Debugging Support for TokenNet Computations

KEVIN MITCHELL
Agilent Measurement Research Laboratory

TokenNets have been proposed as a way of simplifying the exploitation of multicore machines in the streaming domain. Providing high-level debugging support for such an approach presents a number of challenges. We describe the motivation behind the design of a TokenNet debugger and details of its implementation.

Key Words and Phrases: TokenNet, Parallelism, Debugging, Visual Studio

1. INTRODUCTION

TokenNets[Barford et al. 2009] have been proposed as a way of exploiting the power of multicore machines in the streaming domain whilst hiding many of the complexities usually associated with parallel programming. The approach was partly inspired by the recent work on Concurrent Collections[Knobe and Rao 2009] and also has similarities to some of the earlier research on Colored Petri Nets[Jenson 1997]. A TokenNet instance is comprised of a directed acyclic graph where the vertices denote computations over token streams. A token has two components, an index tuple and a data value. The index tuple, formed from a sequence of integers, uniquely identifies a token in a token stream. A token's type is defined by the length of the index tuple coupled with the type of the data component. Each edge in the graph is associated with a token type, imposing a uniform structure on the tokens flowing over the edge.

Where the data component of a token is a simple value, for example a floating point number, then we do not need to concern ourselves with questions of access rights to this data. However, in more interesting cases the data might consist of a reference to some shared structure, for example an array of values. There are two cases to consider here. In the case where the array is treated as an immutable object then there is no harm in having multiple tokens in existence referencing the same array. However, in the case of mutable objects a valid TokenNet computation graph must ensure that there is only one active token referencing the object in existence at any point in time. In this case the reception of a token by a vertex can be viewed as conferring the right to alter the referenced data. If other parts of the graph subsequently require access to this data the vertex can either pass the token to a downstream node for further processing, or embed the reference in a new token before passing it on.

In the simplest scenario a vertex receives a token from a single input stream, performs some operation on the data, and forwards a result token. The situation becomes more interesting when a vertex has two or more incoming edges. In this case the code associated with the vertex can only perform a computation when there are matching tokens on each of the input edges. In the simplest case two tokens match when they have identical index tuples. In more complex cases we associate projection functions with each incoming edge. Consider the case of two input edges, e_1 and e_2. A token t_1 on edge e_1 then matches with token t_2 on edge e_2 if the projection function π_1 when applied to the index tuple of t_1 produces the same value as the result of applying projection function π_2 to the index tuple for t_2. The simple case just uses the identity function for the projections π_i. This concept of matching generalizes to the case where there are more than two incoming edges.

If a token arrives at a vertex and there are no matching tokens on the other incoming edges then the token is queued until a match is found. Thus tokens can overtake each other, and tokens can potentially be blocked indefinitely if no match is ever found. Tokens are typically processed as soon as a match is identified and so multiple threads can be running code within a vertex simultaneously. This maximizes the potential

Author's address: K. Mitchell, Agilent Labs Scotland, South Queensferry, Scotland EH30 9TG

for parallelism, but in some examples, for example where a vertex maintains state, it may be desirable to place constraints on the extent of this parallelism. We can impose a partial order on the token indices, and then require the vertex code to be executed sequentially with respect to this ordering, or even require only a single instance of the vertex code to be active at any time. The TokenNet approach defines various vertex types to capture these different behaviors. See [Barford et al. 2009] for a more comprehensive description of the motivation behind this approach, its semantics, and some examples.

Debugging TokenNet applications can be challenging. Whilst some of the pitfalls associated with parallel programming are largely avoided by the TokenNet approach, the computation is still spread across multiple nodes in the graph, with queued tokens on the graph edges. It would be unfortunate if a user was insulated from many of the complexities of parallelism when writing a program, only to be confronted with the same complexities during the debugging process. Ideally we would like a higher-level view of the debugging process to complement the TokenNet approach. This report describes some initial steps towards this goal.

2. DEBUGGING TOKENNET APPLICATIONS

The current TokenNet implementation is targeted at C# applications. Programs are built using Visual Studio 2010 and .NET 4, exploiting the new support for parallel tasks in this release, and debugged using the Visual Studio Debugger[Microsoft 2009c]. In addition to the usual debugging operations, for example setting breakpoints, examining data values and the call stack, the debugger also provides support for parallel programming. The Threads panel displays the current list of active threads, and allows you to switch between threads whilst inspecting the program's current state. Visual Studio 2010 extends this support with a Parallel Tasks panel, allowing the user to inspect the currently created tasks, and the mapping between tasks and threads. Whilst all these facilities provide a detailed view of the computation at a microscopic level, it is often difficult to extrapolate from this level to get an overview of the graph computation itself. Indeed even the structure of the graph is implicit, with the information spread across a large number of variables.

