TokenNets: An Approach to Programming Highly Parallel Measurement Science and Signal Processing

Lee Barford Kevin Mitchell Tom Vandeplas

January 7, 2010

1 Introduction

All industries that provide value-added through the use of software are facing a "multicore crisis." Clock rates of central processing units (CPUs) stopped rising at their former exponential rate in about 2004. Instead, processor manufacturers are increasing the number of parallel execution units or *cores* provided per CPU. The number of cores is now increasing exponentially. In order for product performance to grow with improvements in processors, it will be necessary to program in such a way that this ever-growing number of parallel processors is used effectively.

However, for the sort of algorithms typically found in applications where the computational complexity lies in signal processing and measurement science algorithms, currently available programming tools will fall short in exploiting the numbers of cores soon to be available. Gustafson's Law gives an straightforward (but approximate) way of exploring the limits of performance improvements available from parallelization. The Law can be written

$$T_P = T_{\text{ser}} + \frac{T_{\text{par}}}{P}, \tag{1}$$

where T_P is the time required for a given piece of code to run on P processors, T_{ser} is the time spent in serial (un-parallelized) code, and T_{par} is the time spent in the parallelized portion. In order to maximize performace as P increases, the *serial fraction* (also called the *Karp-Flatt Metric*) [10]

$$e = \frac{T_{\text{ser}}}{T_1} = \frac{T_{\text{ser}}}{T_{\text{ser}} + T_{\text{par}}} \tag{2}$$

needs to be minimized.

Currently available tools such as Intel Threading Building Blocks [12, 2], Microsoft Task Parallel Library's `For` and `ForEach` loops and Microsoft PLINQ [14], Cilk [4] and its commerical successor Cilk++, OpenMP [7], and the Matlab Parallel Computing Toolbox [3] all provide capabilities for parallelizing loops within a subroutine, function, or method. However, the manner in which these approaches embed parallel programming within serial languages require that the sequential semantics of statements be preserved. In particular, consider two parallel loops such as

```
parallel_for(int i=0; i!=iend; i++) {
    ...
}
```

"Teflon," if used herein, means "fluoropolymer" or "PTFE." Such use was improvident, as Teflon® is a registered trademark of E.I. du Pont de Nemours and Co. in the US and elsewhere used as a brand of fluoropolymer additive, coating or resin and not a finished product.

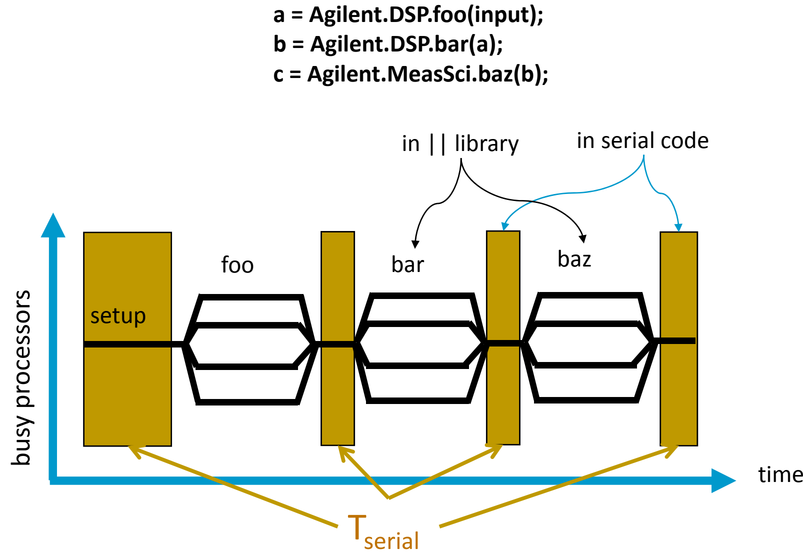


Figure 1: Figure illustrating unnecessary serialization when only intra-routine parallelization is used.

```
parallel_for(int j=0; j!=jend; j++) {
    ...
}
```

A *barrier*, a point where threads of execution wait for all to reach the same point in the code, is required between the two loops. The barrier is needed to guarantee the semantics, required by the serial language, that the entire first loop executes before any of the second loop executes. However, this constraint may or may not actually be required for the correctness of the program. An example of this would be if the second loop does not read any memory written by the first loop. For similar reasons, using current tools parallelism is extremely difficult or impossible to realize across function or method boundaries. Figure 1 illustrates the resulting behavior obtainable with these currently available tools. Parallel sections alternate with serial sections and the serial fraction is large. This problem of not finding enough parallelism has been experimentally verified to be a real one in the areas of measurement science and signal processing algorithms. Figure 2 shows a performance analysis of a routine that identifies the starting time of a data frame to sub-sample accuracy. The routine was parallelized usign Cilk++. It was timed using one through eight cores of an eight core (two dies of four cores each) Xeon class server. Twenty repititions were done for each measurement. The parameters of (1) were fit to the measurements using a linear least squares procedure. Although the extrapolated performance shown is probably not extremely accurate, it is nonetheless likely that little perforanace improvement can be expected beyond 16 cores.

In order to get anything like P times speedup for P processors as P increases to eight and beyond, it will be necessary to find and exploit parallelism in signal processing and measurement science algorithms that crosses function and loop boundaries. Since parallel speedup within individual algorithms, routines, or functions will not be sufficient to make full use of the numbers of cores that will soon be available, it is desirable to acheive the situation illustrated in Figure 3. In this figure, an operation on data can be run as soon as the

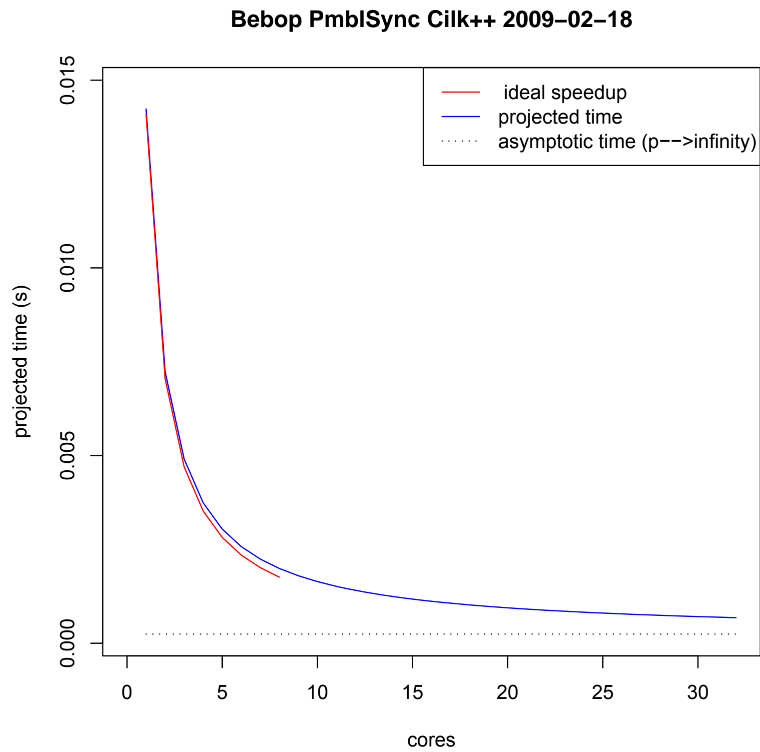


Figure 2: Speedup of a frame synchronization routine showing projected performance increase with number of cores.

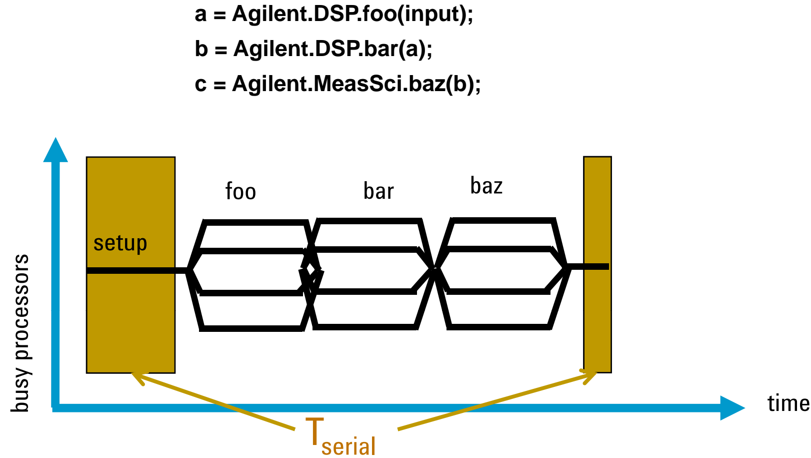


Figure 3: Parallelization of loops and routines with serialization only at points with true dependencies.

operations' inputs are available. A barrier is required if it is required by true dependencies, that is, that all future work depends on the results being waited for.

This report describes a language-independent paradigm, called TokenNets, intended to provide the sort of minimally constrained parallism illustrated in Figure 3. The report also describes a first implementation of TokenNets, called TokenPilot, and the first significant application written in TokenPilot, which does signal processing often required for jitter analysis of serial signals. While writing this application code, we identified several features for making programming easier that could be added to TokenNets. These features are discussed. The report concludes with a summary of the advantages we see in the TokenNets approach over other approaches that are available or discussed in the literature.

2 TokenNets

TokenNets is a language-independent scheme for writing DSP and measurment science algorithms and entire measurement applications. Rather than expressing how code should run in parallel, it provides a means for expressing the constraints on parallelism. Parallel execution may procede in any way not prohibited by the constraints. The programmer writes serial code–never parallel code–that is invoked by the TokenNets implementation. In other words, TokenNets separates the concerns of (1) specifying computations and (2) initiating and coordinating computations. The TokenNets implementation hides from the programmer the messy and error-prone details of parallel programming such as processes, threads, and locks or other mutual exclusion mechanisms.

We will first present the construction and behavior of a TokenNet without reference to what language the computations are written in. First, what a TokenNet is and how it functions is described fairly informally by describing two small example algorithms in TokenNets. Then, a more formal description will be given. In the following section we will decribe the first implementation of TokenNets, called TokenPilot, and dicuss some of the language- and system-dependent features of that implementation.

