ADDITIVE

PUBLIC gnUserId, ;
       gnDeptId, ;
       gnFontSize ;

gnFontSize   = _Screen.FontSize
glFontBold   = _Screen.FontBold
glFontItalic = _Screen.FontItalic

ON SHUTDOWN DO MarkusShutDown

Local Parallel as Parallel
Parallel = NewObject("Parallel", "ParallelFox.vcx")

*-- Default - using all my logical CPUs.
Parallel.SetWorkerCount(Parallel.CPUCount)

Parallel.StartWorkers(FULLPATH('MainParallelMainWorker.prg'),,,.T.)
Parallel.Do(FULLPATH('setmarkusworkers.prg'),,,.T.)

RETURN
```

Figure 6: Added the ParallelFox and WorkerMgr to my main application

The code for the ON SHUTDOWN also had to change to shut down all the workers. Please see figure 7 for the code listing. Please note the ***Parallel.StopWorkers(),*** telling ParallelFox to stop and shut down all the workers. The workers will finish processing what is in the queue. If you want the workers to stop immediately, you will pass in a parameter of .T. (true).



markusshutdown.prg

```
* ----------------------------------------------------------
* MarkusShutDown - Shutdown for Markus Project
* ----------------------------------------------------------
* Author:      Jody L. Meyer
* Date:        10/2011
* ----------------------------------------------------------
* Update Notes: JLM - 2011 - Updated this demo project to
*    include ParallelFox written by Joel Leach
* ----------------------------------------------------------
LOCAL Parallel AS Parallel
Parallel = NEWOBJECT('Parallel', 'ParallelFox.vcx')

ON SHUTDOWN

Parallel.StopWorkers()

CLOSE ALL

*-- Wait until worker shutdown is complete
WAIT  "Shutting Down Workers..." WINDOW NOWAIT
Parallel.Wait()

WAIT CLEAR

* Exit app
SET PROCEDURE TO
CLEAR EVENTS
CLOSE ALL
RETURN
```

Figure 7: Modified ON SHUTDOWN program to shut down the ParallelFox workers

To start my ParallelFox workers, I needed to tell ParallelFox how many workers I wanted and start the workers. I added code to the main application code:

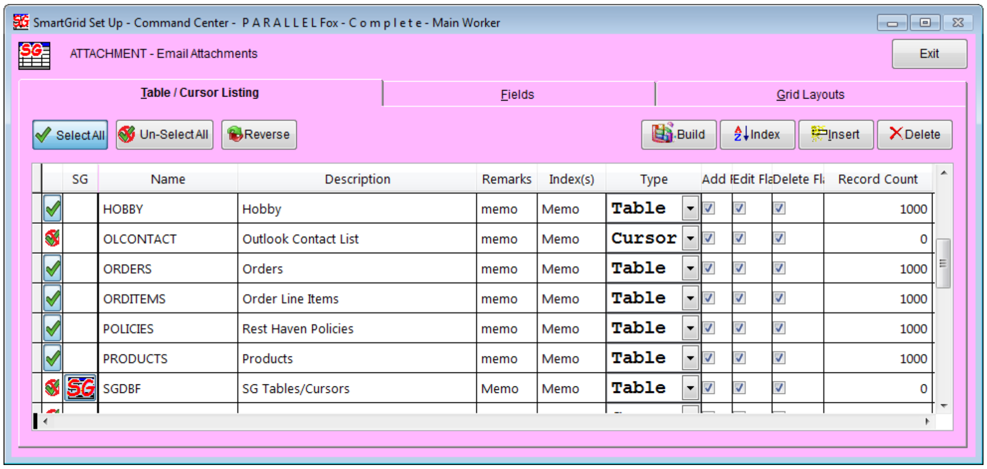
```
* Set ParallelFox Worker Count to all available logical CPUs
Parallel.SetWorkerCount(Parallel.CPUCount)  && If Debugging code, set to 1

* Start the workers - .T. is in DEBUG mode. For production, this will be set to .F.
Parallel.StartWorkers(FULLPATH('MainParallelMainWorkers.prg'),,.T.)
```

Once my workers are started, I want to set their environment. Remember, the workers are in their own separate environment and you will need to set each of the workers' environments to know about the main application. To do this, I told ParallelFox to do the workers' set environment program:

```
* Set all the workers' environments to be application aware
* Note: .T. tells ParallelFox do run this program on all workers
Parallel.Do(FULLPATH('setmarkusworkers.prg'),,.T.)
```

The SetMarkusWorkers.prg code sets all the same class libraries, pathing, etc. that the main application program has. In this way, I ensure that the workers can use the application's classes, methods, procedures, tables, etc. that they might need to run the worker code.

In my form, frmSGSetup.scx (see figure 8), that rebuilds and reindexes user selected tables, I changed the load Event to include ParallelFox by adding the oParallel project to the form. The last new line of code, I called the ParallelFox.BindEvent() to bind the worker's complete event to the form's MthIncrementProgress method which updates the progress bar.

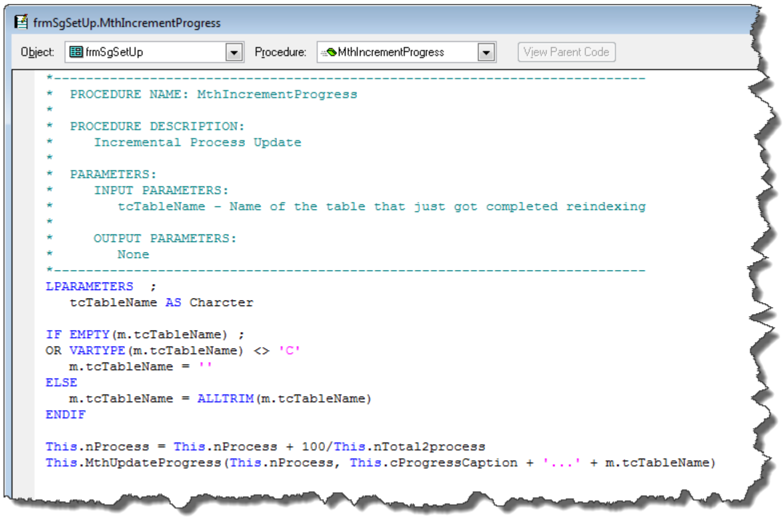

SmartGrid Set Up - Command Center - PARALLEL Fox - Complete - Main Worker
ATTACHMENT - Email Attachments
Exit
Table / Cursor Listing
Fields
Grid Layouts
Select All
Un-Select All
Reverse
Build
Index
Insert
Delete

	SG	Name	Description	Remarks	Index(s)	Type	Add I	Edit Fla	Delete Fl	Record Count
✓		HOBBY	Hobby	memo	Memo	Table	☑	☑	☑	1000
✗		OLCONTACT	Outlook Contact List	memo	Memo	Cursor	☑	☑	☑	0
✓		ORDERS	Orders	memo	Memo	Table	☑	☑	☑	1000
✓		ORDITEMS	Order Line Items	memo	Memo	Table	☑	☑	☑	1000
✓		POLICIES	Rest Haven Policies	memo	Memo	Table	☑	☑	☑	1000
✓		PRODUCTS	Products	memo	Memo	Table	☑	☑	☑	1000
✗	SG	SGDBF	SG Tables/Cursors	Memo	Memo	Table	☑	☑	☑	0

Figure 8 FrmSGSetup.scx: This form Rebuilds / Reindexes user selected tables. Please note the pink background is for demonstration purposes only.

```
*frmSGSetUp.Load
This.AddProperty('oParallel',NULL,1,'ParallelFox Object')
This.oParallel = NewObject('Parallel', 'ParallelFox.vcx')

* Bind the Workers.Complete event to the frmSGSetup.MthIncrementProgress
This.oParallel.BindEvent('Complete',ThisForm,'MthIncrementProgress')
```

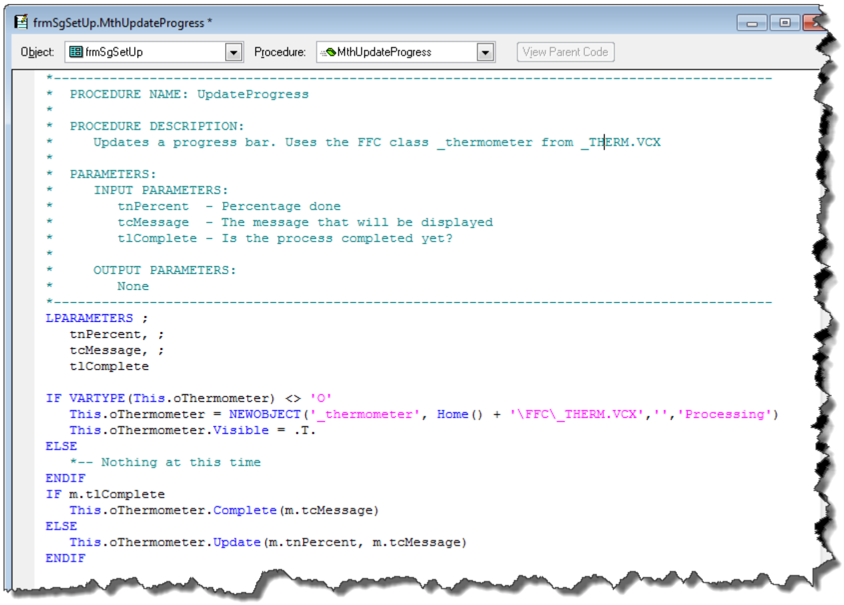
The form's MthIncrementProgress code is listed in Figure 9:


frmSgSetUp.MthIncrementProgress
Object: frmSgSetUp
Procedure: MthIncrementProgress
View Parent Code