Ideally we would like to be presented with an overview of the TokenNet graph during a debugging session. From this view it should be possible to see the tokens currently being processed or queued, and be able to drill down into any of these to see more detail if required. However, we would like to complement the existing debugger, not replace it, and so we also need to be able to tie the two worlds together. For example, if a token is currently being processed by a vertex it should be possible to determine the ID of the task performing this work, thus enabling us to inspect the computation in greater detail using the task and thread views. Figure 1 illustrates the kind of hybrid view we are aiming for. The left-hand pane provides an overview of the current graph, and the top-right pane provides more details of an element within the graph. The details in this pane can be correlated with the task and thread information provided by the Visual Studio debugger, as shown in the bottom two panes on the right-hand side of the figure.

3. THE TOKENNET LIBRARY

The TokenNet library was designed to support high-speed parallel processing of tokens on multicore machines. Each time a token is passed to a downstream node a separate task is created to process this token. The library uses .NET's Task Parallel Library (TPL)[Leijen and Hall 2007] to efficiently allocate tasks to threads and cores. For each outgoing edge a vertex simply has to maintain a reference to the destination vertex's method for processing tokens received along this edge, a delegate in C# terminology. Unfortunately whilst such a representation is sufficient to express a TokenNet computation graph, it leaves much to be desired from the perspective of supporting a debugger.

From a method reference it is possible to use .NET's reflection mechanism to ascertain the enclosing vertex, and we can continue this traversal process to determine all the downstream nodes from this point. However, as each vertex is unaware of its upstream neighbors we will only be able to reconstruct part of the graph unless we are fortunate enough to start with a reference to the graph's root node. To make matters worse, much of the state of the computation is implicit in the representation of the graph, for example in tasks queued within the scheduler. It would certainly be possible to tease out of lot of this information using a combination of the EnvDTE API[Microsoft 2009b] and reflection. However, we assume that the debugger will only be used when the application is compiled in debug mode, and so we will already be paying a

