Retargeting Embedded Software Stacks for Many-Core Systems

Dr. Sumant Tambe, Real-Time Innovations (RTI), Inc., Sunnyvale, CA
Prof. James Anderson, University of North Carolina, Chapel Hill, NC
Contact: sumant@rti.com

Keywords: Multi-core, Many-core, Concurrency, Data-Centric Middleware, Component-based Software Engineering, Message-passing, Data Distribution Service, Scheduler

Introduction

As recent technology trends usher us into the many-core era, we need novel techniques that allow high-performance embedded applications to exploit massive local concurrency. To position software applications to do more on machines with more cores—i.e., re-enabling the proverbial free lunch—requires substantial restructuring of the embedded software stacks, which includes applications, middleware, and the operating system. New operating system and middleware mechanisms are required to handle multi-threading, scheduling, resource-sharing, and communication in many-core systems.

As contemporary many-core processors have cores in high double digits, keeping all the cores busy is not simple. Existing software stacks are rarely designed to adapt and scale on machines with up to a dozen cores. Multi-threading is a popular choice for implementing concurrency in infrastructure software. It is, however, extremely hard to get right without the systematic use of the proven practices and patterns of managing concurrency.

Concurrency is that elephant in the story of blindfolded men who make wildly different perceptions about the elephant depending on where they touch it. That is, concurrency has many dimensions. Further, whether you are modernizing an existing code-base or starting with a clean slate will determine your perception. Nevertheless, the enduring solution is likely to use timeless techniques as the foundation, combined with the modern technology. So Real-Time Innovations (RTI) and University of North Carolina (UNC) came up with just that when we set out on a path-finding mission to modernize an existing data distribution middleware. We found four promising ways, which often intersect:

- Concurrency patterns for effective multi-threading,
- Component-based software design for scalable many-core applications,
- Hardware-accelerated messaging transports for inter-core communication, and
- Smart scheduling for many-core processors.

Concurrency patterns are specific multi-threading techniques that help improve responsiveness and the overall throughput. The component-based software design supports effective data and functional partitioning, which is the key to enabling *shared-nothing parallel programming*—an enduring principle. Further, it helps developers specify dataflow requirements between the components and manage their lifecycle. Modern messaging transports such as Tilera iLib [1] library and Multicore Communication API (MCAPI) [2] enable message-passing between cores. When high-throughput networks are available

directly in the processor, the inter-core communication latency is reduced to just a few cycles. Finally, smart scheduling algorithms use the dataflow requirements to assign components to clusters of cores to efficiently use the underlying processing capacity and minimize data movement.

Let's dive deeper.

Concurrency Patterns for Effective Multi-threading

Adopting multi-threading best practices and patterns often results in a high ROI if your infrastructure software is already multi-threaded. Concurrency patterns [14] help infrastructure software scale on multi-core platforms. A large number of research papers, articles, and books have been written on improving concurrency using explicit multi-threading. It is used widely and we expect its use to grow as the concurrency patterns are better understood.

The Leader/Followers [3] concurrency pattern is a proven thread-management technique that allows multiple threads to take turns and share a set of event sources in order to detect, dispatch, and process service requests. Little or no synchronization between threads is necessary when they execute logically independent service requests. This pattern also minimizes latency because of the multiple threads.

We applied the Leader/Followers thread-pool in RTI Connext DDS to support concurrent DataReaders. Figure 1 shows a simplified test scenario where process #1 is publishing various *shapes* objects to process #2. Process #2 was tested with up to 10 concurrent subscribers served by 10 threads organized in the Leader/Followers fashion. This initial thread-pool setup helped achieve a nearly 250% increase in the overall messaging throughput when tested on a CPU-bound, mostly data-parallel work-load.

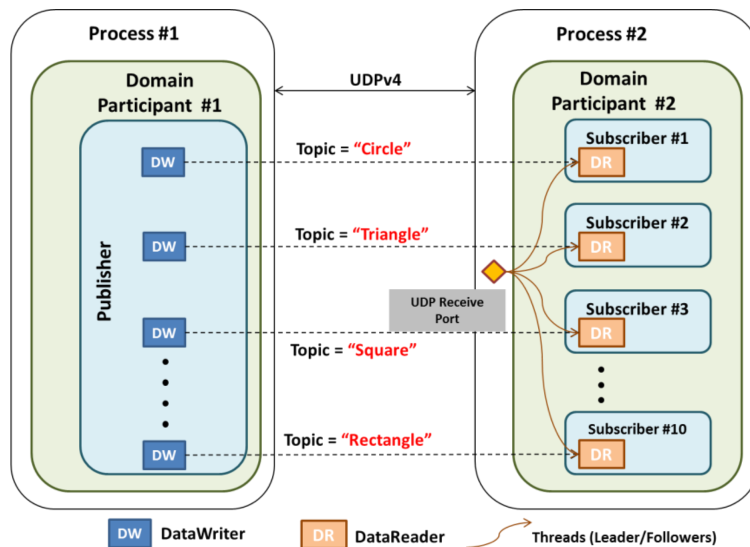

Figure 1: Leader/Follower Thread-pool in RTI Connext DDS Serving Concurrent DataReaders

