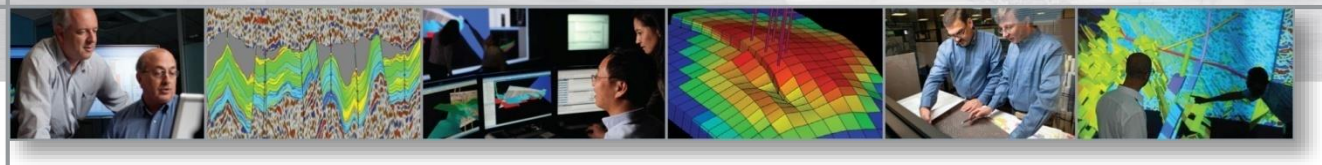
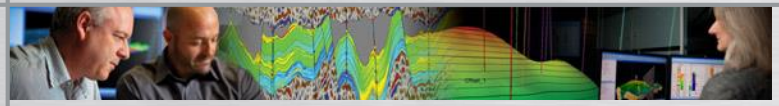
Parallelizing Ocean plug-in computations using the Background Worker + PFX pattern

Dmitriy Repin
Program Architect, Schlumberger PTS

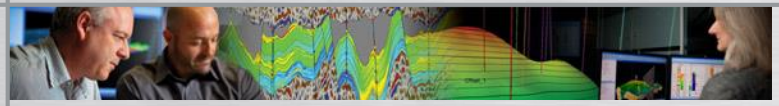
Ocean Development Framework User Meeting
Houston, October 24, 2014





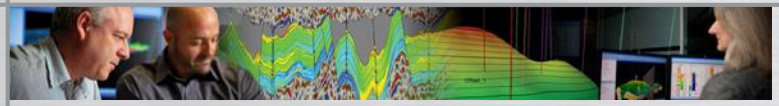
User requirements for a Petrel plug-in

- Deliver the result faster
  - Do the work in parallel, I've got the cores!
- Stay responsive to user interactions
  - Do not freeze the Petrel UI
  - Allow me to do something else (e.g., view the data) while the computation proceeds
- Report your progress
- Provide a way to cancel the computation, if needed



Petrel ecosystem constraints

- Windows Forms / WPF thread affinity
  - Controls can only be used on the main application thread.
  - Must use ISynchronizeInvoke.Invoke() to access from other thread
- Most of the Ocean API (with small exceptions, e.g., Seismic and StructuralFramework) does not support asynchronous access
  - Ocean domain objects must be accessed on the main application thread
  - Petrel does not use locking while accessing objects through UI
  - No application-wide deadlock prevention strategy



Given the limitations, what is possible?

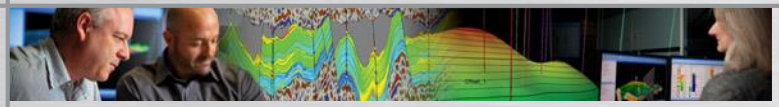
- ***Live dangerously:*** Perform parallel computations in the background without locking the data
  - *Be ready to crush - user can delete/change inputs at any moment*

- ***Freeze conservatively:*** Spawn threads / tasks from the main thread and block it until the computation is complete.
  - *Good for a short-running operations*
  - *Beware of Application.DoEvents() use for progress reporting!*

- ***Use optimistic concurrency:***

  1) Copy the data to a local buffer
  2) Perform parallel computations in the background using the buffer
  3) Copy the result back, if possible

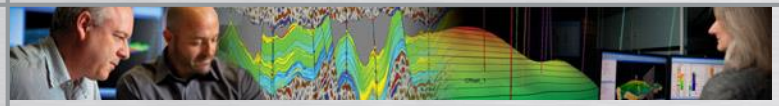
  - *Good for long-running operations when the computation is more expensive than the data/result copy*