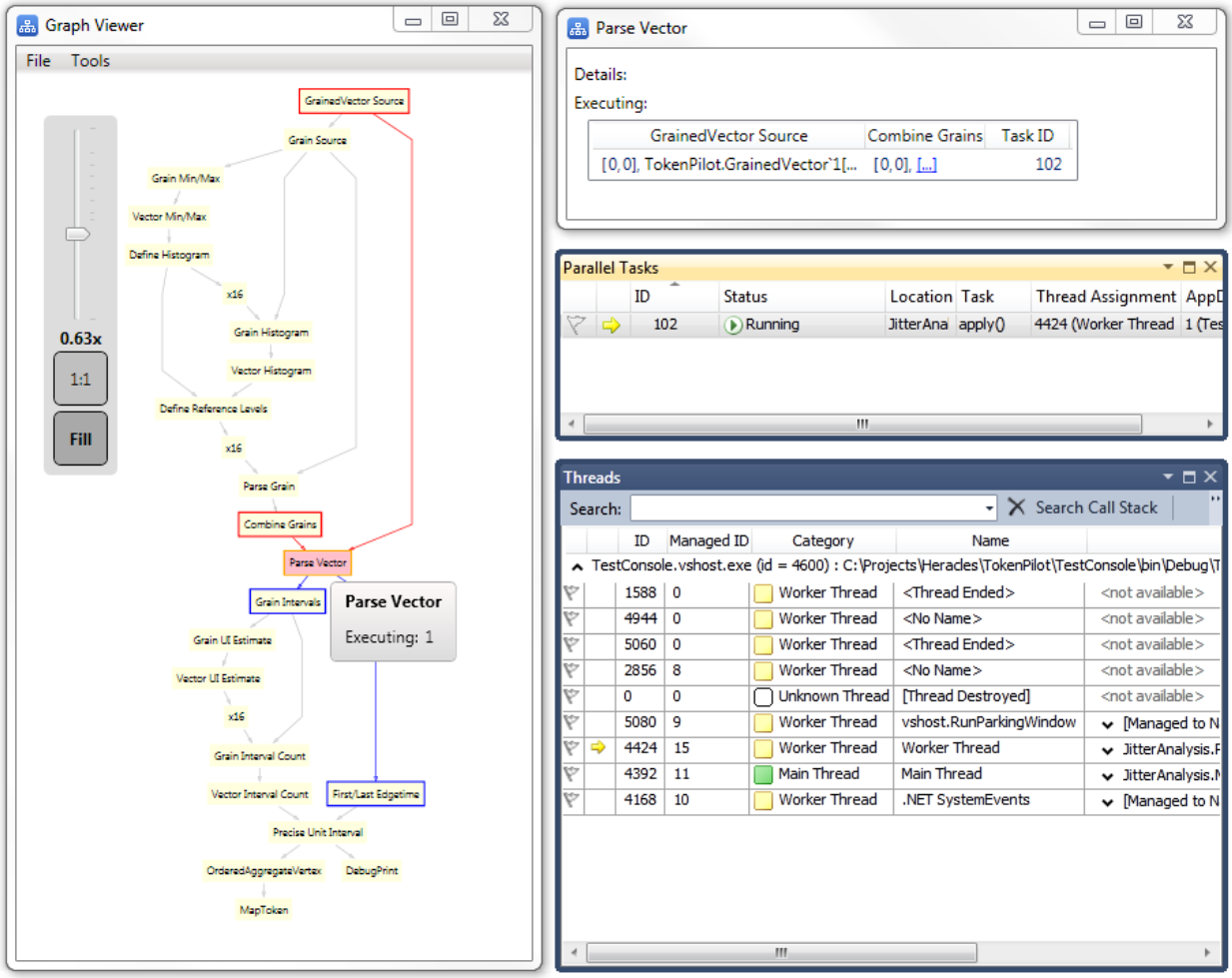

Fig. 1. A Hybrid Debugger

performance penalty in such cases. A simpler approach is therefore to modify the library to maintain more explicit information about the state of the computation, and the graph structure, when compiled in debug mode. The library was updated accordingly, and we also took the opportunity to associate a globally unique identifier (GUID) with each graph which changes whenever the graph structure is altered. This simplifies the design of the graph viewer as it allows us to rapidly distinguish between needing to display a new graph and simply updating the state of an existing one.

4. DEBUGGER VISUALIZERS

As mentioned in Section 2, we do not want to write a new debugger, but rather complement the existing one. So how do we extend a Visual Studio debugger? The usual way to extend Visual Studio is with an add-in[Microsoft 2009a]. However, this does not give us access to the state of the application being debugged. We could actively instrument the application, using a communication mechanism to transmit changes of state to a graph viewer, but this would be too intrusive. Furthermore it would be very time-consuming to update the graph view whenever the application state changed. In most cases it is sufficient to determine the graph state when something interesting happens, for example when we reach a breakpoint. But at that point the application is halted, and so we cannot rely on it providing us with the information about its

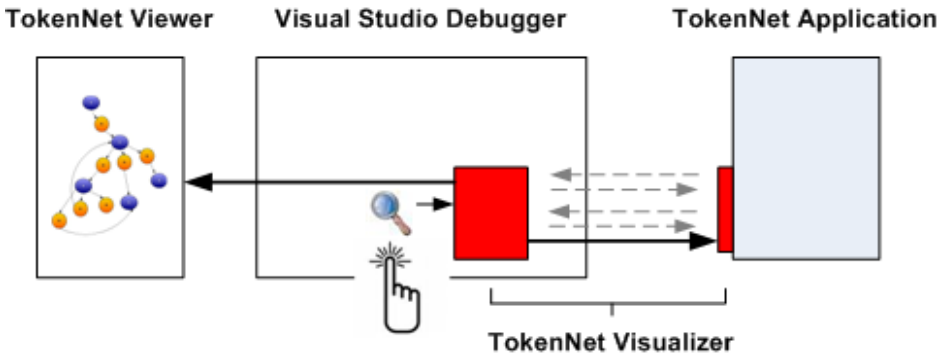

Fig. 2. The architecture of a visualizer-based graph viewer

current state. Fortunately Visual Studio provides us with an alternative.

Debugger visualizers[Microsoft 2009d] are a component of the Visual Studio debugger user interface. They can be used to create a dialog box or other interface to display a variable or object in a format that is appropriate to its data type. The Visual Studio debugger comes with a set of standard visualizers. However, Microsoft also provides a mechanism that allows users to develop and install their own visualizers within the debugger. Most visualizers present a read-only view of the observed data, but it is also possible to write a visualizer that allows you to edit the data as well. Visualizers are represented in the debugger by a magnifying glass icon. When you see the magnifying glass in a DataTip, in a debugger variables window or in the QuickWatch dialog box, you can click on the magnifying glass to select a visualizer appropriate to the data type of the corresponding object. A debugger visualizer has two components, and the first of these runs within the debugger process. To retrieve information about the application being debugged it communications with a second component that runs within the address space of the application under test, as illustrated in Figure 2.

