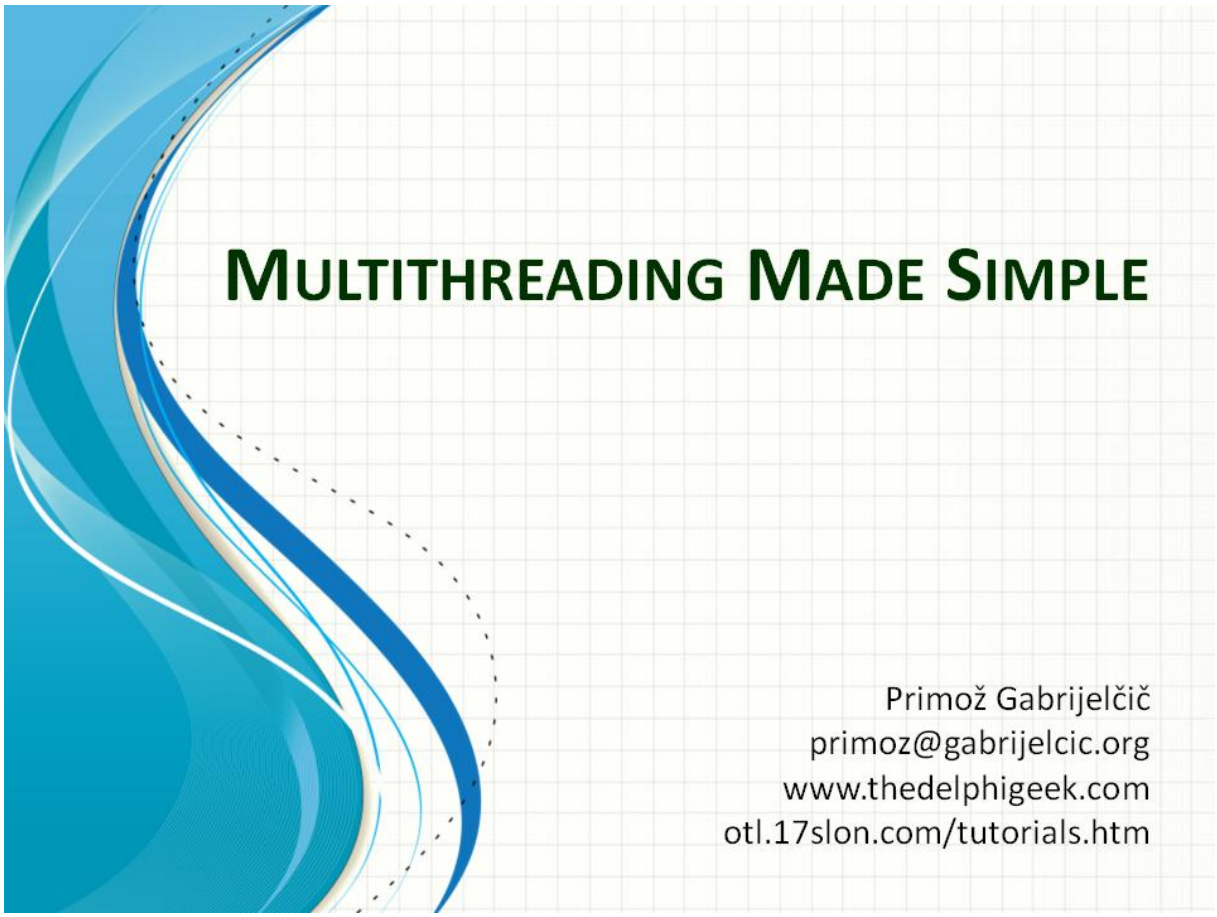
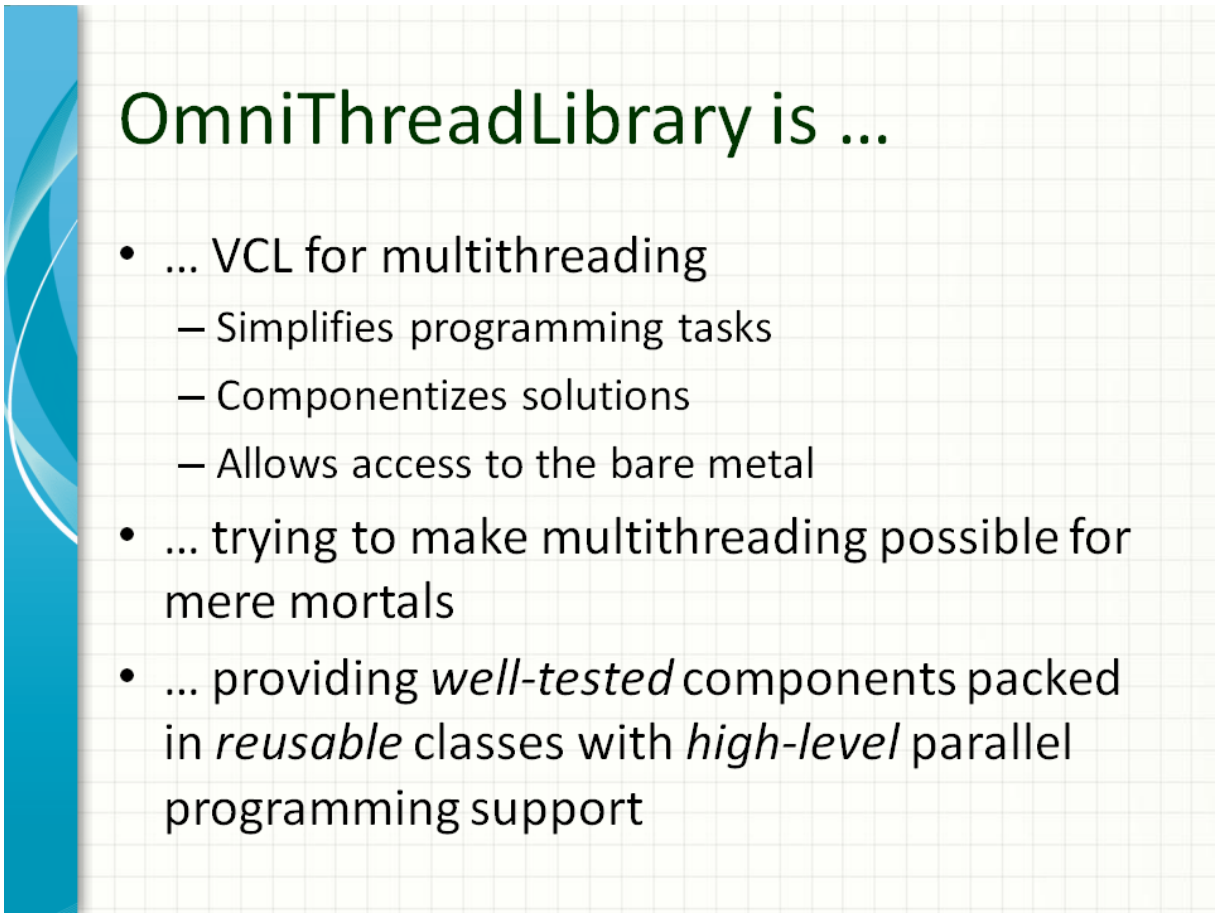
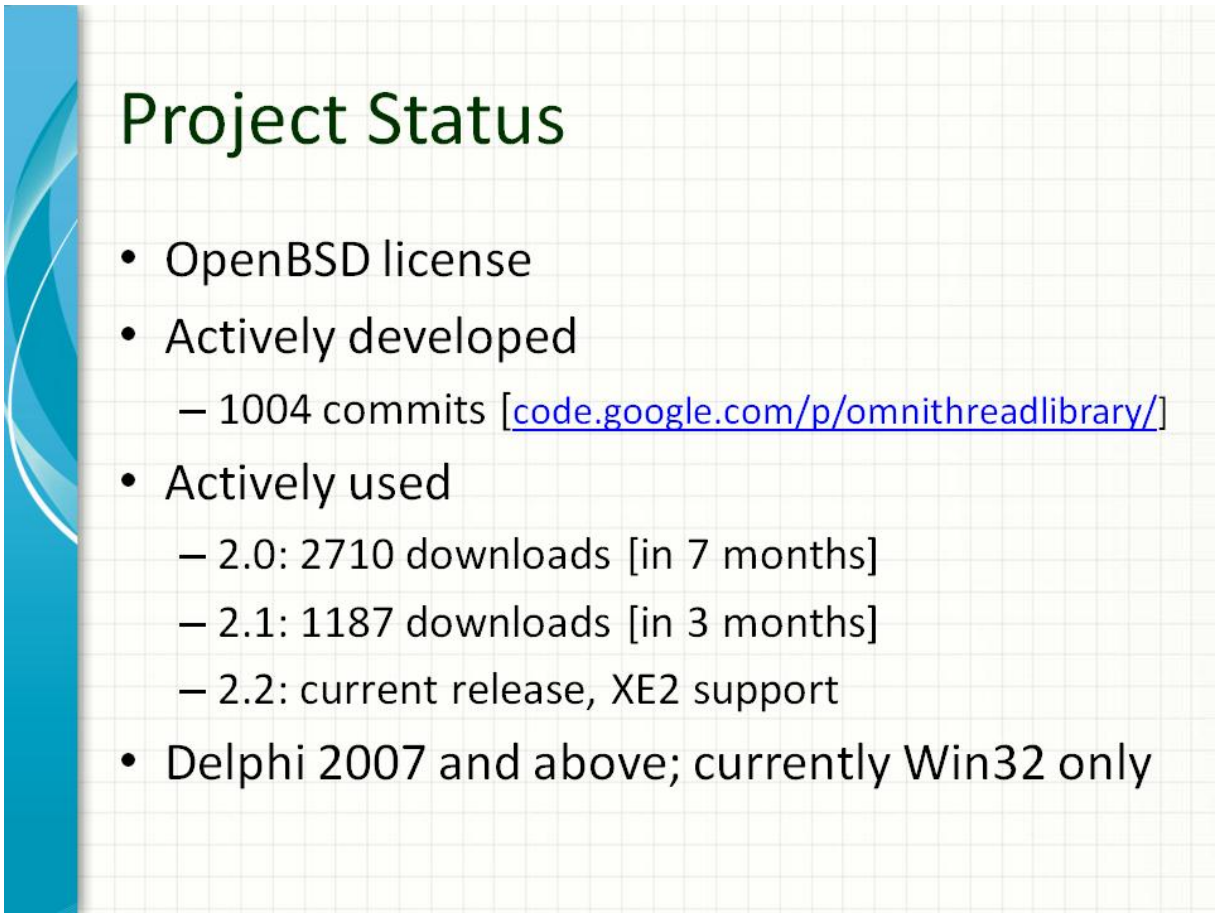