```
*-------------------------------------------------------------------------
*  PROCEDURE NAME: MthIncrementProgress
*
*  PROCEDURE DESCRIPTION:
*     Incremental Process Update
*
*  PARAMETERS:
*     INPUT PARAMETERS:
*        tcTableName - Name of the table that just got completed reindexing
*
*     OUTPUT PARAMETERS:
*        None
*-------------------------------------------------------------------------
LPARAMETERS  ;
   tcTableName AS Charcter

IF EMPTY(m.tcTableName) ;
OR VARTYPE(m.tcTableName) <> 'C'
   m.tcTableName = ''
ELSE
   m.tcTableName = ALLTRIM(m.tcTableName)
ENDIF

This.nProcess = This.nProcess + 100/This.nTotal2process
This.MthUpdateProgress(This.nProcess, This.cProgressCaption + '...' + m.tcTableName)
```

Figure 9: The form's MthIncrementProgress is bound to the worker's complete event which will enable reporting progress back to the user.

The last piece of code to run the thermometer progress bar is MthUpdateProgress. See figure 10. This code just updates the thermometer progress bar form's percentage and message.



```
*-------------------------------------------------------------------------------------
*  PROCEDURE NAME: UpdateProgress
*
*  PROCEDURE DESCRIPTION:
*     Updates a progress bar. Uses the FFC class _thermometer from _THERM.VCX
*
*  PARAMETERS:
*     INPUT PARAMETERS:
*        tnPercent  - Percentage done
*        tcMessage  - The message that will be displayed
*        tlComplete - Is the process completed yet?
*
*     OUTPUT PARAMETERS:
*        None
*-------------------------------------------------------------------------------------
LPARAMETERS ;
   tnPercent, ;
   tcMessage, ;
   tlComplete