Debugger visualizers typically behave in a modal fashion. They create a dialog providing a visual or textual representation of the value being inspected, and the user has to dismiss this dialog before continuing with the debugging process. Such behavior is undesirable in our situation as we wish to use the graph view in conjunction with the normal debugging behavior, which requires both the debugger and graph viewer to be active at the same time. Furthermore we do not want to create a new graph view whenever we access the visualizer. A better approach would be to display a single graph view that is updated by each run of the visualizer. Of course that doesn't stop us cloning such a view to persist it if required.

5. COMMUNICATING WITH THE GRAPH VIEWER

Visual Studio can be extending by automating its existing features or by adding new ones. An add-in is a compiled DLL that runs in the Visual Studio integrated development environment (IDE). An add-in can define menu items and panels that can be docked within the IDE. The current version of the TokenNet library makes extensive use of tasks, and so relies on Visual Studio 2010/.NET 4. Unfortunately at the time of writing these were both at the Beta development stage, and were not entirely backwards compatible. In particular, the library we wished to use for displaying graphs, Graph#[Palesz 2009], did not build under this release of .NET. We therefore made the decision to build the graph viewer as a separate process. This allowed us to use .NET 3.5/Visual Studio 2008 for the viewer whilst continuing to use .NET 4 for the application under test. Using a separate process has the additional advantage that we can export graphs from a running application and then view them without having to run an instance of Visual Studio. A user is largely unaware of the view being provided by a separate process as the debugger visualizer takes care of starting the process if it is not already running. The main limitation of the current approach is that the window does not integrate with the rest of the IDE. Once the final versions of Visual Studio 2010/.NET 4 are released, and libraries are updated to work with them, it should be a simple task to package up the

viewer as an add-in, if required. This will allow us to manipulate the view just like all the other Visual Studio views, for example docking it within the main IDE display.

Irrespective of whether the viewer is run as a separate process or an add-in, we still need to provide a mechanism for communicating between the debugger visualizer and the graph viewer. Debugger visualizers run in a sandbox that restricts the ways in which they can communicate with other components even if, in the case of an add-in, they are running within the same process. This makes it difficult to communicate via shared memory, for example. To allow the visualizer and viewer to run in separate processes we used Microsoft's Windows Communication Foundation (WCF)[Microsoft 2009e], a part of the .NET Framework that provides a unified programming model for rapidly building service-oriented applications. It seemed reasonable to assume that the viewer and visualizer would be running on the same machine, and we therefore established an WCF connection between these two components using named pipes.

WCF is really aimed at building service-oriented applications that communicate across the web and the enterprise. In particular, it does not assume that both ends of the connection are running .NET. Whilst this makes sense in the general case, for the graph viewer such decoupling of implementation from interface is unnecessary and restricting. We would like the visualizer and viewer to be able to share class definitions, and respect the inheritance hierarchy of any objects they exchange. Whilst in theory this can be done in WCF, in practice achieving this level of shared awareness is not easy. For simplicity we eventually decided to just serialize and deserialize complex objects within our own code, simply using WCF to establish the communications. This solution is not ideal as the interface between the two components becomes more opaque than it should be, defining methods that pass byte array representations of objects, rather than the objects themselves. However, placing the serialization in our hands, rather than relying on WCF, avoided a lot of the complications involved in exchanging complex data defined in multiple assemblies across a WCF link.

What data needs to be transported across the WCF link? Clearly the visualizer needs to pass a representation of the computation graph to the viewer. The graph structure of a TokenNet application is typically small, and so passing the topology of the graph to the viewer is a simple matter. But TokenNet graphs contain tokens, and exchanging token information is more problematic. A token has two components, the index tuple and the data value. The type of the data value depends on the particular link that carries it, and this is application specific. In the worst case it could be a reference to a data structure that takes up most of the memory of the running application. We clearly do not want to pass all the data across to the viewer in such a case as it would be far too time-consuming. Complex data is typically hierarchical in nature. Even large arrays can be viewed in this fashion by considering them as being formed from multiple smaller segments. We can exploit this hierarchy by passing across to the viewer just the top-level details of the data, a précis, and then providing a mechanism, described in Section 7.1, for requesting further details.

