

# **A ManyCore Opportunity: Programming Data-driven Applications with MicroServices.**

Beth Plale and Dennis Gannon

Computer Science, School of Informatics, Indiana University

## ***The Application Challenges***

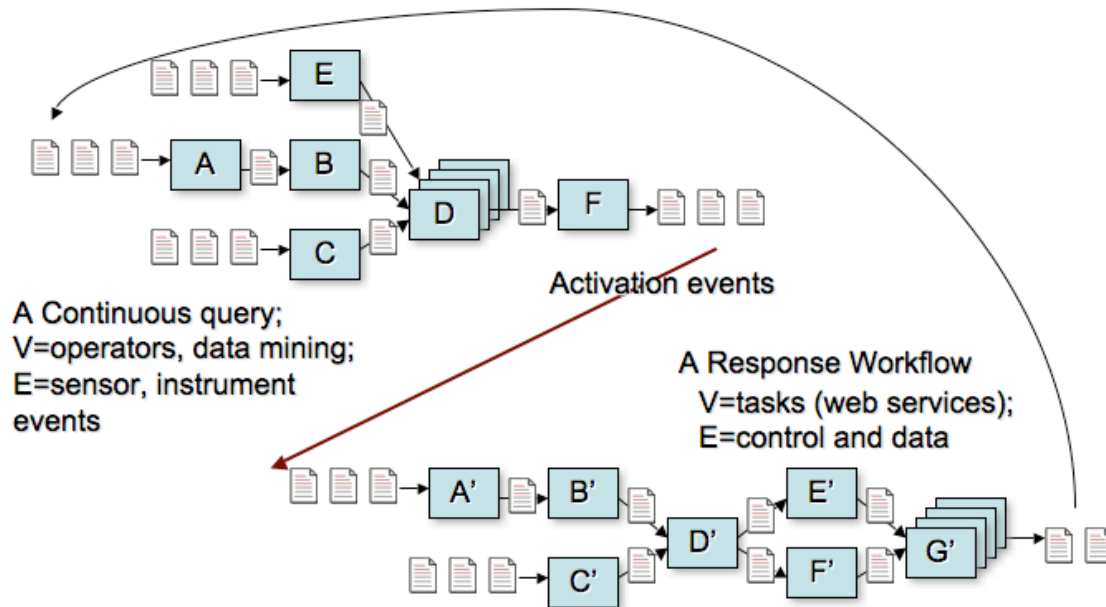
Access to data and the ability to continuously process it and extract knowledge in real-time may be the killer application for multicore and future manycore systems. The data may be in very large databases, remote data archives searchable via local metadata catalogs or it may be streaming from instruments or sensors. Specific applications will be predicting severe weather, discovering patterns in large scientific data sets such as is done in drug discovery and digital sky searches, responding to disasters, processing computer vision data, monitoring complex systems and responding to them in real time such as robot control and motion planning. We do not expect end users to “program” these systems. Rather the Web and the success of database query languages tell us that users will pose “reactive queries” of the form “Discover all the known sources of cheap hydrogen for my car within this area and notify me when changes occur.” Or, “Monitor the weather over Chicago and when a supercell is detected run a tornado forecast.” “Scan all the published literature on the effect of compound X on protein Y and notify me when new discoveries are published.” “Watch the lobby of the building looking for the person in this photo and then greet her and notify me.” These query response scenarios are data driven and persistent. They require an agent-like or rule-based framework that is constantly acting on the users behalf. They are also typically realized as pipelined, complex event-driven workflows that must adaptively use resources (cores and bandwidth) to keep up with irregular data arrival rates.

The infrastructure challenge is *to design a programming framework for complex data-driven knowledge discovery through which users interact by posing complex, related event-action scenarios such that when the conditions forming a scenario occur, a complex response is initiated automatically.* Furthermore, we must blend these declarative queries with policy annotations such as “maximize output rate” or “minimize detection latency” which can have the effect of throttling or expanding the amount of concurrency applied to the system.

## ***A Technical Approach***

It is possible to *translate these continuously executing queries and rule patterns into data pipelines composed of agile “micro services” that can scale and efficiently on manycore processors.* A common pattern for these queries takes the form of “When ... Then ...”, where the “When” part is data mining continuous query that monitors the external world and the “Then” part that is the response workflow. As illustrated below, the query takes the form of a pipelined DAG that is receiving events from the external world and applying a set of logical data mining filters and relational algebra-like operators. Some event streams may be generated by database searches or data catalog crawlers. The

domain of complex event processing has already characterized many such patterns and primitives.



When the query is satisfied an activation event is sent to a response workflow. Concurrency can occur at three levels in this system. First, the continuous query is pipelined (with special treatment of join like operations). Second, for pipeline stages that are bottlenecks, multiple instances of that component may be allocated in parallel (as is illustrated at node D and G' above.). This type of concurrency can be seen as a type of “map-reduce” applied to a pool of messages. Finally, in the case of the response workflow, in addition to pipelining, multiple copies of the entire graph can be allocated. Given a sufficiently large library of application workflow components, it is possible to compose arbitrarily complex response workflows. Many examples of this style of workflow as composed services now exist in eScience and it has strong roots in computer graphics. In some cases responses can feed back into the query event stream. For example, a feedback may refine or redirect a search.

Standard thread systems and synchronization mechanisms can be used to implement arbitrarily complex data processing pipelines, but they suffer from lock contention and long thread creation, tear-down and context switching times. Managed thread pools can mitigate the thread creation/tear-down costs and Software Transactional Memory (STM) can be used to avoid lock problems, but the traditional model of allocating one or more threads to a service still does not scale because of long context switch times. Our approach is based on the concept of a **micro-service**, which we define as web service-like entity that can handle large streams of events coming from external sources as well as in-memory messaging between services in the same process. Rather than binding a thread to a service, a single thread may execute an entire pipeline path, thus avoiding synchronization between coupled services. Micro-services differ from objects in an object oriented system because they are largely stateless and are message oriented, i.e. rather than implementing complex method signatures, they support a small set of “http-

like” operations (GET, PUT, SUBSCRIBE, QUERY ...) where the operand is a binary encoded XML SOAP message. Because micro-services are extremely lightweight, very large numbers (hundreds) can be managed by a pool consisting of one thread per core, all within a single process.

The most promising implementation of this concept comes from the Microsoft Robotics Studio’s Concurrency and Coordination Runtime (CCR), which manages ports, queues, and has lightweight implementations of the join pattern based on arbiters, and the Decentralized Software Services (DSS) stack, which implements the service level.

## **Challenges**

In their current form multicore processors are limited by primitive synchronization operations that lock the bus and prevent all cores from making progress. The micro-service model described above does not completely avoid locks, so this will continue to be a problem until the next generation of processors addresses these issues. False sharing and other types of cache conflicts are an additional problem that bedevil performance of current multicore systems. This problem can be mitigated by increasing core private cache sizes.

However, for the applications and approach described here, the greatest challenge will be managing the data throughput. If all the data in a particular event pipeline can reside in the on-processor memory hierarchy, performance should be excellent. Will the applications have the ability to control write-back to memory? If this is true, the question to ask is when is it better to throttle the number of concurrent streams to minimize cache-to-memory spills than to attempt to increase concurrency and use more cores for a particular query? Which types of continuous query response patterns will result in more data reuse from cache per each data write?

We assumed above that all the mirco-services were all living in one shared managed VM which uses all the cores of a single processor. Scalability will require multiple multicore processors either in a shared memory configuration or in a distributed environment with a fast switch or network connecting different instances of the OS. How will this model data-driven micro service model scale in this context?