IF VARTYPE(This.oThermometer) <> 'O'
   This.oThermometer = NEWOBJECT('_thermometer', Home() + '\FFC\_THERM.VCX','','Processing')
   This.oThermometer.Visible = .T.
ELSE
   *-- Nothing at this time
ENDIF
IF m.tlComplete
   This.oThermometer.Complete(m.tcMessage)
ELSE
   This.oThermometer.Update(m.tnPercent, m.tcMessage)
ENDIF
```

Figure 10: The form's MthUpdateProgress updates the thermometer progress bar form.

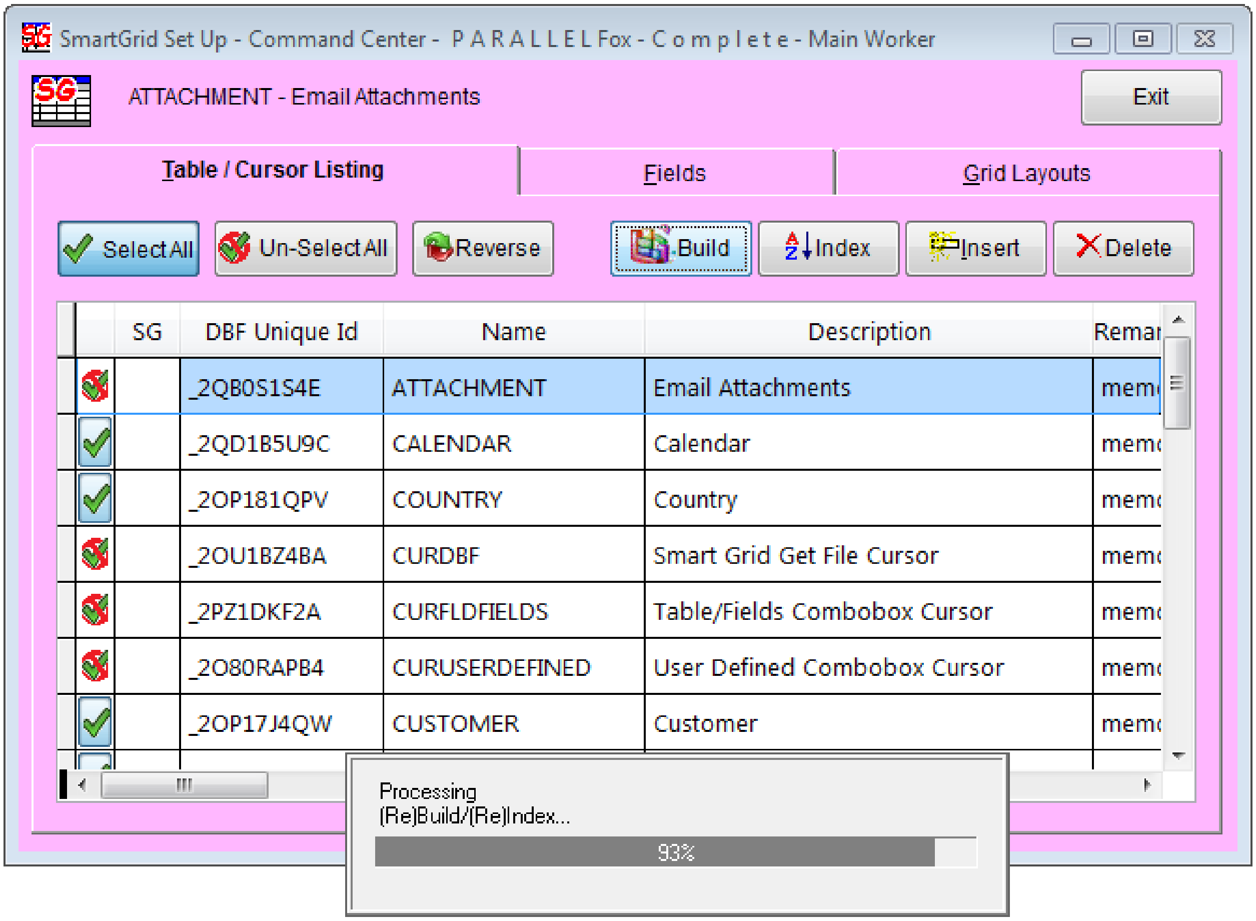
Now I turned my attention to the main process of rebuilding tables within this form. In my frmSGSetUp.MthProcessDbfBuildStructureInParallel method, I changed the code from being a very large SCAN….ENDSCAN to something quite simple, a call to the Worker via Parallel.CallMethod(). Please note in the code listed below, the CallMethod() passes in a number of parameters:

- lpProcessingDbfBuildStructureInParallel – the rebuild tables method within the SGSetupBizObj class
- SGSetupBizObj – the name of the class object that contains the lpProcessingDbfBuildStructureInParallel method
- FullPath("SGSetup.prg") – the name of the program that defines the SGSetupBizObj class object
- m.tlSilent – Don't display messaging for the reindexing process
- m.lcDbf – Name of the table
- &lcRecordSource..mIndex – the index tags contained in a memo field for the table
- &lcRecordSource..DbfGuidId – the unique id for the table record

```
* FrmSGSetup.MthProcessDbfBuildStructureInParallel
SELECT (m.lcRecordSource)
SCAN FOR lSelect  && Selected records pointing to tables that need to be rebuilt
   m.lcDbf = ALLTRIM(&lcRecordSource..cCursor)

   * Queue up the work for the ParallelFox Workers using the ParallelFox CallMethod()
   * lpProcessingDbfBuildStructureInParallel is the method in the SGSetupBizObj.prg
   * m.tlSilent tells the process to be silent
   * m.lcDbf tells the process what table to rebuild
   * &lcRecordSource..mIndex is the indexing commands to reindex the newly built table
   * &lcRecordSource..DbfGuidId is the unique id for the table that is getting rebuilt
   This.oParallel.CallMethod( ;
        'lpProcessingDbfBuildStructureInParallel', ;
        'SGSetupBizObj', ;
        FullPath("SGSetup.prg"),,, ;
        m.tlSilent, ;
        m.lcDbf, ;
        &lcRecordSource..mIndex,;
        &lcRecordSource..DbfGuidId)

ENDSCAN

* Tells ParallelFox to wait for all queued work to be completed before continuing to the next
  line of code.
This.oParallelWait()

* Update the progress bar to be completed at 100%
This.MthUpdateProgress(00, 'Completed Rebuild', .T.)