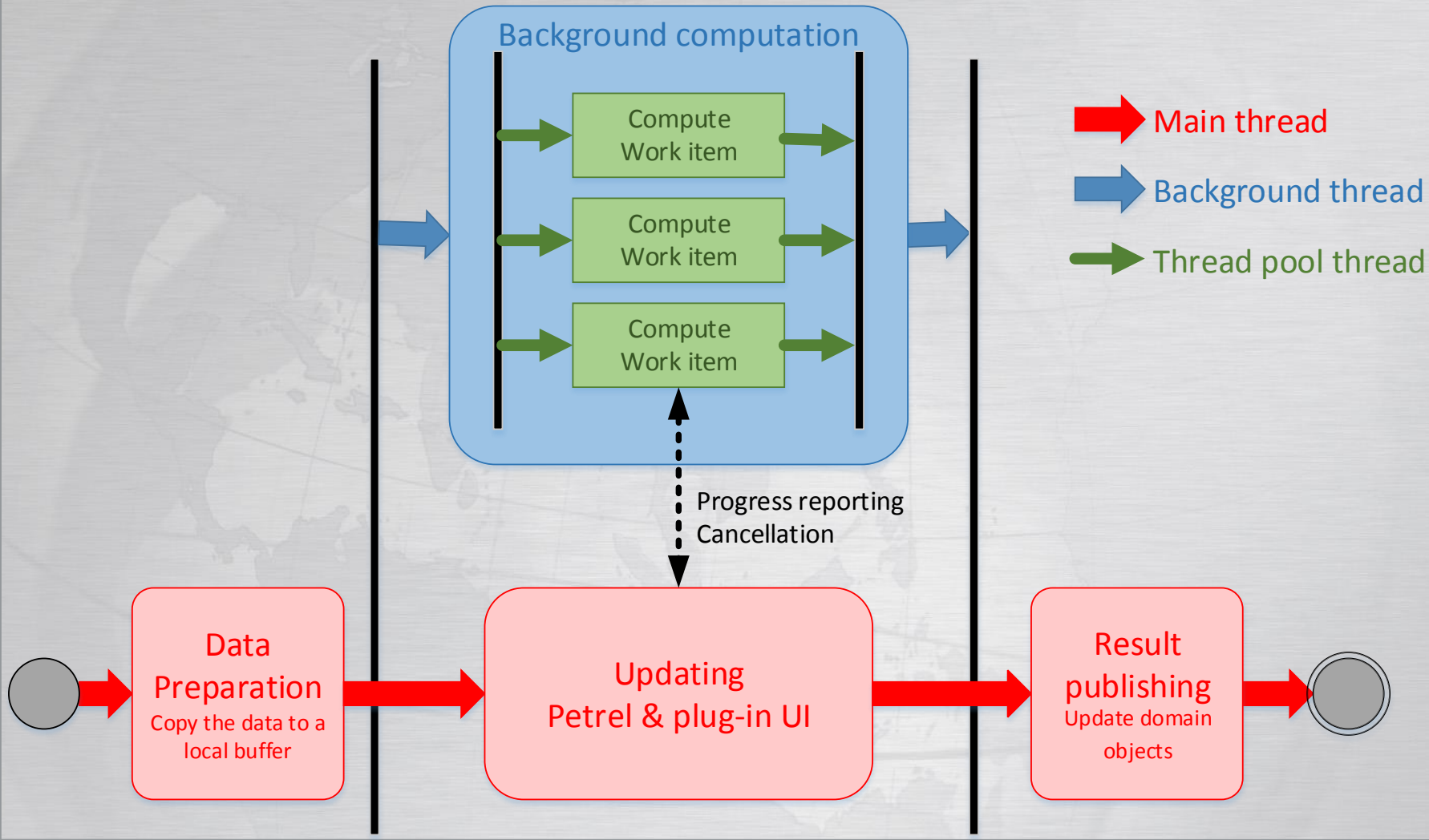
Our original implementation (before Petrel 2014)

BackgroundWorker + Parallel.For = <u>Usability</u>

- Petrel is alive even with near 100% CPU load
- Provide staged progress reporting
  - Step <N> of <L>: <M>% competed
- Provide cooperative cancellation
  - It takes some time to stop computation ☹



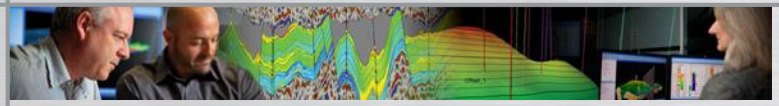
Optimistic concurrency – activity diagram