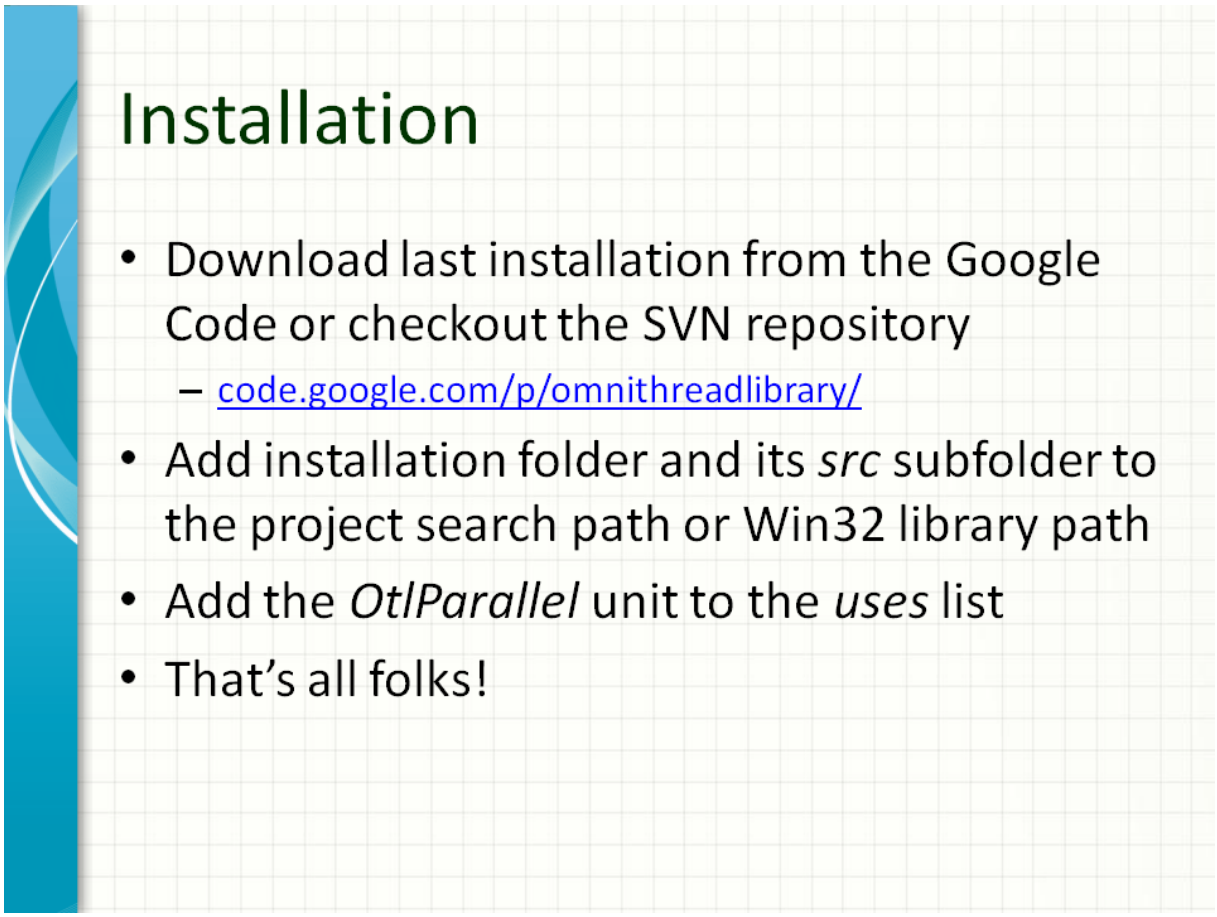
Hello, I'm Primož Gabrijelčič and today I'll talk about multithreading – and not just any multithreading but a multithreading in its simplest form – at least if we limit ourselves to the Delphi language. I'll be presenting the OmniThreadLibrary project, an open source library that tries to bring back fun to the multithreaded programming! Due to a very limited time, I'll be focusing on the high-level constructs only (I will only mention low-level primitives in passing) and even there I'll be only able to cover the basics. You are therefore invited to read more about the OmniThreadLibrary on my blog (www.thedelphigeek.com). Another good place to start is the OmniThreadLibrary home page, otl.17slon.com, and especially the *tutorials* page (link on your screen) which links to important blog articles. I will also assume that you know a little about multithreading programming and troubles associated with that (such as data sharing and synchronization problems).



I'll start with a few words about the OmniThreadLibrary. It was designed to become a "VCL for multithreading" - a library that will make typical multithreading tasks really simple but still allow you to dig deeper and mess with the multithreading code at the Win32 API level (and soon also on the Win64 level). Initially the focus was on well-tested low-level components that made multithreaded programming much simpler as with the TThread (although I must say that there's nothing wrong with the TThread – it is used in the OmniThreadLibrary internally to manage threads) and then (in release 2.0) the focus moved to high-level primitives (such as *parallel for*) which I'll be talking about today. If I had to point out one specific feature of the OmniThreadLibrary I'd mention that it is not focused on threads but on *tasks*. In other words, you tell the system **what** you want to run in a context of a different thread and not **how** to run it. And that makes all the difference.

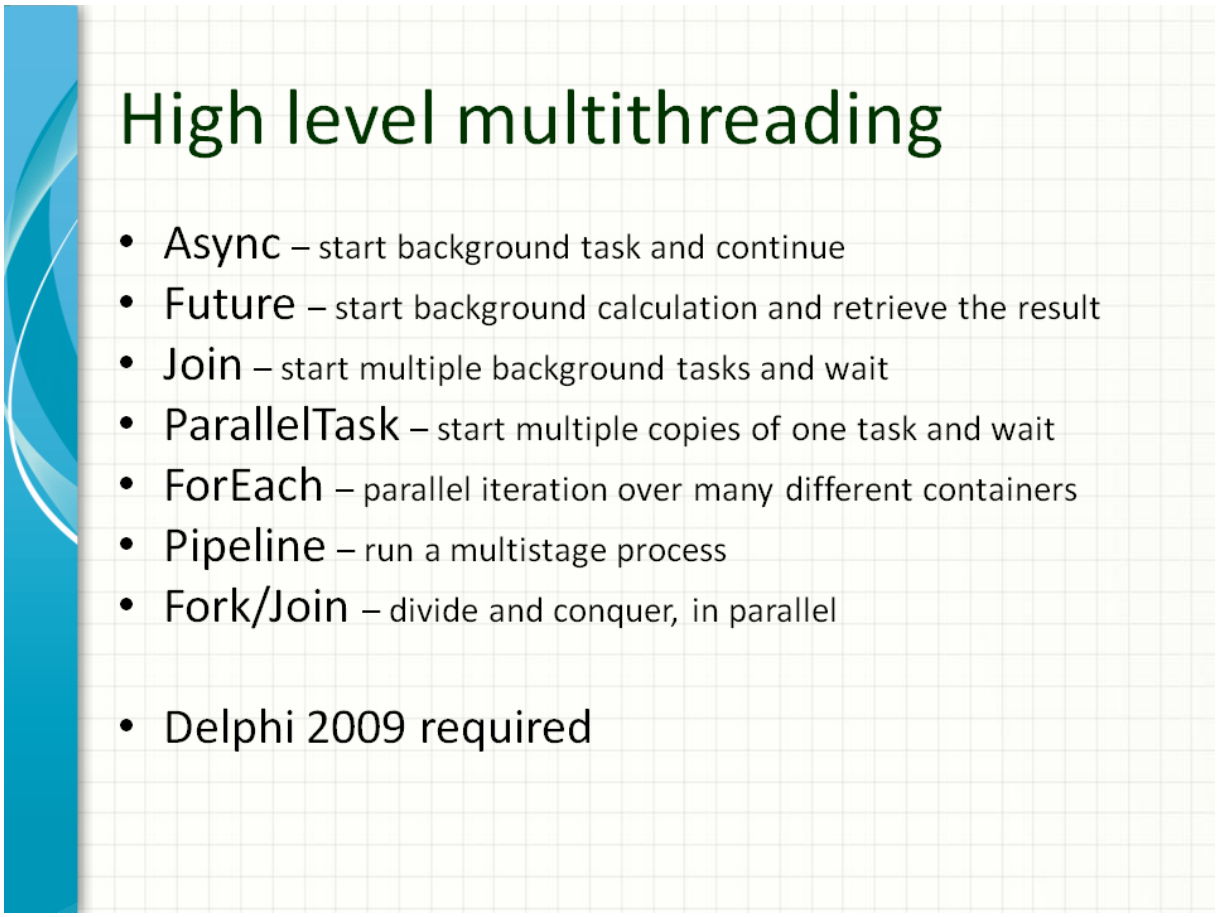


A few words about the project itself. It is released under the OpenBSD license, which is one of the most “forgiving” licenses and doesn’t affect your commercial applications in any way. Started in July 2008, it lives in the Google Code repository and is actively developed with 1004 commits and 9 releases. Current release 2.2 supports Delphi XE2, but only on the Windows 32-bit target. Support for Windows 64-bit mode is coming before the end of the year and then I’ll port the library – as much as possible – to the OS/X target. (At the moment I have no idea about what is possible to do on the iOS platform.)