Data Distribution Service (DDS) [4] is a standard managed by the Object Management Group (OMG). Several satellite standards have been defined around DDS, including C++ and Java APIs [5] [6] and a wire-interoperability protocol [7]. DDS defines a data-centric publish-subscribe architecture for connecting anonymous information providers with information consumers. Like SOA, DDS promotes loose coupling between system components. A distributed application is composed of data providers and consumers, each potentially running in a separate address space, possibly on different computers. A data provider publishes typed data-flows, identified by names called *topics*, to which consumers can subscribe. RTI Connext™ DDS is the industry-leading implementation of the DDS standard.

Component-based Software Design for Shared-Nothing Parallel Programming

Designing scalable, adaptable, and maintainable applications for many-core architectures requires three key constituents:

1. Functional partitioning without shared state to enable Multiple Instruction Multiple Data (MIMD)-style parallelism
2. High-throughput messaging infrastructure for inter-component communication (i.e., data-flow)
3. Effective resource allocation and scheduling algorithms to exploit local massive concurrency within a reasonable power budget

Component-based design is well-aligned with functional partitioning because components are designed to be modular, cohesive, and independently deployable. Components facilitate the development of pipelined software architectures, which can exploit concurrency as much as the depth of the pipeline. For example, Figure 2 shows a pipeline for an image-processing application where every block represents a component that can execute concurrently with other components. Each arrow represents an explicit messaging channel more amenable to monitoring and control than implicit communication such as function invocation. The components communicate with each other exclusively using messages and do not share state in any other way.

Figure 2: Image pipeline example flow

This architecture has roots in the Actor model [10], which advocates lock-free, *shared-nothing* concurrency and asynchronous message-passing between actors. A number of commercial and open-source platforms are using multiple agents [11] to simplify programming of massively parallel architectures. The next section describes how components deployed on a single many-core host can communicate efficiently without tight coupling.

High-Performance Data-Centric Messaging for Intra-Node Communication

Shared memory is still the preferred communication method on commodity hardware with a handful of cores. This trend, however, is reversing due to performance and correctness issues on many-core architectures. Shared-memory techniques have limited scalability on many-core processors due to the overhead of cache coherence protocols. Moreover, shared-memory systems resist composability—the ability to build complex applications by composing smaller modules. Judicious use of message-passing and shared memory is the preferred way of sharing data on many-core processors.

Message passing is well aligned with component-based programming. Components that use message passing for communication provide a powerful mechanism of isolation since sharing takes place via message exchanges. Validation is simplified because the programmer only needs to check the externally observable component interaction to derive correctness properties for the system as a whole.

Chip manufacturers have developed novel processor architectures with on-chip high-performance interconnects for efficient message-passing across cores. See Figure 3. Vendor-specific message-passing libraries (e.g., RCCE [12] from Intel, iLib from Tilera) and messaging standards such as MCAPI are available to program this new crop of processors.