5.1 Timeouts

There are, in fact, two communication links in our solution, as illustrated in Figure 2. The link between the user process under test and the debugger is hidden inside the debugger visualizer mechanism. In theory this is not a problem as we should not really care about the details of how the two components exchange information. Unfortunately, in reality the communication mechanism provided uses a very short timeout when waiting for a response from the user-side implementation of the visualizer. If this component does not respond fast enough, for example because it tries to send too much data back to the debugger, the timeout is exceeded and the visualizer fails. Unfortunately the timeout is so small that even attempting to pass relatively small amounts of data across the link can trigger a timeout. As the construction of the link is performed implicitly there is no opportunity within the provided API to alter any of the timeout settings. It turns out there is an undocumented way of altering this behavior, although it is far from ideal. In response to a question we posted to the Microsoft team responsible for this aspect of the system Andrew Hall replied with the following:

> This functionality is governed by one of the "*Timeout" registry keys under `"HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\10.0\\Debugger"`.
>
> How the timeouts work for a visualizer is the language's Expression Evaluator (EE) uses the timeout

of the mechanism from which the visualizer is opened. For example, if you open the visualizer from a DataTip, the timeout will be the DataTip's timeout value, if you open the visualizer from the watch window, the timeout used will be the timeout for the watch window, etc. Therefore I can't tell you exactly which key(s) will solve this problem for you, so you may try increasing the values of all "*Timeout" keys.

There is an alternative approach that side-steps the timeout problem, albeit at the cost of increased execution time. We can split the transfer of the state into a number of separate exchanges of data. This is the approach adopted by the current implementation. The debugger visualizer first extracts a list of vertices from the user application. Then for each vertex it requests the list of outgoing edges. It then retrieves the tokens associated with each vertex and edge using a separate request for each one. And finally it retrieves the current thread and task state. All this information is then merged into a coherent whole and passed to the graph viewer over the WCF link. Each exchange involves a relatively small amount of data, at least for the examples that have been tried, and so there are no timeouts. Of course even this degree of decomposition may not be adequate in the general case. The number of tokens that could be queued on an edge, or active on a vertex, is application-dependent, and so there may be situations where we could still suffer from timeouts. To avoid this we would really need to split the data transfers into even smaller units of work, for example only passing at most N tokens in each exchange. But clearly this is not an ideal direction to go in; each message exchange has a non-negligible overhead which quickly becomes the bottleneck when updating the graph viewer with the current execution state.

What other alternatives might be available to us? The code that runs within the user address space has some restrictions, for obvious reasons. For example, you cannot create a thread, start it running, and then return control back to the debugger. So approaches such as communicating asynchronously from the halted user application to the viewer are not available to us. Communicating synchronously from the halted user application to the viewer would also not help as the link between the visualizer and the user application would still time out. In summary, until Microsoft adds an API to provide finer control over the link between the visualizer and the debugged application we are left with no options other than using undocumented registry settings, or splitting the communication into multiple stages.

5.2 Tasks

One of the more frustrating aspects of the current IDE is that it occasionally displays information that is not conveniently accessible through any published APIs. The information displayed in the Parallel Tasks panel, see Figure 1, is a good example of this situation. We would like to highlight nodes in the graph that are currently active, where there are threads executing tasks processing tokens associated with the nodes. To do this we need to construct a map between Thread IDs and Task IDs. The Parallel Tasks panel is a view of such a map. Unfortunately the code implementing this panel constructs the view by rummaging around in the active call stacks, amongst other places, and there is currently no API that exposes this information. Visual Studio provides an automation interface, EnvDTE, which allows a component, such as a debugger visualizer, to explore and automate various aspects of the IDE's behavior. Unfortunately, whilst this interface allows us to retrieve an object representing the Parallel Tasks panel, it's contents are opaque to us. We must therefore look for an alternative strategy for obtaining this information. Our current approach uses EnvDTE to evaluate debugger expressions on each active thread. The following code fragment illustrates the approach:

```
EnvDTE80.DTE2 dte2 = (EnvDTE80.DTE2)
  System.Runtime.InteropServices.Marshal.GetActiveObject("VisualStudio.DTE.10.0");
Debugger4 debugger = (Debugger4)dte2.Debugger;

foreach (EnvDTE.Thread thread in debugger.CurrentProgram.Threads) {
  foreach (EnvDTE.StackFrame frame in thread.StackFrames) {
    EnvDTE.Expression e = debugger.GetExpression3("System.Threading.Tasks.Task.Current.Id",
                                                  frame, true, false, true);
    if (e != null && e.IsValidValue)
      threadToTaskMap[thread.ID] = Int32.Parse(e.Value);
```

```
        break;
      }
    }
```

6. GRAPH#

The structure of a TokenNet graph is clearly application-dependent, and so the graph viewer must be able to display any graph derived from such an application in a pleasing and informative fashion. Fortunately, TokenNet graphs are acyclic, and so there is a natural flow of tokens though the graph, from top to bottom, or left to right, depending on how we chose to orientate the graph. The Sugiyama algorithm[Eiglsperger et al. 2005] is ideal for automating the layout of such graphs. Unfortunately there are very few free graph layout libraries, and even fewer that support the Sugiyama algorithm. Fortunately a recent arrival on the scene, Graph#[Palesz 2009], looked suitable for the task.