Installing OmniThreadLibrary is very simple. Firstly, download the latest release from the Google Code or checkout the SVN repository. (Following the repository HEAD is typically fine – I try very much to not commit buggy code and all volatile development work is done in branches.) Secondly, unpack the release to some folder and add this folder and its *src* subfolder to the project search path or to the Win32 library path. Thirdly, add the *OtlParallel* unit to the *uses* list. You're ready to go!

OtlParallel contains all the high-level stuff discussed today. Sometimes you'll also have to *use* other OmniThreadLibrary units like OtlCommon or OtlTaskControl. I'll come back to them later in the presentation.



The topic of today's talk are high-level OmniThreadLibrary constructs. In order of appearance, they are:

1. Async. It allows you to start an independent background task (that is a piece of code running in a separate thread) and forget about it. Background task can communicate with the owner (typically the main thread) and owner can be notified when the background task completes execution.

2. Future. Similar to Async, a future is an independent background task with a twist – it returns the result of the execution back to the owner. As the Async it supports the communication and completion notification (actually all high-level primitives except the Fork/Join support those two functions). In addition to that it also handles exceptions in the background task and raises them in the owner.

3. Join. Allows you to start multiple background tasks and optionally wait for them to complete execution. Join also provides good exception handling, but it is not simple enough to be explained in a minute so please refer to my blog to learn more about it.

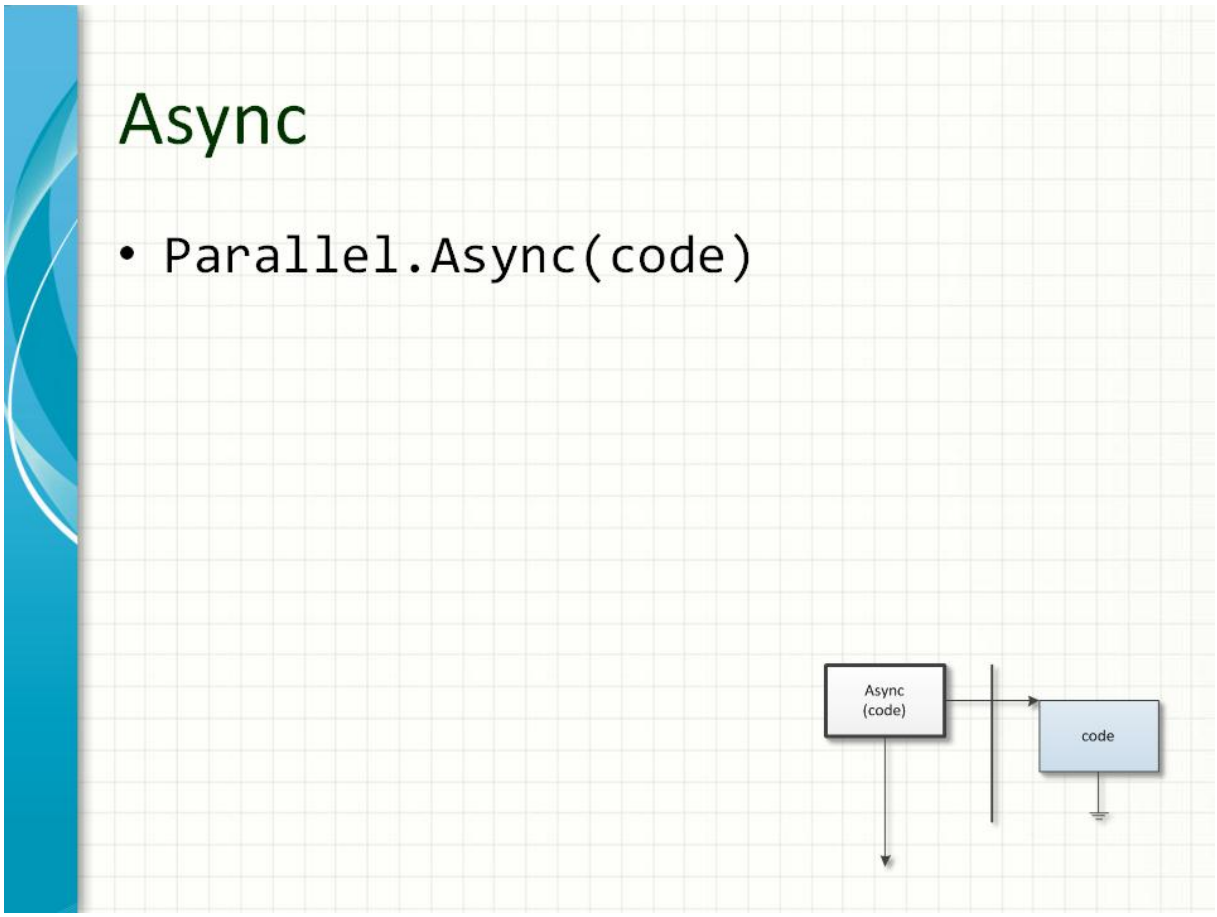
4. ParallelTask. A variant of the Join (it is actually implemented internally using the Join) starts multiple copies of your code in multiple threads and optionally waits for them to complete execution.

5. ForEach. A parallel variant of the *for* statement can iterate over integer ranges (just as the *for* statement does) and over various types of containers (similar to the *for in* statement). It offers many configuration options, only a few of which I'll mention in todays presentation.

6. Pipeline. Runs processes that can be described as a data flow between multiple stages. Exception handling is built-in but is again too complicated to be explained during this session so – again – please refer to my blog.

7. Fork/Join. A parallel variant of the "divide and conquer" programming approach allows you to parallelize tasks which can be expressed in subtasks. Think of recursion and QuickSort and you'll get the right idea.

All those primitives extensively use advanced Delphi features such as anonymous methods and are therefore supported only in Delphi 2009 and newer.



Let's start with the simplest construct, Async. To use it, call Parallel.Async (you'll notice that all high-level constructs start with the Parallel prefix) and pass it some code. This can be a parameterless procedure, method or anonymous method. [If you look at the code, you'll see that there are only two overloaded declarations of Async, both expecting an anonymous method as a parameter. Support for procedures and methods comes automatically courtesy of the Delphi compiler.]

http://www.thedelphigeek.com/2011/04/simple-background-tasks-with.html

http://www.thedelphigeek.com/2011/07/life-after-21-async-redux.html

The diagram in the bottom right corner of the screen explains the execution model. When you call Async, code is started in a new thread (indicated by the bold vertical line) and both main and background thread continue execution. At some time, background task completes execution and nothing special happens.

A note for advanced listeners – when I said "is started in a new thread", I lied a little. OK, I lied a lot. All high-level primitives manage a thread pool. Background thread is always taken from a thread pool and only if there is no thread waiting for the work, a new thread is created.

http://www.thedelphigeek.com/2011/09/life-after-21-parallel-data-production.html

Let's switch to the code now.

I have a simple demo application prepared which will help me with the presentation. As I have little time and lots to show, I've prepared all (well, almost all) code in advance.

To demonstrate the Async, I'm using it to fetch a page from the internet with a simple WinInet synchronous call. Let's first see the synchronous version:

```
procedure TfrmMultithreadingMadeSimple.btnSyncGETClick(Sender: TObject);
var
  page: string;
  time: int64;
begin
  time := DSiTimeGetTime64;
  HttpGet('17slon.com', 80, '/gp/biblio/articlesall.htm', page, '');
  lbLogAsync.Items.Add(Format('Sync GET: %d ms; page length = %d',
    [DSiElapsedTime64(time), Length(page)]));
end;
```

First I'm getting the current time (using the timeGetTime multimedia function which offers a millisecond accuracy), then I call a helper function HttpGet (included in the MMSHelpers unit) and at the end I show the total time used for the call and the length of the returned data (just to check that anything was returned at all).

Let's run the demo now.

As you can see, there's some action already going on. The progress bar at the bottom is updated four times a second from the timer. This will show us very clearly when the main thread is execution long operation (the progress bar will stop). If, for example, I now click on the "Sync GET" button to execute the code we were just examining …

… the code would stop for about two seconds while the web page is being retrieved.