* Release the progress bar form
This.oThermometer.Release()
```

The hard work of rebuilding the table was now put into the hands of many workers instead of just one process.

With these code changes, the form's rebuild/reindex process can now be run in parallel. Please see figure 11 which is a snap shot in time while the selected tables are being rebuilt. Please note the progress bar. Remember, I bound the worker's Complete event to the form's MthIncrementProgress method by using the ParallelFox.BindEvent() to update the progress bar. If I had not done this, the progress would not have been displayed properly.


SmartGrid Set Up - Command Center - PARALLEL Fox - Complete - Main Worker
ATTACHMENT - Email Attachments
Exit
Table / Cursor Listing
Fields
Grid Layouts
Select All
Un-Select All
Reverse
Build
Index
Insert
Delete

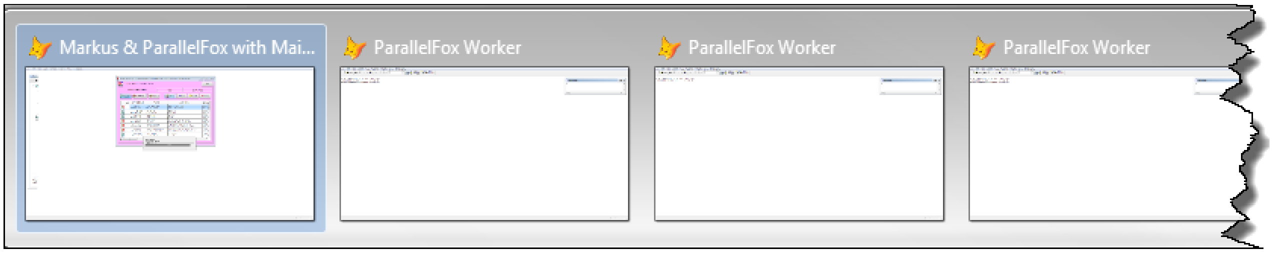
	SG	DBF Unique Id	Name	Description	Remar
✗		_2QB0S1S4E	ATTACHMENT	Email Attachments	mem
✓		_2QD1B5U9C	CALENDAR	Calendar	mem
✓		_2OP181QPV	COUNTRY	Country	mem
✗		_2OU1BZ4BA	CURDBF	Smart Grid Get File Cursor	mem
✗		_2PZ1DKF2A	CURFLDFIELDS	Table/Fields Combobox Cursor	mem
✗		_2O80RAPB4	CURUSERDEFINED	User Defined Combobox Cursor	mem
✓		_2OP17J4QW	CUSTOMER	Customer	mem

Processing
(Re)Build/(Re)Index...

93%

Figure 11: Smart Grid's Data Dictionary form during a process of rebuilding tables in Parallel.

Also, please see figure 12 for a snap shot of the main processing window and 3 of the 8 worker windows in debug mode during the rebuild process.



Conclusion of this process: Once I understood what I needed to do to get ParallelFox working, it was pretty easy to incorporate. However, rebuilding and reindexing tables wasn't any faster than what was done before in a synchronous mode. It could be because of the disk I/O. It's something to consider with other processes that require lots of reading and writing to disk. I was a little disappointed but I did learn something valuable: Not all processes will see benefit from ParallelFox. The only way to know for sure is to try.

Deployment

- Must include ParallelFox.vcx in your project.
- Your worker code can be compiled into your main app/exe. But it doesn't have to be. You can choose to create a separate app/exe for your worker code but you must remember to include ParallelFox.vcx in that project as well.
- ParallelFox.exe should be installed and registered. This is an EXE, so RegSvr32 does not work with it. If your install program doesn't handle it, register by executing ParallelFox.exe /regserver. **HIGHLY IMPORTANT:** Registering ParallelFox requires elevation on Windows Vista/7. Without elevation, ParallelFox.exe /regserver will fail, but it will not notify the user of the failure! Your installation package will likely have already requested elevation, so this shouldn't be a problem. If you will be using FoxPro code to register ParallelFox, see INSTALL.PRG for an example of how to handle this.

Brain Storming

This part is up to you and me on an individual basis. The person that knows your applications the best is ... You. Some possible processes to consider:

- Report Processing
- Month-End Processing
- Processing data to Excel / Word
- Data Conversions
- Rebuilding or Reindexing Tables … or maybe not. I didn't notice benefit from ParallelFox with this in my demonstration code.
- Executive Information
- Forecasting Processes
- ? ? ? ? .... This part is up to you

So, with that said, do your homework. Check out the videos on VFPx and the help that comes with ParallelFox. There are:

- 4 Videos
- 21 Program examples
- Extensive, well-written help

Final Thoughts