EAP and BackgroundWorker

- **E**vent-based **A**synchronous **P**attern from Microsoft
  - Perform an operation asynchronously
  - Receive a notification when the operation completes
  - Communicate with the operation using events/delegates
  - Support for cooperative cancellation and progress reporting
- BackgroundWorker

	RunWorkerAsync()	Starts execution of a background operation.
	DoWork	Occurs when RunWorkerAsync is called, will perform the actual computation
	RunWorkerCompleted	Occurs when the background operation has completed or has been canceled
	CancelAsync()	Requests cancellation of a pending background operation
	CancellationPending	Check this value in the DoWork handler. If TRUE, stop the computation
	ReportProgress(Int32)	When called from the DoWork handler, it will rise ProgressChanged event on the main thread
	ProgressChanged	Use it to update the UI



Simplified example of BackgroundWorker

```
backgroundWorker.RunWorkerAsync();
backgroundWorker.CancelAsync();
```

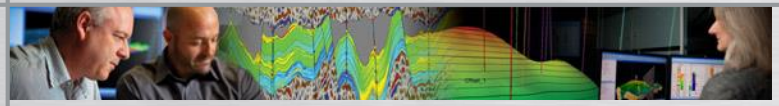
```
void backgroundWorker_DoWork(object sender, DoWorkEventArgs e) {
  BackgroundWorker worker = sender as BackgroundWorker;

  for (int i = 1; i < WorkItemCount; i++) {
    if (worker.CancellationPending == true) {
      e.Cancel = true; break;
    }
    SmoothInline(data, i);
    worker.ReportProgress((int)(100.0f*i/WorkItemCount);
  }
}
```

TODO: Parallelize this

```
void backgroundWorker_ProgressChanged(object sender, ProgressChangedEventArgs e) {
   resultLabel.Text = (e.ProgressPercentage.ToString() + "%");
}
```

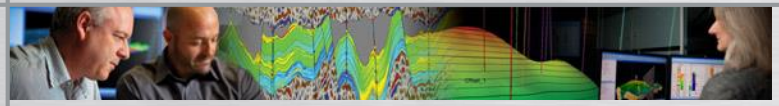
```
void backgroundWorker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
  if      (e.Cancelled == true){ resultLabel.Text = "Canceled!"; }
  else if (e.Error     != null){ resultLabel.Text = "Error: " + e.Error.Message; }
  else                         { resultLabel.Text = "Done!“ + e.Result.ToString(); }
}
```