You don't have to do much to convert this code into a background task.

```
procedure TfrmMultithreadingMadeSimple.btnAsyncGETClick(Sender: TObject);
var
  time: int64;
begin
  time := DSiTimeGetTime64;
  Parallel.Async(
    procedure
    var
      page: string;
    begin
      HttpGet('17slon.com', 80, '/gp/biblio/articlesall.htm', page, '');
    end);
  lbLogAsync.Items.Add(Format('Async: %d ms', [DSiElapsedTime64(time)]))
end;
```

The template code is still the same – store the time, run some code, display the time difference – but this time we are using Parallel.Async to start the operation in background. I've written a simple parameterless anonymous method that wraps the HttpGet call and – for now – ignores the returned page content.

If I now click on the “Async GET” button, I can see that Async call itself only needed 23 milliseconds. The actual HttpGet operation is executing in the background. As the result of HttpGet is (momentarily) thrown away, there’s only one way to prove my words – in the debugger.

I will put a breakpoint on the HttpGet call …

… and click the button again.

You can see in the Thread Status window that the code is really running in a background thread.

So how can we get the result back to the main thread? There are few different ways, one of which is the use of internal _Invoke_ mechanism, which works very similarly to the TThread‘s Queue method.

To use it, we’ll have to write a different anonymous method accepting the low-level _IOmniTask_ interface and add OtlTask to the _uses_ list.

```
procedure TfrmMultithreadingMadeSimple.btnAsyncGETResultClick(Sender: TObject);
var
  time: int64;
begin
  time := DSiTimeGetTime64;
  Parallel.Async(
    procedure (const task: IOmniTask)
    var
      page: string;
      time: int64;
    begin
      time := DSiTimeGetTime64;
      HttpGet('17slon.com', 80, '/gp/biblio/articlesall.htm', page, '');
      time := DSiElapsedTime64(time);
      task.Invoke(
        procedure
        begin
          lbLogAsync.Items.Add(Format('Async GET: %d ms; page length = %d',
            [time, Length(page)]))
        end);
    end);
  lbLogAsync.Items.Add(Format('Async: %d ms', [DSiElapsedTime64(time)]));
end;
```

This interface represents a single task and you can use it to communicate with the main thread or to invoke some code in the main thread, which is the approach I’m using here.

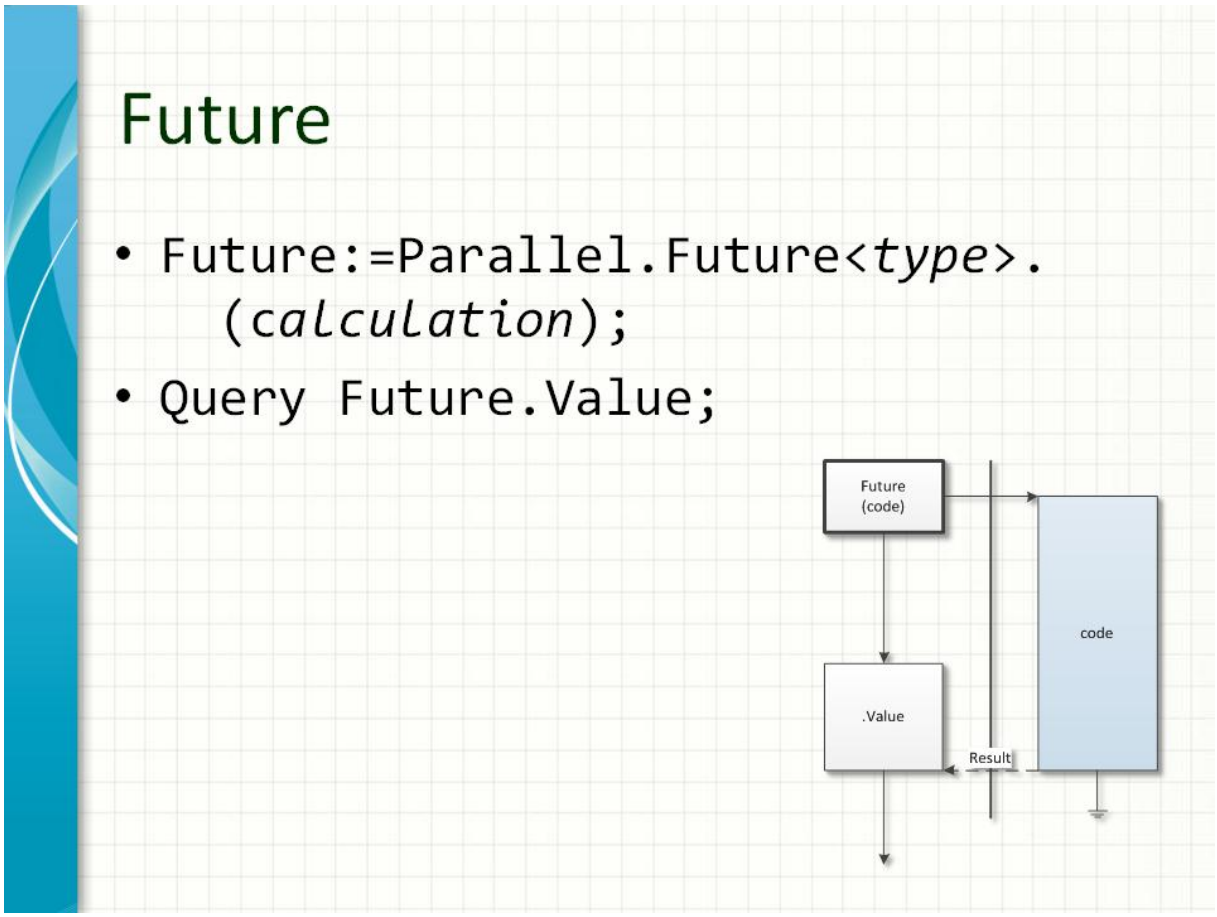
When HttpGet returns, the code will use task.Invoke to execute some code in the main thread and this code will update the user interface.

As you can see, we are now timing two operations – the Async call and the execution time of the background task.

Let’s see how it works …

Async only needed 2 milliseconds (because the thread executing the task was already ready and waiting for work in the thread pool) but the background task needed “normal” second and a half. As you can see yourself, the GUI was responsive all the time (the green bar is being constantly updated).

Enough of the Async, let’s continue with the Future.



A future is a background calculation that returns a result. To create the task, call Parallel.Future (providing the type returned from the calculation). To get the result of the calculation, call the .Value method on the interface returned from the Parallel.Future call.

http://www.thedelphigeek.com/2010/06/future-of-delphi.html

http://www.thedelphigeek.com/2011/07/life-after-21-exceptions-in.html

When you call the Parallel.Future, a background task is started immediately. The task will continue with it's (possibly long) execution and the main thread can do some other work. When you need the result of the background calculation, call the .Value method, which will return the result immediately if it is ready or which will wait on the background code to complete its work if necessary.

To demonstrate the use of the Future construct, I've rewritten the HttpGet demo.

```
procedure TfrmMultithreadingMadeSimple.btnFutureGETClick(Sender: TObject);
var
  getFuture: IOmniFuture<string>;
  page: string;
  time: int64;
begin
  time := DSiTimeGetTime64;
  getFuture := Parallel.Future<string>(FutureGet);
  lbLogFuture.Items.Add(Format('Future: %d ms', [DSiElapsedTime64(time)]));
  // do some other processing
  Sleep(1000);
  page := getFuture.Value;
  lbLogFuture.Items.Add(Format('Future GET: %d ms; page length = %d',
    [DSiElapsedTime64(time), Length(page)]));
end;
```

The demo calls Parallel.Future of string with the FutureGet parameter. Then it logs the elapsed time for the .Future call and does some other processing (simulated here by the Sleep function). At the end it retrieves the result of the FutureGet function.

FutureGet is just a simple function returning string – the contents of the retrieved page.

```
function TfrmMultithreadingMadeSimple.FutureGET: string;
begin
  HttpGet('17slon.com', 80, '/gp/biblio/articlesall.htm', Result, '');
end;
```

Let's see how it works in practice.

The .Future call needed only 17 ms but the total execution time is still about a second and a half. Main thread spent one second of that in the Sleep call and the rest in the .Value where it waited for the future to return a result.

If you don't want to block in the .Value call, you have three options. One is to occasionally call the IsDone function, another is to use TryValue instead of Value and the third one is to set up a termination handler.

```
procedure TfrmMultithreadingMadeSimple.btnFutureGETTerminateClick(Sender:
    TObject);
begin
  FGetFuture := Parallel.Future<string>(FutureGet,
    Parallel.TaskConfig.OnTerminated(FutureDone));
end;
```

By providing second parameter – a "task configuration" – to the Parallel.Future call we can set up a parameterless method/procedure/anonymous code that will be executed when the background calculation is completed.