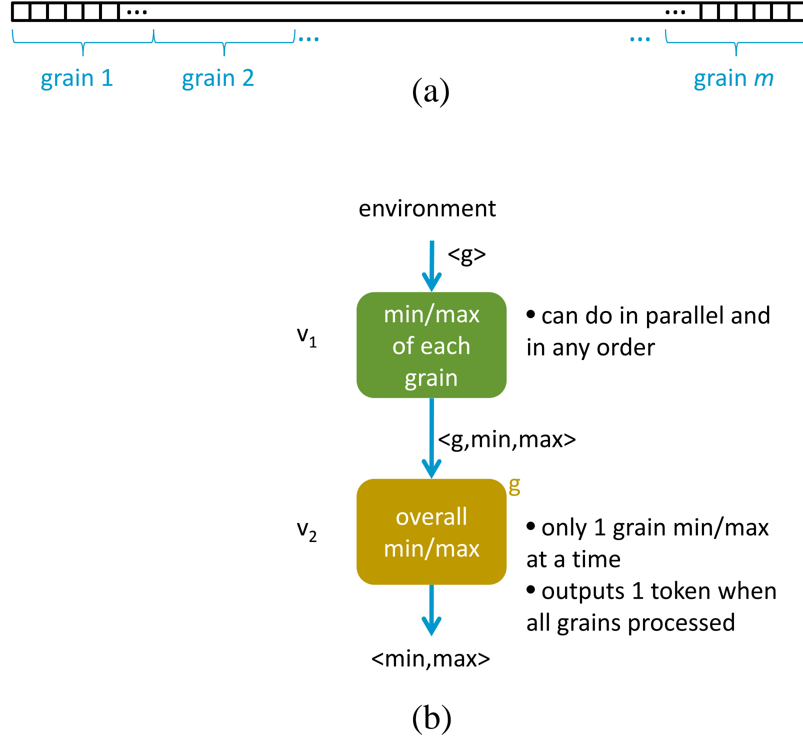


Figure 4: Example TokenNet that computes minimum and maximum of an array. (a) Array broken up into *grains* or contiguous segments. (b) The TokenNet.

2.1 Informal Description

We will describe TokenNets informally by discussing two small example parallel algorithms expressed using TokenNets. These examples serve to show how considerable parallelism can be obtained using simple TokenNets. (Neither represents the parallel algorithm with the highest parallel speedup for its task. But these other algorithms from the literature are rather more complicated. Presenting these more complicated algorithms would put attention on them rather than on TokenNets.)

Figure 4 shows a small TokenNet that is used to control the parallel computation of the minimum and maximum of a (presumably large) array. The array is divided into a number m of contiguous segmets called *grains*, illustrated in Figure 4(a). m typically will be somewhat more than the number of processors or cores P. Figure 4(b) showns the TokenNet. It consists of two nodes v_1 and v_2.

A program, referred to as the *environment*, that calls the minimum/maximum algorithm places *tokens* that contain indices of the grains, $g = 0, 1, \ldots, m-1$. This creation and placing of tokens is performed through an application program interface (API) provided by the TokenNets implementation.

For each token received, vertex v_1 computes the local minimum and maximum within the grain with the token's grain index g. Then, the function creates and returns a token containing the data g, the minimum, and the maximum. v_1 places no constraint on parallel execution. An unlimited number of computations of the grainwise minimum and maximum can take place in parallel and in any order.

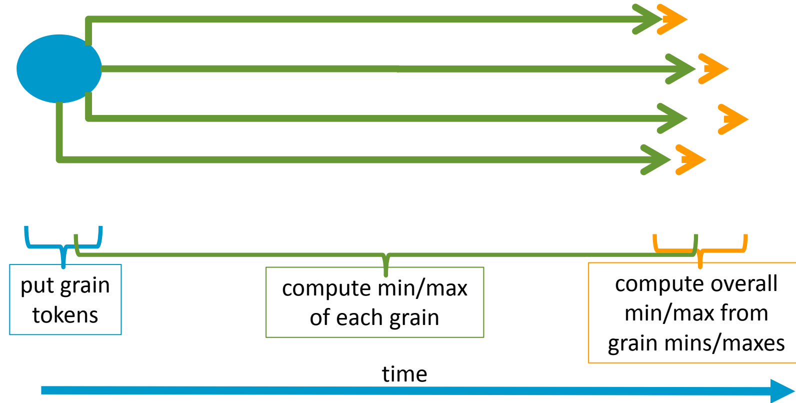


Figure 5: Hypothetical timeline of execution of the TokenNet in Figure 4 with number of cores $P = 4$ and number of grains $m = 4$.

Tokens output by v_1 are sent to v_2 along the edge that connects them. Vertex v_2 computes the minimum and maximum of the entire array from the grain-wise minima and maxima. v_2 has internal state: the minimum and maximum of the local minima and maxima from the tokens it has processed so far, and the number of tokens processed so far. When applied to a token (g, min, max), v_2's execute function f_{v_2} compares the running minimum and maximum to min and max and updates its internal state. When all m local minimum/maximum tokens have been processed, f_{v_2} outputs a token containing the overall minimum and maximum of the array. In order that the internal state of v_2 be updated consistently, only one token can be "running" at a time in v_2. The parallelism in v_2 must be constrained: execution must be limited to one token at a time.

Figure 5 shows a hypothetical timeline of the execution of the TokenNet in Figure 4. An execution of a local min-max finding routine begins as soon as the proper token is deposited into v_1. Those executions take much longer to run than the m executions of f_{v_2} because the number of array elements in each grain is much larger than 1. Executions of f_{v_2} can begin as soon as partial results–the local minima and maxima–are available.

Figure 6 shows another example TokenNet that does transition localization (sometimes called edge finding) in a one dimensional signal. The purpose of the net is to find the times (or sample indices) at which the signal transitions between its low and high states. Three voltages are given, called the low, medium, and high reference levels. Essentially, a transition is recorded when the voltage crosses the mid reference level, subject to criteria that make it so that glitches are not considered transitions. A complete description of what constitutes a transition is found in the relevant IEEE standard [1].

The TokenNet in Figure 6 localizes all the transitions in a signal stored as samples in a set of grains. The top vertex in the figure takes two tokens at a time:

- one that contains the grains (indexed by acquisition a, channel c, and grain g and
- the other that contains the reference levels that have been computed previously, presumably by another subgraph of the TokenNet.

Code associated with the vertex is executed when its edges have tokens with equal values of a, c, and g. The vertex's code localizes the transitions within one grain specified by its input

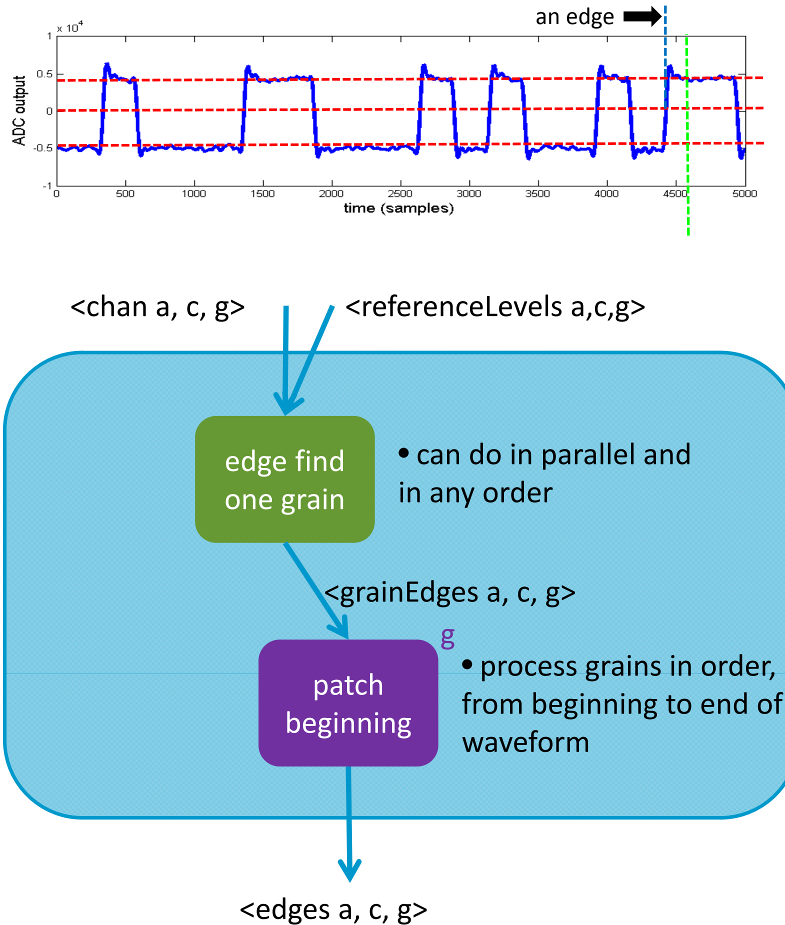


Figure 6: TokenNet that performs transition localation (edge finding) in a one-dimensional signal

tokens. It outputs a token containing the a, c, and g indices along with the localizations. The token also contains the logical state (e.g., low, high, or indeterminate) of the signal at the end of the grain. The vertex's parallelism is unconstrained, like vertex v_1 in the first example.

Now these localizations may not be complete. It is possible to miss the first edge at the beginning of each grain because it may not be possible to determine correctly the state of the signal at the boundary between grains. The purpose of the second vertex is to identify when that occurs by examining the end of each grain and the beginning of the succeeding grain, then incorporate any missed transitions into its output tokens. The second vertex receives the tokens produced by the first vertex. However, the second vertex needs to examine logical state of the signal at the end of the i-th grain in order to determine whether a transition is missed near the start of the $i+1$-st grain. So, the second vertex also constrained in its parallelism and in its execution order. That vertex can only be "running" one token at a time. Furthermore, it has to run the tokens in the order $g = 0, 1, 2, \ldots$.

We have seen that TokenNets are directed graphs on which tokens flow. Tokens both contain (or contain references to) read-only data and control execution. A vertex has an

associated function that runs when the right tokens are sent to it. Constraints on parallel execution of a vertex are sometimes necessary. A vertex can specify that it can run on at most one token at a time. A vertex can also specify that it must run on tokens in a certain order. We now turn to making these notions more precise.

2.2 Formal Description

A *TokenNet* is an directed acyclic graph (DAG) with $G = (V, E)$ with vertices V and edges $E \subset V \times V$. Let $\text{in}(v) = \{e \in E | (e, v) \in V\}$ be the in-edges of vertex v. There is a total order on the in-edges of v, so they can be indexed

$$\text{in}(v) = \{\text{in}(v)_1, \text{in}(v)_2, \ldots\}.$$

Let $\text{out}(v) = \{e \in E | (v, e) \in V\}$ be the out-edges of v. Let the source vertices of G be written $S(G) = \{v \in V | \text{in}(v) = \emptyset\}$.

