Task Graph

Name:

Task Graph

Note: The *Task Graph* pattern is a concurrent execution pattern and should not be confused with the *Arbitrary Static Task Graph* architectural pattern (1) which addresses the overall organization of the program.

Problem:

Many computational problems can be broken down into a collection of independent operations. Operations produce outputs that are used as inputs to other operations. The operations can be thought of as forming the vertices of a directed acyclic graph while the graph's directed edges show the dependencies between operations. Operations may be performed in parallel or serially, depending on when inputs become available. The *Task Graph* pattern addresses this problem.

Context:

Problems which can be broken down into independent operations as described above can be implemented as graph of independent tasks with dependencies between them. Each task is a separate unit of work which may take dependencies one or more antecedents (see figure 1 below). Tasks with antecedents may only start when all their antecedents have completed. The result of an antecedent task may be passed to dependent tasks on completion of the antecedent. Data does not stream to dependents during execution, the antecedent completes and then its result is passed to the dependents. The final result of the graph is returned when the last dependent task(s) complete (tasks E and F in figure 1 below).

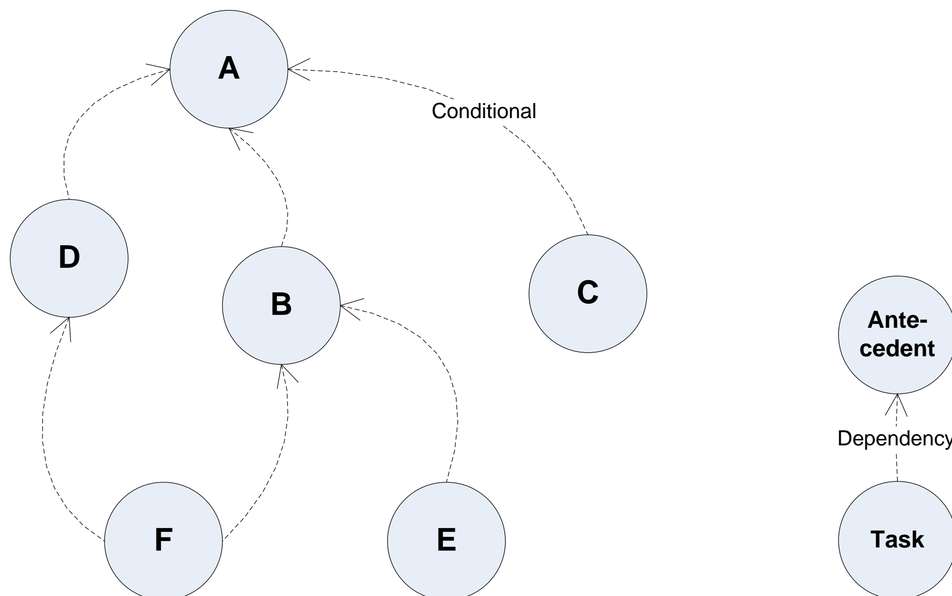

Figure 1: A task graph.

Figure 1 shows task dependencies *not* data flow. Thus tasks B and D are dependent on task A, their antecedent, and cannot start until task A completes. A task may be dependent on multiple antecedents (in the diagram above F is dependent on B and D). The graph is acyclic—the dependencies between tasks are one way—meaning that tasks cannot impose further dependencies on their antecedents.

Task graphs can be considered to come in two flavors depending on when their shape—the vertices and edges—are defined. The following terms are used here:

Statically defined graphs – The shape of the graph is known at compile time.

Dynamically defined graphs – The shape of the graph is not known at compile time so the program defines the graph dynamically at runtime.

In both cases the graph may contain conditional tasks but all tasks and dependencies are defined by the main program before the graph is executed, after which its definition doesn't change. The graph itself is static once defined.

Even though the graph is statically defined it is possible for tasks to be conditional on the outcome of an antecedent. In figure 1 task C only executes if some condition of task A is met. So, even *statically defined graphs* may exhibit some dynamic behavior at runtime depending on whether a condition is met but the overall shape of the graph is known at compile time. Different frameworks and runtimes may support different conditions, e.g. on error, on completion/non-completion, on cancellation (2).