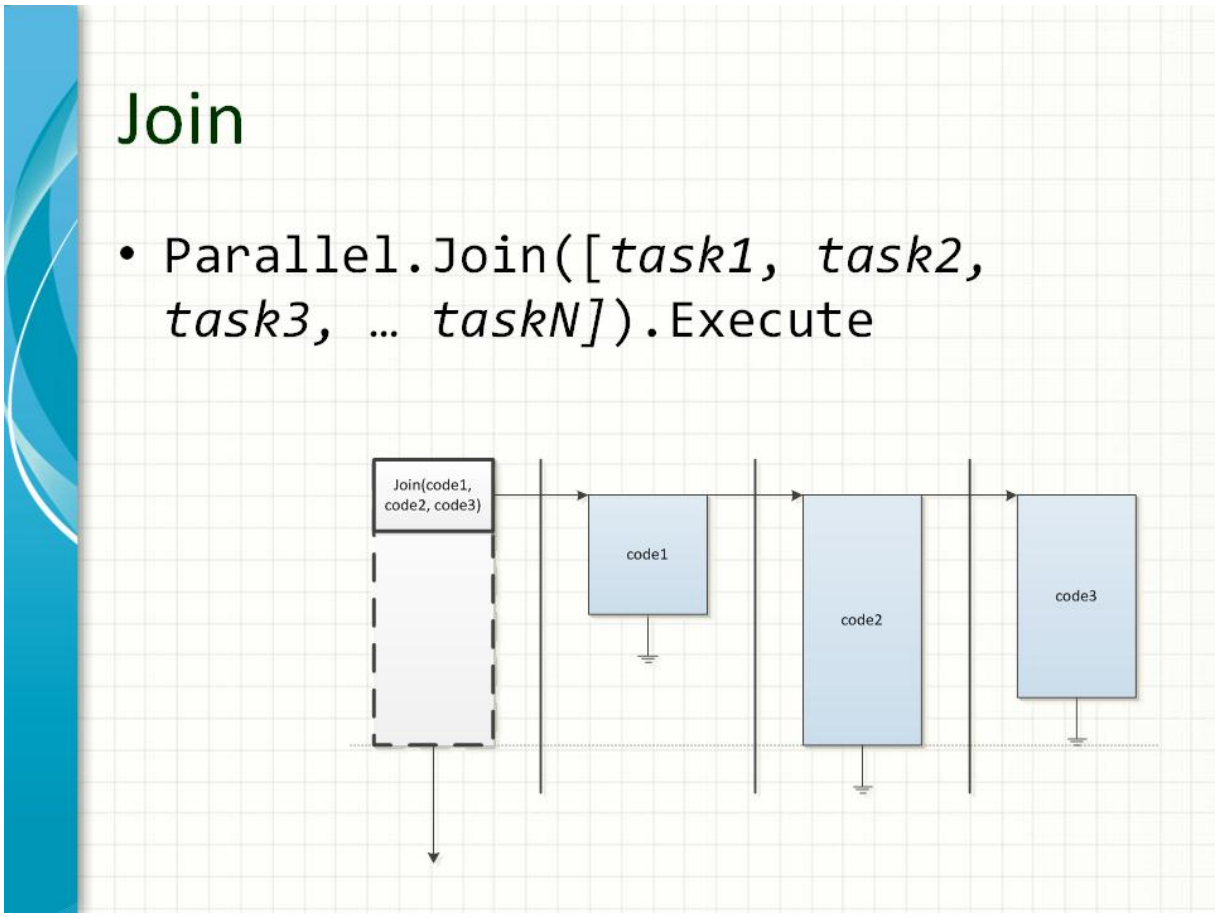
In this case, result of the Future call must be stored in a form field (or another global instance) so that it is not destroyed when the button handler exits.

```
procedure TfrmMultithreadingMadeSimple.FutureDone;
var
  page: string;
  time: int64;
begin
  time := DSiTimeGetTime64;
  page := FGetFuture.Value;
  FGetFuture := nil;
  lbLogFuture.Items.Add(Format('Future Done: %d ms; page length = %d',
    [DSiElapsedTime(time), Length(page)]));
end;
```

In the FutureDone we call .Value to retrieve the result and assign “nil” to the FGetFuture field to free the future interface.

In this case the GUI is not blocking and result is retrieved immediately. Of course, we have to wait a second and a half for FutureDone to be called at all.

Let’s continue with Join.



Join takes multiple code fragments (that is methods, procedures or anonymous methods) and executes each in its own thread. It can optionally wait on all tasks to complete execution or it can continue immediately.

http://www.thedelphigeek.com/2011/07/life-after-21-paralleljoins-new-clothes.html

A very simple demo uses Join to execute two methods that spend all their time sleeping – one for 2 and another for 3 seconds.

```
procedure TfrmMultithreadingMadeSimple.btnJoinClick(Sender: TObject);
var
  time: int64;
begin
  time := DSiTimeGetTime64;
  Parallel.Join(Delay(2000), Delay(3000)).Execute;
  lbLogJoin.Items.Add(Format('Join: %d ms', [DSiElapsedTime64(time)]));
end;
```

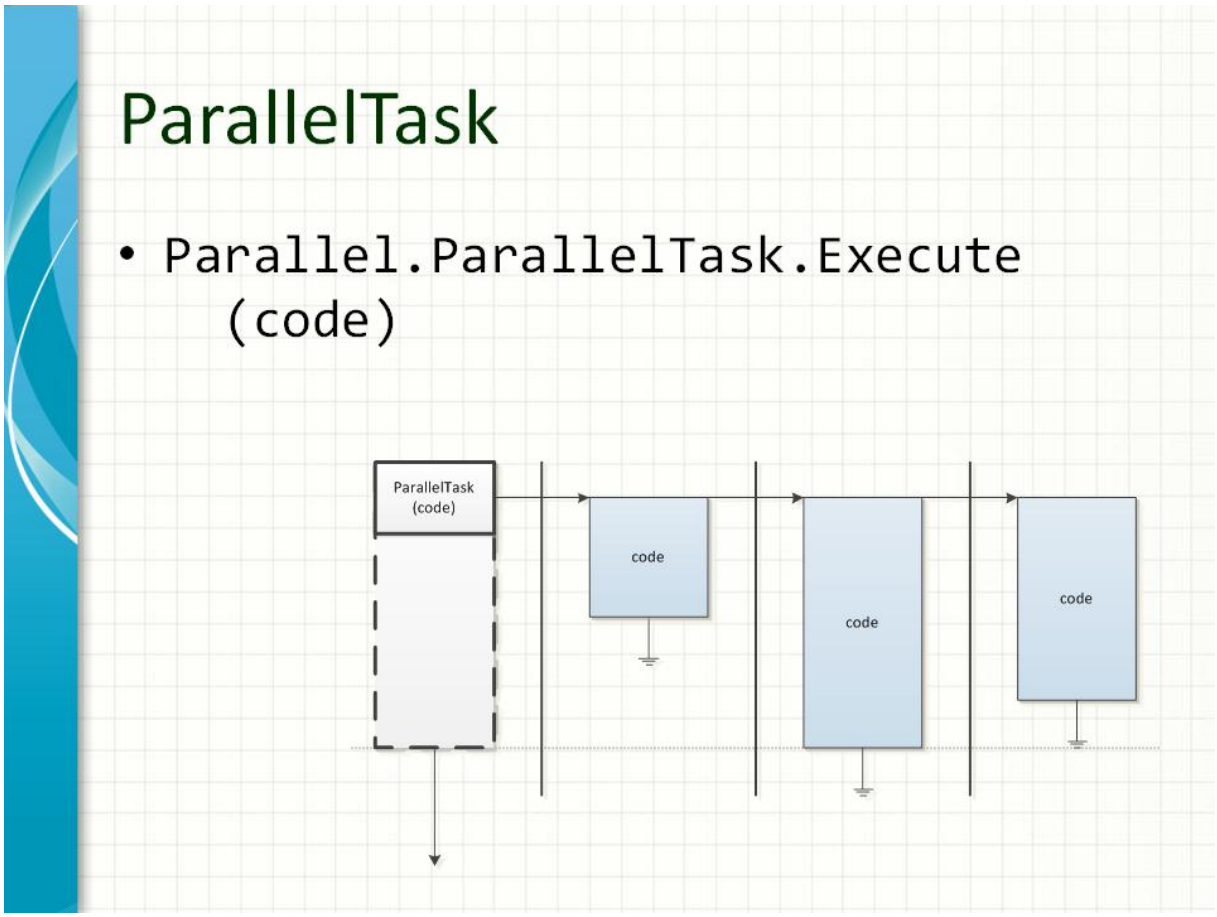
The total execution time is, of course, three seconds.

As we are not using the nonblocking version of Join, the GUI has stopped for three seconds too. I'll show you how to use a non-waiting version in a moment.

Just for fun, the Delay is implemented as a function that returns an anonymous method (which is then executed in the background task).

```
function TfrmMultithreadingMadeSimple.Delay(timeout_ms: integer): TProc;
begin
  Result :=
    procedure
    begin
      Sleep(timeout_ms);
    end;
end;
```

Enough of that, let's move to ParallelTask.



ParallelTask is a specialized version of Join (it is even implemented using Join) that executes same code on multiple cores. By default it will use all cores in the system but you can override this and run less or more copies.

Usage is simple – call Parallel.ParallelTask.Execute and provide a code. This will create multiple threads running the same code. By default, this call will wait on all of them to complete the execution but this behavior can again be overridden.

```
procedure TfrmMultithreadingMadeSimple.BurnCPU;
var
  a: real;
  i: integer;
begin
  a := 1;
  for i := 1 to 100000000 do
    a := Cos(a);
end;
```

To demo ParallelTask I've written a simple code that tries to use one CPU core in a tight loop. If we run it, we'll see that it manages to use about 15% of my system (my machine is running on two Xeon processors, each with two hyperthreaded cores giving the eight cores in total).