Each e is associated with a token type T_e which is a finite cross product of types from the underlying language,

$$T_e = T_{e,1} \times T_{e,2} \times \ldots.$$

We abbreviate the input type $\text{in}(v)_i$ as $IT(v, i)$. The overall *token input type* of a vertex, written $IT(v)$, depends on whether or not the vertex is a source. When $v \notin S(G)$, then $IT(v) = \times_{0 \leq i < \text{length}(\text{in}(v))} IT(v, i)$. On the other hand, when $v \in S(G)$, then $IT(v)$ must be given.

Each edge $e = (v_1, v_2)$ is also associated with a *projection function* $\Pi_e : T_e \to U(v_2)$ where $U(v_2)$ is a tuple type. Note that $\Pi_{e_1} = \Pi_{e_2} = U(v)$ for $e_1, e_2 \in \text{in}(v)$.

The token types of all out-edges of an edge must all be the same. That is, $T_{e_1} = T_{e_2}$ for e_1, $e_2 \in out(v)$. This token type is written $OT(v)$ for short. When v is a sink, that is $\text{out}(v) = \emptyset$, without loss of generality $OT(v)$ is `void`.

Each vertex v is associated with an *execute function* f_v. This function has type $IT(v) \to OT(v)$. The functions f_v are written in the underlying language. Each vertex v has an associated *parallelism constraint* $C(v)$ *and a* matching function m_v of type $IT(v) \to$ boolean. $C(v)$ is an element of unconstrained, exclusive, sequential. When $C(v) =$ sequential, a function $s(v) : U(v) \to \mathbb{N}$, called the *sequence function* of v, must also be given.

A *token* of token type T is an object from the underlying language of type T.

The *environment* is an external body of code that initiates a TokenNets computation. The environment does so by creating and placing tokens t into sources $v \in S(G)$. Once a token t is placed onto a source v, the function $f_v(t)$ may begin to execute.

When a function $f_v(t)$ finishes executing, it returns a token t' of type $OT(v)$. Token t' is logically copied as necessary and a copy of t' is placed on each out-edge $\text{out}(v)$.

The determination of when f_v of non-source vertices may execute depends on $C(v)$. If $C(v) =$ unconstrained, then when there exists token t_1 on $\text{in}(v)_1$, t_2 on $\text{in}(v)_2, ldots$, such that the $\Pi_{\text{in}(v)_i}(ti)$ are all equal tuples, then the following happen:

- t_1 is removed from $\text{in}(v)_1$, t_2 is removed from $\text{in}(v)_2, ldots$, and
- $f_v(t_1, t_2, \ldots)$ begins execution.

The test of $m(v)$, the removal of tokens, and the initiation (but not completion) of $f_v(t_1, t_2, \ldots)$'s execution happen atomically.

If $C(v)$ = exclusive is similar except that only one invocation of $f_v(t_1, t_2, \ldots)$ may be executing at a time. In other words, the same three steps happen when $C(v)$ = exclusive as when $C(v)$ = unconstrained, but all three steps–including the running of $f_v(t_1, t_2, \ldots)$ to completion–happen atomically. Vertices v with $C(v)$ = exclusive are used to perform operations that cannot be done in parallel.

A vertex v with $C(v)$ = sequential is used to handle the cases where an operation must be done in a particular order. If $C(v)$ = sequential, then v behaves as if $C(v)$ = exclusive except that there is an additional constraint on the order of execution. For any value of $t' = \Pi_{(v',v)}(t)$, f_v cannot be executed on token arguments with $s(t') = i + 1$ before it is executed on token arguments with $s(t') = i$ and that execution has completed.

Whatever the value of $C(v)$, when $f_v(t_1, t_2, \ldots)$ is complete, its value (a token) is placed onto every out-edge of v.

Once the environment has placed a token onto a source of the TokenNet graph, execution of the TokenNet terminates until both

- No f_v is executing for any v, and
- No token is waiting on any edge.

As a practical matter, a TokenNets implementation must provide a means of notifying the environment once termination has occured.

3 TokenPilot: The Initial Implementation of TokenNets

The first implementation of TokenNets is TokenPilot. It is written as a set of C# 4.0 classes. TokenPilots uses the lightweight task creation and monitoring facilities provided with .Net 4.0's Task Parallel Library. These facilities are, in turn, implemented on top of a task-stealing scheduler. Such schedulers distribute work dynamically and greedily. Work-stealing schedulers approach optimal load balancing asymptotically [9]. Intuitively, near-optimal load balancing should lead to good parallel speedup. (However, intuition can be wrong in the presence of complex memory hierarchies, cache coherency hardware, and other complications. So this is a plausibility argument for, not a proof, of good performance.) One way of viewing the TokenNets approach is as a way of generating a large number of tasks that keep a near-optimal schedule well-supplied with tasks to schedule.

The TokenNet concepts of graph, vertex, and token are implemented as objects. In what follows, symbols written in italics like v refer to the TokenNet abstractions discussed in the previous subsection. Symbols written in typewriter font link `v` refer to C# objects in the TokenPilot implementation. A graph can be built up out of vertices and edges dynamically, tokens created and inserted into the graph, graph termination monitored, and the graph destroyed–all under program control. Thus, C# programs can write and execute TokenNets. This ability is useful in many instrumentation contexts because measurement algorithms to be run can change dynamically upon user input or commands delivered to an instrument via a computer network.

A token may be any object that provides an `Equals()` method and a `GetHashCode()` method that are consistent in the sense that if two tokens `t1` and `t2` have `t1.Equals(t2)` then `t2.Equals(t2)` and `t1.GetHashCode()==t2.GetHashCode()`. However, for convenience a class hierarchy of tokens is provided that are tuples with low arity or C# types or classes is provided.

A vertex v has a method `Execute()` which is overriden to provide the execute function f_v. v also has a method that provides the projection function $\Pi(e)$ for each of v's in-edges e. These projection methods default to identity functions but may be overridden. A base vertex class is provided with in-degree 1, 2, and so on. The base vertex classes have template arguments for the types of tuples expected on the in-edges. This guarantees that only graphs that are type-correct in the sense that only tokens of expected types can be placed on edges can be created. The base vertex classes are unconstrained. Subclasses provide exclusive and sequential vertices. The constructor of a vertex v takes the vertices at the other end of v's in-edges as arguments. So, only acylic graphs can be created.

The vertex base classes provide a method that subclasses can use to send a token out the out-edges of a vertex. In order to maximize available parallelism, that method immediately creates a task for each out-edge. Each out-edge task determines whether the placement of the token on that edge causes the `Execute()` method of the destination vertex to run or not. If the `Execute)` method is to run, it is run by that task.

Like in other programming paradigms, in TokenNets there are idioms that are often repeated. In TokenNets, rather than a textual form, such an idiom is a commonly-used subgraph. An example is the subgraph that corresponds to the map/reduce [6] parallel programming construct. Figure 4 is an example of a map/reduce subgraph in TokenNets. $v1$ provides the map computation, mapping grains to local minima and maxima. $v2$ is the reduce step, reducing the grain-wise minima to the overall minimum and the grain-wise maxima to the overall maximum. TokenPilot is designed to provide functions that build such common subgraphs. Currently, a function is provided that builds map/reduce subgraphs. The function takes the vertex providing the in-edge as an argument and the mapping function and the reduce function as function-valued arguments.

The TokenPilot program corresponding to the TokenNet in Figure 4consists of two class definitions–one for each vertex–and a main program. There is also a simple `struct` that contains minima and maxima:

```
struct MinMax
{
   public double min;
   public double max;
}
```

The class of the vertex that finds the grain-wise mimima and maxima (v_1) is declared

```
class GrainMinimizerVertex :
    Vertex<TokenIndices1, VectorGrain<double>, TokenIndices1, MinMax>
{
    ...
```

The subclass `Vertex<TokenIndices1, VectorGrain<double>, TokenIndices1, MinMax>` is the class of vertices that have one in-edge that can send tokens containing one data item of class `VectorGrain<double>` and have out-edges that can send tokens containing one data item of class `MinMax`. A `VectorGrain<double>` contains a reference to one grain from a vector (array) of doubles. The `Execute()` function uses .NET extension methods to compute the grain-wise minimum and maximum without any explicit loops. The method `this.SendToken()`, provided by the `Vertex` base class sends copies of the resulting token to any vertex with an in-edge originating in this vertex:

```
protected override
void Execute(Token<TokenIndices1, VectorGrain<double>> tok1)
{
    VectorGrain<double> grain = tok1.Data;
    MinMax mm;
    mm.min = grain.Min();
    mm.max = grain.Max();
    this.SendToken(
        new Token<TokenIndices1, MinMax>(tok1.Indices, mm));
}
```

The other vertex (v_2) needs to have the exclusive constraint on parallelism. So its class derives from a TokenPilot-provided class that ensures that only one invocation of the `Execute()` method can be executing at a time.

```
class OverallMinMax :
    MutuallyExclusiveVertex<TokenIndices1, MinMax, TokenIndices0, MinMax>
{
    ...
```

The `Execute()` method tracks the number of tokens seen and outputs a token when the all the necessary tokens have been received.

```
protected override
void Execute(Token<TokenIndices1,MinMax> tok1)
{
    MinMax tokenMinMax = tok1.Data;
    this.mm.min = Math.Min(this.mm.min, tokenMinMax.min);
    this.mm.max = Math.Max(this.mm.max, tokenMinMax.max);
    grainsSeen++;
    if (grainsSeen == numInputTokens)
    {
        Console.Write("Overall Min=");  Console.Write(this.mm.min);
        Console.Write(" Overall Max=");  Console.WriteLine(this.mm.max);
        this.SendToken(
            new Token<TokenIndices0, MinMax>(new TokenIndices0(), mm));
    }
}
```

The main program creates and fills a vector with data, builds a graph out of the two vertices, inserts tokens into the graph, and waits for the computation to complete.

```
static void Main()
{
    // Create vector containing data to min & max
    int size = 1<<16;
    int grainSize = size>>4;
    GrainedVector<double> v = new GrainedVector<double>(size, grainSize,
                                                  (int i) => Math.Sin(i));

    // Create the graph
    Graph graph = new Graph();
```

```
        GrainSource<double> source = new GrainSource<double>(graph);
        GrainMinimizerVertex v1 = new GrainMinimizerVertex(source, graph);
        OverallMinMax v2 = new OverallMinMax(v1, v.Grains.Count, graph);

        // Run the graph and wait for it to terminate
        source.PutGrains(v);
        graph.WaitAll();
    }
```

The output of the program is shown in Figure 7.