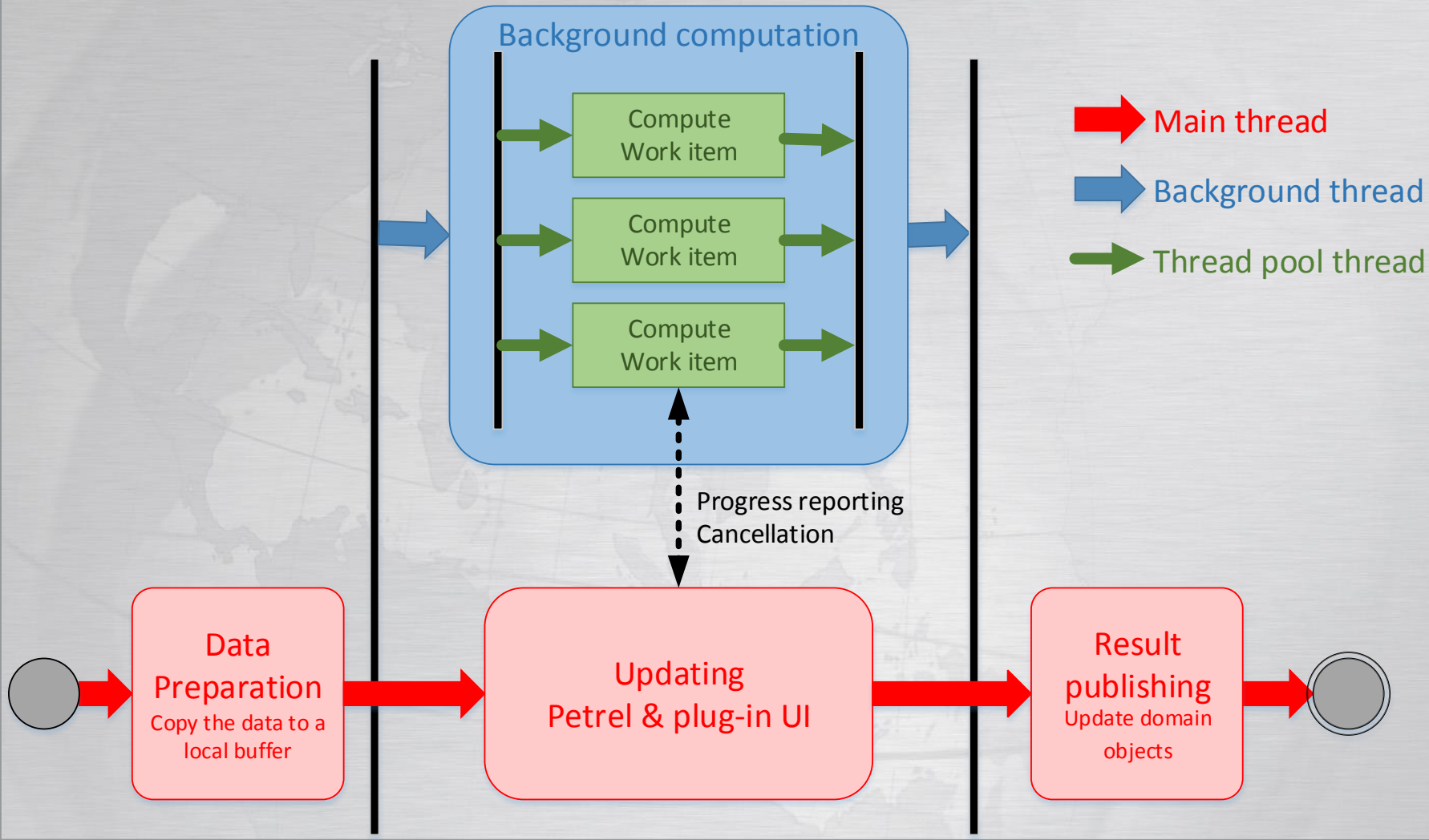
System.Threading.Tasks.**Parallel.For**

- Parallel Extensions for the .NET Framework (PFX) include
  - Parallel LINQ or PLINQ
  - The System.Threading.Tasks.**Parallel** class
  - The System.Threading.Tasks.**Task** parallelism constructs
  - The concurrent collections
- **Parallel** class supports basic data parallel computations:
  - Operation is performed concurrently on elements in a array
  - Scheduled on the thread pool and managed by the Task Scheduler
  - Blocks until all work is completed
  - After an exception, workers are stopped and AggregateException is thrown

```
System.Threading.Tasks.Parallel.For(0, WorkItemCount,
 (itemIndex) => {SmoothInline(data, itemIndex); }
);
```

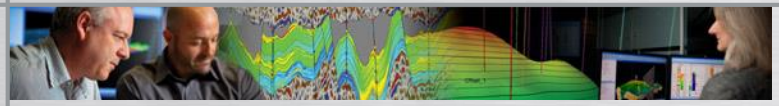


Optimistic concurrency – activity diagram



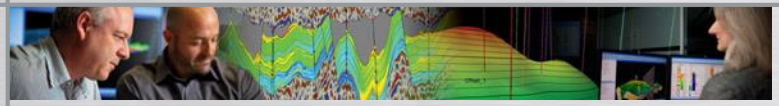
Putting it all together

- Data preparation **(main thread)**
  - Copy the Ocean domain object data to a local computation parameter buffer
  - Initialize a background worker. Run it asynchronously with the buffer as an argument
- Keep updating Petrel / plug-in UI **(main thread)**
  - Listening to ProgressChanged event and when it is received, update the progress bar
  - Call CancelAsync() if user wants to stop the computation
- DoWork event handler **(background thread)**
  - Divide the computation work into an number of small work items and schedule them for execution using Parallel.For()
  - Block the DoWork handler until all items are processed
  - Upon the DoWork completion, the RunWorkerCompleted event will be raised
- Compute work item / Parallel.For body (**thread pool thread**)
  - Check CancellationPending and quit if cancellation was requested
  - Perform a part of the computation and report the progress
- RunWorkerCompleted event handler (**main thread**)
  - Check the RunWorkerCompletedEventArgs
  - If no errors or cancellation occurred, update the domain object using the computation result