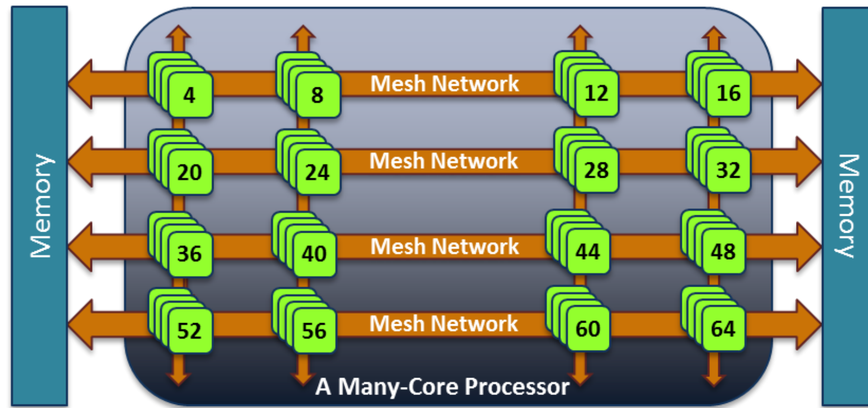

Figure 3: Schematic of a many-core processor

The message-passing libraries are specialized for closely distributed environments. They are often lightweight and offer higher performance due to on-chip hardware-level support for message-passing between cores. The programming model offered by these libraries is very similar to that of the Message Passing Interface (MPI). MPI is particularly suitable for *fully synchronized* (i.e., matched send/receive calls) communication, which is often preferred in low-level modules because the most basic building blocks can be implemented in hardware and higher-level abstractions can be built without much overhead.

The MPI-style programming model, however, is not suitable for many real-time distributed systems where dynamic workloads, changing topologies, and stringent Quality-of-Service (QoS) requirements must be supported. The DDS programming model is particularly suitable in such environments thanks to its built-in discovery protocol, nearly two dozen QoS policies, and a publish-subscribe communication model. Table 1 summarizes the key architectural differences between DDS and MPI-style programming. DDS provides the necessary anonymous publish-subscribe programming model to create large-scale real-time distributed systems.

A new technology is needed to combine the best of both worlds.

When multiple communicating applications are deployed on a single many-core host, the infrastructure should use the most appropriate communication mechanism to share data. The details of low-level APIs should be hidden from the applications so that a different transport can be plugged-in without changing the application sources. For example, RTI Connext DDS provides a powerful data-centric publish-/subscribe programming model for real-time applications. When these applications run on a single many-core host, a variant of the pluggable transport [13] layer abstracts the details of the low-level transport. Through the pluggable transport, RTI Connext DDS uses the MPI-style messaging primitives internally. Applications use the standard DDS API without losing efficiency and/or loose coupling on many-core platforms.

Table 1: Key architectural differences between DDS and MPI-style programming APIs

	Data Distribution Service (DDS)	API of iLIB, RCCE, MCAPI (MPI-style programming)
Programming Model	Anonymous publish-subscribe	Fully synchronized (i.e., matching send/receive calls)
Message Architecture	Data-centric middleware (data ≠ messages, knows schema)	Message-centric (just byte arrays or scalars, data == messages)
Coupling	Loosely coupled	Tightly coupled
Optimized for	Widely distributed	Closely distributed in case of iLIB, RCCE, and MCAPI. (In general, MPI is suitable for HPC clusters.)
Topology	Hides physical topology	Uses physical topology
System Calls	Uses different system calls depending upon transport	Avoids system calls when special instructions for using high-speed interconnect are available

RTI, in collaboration with UNC, implemented a prototype transport using OpenMCAPI—an implementation of the MCAPI standard. Using this transport plugin, we were able to execute existing applications and RTI infrastructure services without any change to the source. This novel solution allows applications to use the most suitable transport on a given platform without changing the programming model and/or the source code. As operating systems gain more capabilities for many-core systems, the middleware and applications can take advantage of these capabilities simply by developing new pluggable transports.

Many-Core Resource Allocation and Scheduling Algorithms

The infrastructure middleware must ensure an optimal allocation of concurrent components to clusters of cores and schedule their execution so that the instruction throughput can be maximized. RTI and UNC developed new techniques for efficient allocation of flow-based computations to enable scalable performance on a many-core platform (Intel Single-Chip Cloud Computer).

Processing Graph Method (PGM) [9] is an expressive formalism for representing flow-based computations. In PGM, a computation is represented as a directed acyclic graph (DAG), in which each vertex is a task (sequential program) and each edge is a typed, first-in first-out (FIFO) queue that connects a producing task with a consuming task; such queues abstract message communication. We are leveraging prior work on PGM graphs to determine how to best allocate flow-based computations on many-core platforms. Specifically, we specify application-specific component assembly as PGM graphs, to schedule them using deadline-based clustered scheduling algorithms, and to analyze them for schedulability

assuming that deadlines are “soft” and can be missed by a bounded amount. Note that if bounded deadline tardiness is ensured, long-term processing rates will be as prescribed.

RTI and UNC are developing techniques for assigning PGM nodes to clusters of cores within a many-core platform. Such techniques are extensions of similar existing techniques [9] for clustered scheduling of PGM graphs in networked systems. These techniques will utilize the cores efficiently and will avoid excessive data movement across cores.

In summary, 100s of cores will soon stop being a novelty. The onus is on the programmers to get the most out of the processor, which is essentially a supercomputer. Data-centric middleware offers a powerful programming model combined with scalability and flexibility necessary to cope with the ever-changing landscape of many-core processors.