The design of the library relies heavily on the design and philosophy of Windows Presentation Foundation (WPF)[MacDonald 2008]. In WPF you typically create visible UI elements in the declarative XAML markup language, and then separate the UI definition from the run-time logic by using code-behind files, joined to the markup through partial class definitions. Graph# is built as a WPF component, allowing the look and feel of the generated graphs to be controlled by defining suitable definitions in XAML. The approach works well for vertices as a vertex typically occupies a clearly defined area of the screen. The approach is more problematic for edges, particularly those represented by multiple lines. For example, attaching a label to a vertex is a simple operation. Labeling an edge is a lot more complex because it is less clear where the label should be displayed. However, the main disadvantage of Graph# is an almost complete lack of documentation and support, at least at the time of writing. This made the task of developing with Graph# far more time-consuming than it could have been, although developing similar functionality from scratch would have been even more so. The role of Graph# is to layout and display graphs, not define an API to represent them. This task is delegated to another library, QuickGraph[Pelikhan 2009].

7. THE GRAPH VIEWER

The current state of a TokenNet computation is not easy to display in a concise fashion, and so we adopt a hierarchical approach. The initial view just shows the graph structure consisting of the vertices that perform the token processing, and the edges connecting these vertices. The graph can be easily zoomed and panned, allowing relatively large graphs to be viewed. The active vertices, those currently processing tokens, are highlighted, and the vertex that triggered the breakpoint is identified. Tokens can be queued on edges waiting for a match from another incoming vertex edge. The size of this queue is used to determine the width of the edge in the graph view. Given that the size could vary considerably depending on the example we use a logarithmic scale. This allows the edges containing pending tokens to be easily identified whilst not obscuring the view with excessively wide lines. One of the ways in which a TokenNet application can fail is through a token leak. If tokens arrive at a vertex and never find a match because of a bug in the application they will gradually accumulate at the edge. Such errors should be easy to spot from the graph view once the application has been running for a while. The viewer also uses vertex and edge tooltips to display a more detailed summary of the element's state, for example the number of tokens queued and being processed.

7.1 Exchanging token data

Detailed information about the computation's state can be requested via context menus attached to the graph's components. What should we display in the vertex and edge views presented in response to such a request? Clearly we need to display a list of the relevant tokens, but each token contains a data component whose type and complexity is application-specific. The initial view displays the token indices plus a précis of the token data, as illustrated in the panel at the top-right of Figure 1. Of course in some cases, where the data is just a simple value for example, the précis is just the string representation of the value itself. But in general the précis only contains a partial representation of the data, and is displayed as a hyperlink. If the user clicks on this link the viewer will display a more detailed view of the data associated with the token. But where does this additional data come from?

As mentioned in Section 5, the debugger visualizer only transfers a partial view of the data component of a token to the graph viewer. Each vertex and edge has a depth associated with it, and this limits the amount of detail available for each data item. If you want to view the data in more detail then the viewer has to request more details from the client. This seemingly simple request is in reality surprisingly problematic as the state of the client is unknown at this point. It might be running, having been resumed from a breakpoint, or may even have terminated. This is the price we pay for not making the viewer a modal dialog created by the visualizer. We therefore require some assistance from the user. When the user requests more details for a token's data component this request is recorded by the viewer. More precisely, the data depth associated with the vertex or edge containing the token is incremented, and the request is added to a list of pending requests. When the viewer is next resynchronized with the debugger visualizer these requests are passed to the visualizer, and in response it passes back the requested information and the viewer's display is updated. To remind the user that the viewer requires resynchronization with the debugger a flashing icon is displayed at the bottom of the view.

Of course ideally we would like to perform the synchronization step automatically. However it can only be triggered by pressing the magnifying glass icon, for example in a row of the local values view or a watch window. Whilst the EnvDTE mechanism can be used to perform tasks such as selecting a window or menu item remotely, it cannot drill down inside panels to press an embedded icon, or interrogate the fields within a local view to see if any of them represent a token graph or vertex. So at present this has to be performed manually. In reality the extra overhead and inconvenience of manually pressing the icon is small in the larger scheme of things.