4 Example

4.1 Introduction

The example is based on the IEEE-181-2003 standard [1] for the description of waveforms. We will analyze a pulse waveform and determine the time interval error. For the sake of the example, some simplifications have been made to the actual algorithm. For a given waveform, the time interval error can be determined as illustrated in Figure 8.

The first step in the process will determine the *state levels*, or *reference levels* using a histogram, for simplicity a bimodal amplitude distribution is assumed. To generate the histogram, the amplitude range must be divided in M unique amplitude intervals. The amplitude interval is called the histogram bin width, Δy. The amplitude range, y_R, is determined by the minimum and maximum amplitude values: $y_R = y_{max} - y_{min}$. For equal sized bins, Δy is found by dividing y_R by M:

$$\Delta y = \frac{y_R}{M} = \frac{y_{max} - y_{min}}{M}.$$

Next the *state level boundaries* are determined based on the histogram bin counts. The histogram is split into the upper and lower histogram. The *low state level* is given by the mode of the lower histogram, the *high state level* is given by the mode of the upper histogram.

Once the *low* or and *high state* levels have been determined, the *reference levels* can be calculated directly from those parameters.

$$A = \text{level}(s_2) - \text{level}(s_1)$$

$$y_{10\%} = \text{level}(s_1) + \frac{|A|}{100}10\%$$

$$y_{50\%} = \text{level}(s_1) + \frac{|A|}{100}50\%$$

$$y_{90\%} = \text{level}(s_1) + \frac{|A|}{100}90\%$$

where

A is the amplitude of the waveform

level(s_1) is the *low state level*

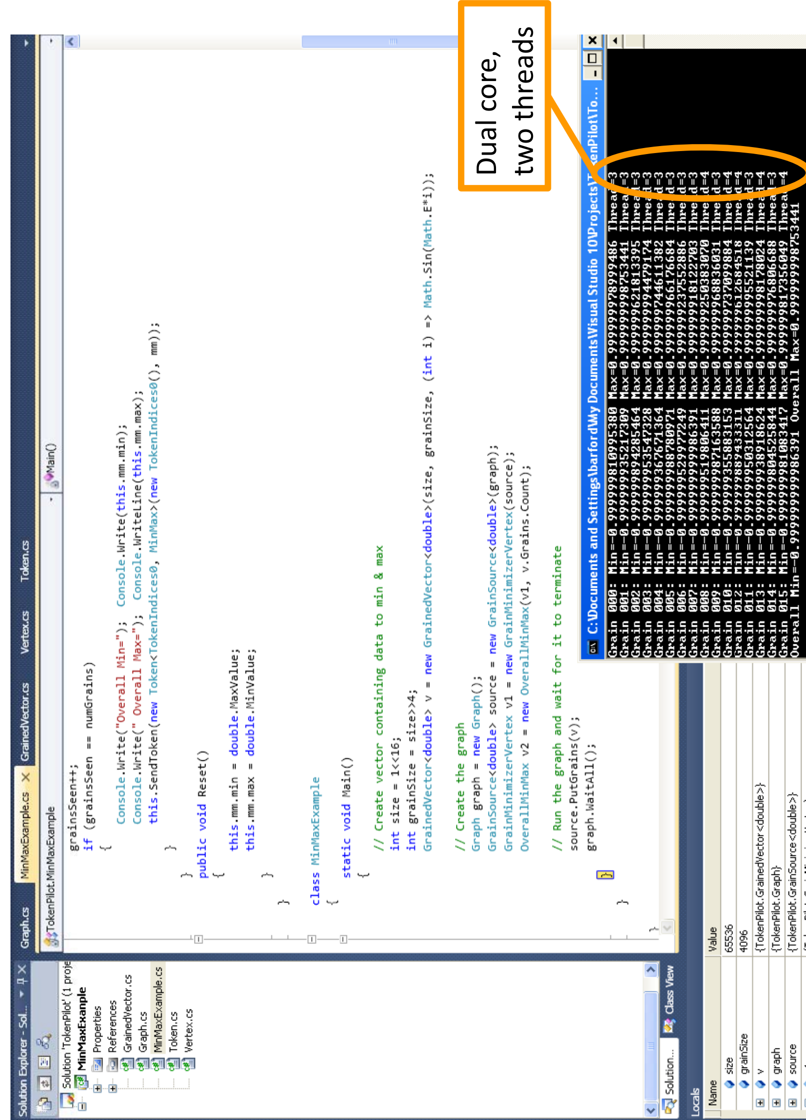


Figure 7: Screenshot of code and execution trace of TokenPilot implementation of the minimum/maximum example of Figure 4

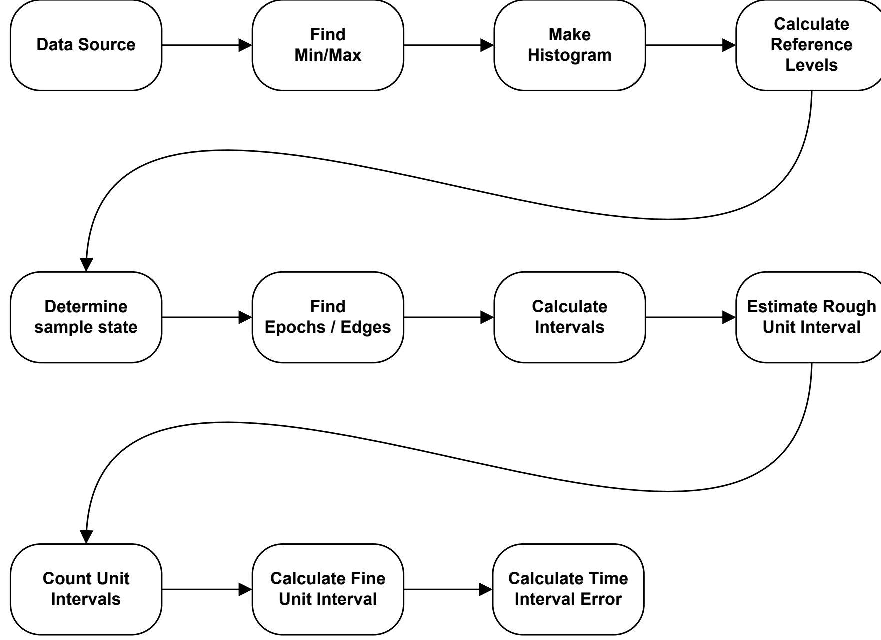


Figure 8: Jitter Analysis Block Diagram

level(s_2) is the *high state level*

$y_{x\%}$ is the value of the reference level.

Next the waveform is scanned for for subepochs, edges and transitions, this is called the parsing process . The first step in the parsing process assigns *state levels* to each amplitude value by comparing the waveform value to the upper and lower *boundaries* of the states defined by the reference levels. If the waveform value is contained withing the boundaries of a state, the waveform value is assigned a state level. If a waveform value is not within the boundaries of any state the value is a assigned the *undefined state.*

$$\text{state}(t_n) = \begin{cases} \text{lowstate}, & \text{for } y_{min} \leq t_i \leq y_{10\%} \\ \text{undefinedstate}, & \text{for } y_{10\%} < t_i < y_{90\%} \\ \text{highstate}, & \text{for } y_{90\%} \leq t_i \leq y_{max} \end{cases}$$

The waveform will be decomposed into subepochs based on the assigned state levels. A subepoch is defined by the state value and the start time. To filter out false subepoch states the length of each subepoch defining an assigned state is compared against a predefined minimum *state* duration. If the requirement is not met, the subepoch is assigned the undefined state and merged together with adjacent subepochs with an undefined state level.

$$\forall t \in [t_x, t_y] : \text{state}(y_t) = S \ , \ t_y - t_x > d_{min}$$

After the subepochs have been merged, we will look for *state transitions.* A *transition* is defined by either two adjacent subepochs with different *state values* or alternatively by three adjacent subepochs where the state level of the second subepoch is undefined while the state level of the first subepoch is different from the state level of the third subepoch. Each transition or edge is defined by it's interpolated $y_{50\%}$ crossing time. The edge times

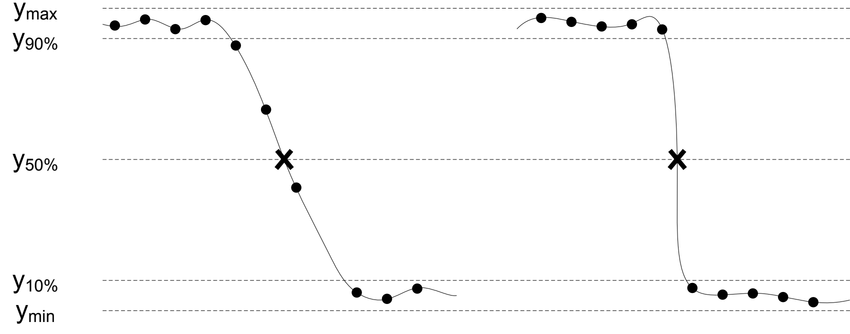


Figure 9: Valid transitions between state levels

are used to determine the *unit time interval* of the *composite waveform*. This is a two step process, first a rough estimate of the unit time interval is determined. One way to define the rough estimate is by looking for the minimum time interval between two edges. A probably better method would be to use the 25% percentile of the historgram of the time intevals, this will filter out possible outliers. For sake of simplicity we will use the first method in this example.

Using the rough estimation of the unit time interval we can determine the total number of unit time intervals between the first and last edge in the composite waveform by summing the quantified number of estimated unit intervals between all pairs of two adjacent edges. Now the total number of unit time intervals is known we can calculate the precise unit time interval. Assuming the first and last edge in the composite waveform have no time error, the unit time interval is the time between the first and last edge divided by the number of unit time intervals.

$$\Delta t_i = t(E_{i+1}) - t(E_i)$$
$$\theta = min(\Delta t_i)$$
$$N = \sum_i \left\lfloor \frac{\Delta t_i}{\theta} + 0.5 \right\rfloor$$
$$\Theta = \frac{t(E_n) - t(E_0)}{N}$$

where

$t(E_i)$ is the time of the i-th edge

Δt is the time interval between two adjacent edges in the composite waveform

θ the rough estimate of the unit time interval

N is total number of unit time intervals

Θ is the unit time interval.