In contrast, *dynamically defined graphs* are defined at runtime. Dynamically defined graphs fall into two broad types, graphs with known dependency patterns between tasks but whose size only known at runtime and graphs whose tasks and dependencies are only determined at runtime.

Figure 2 shows a task graph for a calculation on a grid of variable size. The calculation for grid element (m, n) is dependent on elements (m-1, n) and (m, n-1) but the overall size of the grid is only known at runtime.

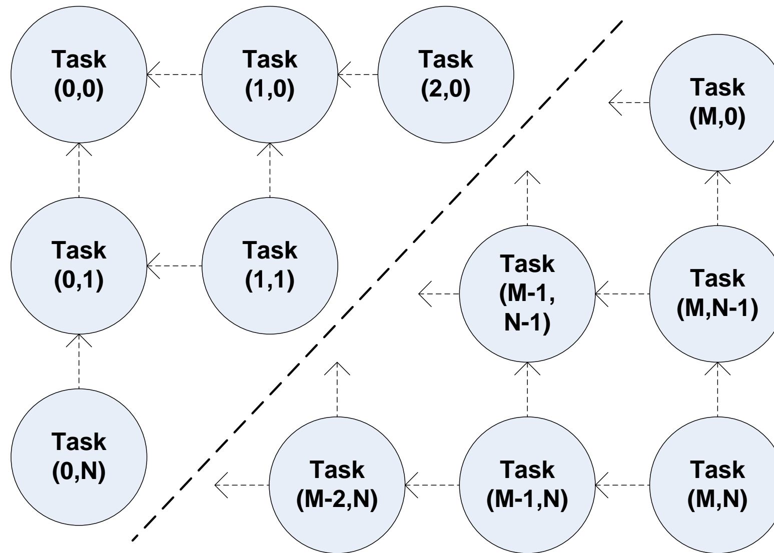

Figure 2: A dynamically defined task graph with a known dependency pattern.

A problem may also be specified in terms of tasks and dependencies which are only know at runtime. In this case the result might look similar to figure 1 but be created by the application at runtime based on input data. Even in this case the task graph does not evolve during execution—tasks do not create other dependent tasks—conditional tasks may be present in the graph but the task and the conditions under which it executes are defined by the application prior to execution of the graph. See the Known Uses section for further discussion of a dynamically defined graph.

Forces:

Universal forces:

1. **Variable vs. uniform task size** – Breaking down the problem into a graph containing tasks of uniform size may be harder but will usually result in a more efficient implementation. A graph which contains highly variable task sizes, e.g. many smaller tasks and one or two very large tasks will typically result in an overall execution time dominated by the larger tasks and is unlikely to be very efficient. Overall a few very small tasks is probably acceptable, but you don't want too many of them, you also don't want your biggest task to be a lot bigger than any of the others.

2. **Large vs. small tasks –** A graph of very small task sizes, even if uniform, may also be inefficient. If the task creation and management overhead becomes significant relative to the amount of work carried out by the task. Conversely, if the tasks are too big, there won't be enough parallelism and lots of time is wasted waiting for the biggest task to finish.

3. **Statically vs. dynamically defined graphs** – Statically defined graphs represent the simplest usage of the pattern but also place significant limitations on its applicability as all tasks and dependencies must be know up front. For problems where the task breakdown is only known at runtime a dynamically defined

task graph is required. Dynamic tasks may exploit more of the available parallelism of the problem but also add to the complexity of the solution.

Implementation forces:

1. **Task breakdown vs. hardware capacity** — Implementations of the task graph pattern may create an arbitrary number of tasks. This may be more than the number of processing elements (PEs) available in hardware. Implementations may limit the number of tasks created to avoid oversubscription of processing elements or reuse processing elements for subsequent tasks but limit the number of tasks being executed simultaneously. Conversely, creating fewer tasks than the number of available PEs will result in underutilization. Efficient implementations usually create significantly more tasks than PEs as this allows the scheduler to load-balance. This is particularly important for task graphs with non-uniform task sizing.

2. **Task granularity vs. startup & communication overhead** — Depending on the communication cost between processing elements and the startup and execution overheads for a task, using larger grained tasks will reduce communication and task creation overhead. Smaller grained tasks may result in better load balancing but the application will waste time creating and communicating with them.