Parallel version of the code is very simple – it just calls ParallelTask.Execute and passes the BurnCPU code as a parameter.

```
procedure TfrmMultithreadingMadeSimple.btnParallelTaskBurnCPUClick(Sender:
    TObject);
var
  time: int64;
begin
  time := DSiTimeGetTime64;
  Parallel.ParallelTask.Execute(BurnCPU);
  lbLogParallelTask.Items.Add(Format('ParallelTask: %d ms', [DSiElapsedTime64(time)]))
end;
```

If we now run the parallel version, we'll see that the system is 100% busy but that the GUI is unresponsive. This is because ParallelTask waits for all tasks to terminate by default.

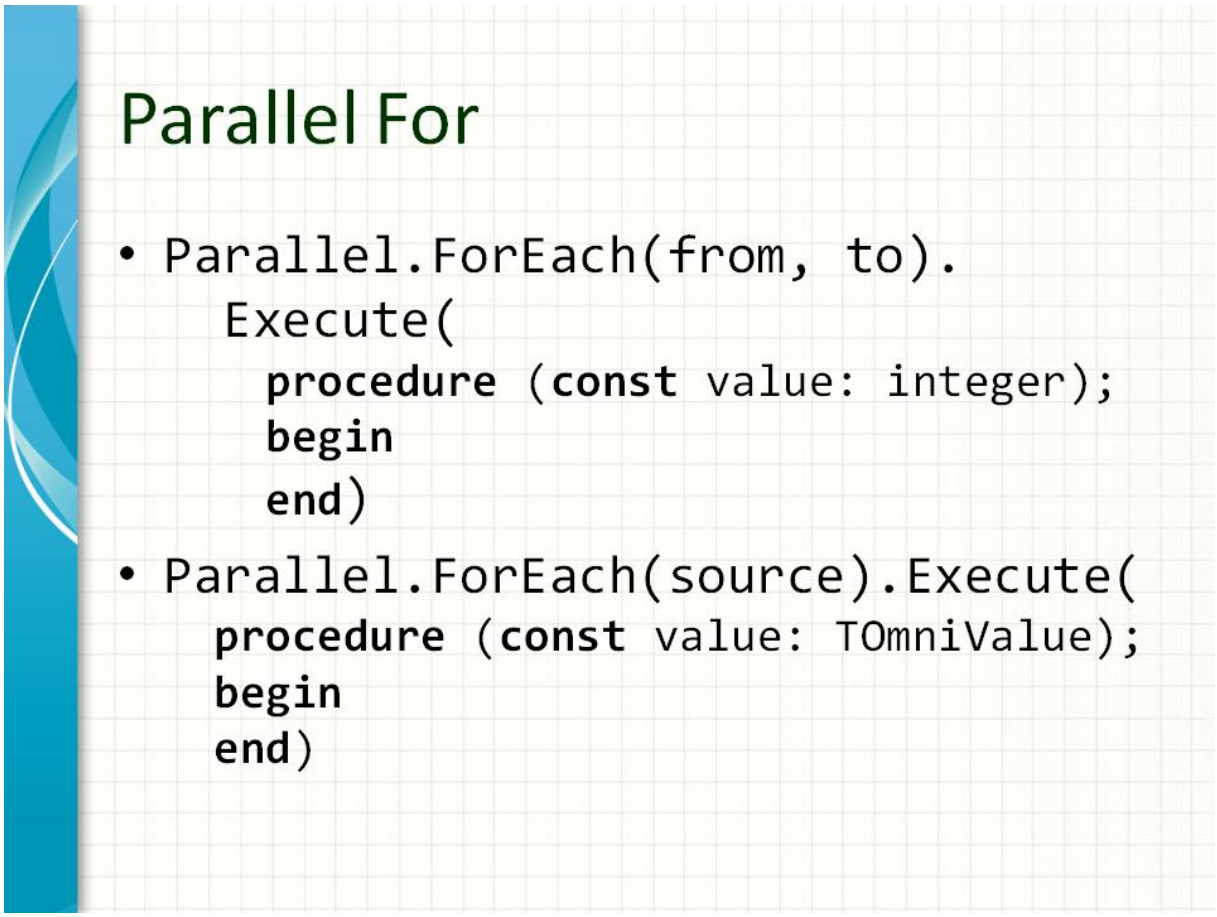
To change this, we have to make one tiny modification to the code – add .NoWait call.

Well, we have to make two changes. We also have to store interface returned from ParallelTask into global field so that it is not destroyed when the event handler exits.

```
procedure TfrmMultithreadingMadeSimple.btnParallelTaskBurnCPUClick(Sender:
    TObject);
var
  time: int64;
begin
  time := DSiTimeGetTime64;
  FBurnCPU := Parallel.ParallelTask.Execute(BurnCPU);
  lbLogParallelTask.Items.Add(Format('ParallelTask: %d ms', [DSiElapsedTime64(time)]))
end;
```

Now the system is busy but the GUI is still responsive. The system is not 100% busy because one core is not used for background execution. Because the .NoWait is used, the ParallelTask assumes that you want to do some work in the main thread and it leaves one core free.

Let's move on. Next on the list is parallel for – a very important but also quite complicated high-level construct.

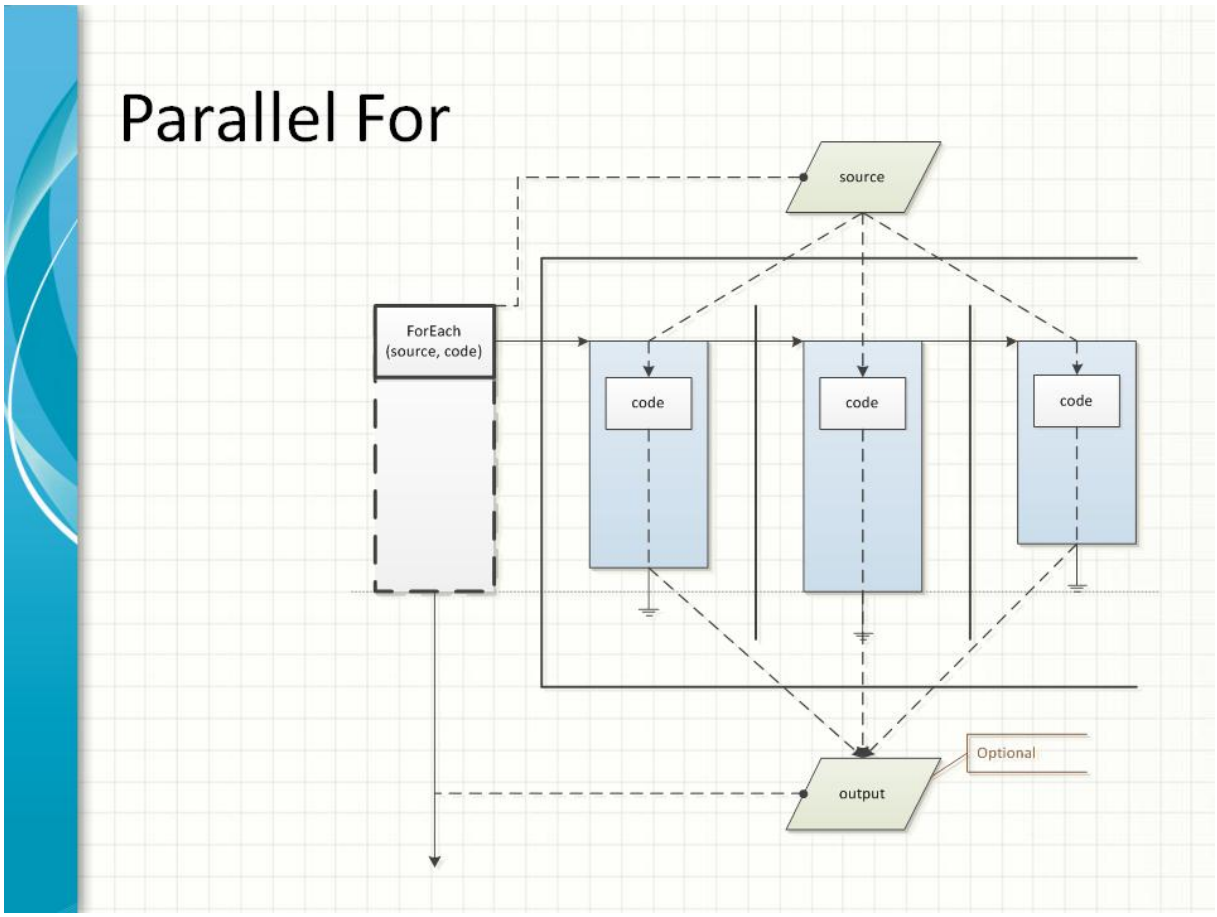


Parallel For (actually called ForEach because For would clash with the reserved keyword for) is a construct that enumerates in a parallel fashion over different containers. The most typical usage is enumerating over range of integers (just like in the classical for), but it can also be used similar to the *for..in* statement.

Typical usage is to call Parallel.ForEach, provide lower and upper iteration bound (as integers) and then call Execute on some code that accepts an integer parameter. This code will be called many times, once for each possible value in the iteration range. Of course, this code will be called in many threads, possibly at the same time.