Finally, the time error for each edge can be determined by comparing its measured time against the expected time which is an integer multiple of the unit time interval calculated in the previous step.

```
class GrainSource<T> : Source<TokenIndices2, GrainedVector<T>>
   {
       private int _FrameId;

       public GrainSource(
             Graph graph,VertexDescriptionAttribute description = null)
           : base(graph, description)
       {
           _FrameId = 0;
       }

       public void PutGrains(GrainedVector<T> input, int Channel)
       {
           this.SendToken(new Token<TokenIndices2, GrainedVector<T>>
               (new TokenIndices2(_FrameId, Channel),input));
           _FrameId++;
       }
   }
```

Figure 10: *GrainSource* definition

4.2 Implementation

The example will be implemented using the TokenPilot library. The basic concept of the library is to break down the algorithm into smaller tasks, which can then be executed in parallel. The implementation examples below will highlight different implementation options and challenges. The code below is by no means complete. For the complete source code please contact the authors.

As a first step the input data, referenced to as the input *vector*, will be split into smaller chunks or *grains*. The number of grains is chosen based on the available cpu cores and the nature of the algorithm. For this case the number of grains is set to twice the number of available cores. In graphical representation, the indices will be marked by $< x, y, z >$, the data is marked as $\{d\}$.

A class `GrainedVector` manages a vector of data and provides for breaking it up and acessing it as a set of grains. A data source type will be implemented that will feed tokens containing data of type `GrainedVector` into the graph. For the sake of the example, first a data source vertex is created that sends out tokens of type `GrainedVector` (Figure 4.2). Next a vertex is added that will split the `GrainedVector`s into their grains. In a real world situation, it would be preferable to combine both operations in a single vertex to reduce overhead. This example illustrates how custom types are used to define a vertex. Later on we'll illustrate how a vertex can be created without the need of an extra custom class.

At the highest level of granularity, the `GrainedVector`, token indices mark the identity of the vector (say as a sequence number indexing data acquisitions) and the identity of the measurement channel the vector came from. This channel ID illustrates how extra identifiers can be added to a token.

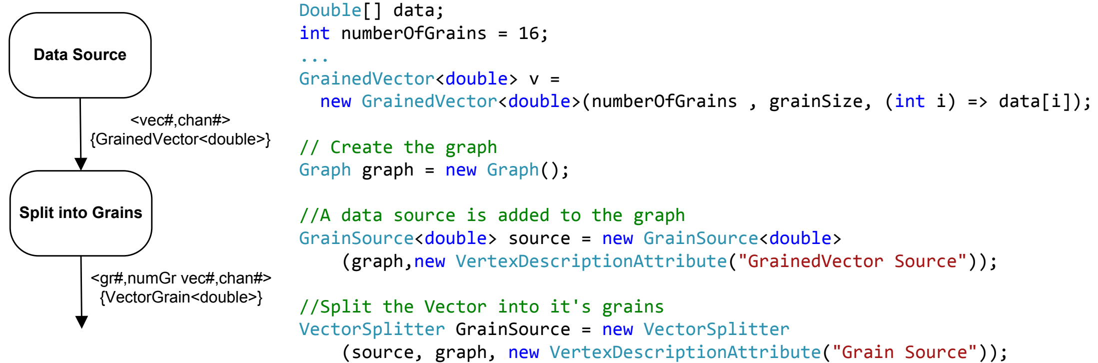
After the vector is split into grains additional indices will mark the identity of the grain and communicate the total number of grains in the vector.   using the `GrainsSource` and `VectorSplitter` classes, an initial graph is created. As illustrated in Figure 4.2, a `GrainedVector` is initialized from an array of numbers, which are samples. Notice the indexing of the array using a lambda function, a feature that provides a flexible means of defining the data in a `GrainedVector`.

```
class VectorSplitter : Vertex<TokenIndices2,GrainedVector<double>,
                              TokenIndices4,VectorGrain<double>>
{
   public VectorSplitter(
       TokenSender<TokenIndices2, GrainedVector<double>> inVertex,
       Graph g, VertexDescriptionAttribute d = null)
       : base(inVertex, g, d) { }

   protected override void Execute(Token<TokenIndices2,
                                        GrainedVector<double>> tok1)
   {
       GrainedVector<double> data = tok1.Data;
       TokenIndices2 ind = tok1.Indices;

       int numGrains = data.Grains.Count;
       for (int i = 0; i != numGrains; i++)
       {
          this.SendToken(new Token<TokenIndices4, VectorGrain<double>>
                 (new TokenIndices4(i, numGrains ,ind[0], ind[1]),
                    data.Grains[i]
                 ));
       }
   }
}
```

Figure 11: `VectorSplitter` definition




```
Double[] data;
int numberOfGrains = 16;
...
GrainedVector<double> v =
  new GrainedVector<double>(numberOfGrains , grainSize, (int i) => data[i]);

// Create the graph
Graph graph = new Graph();

//A data source is added to the graph
GrainSource<double> source = new GrainSource<double>
    (graph,new VertexDescriptionAttribute("GrainedVector Source"));

//Split the Vector into it's grains
VectorSplitter GrainSource = new VectorSplitter
    (source, graph, new VertexDescriptionAttribute("Grain Source"));
```

Figure 12: Initial Graph containing a data source

4.2.1 Reference Level Calculation

To calculate the reference levels the overal minimum and maximum amplitude values have to be determined. This is done in a way similar to that described in section 2.1.

The implementation will make use of the `MapToken` and `ReduceToken` classes that are provided in the TokenPilot library. These types allow a user to create functional vertices without the need to define a custom class by making use of lambda functions as parameters. To improve readability of the code, this method is only used for simple vertices.

The `MapToken` type will apply the provided lambda function to every token presented on its in-edge. The result of this specific `MapToken` vertex is the `MinMax` of the input grain, where `MinMax` is a type containing a minimum and maximum value. Since `MinMax` has the addition operator defined two values can be easily combined by adding them. To calculate the `MinMax` of the enumerable `VectorGrain` the C# `Aggregate` function is used.

The `ReduceToken` class combines a limited number of different tokens based on a common feature. In this case `ReduceToken` is used to combine the partial `MinMax` results generated by the previous step. To define the common feature we rely on the indices, only tokens with matching last two indexes will be combined. In this example the last two indexes represent the vector ID and channel ID. The second index, the grain count, is used to limit the number of tokens that are combined. The output of this vertex will be a token containing the `MinMax` value of an input vector, identified by the vector ID and the channel ID. Internally the `ReduceToken` vertex will define vertex local storage to accumulate the different input tokens. Once all the required tokens are available an output token is generated after which the input tokens are discarded.

Finding the histogram and reference levels are handled similar to how minima and maxima are computed. To generate a histogram we will first define the edges and bin width of the histogram, these values are based on the `MinMax` of the vector and again combined in a data type called `HistogramDescription`. A single token containing a `HistogramDescription` will be generated per `Vector`. To generate the histogram itself a `Vertex` will calculate the partial histograms for each grain which will then be combined into the overall histogram for the vector, similar to how minimum/maximum is computed. However, to calculate a partial histogram we'll need two inputs: the grain containing the samples, and the `HistogramDescription` containing the information on how to create each histogram. Since for each vector only one `HistogramDescription` is available while there will be a number of grains per vector, we repeat the `HistogramDescription`. Every grain token will now have a `HistogramDescription` token with matching indices. Based on the histogram, a single token per vector will be generated containing a data type defining the reference levels for that specific vector.

4.2.2 Waveform Parsing

The waveform parsing step identifies edges in a vector based on the reference levels. Again, we will start by parsing each grain individually, similar to the way we constructed the histogram. For each grain we'll start by defining *epochs*, each of which may contain one edge or glitch. Using the epoch definitions we'll look for edges. Every set of three epochs can have one out of two edge locations, see Figure 4.2.2.

However, the boundary epochs cannot be defined as they might be the continuation of an epoch defined in the previous or next grain. In the worst case, after combining the

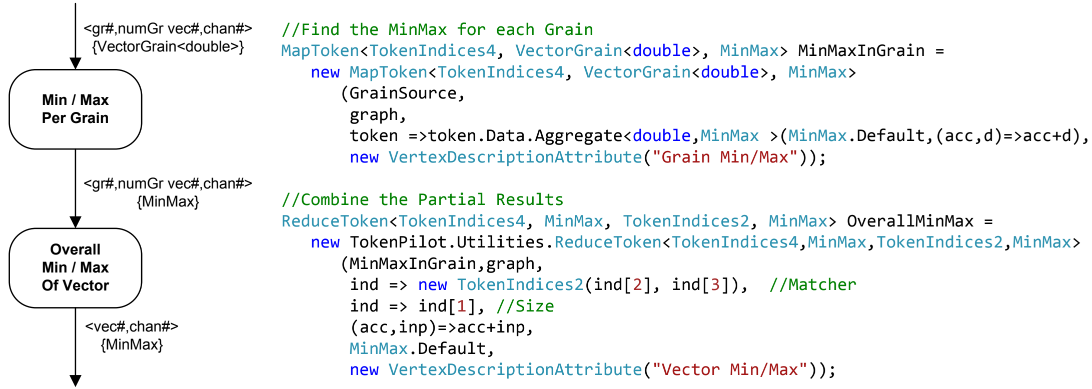


Figure 13: Find Minimum,Maximum

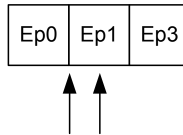


Figure 14: Detected Egde locations in a set of three Epochs

grains an edge might even be thrown away because it doesn't don't meet the minimum length criterion. Because of this the first and last four possible edge locations cannot be detected by making use of a single grain, see Figure 4.2.2.

For each grain, a token will be generated containing the detected edges for that grain plus the information on the boundary epochs to allow overlap processing in the next vertex. Combining the grains to detect the remaining edges imposes a new issue: not only we'll have to combine the grains of a specific vector, we'll also have in process them in an ordered. way. `TokenPilot` provides a helper class to facilitate this: the `OrderedAggregateVertex` will combine the data in a set of matching tokens into a token containing an ordered array of the data.

Next we implement a vertex that combines the elements of the ordered array. Rather than creating a single output token containing all the edges in the vector we will complete the individual grains with the missing edges. In addition to that, in order to make processing in the next stages more easy, we will repeat the last edge of a specific grain in as the first edge in the next grain. The generated tokens will contain an ordered array of edge times.