There is so much more to ParallelFox than what is talked about here in these few pages. Joel has included critical programming where workers might need to create a table and the other workers must wait. He has OnError handling from global to methods as well. We didn't cover transactional processing or terminal server considerations. We never looked at Joel's source code. But you can. It's all up on VFPx.

Every one of us has a time consuming process in code that would fit nicely with ParallelFox. Joel has wrapped ParallelFox up nicely and put a pretty bow on it just waiting for you to open the lib and see what's inside. I am hoping that by going through this small exercise you come to appreciate Visual FoxPro at a whole new level… paralleled. Tap into the awesome power of the available computer's CPU speed by plugging in ParallelFox. See your application processing soar bringing added value to your customers with a minimal amount of effort thanks to Joel and ParallelFox.

Joel has done a tremendous amount work and has taken care to do it well. The videos, extensive help, code examples and ease of use, all speak of Joel's passion for ParallelFox and the FoxPro community. But, it could be better with your help by:

- Getting involved and
- Provide feedback via VFPx Issue Tracker and Discussion areas

Thank you Joel Leach! Without you, we wouldn't be able to even consider the possibilities of parallel processing.

References:

***VFPx - ParallelFox** at*
http://vfpx.codeplex.com/wikipage?title=ParallelFox&referringTitle=Home

***Training videos by Joel Leach on VFPx.** Joel does an excellent job setting your mind thinking about Parallel processing using ParallelFox. His goals for ParallelFox are:*

- Do things the "Fox" way
- Feels like an extension of the Fox language
- Supports full functionality of VFP and it "just works"
- Very important is that it works well with existing applications
- Evolve to meet the needs of the community

1. *"Introduction to Parallel Fox"* at: http://www.mbs-intl.com/vfpx/parallelfox/ParallelFox_Intro/ParallelFox_Intro.html
2. *"Running Code in Parallel"* at: http://www.mbs-intl.com/vfpx/parallelfox/ParallelFox_Code/ParallelFox_Code.html

3. *Worker Events - Part 1* at: http://www.mbs-intl.com/vfpx/parallelfox/ParallelFox_Events/ParallelFox_Events.html
4. *Working Events – Part 2* at: http://www.mbs-intl.com/vfpx/parallelfox/ParallelFox_Events_2/ParallelFox_Events_2.html
   -

Examples that come with ParallelFox:

Sequence Shown	Program Name	What Video
1	Steps_Before.prg	Running Code in Parallel
2	Steps_After.prg	Running Code in Parallel
3	Call_Before.prg	Running Code in Parallel
4	Call_After.prg	Running Code in Parallel
5	CallMethod_Before.prg	Running Code in Parallel
6	CallMethod_After.prg	Running Code in Parallel
7	DoCmdScript.Before.prg	Running Code in Parallel
8	DoCmdScript_After.prg	Running Code in Parallel
1	Complete_Before.prg	Worker Events: Part 1
2	Complete_After.prg	Worker Events: Part 1
3	ReturnData_Before.prg	Worker Events: Part 1
4	ReturnData_After.prg	Worker Events: Part 1
5	ReturnCursor_Before.prg	Worker Events: Part 1
6	ReturnCursor_After.prg	Worker Events: Part 1
1	ProgressForm.prg	Worker Events: Part 2
2	ProgressAlert.prg	Worker Events: Part 2
3	ProgressEvent.prg	Worker Events: Part 2
4	Custom_CallBack.prg	Worker Events: Part 2

5	LongEvent.prg	Worker Events: Part 2
6	AutoYield.prg	Worker Events: Part 2
7	LocalVars.prg	Worker Events: Part 2

My Computer that I use to demonstrate ParallelFox:

- Toshiba Qosmio X500-0895S
- Intel® Core™ i7CUP Q740 @ 1.73 GHz
- Quad Core Process
- Installed memory – 6GB
- 64-bit Operating System
- Windows 7 Professional