3. **Task granularity vs. startup latency** – Depending on the startup cost of creating tasks or assigning them to processing elements, implementations may use larger grained tasks to reduce overhead when starting new tasks. Finer grained tasks may result in better load balancing at the expense of additional startup cost and resource usage. Frameworks reduce startup latency by reusing PEs from a pool at the expense of additional management overhead and possibly maintaining unused PEs.

Solution:

The *Task Graph* pattern addresses this problem by breaking down the computation into a series of independent tasks with clearly defined dependencies to form a directed acyclic graph. The acyclic nature of the graph is important as it removes the possibility of deadlocks between tasks, provided the tasks are truly independent. When specifying the graph it is really important to understand all dependencies between tasks, hidden dependencies may result in deadlocks. Tasks are then executed on the available processing elements. The overall task graph is considered complete when all the tasks have completed.

Break down the problem using the *Task Decomposition*, *Group Tasks* and *Order Tasks* patterns to decompose the tasks and analyze the dependencies and non-dependencies between them (see: *Patterns for Parallel Programming*, chapter 3 (3)). In summary:

Task Decomposition – Identify independent tasks which can execute concurrently.

Group Tasks – Group tasks to identify temporal dependencies and truly independent tasks.

Order Tasks – Identify how tasks must be ordered to satisfy constraints among tasks.

This approach may lead to several different possible graphs. Use the *Design Evaluation* pattern (see: *Patterns for Parallel Programming*, chapter 3 (3)) to assess the appropriateness of each proposed task graph for the target hardware. In summary:

> How many PEs are available? A task breakdown which results in many more tasks than PEs usually results in a more efficient implementation. However additional tasks represent additional overhead which may reduce the overall efficiency.
>
> How much data is passed between tasks? Does the proposed task graph use the right level of task granularity based on communication latency and bandwidth.
>
> Is the proposed design flexible, efficient and as simple as possible? These may be conflicting requirements especially if more than one target platform is under consideration.

At runtime tasks are then assigned to a pool of processing elements based on the task dependencies and the availability of PEs (see example).

Discussion. The resulting task graph does not inherently load balance the computation (like *Master/Worker*) or have known a load characteristic for a given graph size and number of PEs (like *Pipeline*). Tasks may be of differing sizes which may result in bottlenecks depending on the layout of the graph. The Task Graph pattern does nothing to resolve these issues; they are left to the programmer implementing the graph.

Error Handling. Individual tasks may encounter errors. Implementations need to consider how to handle errors and propagate error state back to the application, either by adding additional flags or data to the task result or by using features specific to the implementation language or framework. For example, the Task Parallel Library aggregates exceptions from tasks within the graph into an `AggregateException` exception (2).

Examples:

Two examples are given here, one for a statically defined graph and another for a dynamically defined graph with a known dependency pattern but unknown size.

A Statically Defined Graph

Here is a hypothetical system which loads both current (last trading period) and historical financial data from different sources and runs a risk model against both sets of data before comparing the result to verify the model’s relevance to current trading conditions.