References

[1] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, Anant Agarwal, “On-chip Interconnection Architecture of The Tile Processor”, IEEE Micro, 27(5):15–31, 2007

[2] The Multicore Communications API (MCAPI), http://www.multicore-association.org/workgroup/mcapi.php. Accessed September 2012

[3] Douglas C. Schmidt, Carlos O’Ryan, Michael Kircher, Irfan Pyarali, and Frank Buschmann; Leader/Followers; http://www.cs.wustl.edu/~schmidt/PDF/lf.pdf. Retrieved September 2012

[4] The Data Distribution Service specification, v1.2, http://www.omg.org/spec/DDS/1.2/

[5] DDS-PSM-Cxx: ISO/IEC C++ 2003 Language DDS PSM, http://www.omg.org/spec/DDS-PSM-Cxx/1.0

[6] DDS-Java: Java 5 Language PSM for DDS, http://www.omg.org/spec/DDS-Java/1.0/Beta3/PDF

[7] The Real-Time Publish Subscribe DDS Wire Protocol, v2.1, http://www.omg.org/spec/DDSI/2.1/

[8] D. J. Kaplan, “An introduction to the processing graph method”. In the proceedings of Engineering of Computer-based Systems”, 1997

[9] C. Liu and J. Anderson. “Supporting graph-based real-time applications in distributed systems”. In the Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. IEEE Press, August 2011

[10] Gul Agha, “Actors: A Model of Concurrent Computation in Distributed Systems”. Doctoral Dissertation. MIT Press, 1986

[11] Michael Wooldridge, “An Introduction to Multi-Agent Systems”. Publisher John Wiley & Sons Ltd, 2009

[12] RCCE: a Small Library for Many-Core Communication (RCCE Specification), http://techresearch.intel.com/spaw2/uploads/files//RCCE_Specification.pdf

[13] Douglas C. Schmidt, Carlos O’Ryan, Ossama Othman, Fred Kuhns, and Jeff Parsons, “Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware”, http://www.cs.wustl.edu/~schmidt/PDF/pluggable_protocols.pdf

[14] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, “Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects”, Wiley 2000, ISBN-10: 0471606952

Bios

Sumant Tambe. Dr. Sumant Tambe is Senior Software Research Engineer at Real-Time Innovations, specializing in standards-based data-centric and component-oriented middleware for distributed, real-time, and embedded (DRE) systems. His expertise includes static and adaptive fault-recovery protocols for component-based, multi-tier real-time systems, publish-subscribe middleware, real-time embedded systems, model-driven deployment and configuration of QoS-enabled middleware, and techniques for component behavior and dependability modeling.

Dr. Tambe is the chair of Java 5 Language Platform Specific Model (PSM) for DDS finalization task force and a voting member on C++ Language PSM for DDS and Remote Procedure Call (RPC) over DDS standards in the Object Management Group (OMG). He is a regular speaker at several technology conferences, workshops, OMG meetings, and code camps. Dr. Tambe is the author of "More C++ Idioms" wikibook and also the "C++ Truths" blog.

Dr. Tambe has a Ph.D. in Computer Science from Vanderbilt University and an M.S. degree in Computer Science from New Mexico State University.

James H. Anderson. Prof. James H. Anderson is a professor in the Department of Computer Science at the University of North Carolina at Chapel Hill. He received a B.S. in Computer Science from Michigan State University in 1982, an M.S. in Computer Science from Purdue University in 1983, and a Ph.D. in Computer Sciences from the University of Texas at Austin in 1990.

Before joining UNC-Chapel Hill in 1993, he was with the Computer Science Department at the University of Maryland between 1990 and 1993. In 1995, Dr. Anderson received the U.S. Army Research Office Young Investigator Award, and in 1996, he was named Alfred P. Sloan Research Fellow. He won the Computer Science Department's teaching award in 1995, 2002, and 2005. He has served as program chair and/or general chair for several conferences, including the ACM Symposium on Principles of Distributed Computing, the IEEE International Real-Time Systems Symposium, the International Conference on Principles of Distributed Systems, the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, and the Euromicro Conference on Real-Time Systems.

Prof. Anderson's main research interests are within the areas of concurrent and distributed computing and real-time systems. He has authored over 170 papers in these areas. He has graduated 11 Ph.D. students and several M.S. students. He is currently supervising eight Ph.D. students and is managing several projects, funded by NSF, ARO, AFOSR, AFRL, and industry.