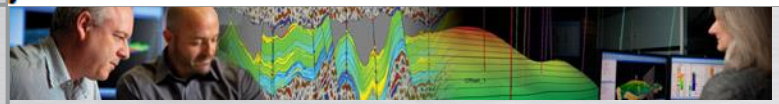
Petrel 2014 implementation using Async/Await

- Visual Studio 2012 (.NET 4.5) introduced a simplified approach to asynchronous programming – **Async / Await** pattern
  - The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code.
  - As a result, you get all the advantages of asynchronous programming with a fraction of the effort.



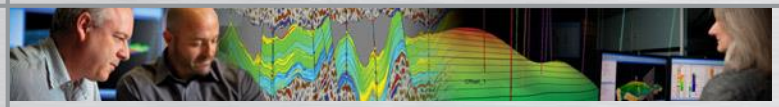
Petrel 2014 implementation using Async/Await

```
public async void Compute(Func<TData, IProgressReporter, TResult> compute, Func<TData> prepare,
   Action<TData, TResult> publish, Action<TData> onCancel, Action<TData, Exception> onException)
{
   this.IsExectuting = true;
   TData data = default(TData);
   try {
      this.reporter = CoreSystem.GetService<IProgressService>().Create(this.Name, cancellationToken);
      data = prepare();
      TResult result = await Task<TResult>.Run(() => compute(data, reporter));
      publish(data, result);
   }
   catch (OperationCanceledException) { onCancel(data); }
   catch (Exception ex) { onException(data, ex); }
   finally {
      this.reporter.Dispose(); this.reporter = null;
      this.IsExectuting = false;
   }
}
```



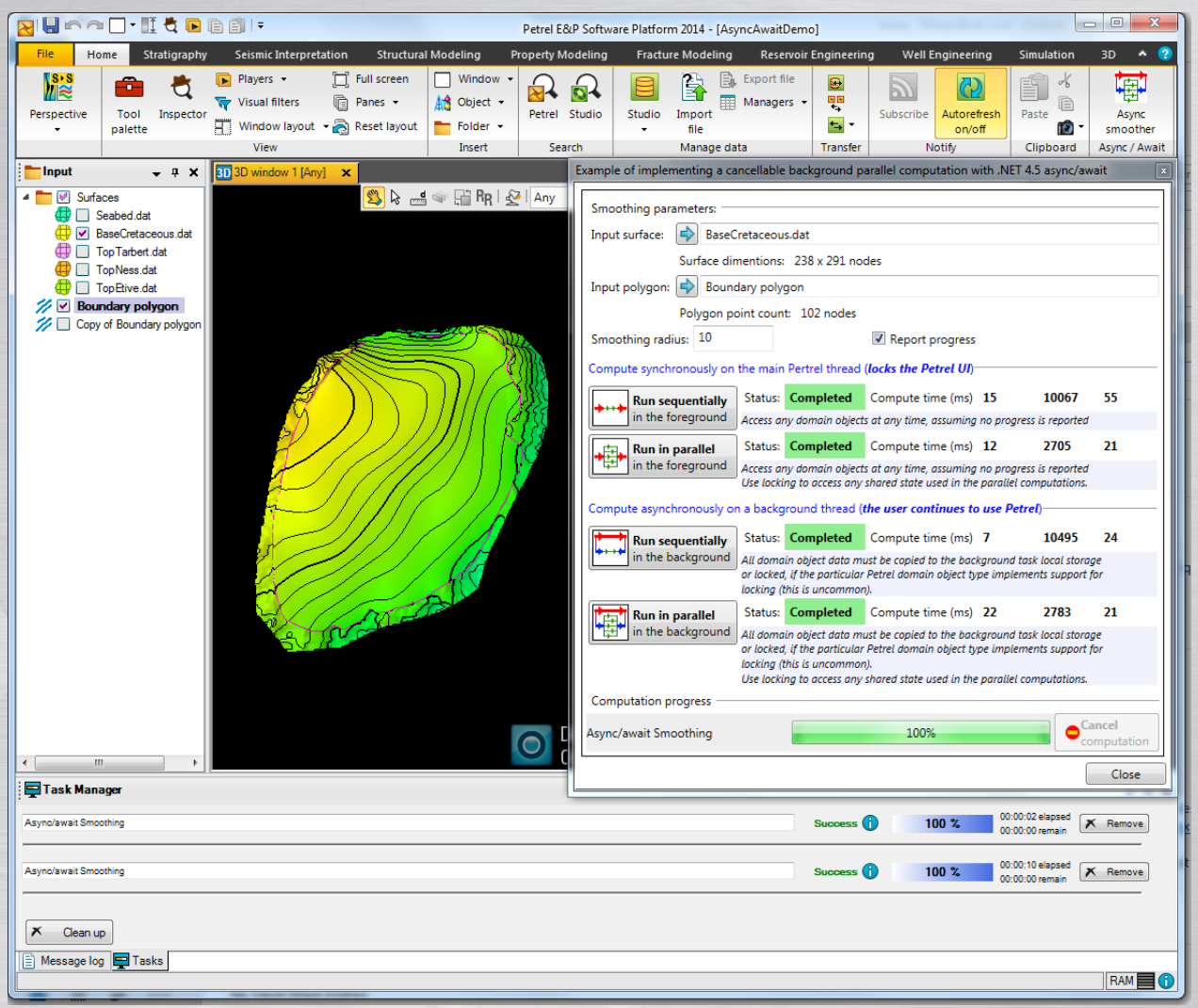
Petrel 2014 implementation using Async/Await