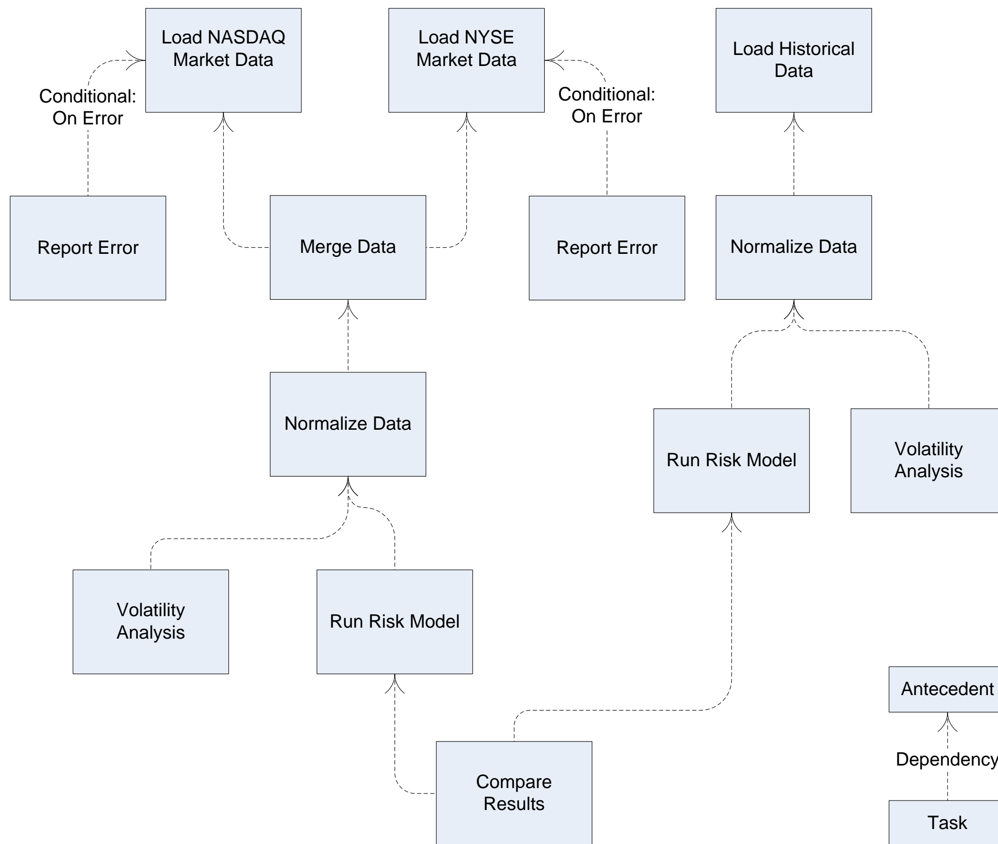

Figure 3: A task graph representing the dependencies in financial modeling application.

The above figure shows task dependencies *not* data flow. Thus task "Merge Data" is dependent on tasks "Load NASDAQ Market Data" and "Load NYSE Market Data". The program reports volatility statistics for market and historical data as well as comparing models. These three results may be returned in any order as the graph does not specific any dependencies between the tasks.

This task breakdown can be expressed as a statically defined task graph in C# using the .NET Framework 4 Task Parallel Library (TPL) (4) and targeted at a multi-core processor. For clarity the example code below defines the graph but the code for the actual tasks is not shown.

```
class Program
{
  private static MarketData LoadNyseData() { /* ... */ }
  private static MarketData LoadNasdaqData() { /* ... */ }
  private static void ReportError(Task<MarketData> task) { /* ... */ }

  private static MarketData LoadFedHistoricalData(){ /* ... */ }
  private static MarketData NormalizeData(Task<MarketData>[] tasks) { /* ... */ }
  private static MarketData MergeMarketData(Task<MarketData>[] tasks) { /* ... */ }
  private static AnalysisResult AnalyzeData(Task<MarketData>[] tasks) { /* ... */ }
  private static ModelResult ModelData(Task<MarketData>[] tasks) { /* ... */ }
  private static void CompareModels(Task<ModelResult>[] tasks) { /* ... */ }

  static void Main(string[] args)
  {
     TaskFactory factory = Task.Factory;
```

```
      Task<MarketData> loadNyseData = factory.StartNew<MarketData>(LoadNyseData);
      loadNyseData.ContinueWith(ReportError, TaskContinuationOptions.OnlyOnFaulted);

      Task<MarketData> loadNasdaqData = factory.StartNew<MarketData>(LoadNasdaqData);
      loadNasdaqData.ContinueWith(ReportError, TaskContinuationOptions.OnlyOnFaulted);

      Task<MarketData> mergeMarketData =
         factory.ContinueWhenAll<MarketData, MarketData>(
            new[] { loadNyseData, loadNasdaqData },
            MergeMarketData);
      Task<MarketData> normalizeMarketData =
         factory.ContinueWhenAll<MarketData, MarketData>(
         new[] { mergeMarketData }, NormalizeData);

      Task<MarketData> loadFedHistoricalData =
         factory.StartNew<MarketData>(LoadFedHistoricalData);
      Task<MarketData> normalizeHistoricalData =
         factory.ContinueWhenAll<MarketData, MarketData>
            (new[] { loadFedHistoricalData },
            NormalizeData);

      Task<AnalysisResult> analyzeHistoricalData =
         factory.ContinueWhenAll<MarketData, AnalysisResult>(
            new[] { normalizeHistoricalData },
            AnalyzeData);
      Task<ModelResult> modelHistoricalData =
         factory.ContinueWhenAll<MarketData, ModelResult>(
            new[] { normalizeHistoricalData },
            ModelData
            TaskContinuationOptions.LongRunning);

      Task<AnalysisResult> analyzeMarketData =
         factory.ContinueWhenAll<MarketData, AnalysisResult>(
            new[] { normalizeMarketData },
            AnalyzeData);
      Task<ModelResult> modelMarketData =
         factory.ContinueWhenAll<MarketData, ModelResult>(
            new[] { normalizeMarketData },
            ModelData
            TaskContinuationOptions.LongRunning);

      Task compareModels = factory.ContinueWhenAll<ModelResult>(
         new[] { modelMarketData, modelHistoricalData },
         CompareModels);

      Task.WaitAll(analyzeHistoricalData, analyzeMarketData,
         compareModels);
   }
}
```