Another way is to provide an *iteration source* which can be any object that implements Delphi-compatible iterator or Windows IEnumerable interface. In that case, the code that is passed to the Execute must accept a TOmniValue parameter. (TOmniValue being a 16-byte record that can accept almost any data, similar to the Delphi's Variant but faster.)

http://www.thedelphigeek.com/2010/06/omnithreadlibrary-20-sneak-preview-1.html



Execution model for parallel for is quite complicated. Internally, ForEach creates a source object which encapsulates access to a data source (an integer range or some other kind of data). This data source is fully thread-safe and is written in a way that minimizes collisions between threads when accessing the source data. It also creates a quite complicated code block that wraps your own code and provides it with that data. Optionally your code can provide an output that flows into an output queue which can be used from the main program.

ForEach waits for all background tasks to complete execution by default but it can be modified with the NoWait modifier (just as in the Join and ParallelTask examples).

To demonstrate the parallel for, I've written a simple loop that loops from 1 to 10 millions, checks each number if it is a prime number and outputs all numbers into an output queue. First 1000 primes are then displayed in the user interface.

```
procedure TfrmMultithreadingMadeSimple.btnSequentialPrimesClick(Sender:
   TObject);
var
  i: integer;
  prime: TOmniValue;
  primeQueue: IOmniBlockingCollection;
  time: int64;
begin
  lbForEachPrimes.Clear; lbForEachPrimes.Update;
  time := DSiTimeGetTime64;
  primeQueue := TOmniBlockingCollection.Create;
  for i := 1 to CMaxPrime do
    if IsPrime(i) then
      primeQueue.Add(i);
  lbLogForEach.Items.Add(Format('Sequential primes 1: %d ms', [DSiElapsedTime64(time)]));
  for prime in primeQueue do begin
    lbForEachPrimes.Items.Add(IntToStr(prime));
    if lbForEachPrimes.Items.Count = 1000 then
      break; //for
  end;
end;
```

The code uses a TOmniBlockingCollection, which is a thread-safe queue that stores TOmniValue objects. Primes are added to this queue in the for loop and then read from in the second for loop. As you can see, it takes 7 seconds and a half to run the code on my machine. The 1000-prime is 7919.

Conversion to the parallel for is pretty straigtforward.

```
procedure TfrmMultithreadingMadeSimple.btnForEachUnorderedPrimesClick(Sender:
   TObject);
var
  prime: TOmniValue;
  primeQueue: IOmniBlockingCollection;
  time: int64;
begin
  lbForEachPrimes.Clear; lbForEachPrimes.Update;
  time := DSiTimeGetTime64;
  primeQueue := TOmniBlockingCollection.Create;
  Parallel.ForEach(1, CMaxPrime).Execute(
    procedure (const value: integer)
    begin
      if IsPrime(value) then begin
        primeQueue.Add(value);
      end;
    end);
  lbLogForEach.Items.Add(Format('Unordered primes: %d ms', [DSiElapsedTime64(time)]));
  for prime in primeQueue do begin
    lbForEachPrimes.Items.Add(IntToStr(prime));
    if lbForEachPrimes.Items.Count = 1000 then
      break; //for
  end;
end;
```

“For” statement was changed with Parallel.ForEach(1, CMaxPrime).Execute and “for” body was wrapped into appropriate anonymous method.

Running the code shows that the parallel version is indeed much faster – it needs only 1500 ms to finish.

We can also notice something interesting – because the code runs in many threads we can not expect that output will be ordered the same way as in the sequential version. As this can be a big problem, OmniThreadLibrary implements *order-preservation parallel for*.

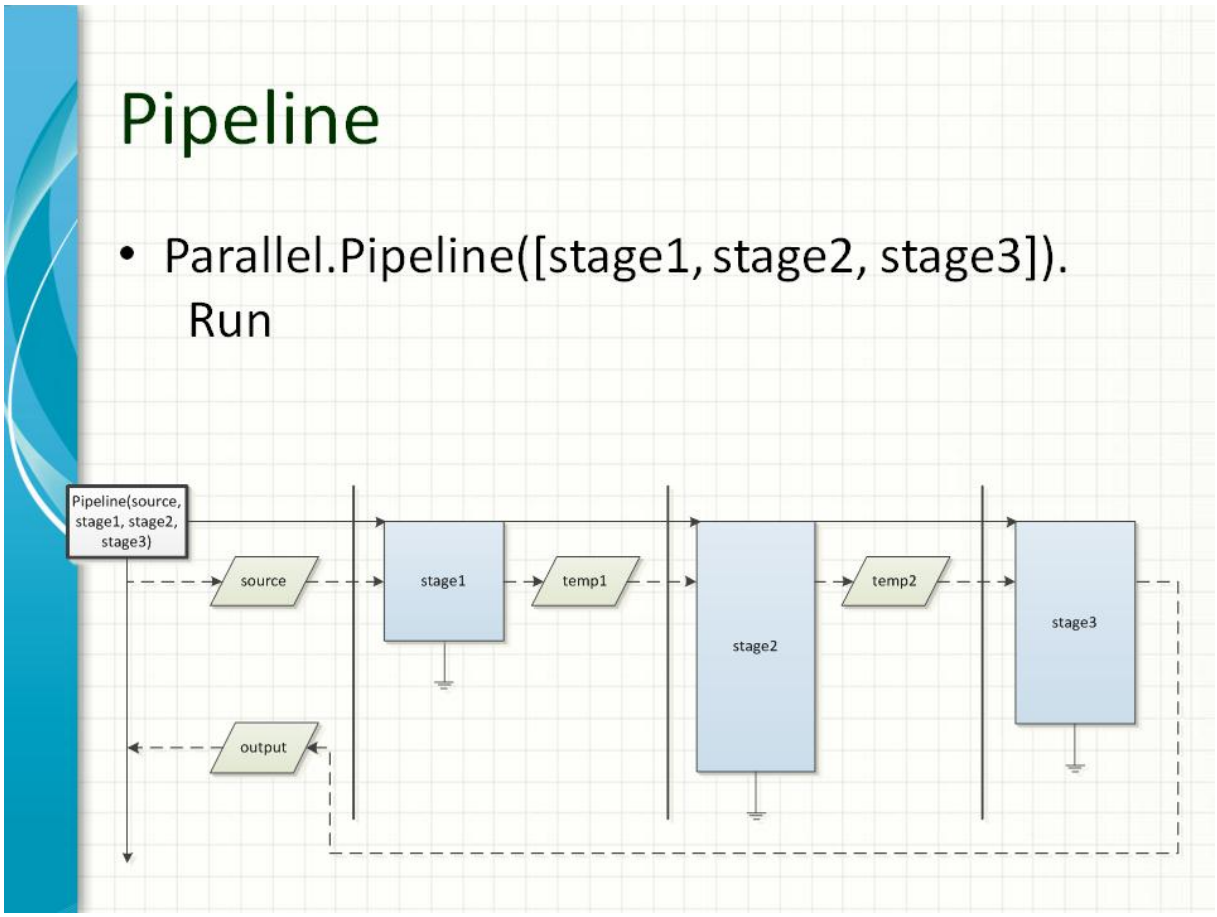
```
procedure TfrmMultithreadingMadeSimple.btnForEachOrderedPrimesClick(Sender:
   TObject);
var
  prime: TOmniValue;
  primeQueue: IOmniBlockingCollection;
  time: int64;
begin
  lbForEachPrimes.Clear; lbForEachPrimes.Update;
  time := DSiTimeGetTime64;
  primeQueue := TOmniBlockingCollection.Create;
  Parallel.ForEach(1, CMaxPrime)
    .PreserveOrder
    .Into(primeQueue)
    .Execute(
      procedure (const value: integer; var res: TOmniValue)
      begin
        if IsPrime(value) then
          res := value;
      end);
  lbLogForEach.Items.Add(Format('Unordered primes: %d ms', [DSiElapsedTime64(time)]));
  for prime in primeQueue do begin
    lbForEachPrimes.Items.Add(IntToStr(prime));
    if lbForEachPrimes.Items.Count = 1000 then
      break; //for
  end;
end;
```

To request order preservation, just add .PreserverOrder call. This brings in some other changes. Output must be written into an IOmniBlockingCollection (which you pass to the OmniThreadLibrary by calling the Into function) and code signature must be modified. Code must accept one parameter, as before, but it must also return one parameter of the TOmniValue type. If this parameter is set in the code, it will be written into the output queue. If not, nothing will be written.

Running the ordered version we can notice two things – the output is always ordered and the execution is slightly slower.

ForEach supports many other features, which are nicely demoed in the test programs that come as part of the OmniThreadLibrary distribution and I could spend all session just talking about it.

As I can’t afford that luxury, let’s move to the next topic.