4.2.3 Unit Time Interval Calculation

Finally we can start calculating the unit time interval. The vertices that do so are create using the techniques explained above. First we will calculate the time between edges in each grain using `MapToken`. Next we look for the minimum interval using a combination of `MapToken` and `ReduceToken`, similar to the example above. The output of this `ReduceToken` vertex is a rough estimate of the unit interval. Next a vertrx is implemented to determine the number of intervals in each grain, based on the rough estimate of the unit time interval. As usual, a second `ReduceToken` reduces the partial results to compute total number of unit

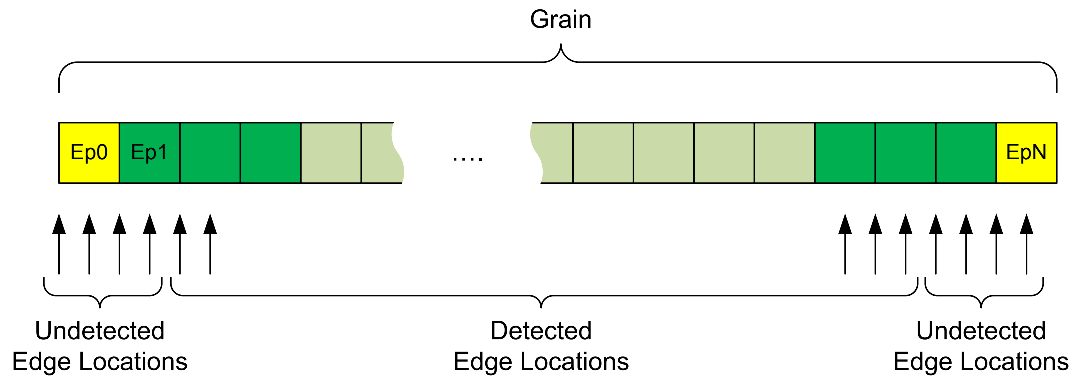


Figure 15: Detected Edge locations in a Grain

```
//Aggregate all partial ParseResults  --> output is ParseResult[]
TokenPilot.Utilities.OrderedAggregateVertex
      <TokenIndices4, ParseWaveForm.ParseResult, TokenIndices2>
   parseResultAggregate = new
    TokenPilot.Utilities.OrderedAggregateVertex
      <TokenIndices4, ParseWaveForm.ParseResult, TokenIndices2>(
          grainEpochParser,
          (TokenIndices4 ind)=> new TokenIndices2(ind[2], ind[3]),//Match
          (TokenIndices4 ind)=> ind[0], //Order
          (TokenIndices4 ind)=> ind[1], //Size
          graph,
          new VertexDescriptionAttribute("Combine Grain Epochs")
       );
```

Figure 16: Using the OrderedAggregateVertex to combine Tokens

```
//Find the first and last edge in the vector
ReduceToken<TokenIndices4, double[], TokenIndices2, MinMax>firstLastEdge=
  new ReduceToken<TokenIndices4, double[], TokenIndices2, MinMax>
  (
     GrainStitcher, graph,
     ind => new TokenIndices2(ind[2], ind[3]),
     ind => ind[1],
     (acc, d) => acc + new MinMax() { min = d[0], max = d[d.Length - 1]},
     MinMax.Default,
     new VertexDescriptionAttribute("First/Last Edgetime")
  );
```

Figure 17: Find the first and last edge using a *MinMax* data type

```
//Based on the first and last edge, calculate the UI
JoinTokens<TokenIndices2,int,MinMax,double[]> calculatePreciseUI =
     new JoinTokens<TokenIndices2,int,MinMax,double[]>
       (
         sumGrainUIs,firstLastEdge,
         graph,
         (T1,T2)=> new double[] {T2.Data.Range/
                                  T1.Data,T2.Data.min},
         new VertexDescriptionAttribute("Precise Unit Interval")
       );
```

Figure 18: Calculate the precise *Unit Interval* using the *JoinTokes* type

intervals within the vector

At the same time, we look for the first and last edge in the vector. We do this using a `ReduceToken` vertex by making use of the addition operation of the `MinMax` type (Figure 4.2.3). The resulting token contains a `MinMax` with the time of the first edge as the minimum, and the time of the last edge as the maximum.

Combining the number of unit intervals and the time between the first and last edge provides a precise unit interval under the assumption that the clock frequency is constant. TokenPilot provides a helper class to facilitate simple operations based on two `Tokens`: `JoinToken` class, see Figure 4.2.3. The complete TokenNet is illustrated in Figures 4.2.3 and 4.2.3.

4.3 Performance

The example was benchmarked both for scalability and raw performance. To benchmark the scalability, speed tests have been run using a variable degree of parallelism. To allow this, a custom scheduler has been implemented which exposes the number used cores through a programmable number of worker threads. Forty repitions were performed for each number of worker threads. A box plot showing the number of worker threads versus verious percentile levels of speedup T_1/T_P is shown in Figure 21. Approximately $7x$ speedup was obtained on eight cores using nine worker threads. The raw performance was benchmarked by comparing the `TokenPilot` implementation against a traditional serial implementation. This experiment exposes the overhead in the current implementation. On the eight core test machine we measured a speedup of about $4x$ when compared to the serial implementation.