The TPL distributes the tasks across threads running on the available cores using task continuations to start new tasks as their antecedents complete.

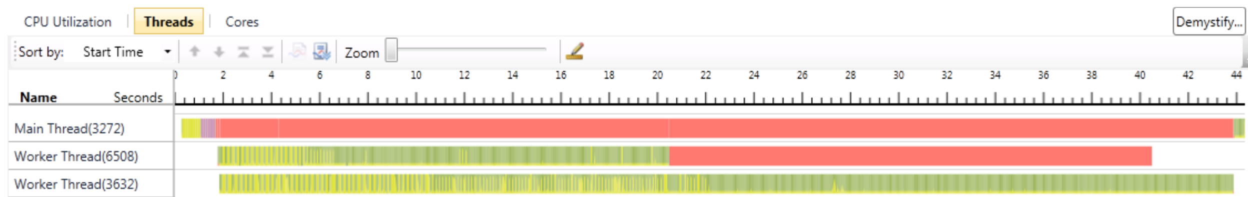

Figure 4: Profiler output showing work distribution across threads.

The Visual Studio 2010 profiler result above shows the main thread and two worker threads from a run on a two core machine (other threads are hidden for clarity). The computation is dominated by the `ModelHistoricalData` task which runs longer than other tasks. This results in only one thread being used for latter half of the computation.

A Dynamically Defined Graph with Known Dependency Pattern

The second example is of a dynamically defined graph of similar to the one shown in figure 2. It applies a method, `processRowColumnCell(int a, int b)` to each element in a integer matrix of size `numRows` by `numColumns`. The method accesses only the values of the cells above and to the left and calculates a new value for cell [a, b] based on them.

```
public static void GridUpdate(int numRows, int numColumns,
  Action<int, int> processRowColumnCell)
{
  Task[] prevTaskRow = new Task[numColumns];
  Task prevTaskInCurrentRow = null;

  for (int row = 0; row < numRows; row++)
  {
     prevTaskInCurrentRow = null;
     for (int column = 0; column < numColumns; column++)
     {
        int j = row, i = column;

        Task curTask;
        if (row == 0 && column == 0)
        {
           curTask = Task.Factory.StartNew(() => processRowColumnCell(j, i));
        }
        else if (row == 0 || column == 0)
        {
           var antecedent = (column == 0) ? prevTaskRow[0] : prevTaskInCurrentRow;
           curTask = antecedent.ContinueWith(p =>
           {
                  processRowColumnCell(j, i);
           });
        }
        else
        {
           var antecedents = new Task[] { prevTaskInCurrentRow, prevTaskRow[column] };
           curTask = Task.Factory.ContinueWhenAll(antecedents, ps =>
           {
                  processRowColumnCell(j, i);
           });
        }
```

```
            // Keep track of the task just created for future iterations
            prevTaskRow[column] = prevTaskInCurrentRow = curTask;
        }
    }

    // Wait for the last task to be done.
    prevTaskInCurrentRow.Wait();
}
```

For clarity error handling code has been left out of both these samples. For further examples of task graphs and other the patterns supported by the Task Parallel Library see Toub 2009 (5).

Known Uses:

The Task graph pattern is used in a number of instances including:

Build tools create dynamically defined task graphs based on component dependencies and then build components based on ordering. They create graphs where neither the graph's size nor dependency pattern is known at runtime. MSBuild (6) is one such system. MSBuild defines a build process as a group of targets each with clearly defined input and outputs using an XML file. The MSBuild execution engine then generates a task graph and executes it. The executables compiled by one target forming the inputs to subsequent dependent target.