Pipeline construct implements high-level support for multistage processes. The assumption is that the process can be split into stages (or subtasks), connected with data queues. Data flows from the (optional) input queue into the first stage, where it is partially processed and then emitted into intermediary queue. First stage then continues execution, processes more input data and outputs more output data. This continues until complete input is processed. Intermediary queue leads into the next stage which does the processing in a similar manner and so on and on. At the end, the data is output into a queue which can be then read and processed by the program that created this multistage process. As a whole, a multistage process functions as a pipeline – data comes in, data comes out.

http://www.thedelphigeek.com/2010/11/multistage-processes-with.html

http://www.thedelphigeek.com/2011/09/life-after-21-pimp-my-pipeline.html

This demo shows a very simple three-stage process that “encrypts” a text file. (And by “encrypt” I mean that it uses a variant of Ceasar’s cypher called ROT13, which is completely unsafe and should not be used for any serious purpose.)

```
procedure TfrmMultithreadingMadeSimple.btnPipelineRot13Click(Sender: TObject);
var
  time: int64;
begin
  time := DSiTimeGetTime64;
  Parallel.Pipeline
    .Stage(PipelineRead)
    .Stage(PipelineEncrypt)
    .Stage(PipelineWrite)
    .Run
    .WaitFor(INFINITE);
  lbLogPipeline.Items.Add(Format('Pipeline ROT13: %d ms', [DSiElapsedTime64(time)]));
end;
```

The code sets up three stages – reader, encryptor and writer, - runs the pipeline and waits on it to complete the work. Non-waiting version is trivially implemented – just call .Run without .WaitFor.

```
procedure PipelineRead(const input, output: IOmniBlockingCollection);
var
  fIn: textfile;
  line: string;
begin
  Assign(fIn, '..\..\AliceInWonderland.txt');
  Reset(fIn);
  while not Eof(fIn) do begin
    Readln(fIn, line);
    output.Add(line);
  end;
  CloseFile(fIn);
end;
```

Reader ignores its input (the *input* parameter) because there's no input to this stage. It opens the file using the good old pascal-style file access, reads it line by line (that's the only reason to use textfile instead of a file stream) and outputs each line to the output queue. IOmniBlockingCollection is again used to implement the queuing system.

```
function ROT13(const s: string): string;
var
  i: integer;
begin
  Result := s;
  for i := 1 to Length(Result) do begin
    if CharInSet(s[i], ['A'..'M', 'a'..'m']) then
      Result[i] := Char(Ord(s[i]) + 13)
    else if CharInSet(s[i], ['N'..'Z', 'n'..'z']) then
      Result[i] := Char(Ord(s[i]) - 13);
  end;
end;
procedure PipelineEncrypt(const input: TOmniValue; var output: TOmniValue);
begin
  output := ROT13(input.AsString);
end;
```

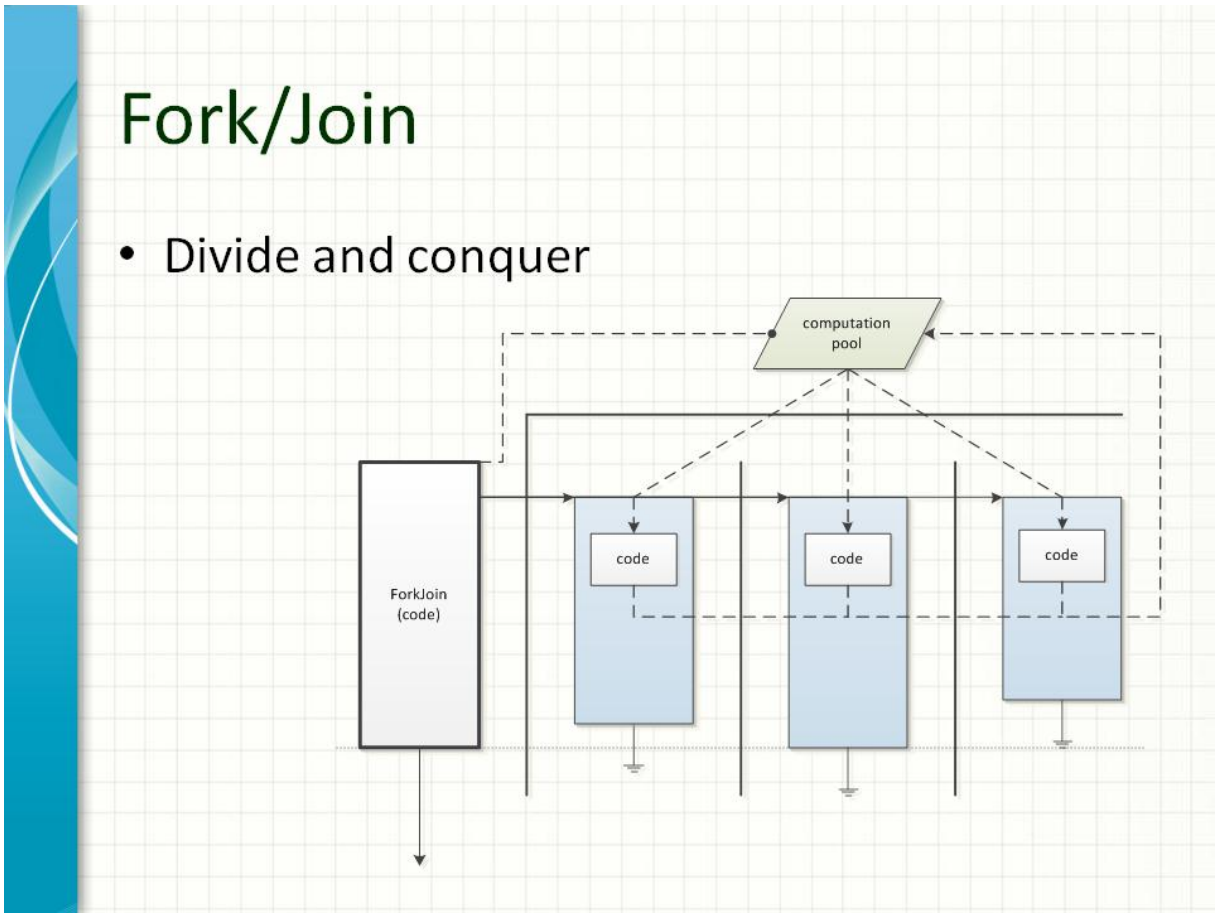
Encryptor is written as a *simplified stage* – that is a stage that doesn't accept input and output **queues** as a parameter but a single element from the queue (which is, of course, a TOmniValue). Queue reading and writing is done in a wrapper provided by the OmniThreadLibrary.

```
procedure PipelineWrite(const input, output: IOmniBlockingCollection);
var
  fOut: textfile;
  line: TOmniValue;
begin
  Assign(fOut, '..\..\AliceInWonderland-rot13.txt');
  Rewrite(fOut);
  for line in input do
    Writeln(fOut, line.AsString);
  CloseFile(fOut);
end;
```

Writer as again a *normal stage*, just as the reader is, but unlike reader it processes only the input queue and ignores the output. Output is written into a file, line by line.

As you can see, the execution is quick. From the input text the program produced "encrypted" output text.

Pipeline construct supports exception handling but I don't have time to go into details; see my blog (as usual).



Fork/Join is an implementation of “Divide and conquer” technique. In short, Fork/Join allows you to:

- Execute multiple tasks
- Wait for them to terminate
- Collect results

The trick here is that subtasks may spawn new subtasks and so on ad infinitum (probably a little less, or you’ll run out of stack ;)). For optimum execution, Fork/Join must therefore guarantee that the code is never running on too many background threads (an optimal value is usually equal to the number of cores in the system) and that those threads don’t run out of work.

To achieve this, ForkJoin creates many worker threads and connects them to a *computation pool*. Computation requests (i.e. subtasks) are written into this pool. They are read by worker tasks, processed and optional new subtasks are inserted back into the computation pool. Due to implementation details, ForkJoin **always** waits on all worker tasks to complete the execution. (Which happens only when the computation pool is empty.)

http://www.thedelphigeek.com/2011/05/divide-and-conquer-in-parallel.html

As the ForkJoin is quite complicated, I’ll just show part of the parallel QuickSort implementation. You’ve seen the OmniThreadLibrary in action and I think that you’ll agree that it is a powerful addition to the programmers toolbox.

TParallelSorter class creates a ForkJoin object in its constructor.

```
constructor TParallelSorter.Create(const data: TData);
begin
  inherited Create(data);
  FForkJoin := Parallel.ForkJoin;
end;
```

QuickSort method, which is initially called on the full array (i.e. *left* parameter is 0 and *right* parameter is equal to array length – 1) for checks if current part of the array is *small enough*. If it contains less than 1000 elements, it is sorted with a simple insertion sort.

```
procedure TParallelSorter.QuickSort(left, right: integer);
var
  pivotIndex: integer;
  sortLeft  : IOmniCompute;
  sortRight : IOmniCompute;
begin
  if right > left then begin
    if (right - left) <= CSortThreshold then
      InsertionSort(left, right)
    else begin
      pivotIndex := Partition(left, right, (left + right) div 2);
      sortLeft := FForkJoin.Compute(
        procedure
        begin
          QuickSort(left, pivotIndex - 1);
        end);
      sortRight := FForkJoin.Compute(
        procedure
        begin
          QuickSort(pivotIndex + 1, right);
        end);
      sortLeft.Await;
      sortRight.Await;
    end;
  end;
end;
```

Otherwise, pivot index is found (details doesn't matter), array (the part of array we are currently working on) is partitioned into two halves and two subcomputations are created – one will sort the left part of the array and another the right. Those computations are automatically added to the computation pool.

The code next waits on both computations to terminate by calling the Await method. While waiting, Await uses the current thread to process other computations waiting in the computation pool and as such makes a full use of the thread.

That brings us to the end of today's session.


QUESTIONS?