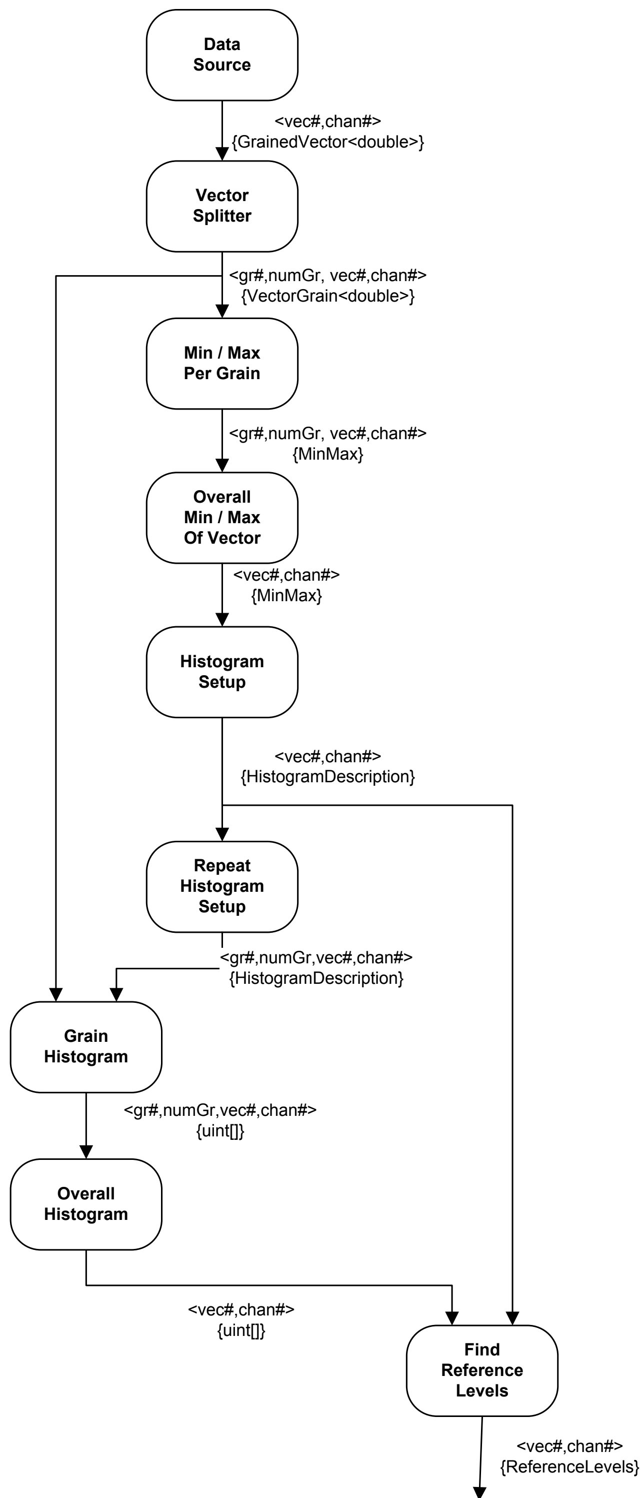


Figure 19: First half of the complete graph

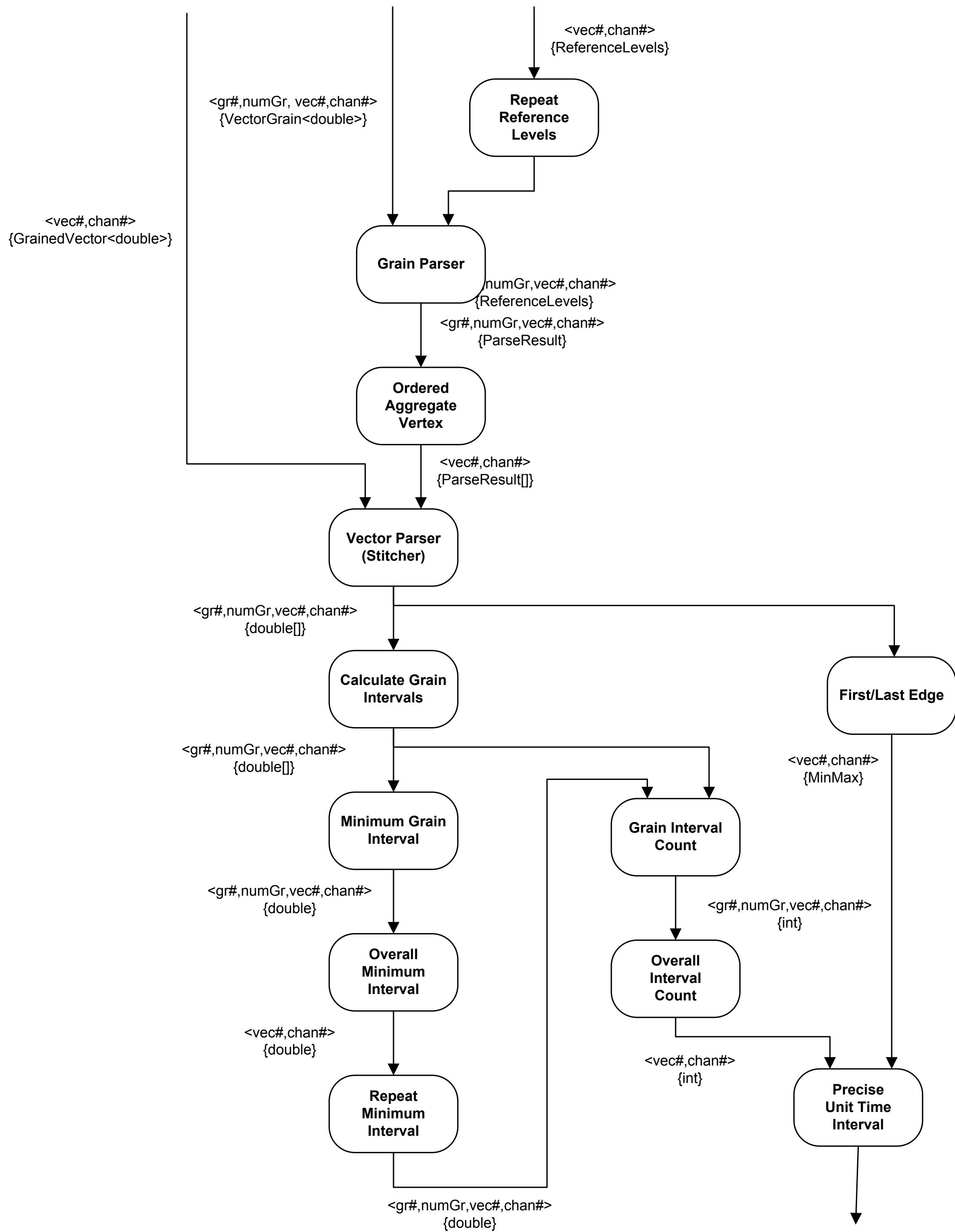


Figure 20: Second half of the complete graph

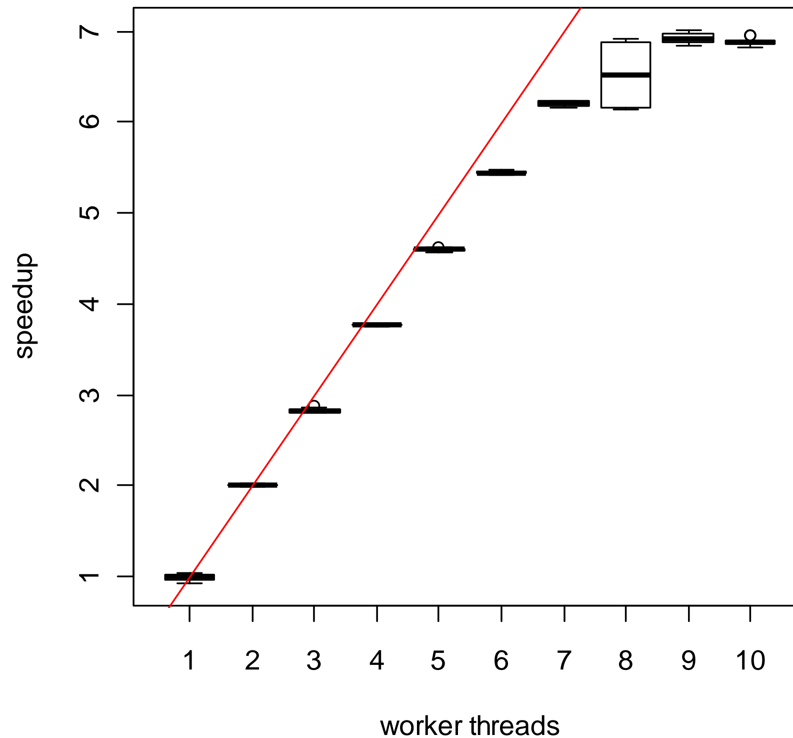


Figure 21: Speedup as a function of number of worker threads. Red line indicates ideal speedup.

5 Related Work

The problem of safely permitting as much work as possible to be carried out using a shared resource is an old one. Perhaps its earliest occurrence was in railroading. By the 1840's, railroads had developed to the point where a railroad would typically have multiple trains that needed to transit a particular route, often in opposite directions. However, few railroads had multiple parallel tracks. The practice of separating trains simply by assigning each train a static schedule were, literally, disastrous. Many railroads developed schemes to ensure that only one train occupied a segment of track at a time by requiring that a physical object called a token be carried in the cab of the locomotive. Originally, there was only one token for one segment of track. Due to the need to send the token back to the starting point if more than one train in a row needed to go the same direction and other complications, often the token was a person–a railroad employee. This person would wear distinctive clothing, often an arm band, and a title such as Token Man, Pilot Man, or (more rarely) the Token Pilot. By the latter nineteenth century semi-automated schemes using electrically-signaled devices dispensing wooden or metal tokens largely replaced the human tokens. The term "token" and the name "TokenPilot" pay homage to this history.

Petri [11] introduced the idea of modeling conurrency by tokens that move around a directed graph. Petri nets are extremely well-known and have given rise to an enormous literature and a number of related schemes, so they will not be discussed further here.

Gelertner's Linda [8] was the most notable *coordination language* for concurrent and distributed systems. When using a coordination language, computation is done in a traditional, serial language. Issues of concurrency and communications are handled as separate code written in the coordination language. In Linda, there is a logically shared *tuple space* that is available to all threads of execution. The tuple space is a bag containing data items, each of which consists of ordered data fields. Communication and coordination is performed by operations such as (1) writing a tuple to the tuple space and (2) pattern matching, reading, and removing a tuple from the tuple space.

Intel Concurrent Collections [5] (abbreviated CnC) is the closest direct ancestor of TokenNets. Like in TokenNets, code is written for CnC by writing functions in an implementation language. Each function is associated with a vertex in a graph. CnC uses *tags* to manage data flow in a manner analogous to tokens in TokenNets. *Step tags* are are special token-like objects that also manage control flow. A vertex's function can be executed when both the required tags and a matching step tag are present in the vertex. CnC graphs can have cycles, so arbitrary control structures can be created using CnC graphs. CnC graphs are specified statically, at compile time, using a textual language. A program reads the textual description of the graph then generates C++ code that implements the parallel control and data flow functionality described by the graph. The CnC programmer writes C++ code containing the functions that the generated code calls to implement the vertices' functionality. CnC makes use of TBB's parallel data structures and work-stealing scheduler but not TBB's parallel control structures.

6 Future Directions

6.1 Structured Tokens

In the current TokenNet implementation the token indices are represented by a flat tuple of integers. However, it could be argued that this approach is too simplistic, and forces the user to write code in the application that would be better supported by the library. For example, consider the situation where we are processing a data stream representing a sampled signal. It would be natural to use one index to hold the sample number. However, if the signal is represented by a large array of values we might split the array into a number of smaller grains to increase the potential for parallel processing. In such as case we would use a second index to record the grain number.

Now consider the situation where a graph vertex V needs to process all the grains within a sample before emitting a result token. The number of grains may vary from sample to sample, particularly if the sample size itself can vary. So the first problem is to know how many grains make up the ith sample. The only way the vertex responsible for splitting up the sample can communicate this information to vertex V is via the tokens themselves; there should be no other form of communication between vertices in a TokenNet graph. A token has two components, an index tuple and a data value. We must therefore either add the number of grains as an additional index to the index tuple, or create a composite data value containing this information along with the "real" data value. Neither of these alternatives is particularly attractive.

For a given sample index the vertex V is presumably maintaining some application-specific state that is updated as each grain for this sample is processed. Once the last grain for the sample has been analyzed the vertex will typically output a token whose data component is built from this sample state, and the storage can then be relinquished. Ideally the book-keeping involved in maintaining sample-specific storage should be delegated to the TokenNet infrastructure, not reimplemented in each new application.

Many operating systems support the notion of thread-local storage, areas of memory that are thread-specific and are automatically reclaimed when the thread expires. The concept frees the developer from maintaining explicit thread-to-memory maps and the recycling of this memory. Ideally a TokenNet developer should be able to rely on a similar level of abstraction, i.e. a notion of token(set)-local storage. What is the lifetime of such storage. In our previous example the lifetime was from when vertex V first saw a grain for a particular sample until all the grains for the sample had been processed by V. More generally, it becomes clear that modeling token indices by a flat tuple is too simplistic; the indices should have some form of hierarchical structure.

Currently a token index tuple such as $(i_1, i_2, \ldots, i_n)$ can be viewed as an element of the index set $I_1 \times I_2 \times \ldots \times I_n$. But there is another way of constructing such indices. The concept is perhaps easiest to describe using an example. Let us generalize our previous example to also include the concept of channel. For example, an N-channel oscilloscope would produce multiple streams of samples, one for each channel, and these could all be interleaved on a single graph edge. In the current version of the TokenNet API we might then attach the token index $(c_0, s_0, g\#, i)$ to the ith grain for sample s_0 on channel c_0, where $g\#$ is the total number of grains in this sample.

Let us now replace the use of a Cartesian product by a mutually-recursive definition of token index tuples and token index sets. Each token index set S will have a defined cardinality, possibly infinite, denoted by $|S|$. Given an index i, where $0 \leq i \leq |S|$, we can construct a new token index (S, i). A token index is therefore formed from a token index set and a number. To complete the mutually recursive definition we also allow a token index set to be constructed from a token index by specifying a cardinality for the new set. The term $\langle (S, i), N \rangle$ therefore represents a new token index set, with cardinality N. To bootstrap the definitions we also define the distinguished element $\perp$ as a token index. Returning to our original example, we can now form the channel index set as $\langle \perp, N \rangle$, where N is the number of channels. A particular channel index, for example for channel 1, would then be represented by $(\langle \perp, N \rangle, 1)$. The index could be used to build a sample index set. In this case the number of samples is potentially unbounded, and so the set would be represented by $\langle (\langle \perp, N \rangle, 1), \infty \rangle$. A token index for sample j would then be represented by $(\langle (\langle \perp, N \rangle, 1), \infty \rangle, j)$. If this sample is broken down into $g\#$ grains then the corresponding token set would then have $g\#$ as its cardinality, and so on. Of course writing tokens in this fashion is rather tedious, and so where the cardinalities of the sets can be deduced from the context we can use the original notation, simply writing the indices as an unstructured tuple.

The primary advantage of the structured approach is that it makes the cardinalities of the underlying sets explicitly available to us, and other library code. We no longer have to view the grain count, $g\#$, as an additional index. We can simply determine this value by retrieving the set that generated the token index, and then querying for the cardinality of this set. The lifetime of token-local storage is also easy to define. In our previous example

we wanted vertex V to maintain some local state for each token index set of the form $\langle(\langle(\langle\perp, N\rangle, c_i), \infty\rangle, s_j), g\#\rangle$. When a token is received whose token index is generated from a set that doesn't match any of the sets currently being processed a new storage element is created. The vertex then expects to see $g\#$ tokens built from this set, including the current one; the set, and its associated storage, can then be reclaimed. The structure imposed on the token indices allows us to automate this process. The application programmer would simply need to indicate when token-local storage is required, and the type of the data required to be stored.