The following frameworks support implementation of this pattern:

- The .NET 4 Task Parallel Library implements this pattern using Continuation Tasks (2)
- Dryad supports the task graph pattern although it adds additional support for streaming data between tasks (7).
- OpenMP 3.0 (8).
- The StarSs programming model (9).

Related Patterns:

Arbitrary Static Task Graph (structural pattern) – Addresses the overall organization of the program rather than an implementation on specific programming model(s).

Task Decomposition, Order Tasks & Group Tasks – The task decomposition, order tasks and group tasks patterns detail an approach to understanding task dependencies. They help the implementer to specify the directed graph prior to implementation.

Several other common patterns bear similarities to Task Graph but actually differ in important respects:

Pipeline – Differs in several important respects from a task graph. Firstly, all pipelines have a fixed (linear) shape with some slight variation for non-linear pipelines. Secondly, the pattern focuses on data flow, rather than task dependencies. Data flows through a pipeline with the same task being executed on multiple data items. Finally, pipelines are usually synchronous with data moving from task to task in lockstep.

Master/Worker – Tasks within the master/worker pattern have a child/parent rather than antecedent/dependent relationship. The master task creates all tasks, passes data to them and waits for a result

to be returned. Worker tasks typically all execute the same computation against different data. The number of workers may vary dynamically according to the problem size and number of available PEs.

Divide and Conquer – Tasks within the divide and conquer pattern have a child/parent rather than antecedent/dependent relationship. The resulting task tree is dynamic with the application creating only the root task. All sub-tasks are created by their parent during computation based on the state of the parent. Data is passed from parent to child tasks; the parent then waits for children to complete and may receive results from its children. So, data can move bi-directionally between parent and children, rather than being passed one way from antecedent to dependent.

Discrete Event – Focuses on raising events or sending messages between tasks. There is no limitation on the number of events a task raises or when it raises them. Events may also pass between tasks in either direction; there is no antecedent/dependency relationship. The discrete event pattern could be used to implement a task graph by placing additional restrictions upon it.

References:

1. **Mattson, Tim.** Our Pattern Language (OPL). *A Pattern Language for Parallel Programming ver2.0.* [Online] [Cited: March 1, 2010.] http://parlab.eecs.berkeley.edu/wiki/_media/patterns/opl_pattern_language-feb-13.pdf.

2. **Microsoft Corporation.** Continuation Tasks. *MSDN.* [Online] [Cited: March 1, 2010.] http://msdn.microsoft.com/en-us/library/ee372288(VS.100).aspx.

3. **Tim Mattson, Beverly A. Sanders and Berna L. Massingill.** *Patterns for Parallel Programming.* s.l. : Addison-Wesley, 2004. ISBN 978-0321228116 .

4. **Microsoft Corporation.** Task Parallel Library. *MSDN.* [Online] [Cited: March 1, 2010.] http://msdn.microsoft.com/en-us/library/dd460717(VS.100).aspx.

5. **Toub, Stephen.** Patterns for Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4. *Microsoft Download Center.* [Online] [Cited: March 1, 2010.] http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee.

6. **Microsoft Corporation.** MSBuild Task. *MSDN.* [Online] [Cited: March 4, 2010.] http://msdn.microsoft.com/en-us/library/z7f65y0d.aspx.

7. —. Dryad. *Microsoft Research.* [Online] [Cited: March 1, 2010.] http://research.microsoft.com/en-us/projects/dryad/.

8. **Oracle Corporation.** OpenMP Tasking. *Oracle Wikis.* [Online] [Cited: March 1, 2010.] http://wikis.sun.com/display/openmp/Tasking.

9. **Barcelone Supercomputing Center.** SMP superscalar. *Barcelone Supercomputing Center.* [Online] [Cited: March 4, 2010.] http://www.bsc.es/plantillaG.php?cat_id=385.

Authors:

Ade Miller (Microsoft Corporation, patterns & practices group) with review feedback from; Tasneem Brutch (Samsung Electronics, US R&D Center), Ralph Johnson (UIUC), Kurt Keutzer (UC Berkeley) and Stephen Toub and Rick Molloy (Microsoft Corporation).