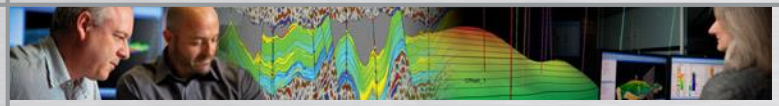
```
Result Compute(Data data, IProgressReporter reporter)
{
    reporter.ResetProgressCount(data.WorkItemCount);
    List<RegularHeightFieldSample> output = new List<RegularHeightFieldSample>();
    Parallel.For(0, data.WorkItemCount, new ParallelOptions(), (int indexI) =>
        {
            if (reporter != null) reporter.ThrowIfCancellationRequested();
            IEnumerable<RegularHeightFieldSample> partialResult = SmoothInline(data, indexI);
            lock (output) { output.AddRange(partialResult); }
            reporter.ReportProgressIncrement();
        }
    );
    return new Result(output);
}
```



Petrel 2014 implementation using Async/Await

Demo



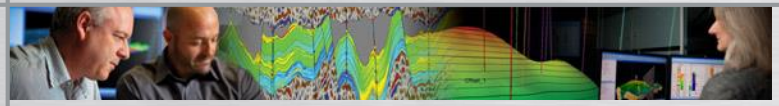


PetroTechnical Services
Global Expertise

Schlumberger

Outstanding issues:

- Major Ocean/Petrel multithreading issues
  - Ocean/Petrel does not use locking while accessing objects and UI
  - No an application-wide deadlock prevention strategy
  - Minor issues
    - To accomodate .NET4.0 Slb.Ocean.Petrel.IProgress needs to support System.Threading.CancellationToken
    - SetProgressText(String) does not work for NewAsyncProgress



References

(1) Pro Asynchronous Programming with .NET, *Richard Blewett, Andrew Clymer*
http://www.amazon.com/dp/B00DREFXNA/

(2) Parallel Processing and Concurrency in the .NET Framework
http://msdn.microsoft.com/en-us/library/hh156548.aspx

(3) Asynchronous Programming with Async and Await
http://msdn.microsoft.com/en-us/library/vstudio/hh191443.aspx

(4) Async in C# 5.0, *Alex Davies*
http://www.amazon.com/dp/1449337163/

(5) Threading in C#, *Joseph Albahari*
http://www.albahari.com/threading/