8. PROFILING

Profiling is another area where we can potentially complement the existing tools provided by Visual Studio. Current tools will display time spent in individual methods and threads. However, we would also like to know how much time was spent executing code in each graph vertex, and be able to annotate the graph view with the results. As multiple vertices may call the same method, profiling at the method level isn't quite what we want. And as the computations are task rather than thread-based then a thread-based profiler isn't quite what we want either.

The approach we adopted was to instrument the TokenNet support library. When the TIMING compilation flag is defined each vertex is modified to maintain a tick count representing the total amount of processor time spent executing code in the vertex. We assume that a task will remain associated with a thread for the duration of its execution. When the task starts executing code for a vertex it first determines the Win32 process thread that's running it, and the total processor time for this thread. On completion of the code for the vertex the same mechanism is used to determine the number of ticks that have passed, and the total is added to the count for the vertex. Figure 3 illustrates a `Timer` support class that can be used with the following method to accumulate the CPU time for an action associated with a vertex:

```
protected void Time(Action action) {
  Timer t = new Timer();
  try {
    action();
  } finally {
    Interlocked.Add(ref processorTimeInTicks, t.Ticks);
  }
}
```

At any point in time, for example at the end of the application, we can export the collected information using the following code fragment:

```
using (var writer = System.Xml.XmlWriter.Create("graph.xml"))
  graph.SerializeToXml(writer,
    (w, g) => { },  // Add additional graph attributes here
```

```
public class Timer {
  [DllImport("Kernel32", EntryPoint = "GetCurrentThreadId", ExactSpelling = true)]
  private static extern Int32 GetCurrentWin32ThreadId();

  private static ProcessThread GetProcessThreadFromWin32ThreadId() {
    Int32 threadId = GetCurrentWin32ThreadId();
    foreach (Process process in Process.GetProcesses()) {
      foreach (ProcessThread processThread in process.Threads) {
        if (processThread.Id == threadId) return processThread;
      }
    }
    throw new InvalidOperationException("...");
  }

  private ProcessThread pthread;
  private TimeSpan totalProcessorTimeStart;

  public Timer() {
    pthread = GetProcessThreadFromWin32ThreadId();
    totalProcessorTimeStart = pthread.TotalProcessorTime;
  }

  public long Ticks {
    get {
      return (pthread.TotalProcessorTime - totalProcessorTimeStart).Ticks;
    }
  }
}
```

Fig. 3. A Timer class for calculating thread CPU time

```
    (w, n) => { w.WriteAttributeString("ticks", n.ProcessorTimeInTicks.ToString()); },
    (w, e) => { }); // Add additional edge attributes here
```

The graph viewer can load in such graphs and color the nodes based on the relative amount of time spent in each vertex, as illustrated in Figure 4.

Using our test examples we find that many of the tick counts stay at zero. At present it is not clear whether this is due to the quantization of the processor time, combined with very short tasks for some vertices, or some other cause. If it is due to excessively short execution times for individual "work" methods then it suggests that some of the vertices do not have the right level of granularity. The TokenNet approach really relies on a fairly coarse-grained level of parallelism, where the vertices take enough time to process each token to largely hide the token synchronization overheads. If the processing time becomes too short then too much of the overall CPU time will be spent on these overheads. We need to explore the timing figures more carefully. We should also see whether there's a better way of obtaining such timings, as the current approach does not seem particularly elegant or efficient. Of course that may simply be one of the prices we pay for running as managed code.

9. FUTURE DIRECTIONS

The graph window acts as a wide-angled lens, providing a good overview of the structure of the TokenNet application. The Visual Studio debugger, in contrast, acts more like a telephoto lens, displaying a very focused view of a small part of the computational state at any one time. This difference in viewpoint can be exploited to make it easier to perform a number of common operations. For example, a frequent task when debugging TokenNet applications is to set and clear breakpoints at the start of the token processing code for a vertex. Ideally you'd like such breakpoints to be conditional, for example where the condition is