The developments described in this section are simply a proposal at this stage, and are not supported by the current version of the API. However, if the TokenNet approach was to be developed further then it would be worth exploring in more detail the merits of such a proposal, and its ramifications on the implementation of the library. From the perspective of the debugger the changes would be relatively minor. At present the maintenance of any token-local storage would be hidden inside the application code, and therefore invisible to the graph viewer. If this state was directly supported by the TokenNet library itself then it would be natural to expose the mapping in the graph view. A developer would be able to inspect the current set of active token sets in a vertex, drilling down into the state associated with each set when required.

6.2 Typed Ports

The current implementation of the TokenNet API defines different vertex base classes based on the number of incoming edges. Maintaining this code is tedious as all the classes follow the same basic pattern. Furthermore, it imposes an artificial upper limit on the number of edges supported by the library; it's unlikely that a developer wanting ten incoming edges to a vertex will find the library defines a vertex class supporting this number of edges for example. Each edge is associated with its own token type. But tokens, as we have seen earlier, have two components, a token index tuple, and a data type. The library uses generic types to abstract away from specific types for these components and so each edge is associated with two type parameters, one for the tuple index and one for the data. Whilst such an approach works well for vertices with a single incoming edge, as the number of edges rises the syntactic clutter arising from all these type arguments starts to overwhelm the code.

In the context of a graph viewer there is a second problem. The API orders the incoming edges to a vertex, whereas a typical graph layout algorithm assumes the edges are unordered. Forcing a layout algorithm to respect the ordering, for example from left-to-right, or top-to-bottom depending on the graph orientation, would severely constrain the degrees of freedom of the layout, potentially leading to greatly increased layout times and poor layouts. Unconstraining the layout algorithm and then annotating the edges with their index might also produce confusing results. The current graph viewer assumes that the context provides enough information to allow the user to deduce edge indexes from the graph view. For example, if the source of the first edge is a vertex of type T_1 and the source of the second edge is a vertex of type T_2 then we rely on the user being able to distinguish vertices of these types in the graph view to be able to determine which edge in the view corresponds to the first edge in the API.

A TokenNet graph exchanges data over typed edges. It can be argued that the focus of the current API is at too high a level, and we should really focus on typed input and

output ports, and the edges that connect them. In this view a vertex contains a set of input ports and output ports. The "work" function associated with the vertex retrieves tokens from the input ports, processes them, and then emits one or more tokens on the output ports. Given such a starting point it is then possible to define the existing API on top of this model if needed, but we can also package up the functionality in other ways as well. Unfortunately there's a complication. The code we need to execute when receiving a token depends on the number and types of all the input ports. This is due to the TokenNet token matching semantics in the case of multiple edges. The current implementation handles this complexity by defining different code to handle each possibility, hence the need for multiple vertex classes. We do not want the user to have to write all this boiler-plate code within the application itself.

One approach to supporting typed ports as the basic building block uses partial classes and a two-pass compilation strategy. Each vertex is defined using a partial class unique to the vertex. Using a partial class it is possible to split the definition of a class or a struct, or an interface over two or more source files. Each source file contains a section of the class definition, and all parts are combined when the application is compiled. The class corresponding to each vertex defines a set of input and output ports, and an execute method. The input and output ports are simply interfaces with no supporting implementation. This allows us to compile our application but it will produce a run-time error if we attempt to execute it. After compilation we run a separate TokenNet code generator that uses .NET reflection to find all the vertex classes within the TokenNet application, identifies the input ports within these classes, and then generates the matching part of the partial class. The generated code defines the implementation for each input port, including the code that performs the token matching across all the input ports. The files that are generated by this process are added to the original solution and then the C# compiler is used once again to recompile the application. The result is an application that is completely define, and so can be executed. The application of the TokenNet code generator, and the second compilation phase, can be driven by a post-build step within Visual Studio, and so the user is largely unaware of it other than the increased compilation time.

The post-processor ensures that the arguments of the execute method match the name and carrier types of the input ports. Although the arguments are still ordered in the work method this ordering is irrelevant; reordering the parameters would simply result in a reordering of the generated code. The edges are wired up by name, not position, as each output port is explicitly linked to the appropriate named input port.

For vertex classes with no explicitly defined constructors the code generator can define a nullary constructor in the generated partial class that initializes the input and output ports. It would typically do this by assigning them instances of auxiliary port classes also defined by the generator, each one capturing the required matching semantics. The situation becomes more complex where there are explicit constructors defined in the user code for the vertex class. In this case we need a little help from the user, in the form of a call to a method to initialize the ports from within the constructor. There are two choices here. If it is acceptable to derive all vertex classes from a common vertex base class then we can define an empty virtual method to initialize the ports within this class. The generated code would then override this method for each vertex class, initializing the ports contained within the class. If, on the other hand, requiring a common base class is too restrictive, noting that in C# a class can only be derived from a single class, an alternative strategy is to rely on

conditional compilation. We cannot simply call a method defined in the generated code, as this will not exist at the time of the first compilation pass. However, we can define a compilation symbol that is only defined during the second compilation. We can then require users to call the code to initialize the ports from their constructors, but wrap the call inside an `#if/#endif` section so that it only gets processed during the second compilation, at which point the corresponding method will be defined.

It is a simple task to implement the existing TokenNet API on top of a port-based alternative. But there are other ways of packaging up this raw functionality that provide interesting alternatives to the current approach. For example, we could define abstractions to support structured graphs in the StreamIt style[13], using pipelines, split-joins and feedback loops, and then call a link method to connect up all the ports in the resulting graph. This allows us to simplify the construction of the graph, when compared to the existing TokenNet approach, but delays the recognition of typing errors until the call to the link method. For example, we might define

```
public interface StreamGraph { void Link(); }

public interface TypedStreamGraph<I, O> : StreamGraph, ... {}
```

and then define a `PipelineStreamGraph<I,O>` class derived from `TypedStreamGraph`. The `PipelineStreamGraph` would maintain a list of `StreamGraph` instances, and so instantiating a `PipelineStreamGraph` instance would be simple using C#'s collection constructor syntax, e.g.

```
new PipelineStreamGraph<TI, TO> {
  new Vertex1(),
  new Vertex2(),
  ...,
  new VertexN()
}
```

Of course at this point there is no guarantee that the output port of a vertex in the pipeline matches the input port type of the next vertex in the pipeline. It is only at the point when the ports are linked together that such errors will be exposed. Defining an appropriate link method in a type-safe fashion can be challenging. Fortunately a combination of generic methods and dynamic types is sufficient for this task. For example, we might define

```
public static void Link<I, T, O>(TypedStreamGraph<I, T> upStream,
                                 TypedStreamGraph<T, O> downStream) {
  upStream.Output.LinkTo(downStream.Input);
}
```

and then implement the Link method within the `PipelineStreamGraph` by

```
public override void Link() {
  foreach (StreamGraph subgraph in subgraphs) subgraph.Link();

  for (int i = 1; i < subgraphs.Count; ++i) {
    dynamic upstream = subgraphs[i - 1];
```

```
      dynamic downstream = subgraphs[i];
      Link(upstream, downstream);
    }
    ...
  }
```

Note that the use of `dynamic` should not suggest that any form of dynamic type-checking is used during the execution of the TokenNet graph. All ports, and hence edges, are strongly typed. We simply use the dynamic mechanism to establish the links. If an erroneous pipeline is constructed, where the typing is not consistent, this will result in the C# runtime being unable to determine the appropriate Link method to call, and an exception will be generated.

7 Conclusion

TokenNets is a scheme for expressing parallel programs where, instead of saying *how* the program is to run in parallel, the programmer says how the parallelism needs to be constrained in order to acheive correct results. We have built TokenPilot, a first implementation of TokenNets/ Experiments so far indicate that this style of programming can lead to scalably parallel programs.

TokenPilot only partially acheived one of our goals, namely making all other synchronization mechanisms superfluous. However, we propose to add structured tokens to TokenNets which will make the use of other synchronization mechanisms unnecessary in the use cases we have identified so far.

We have written a significant measurement computation in TokenPilot, demonstrating that TokenPilot graphs can be composed into full computations while retaining reasonable performance.

References

[1] –, *IEEE Standard on Transitions, Pulses, and Related Waveforms,* IEEE Standard 181-2003, 2003.

[2] –, "Intel Threading Building Blocks for Open Source," Intel Corporation, `http://www.threadingbuildingblocks.org`. Accessed 8 December 2009.

[3] –, *Parallel Computing Toolbox 4 Users Guide,*, The WorkWorks, Inc., 2009.

[4] Blumofe, R. D., et al., "Cilk: an efficient multithreaded runtime system" in *ACM Sigplan Notices,* vol. 30, no. 8, pp. 207-216, August 1995.

[5] Budimlic, Z., et al., "Multi-core implementations of the concurrent collections programming model" in CPC '09: 14th International Workshop on Compilers for Parallel Computers, Springer, January 2009.

[6] Dean, J. and Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters" in *Proc. OSDI'04: Sixth Symposium on Operating System Design and Implementation,* San Francisco, CA, December, 2004.

[7] Chandra, R., et al, *Parallel Programming in OpenMP,* Morgan Kaufmann, San Francisco, 2001.

[8] Gelernter, D., "Generative communication in Linda" in *ACM Trans. Programming Languages and Systems,* vol. 7, no. 1, pp. 80-112, January 1985.

[9] Herlihy, M. and Shavit, N., "Work Distribution" in *The Art of Multiprocessor Programming,* Elsevier, San Francisco, pp. 381-392, 2008.

[10] Karp, A. H. and Flatt, H. P., "Measuring Parallel Processor Performance" in *Comm. ACM*, vol. 33, no. 5, May 1990.

[11] Petri, C. A., *Kommunikation mit Automaten*, doctoral dissertation, Universität Bonn, 1962.

[12] Reinders, J., *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly Media, Sebastopol, California, 2007.

[13] Theis, W., *Language and Compiler Support for Stream Programs,* doctoral dissertation, Massachussets Institute of Technology, 2009.

[14] Toub, S. and H. Shafi, "Improved Support For Parallelism In The Next Version Of Visual Studio," in *MSDN Magazine,* October, 2008.