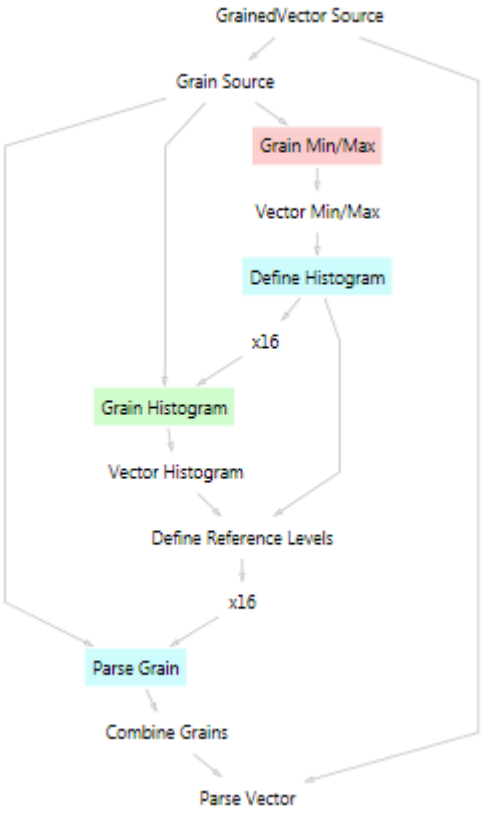

Fig. 4. Highlighting bottlenecks in the profiler

that the token being processed satisfies some property. Selecting a node within the graph, and then toggling a breakpoint flag, would in many cases be a far more natural action than having to find the source file containing the definition of the code for this vertex and then setting the breakpoint in the source view. Furthermore, selecting an existing token in the graph view to form the starting point for the condition would also be very useful. The user might wish to breakpoint the vertex when the successor to an existing token is received, or some other pattern is matched, where the pattern is derived from the selected token. The EnvDTE interface allows us to establish breakpoints programmatically, and so extending the debugging support in this direction would seem a natural next step.

One very useful debugging aid might be to log all tokens passing over a specified edge, or a filtered subset of these tokens. We could obviously do this by altering the source file and recompiling. But we could also potentially do this dynamically. We would use the graph view to select an edge to be logged, and any filtering or other actions to perform on the token stream. At that point the viewer could not alter the running application, any more than it could fetch additional data from the application. However, we could use the "pending request" mechanism, described in Section 7.1, to ask the user to resynchronize the debugger with the viewer. Having halted the application at a breakpoint, if it wasn't already at one, the debugger visualizer would be selected to trigger the resynchronization process. This would have the side-effect of updating the user application to start or stop logging a particular edge. To avoid introducing logging overheads when not required we would simply introduce an additional vertex to perform the logging, and replace the original edge by two edges, one to and one from the new vertex. Of course we would also need a mechanism to undo such a change, and the graph viewer would need to be aware of such logging vertices and exclude them from the view.

Having logged the tokens how do we view them? One approach would be to stream the tokens back to the graph viewer for display and analysis. The logging code would be running in user space, not as part of the debugger visualizer, and so there would be no issues with time-outs in this scenario. We could even instrument the whole graph to show changes in real time using such a mechanism. However, in this case it may be preferable to simply recompile the application in trace mode, and establish a direct connection with the graph viewer, rather than involving the debugger visualizer as a go-between.

REFERENCES

Barford, L., Mitchell, K., and Vandeplas, T. 2009. TokenNets: An approach to programming highly parallel measurement science and signal processing. Tech. Rep. AGL-2009-11, Agilent Measurement Research Laboratory.

Eiglsperger, M., Siebenhaller, M., and Kaufmann, M. 2005. An efficient implementation of Sugiyama's algorithm for layered graph drawing. In *Graph Drawing*. Lecture Notes in Computer Science, vol. 3383/2005. Springer Berlin / Heidelberg, 155–166.

Jenson, K. 1997. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Monographs in Theoretical Computer Science. Springer-Verlag.

Knobe, K. and Rao, G. 2009. Intel Concurrent Collections: Parallelize C++ Programs. In *Intel Developer Forum (IDF 2009)*.

Leijen, D. and Hall, J. 2007. Optimize managed code for multi-core machines. *MSDN*.

MacDonald, M. 2008. *Windows Presentation Foundation with .NET 3.5.* Apress.

Microsoft. 2009a. Add-ins and extensibility. http://msdn.microsoft.com/en-us/library/bb384200(VS.100).aspx.

Microsoft. 2009b. Automation and extensibility for Visual Studio; the EnvDTE namespace. http://msdn.microsoft.com/en-us/library/envdte(VS.100).aspx.

Microsoft. 2009c. Debugging in Visual Studio. http://msdn.microsoft.com/en-us/library/sc65sadd(VS.100).aspx.

Microsoft. 2009d. Visual Studio debugger visualizers. http://msdn.microsoft.com/en-us/library/zayyhzts(VS.100).aspx.

Microsoft. 2009e. Windows Communication Foundation. http://msdn.microsoft.com/en-us/netframework/aa663324.aspx.

Palesz. 2009. Graph#. http://graphsharp.codeplex.com/.

Pelikhan. 2009. QuickGraph. http://www.codeplex.com/quickgraph/.