High Performance Software Architectures: A Connector-Oriented Approach

David Woollard
University of Southern California
Department of Computer Science
Los Angeles, CA 90089-0781
woollard@usc.edu

Nenad Medvidovic
University of Southern California
Department of Computer Science
Los Angeles, CA 90089-0781
neno@usc.edu

ABSTRACT

Scientists in multiple domains have begun conducting investigations using a new paradigm centered on computer simulation as means of experimentation and theory validation. Unfortunately, our ability to program simulations that are equal to the task of truly new science is handicapped by our lack of support for high performance computing abstractions. In this position paper, we explore the potential role of software architectures as a means of encapsulating many of the services required for parallel programming into explicit first-class software connectors.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

Keywords

domain specific software architectures, high performance computing, high performance connectors

1. INTRODUCTION

A growing number of physicists, biologists, chemists, and computer scientists in multiple physical domains have embraced a new scientific process that emphasizes simulation as a fundamental method not only in the evaluation of theory but also in the formulation of new science. This shift has been made possible in recent years as the cost-to-performance ratio of consumer hardware has continued to decrease. Computational clusters consisting of fast networks and commodity hardware have become a common sight in research laboratories.

While parallel hardware has become commonplace, our ability to build software capable of matching the theoretical limits of today's supercomputers has fallen short [11]. Our understanding of parallel programming including the abstractions available to scientists with deep domain knowledge but little computer science experience are significantly lacking.

In the past forty years, we have seen the emergence and obsolescence of a superabundance of parallel languages, libraries, and compilers, each promising to be a silver bullet for the parallel programming practitioner. Potentially panacean domain specific parallel languages such as Sisal [1] have offered serviceable development of parallel scientific applications, but have never gained wide acceptance possibly due to the difficulty in adopting new languages (the death rate of languages has been estimated at well over 99% [15][1]).

Today, scientific programmers have eschewed domain specific parallel languages in favor of parallel libraries for established sequential programming languages such as C and Fortran. OpenMP's pragmas [2] and the Message Passing Interface's (MPI) library calls [4] are two examples of parallel libraries in wide use today. Because of interface bloat [5] and the resulting code obfuscation inherent in repetitive library calls, current efforts such as DARPA's High Productivity Computer Systems Initiative are revisiting language-level support for parallel computing.

A largely unexplored alternative to language-level support for parallel computing is to support the concepts of concurrency, synchronization, and topological ordering in a domain specific software architecture [14, 10, 16]. The discipline of software architectures is a powerful abstraction primarily concerned with the composition of software systems from constituent elements such as computational units (components), the communications facilities between these components (connectors), and the organization of these elements in topologies (configurations).

Software architectures do not require investment in new languages by the developer and so avoid the non-acceptance problem of previous domain-specific parallel languages. Because software architectures are high-level abstractions that provide the developer with a concise model of software systems, they do not require the overlay of parallelism in the form of library calls. Rather than support explicit communications interwoven with computational code, a well-formed software architectural solution could support high performance communications via first-class explicit connectors.

In the rest of this paper, we will first explore the pertinent aspects of high-performance scientific applications including structure and typical memory access patterns. We will then discuss the role of connectors in this domain, considering both the evolution of communications in parallel computation and modern examples of high-performance connectors. Finally, we will discuss the services which a high performance connector must provide in order to support parallel programming at a high level.

[1]In an American Mathematical Association prospectus published in July 1965, it was claimed that over 1,700 special programming languages had already been developed [8]. Today, very little is know about the vast majority of these languages.

2. SCIENTIFIC APPLICATION DOMAIN

Scientific applications are generally defined as computer programs that attempt to model physical phenomena, giving rise to a scientific investigation in which empirical observation, mathematical models, computer algorithms and simulation results are used to cyclically establish and verify human understanding.

The structure of these programs is directly influenced by the application of a transformational algorithm to an n-dimensional set of data in a series of discrete time-steps. This structure usually consists of a central control loop governing time-steps and a nest of loops controlling data access (usually of the same dimension as the data). Algorithms can be data access dependent or independent, though they tend to be time-step dependent.

An example of the process of modeling physical phenomena in a scientific application is molecular dynamics simulation [12]. Classical Newtonian physics gives us the following equations for the velocity and acceleration of molecule i at time t given its trajectory, $\overrightarrow{r_i}$, in space:

$$\begin{aligned} t \in \Re \quad &\mapsto \quad \overrightarrow{r_i}(t) \in \Re^3 \\ \overrightarrow{v_i}(t) = \frac{d\overrightarrow{r}}{dt} \qquad & \overrightarrow{a_i}(t) = \frac{d\overrightarrow{v}}{dt} = \frac{d^2\overrightarrow{r}}{dt^2} \end{aligned} \tag{1}$$

These equations, along with models for the potential energy of molecules such as Lennard-Jones [12] give a mathematical basis for the observable phenomena of molecules interacting in three dimensions. A common discretization of these equations which can yield a computable algorithm is the velocity-Verlet algorithm [12] (discretization to time-steps introduces a Δ into the algorithm). When these equations are formulated as a computer algorithm, the common structure of scientific programs emerges. If we assume an array for storage of both molecule position and velocity, there are multiple un-nested data access loops which can be seen in Figure 1.

```
Initialize $(\overrightarrow{r_i},\overrightarrow{v_i})$ for all $i$
Compute $\overrightarrow{a_i}$ as a function of $\{\overrightarrow{r_i}(t)\}$
for stepCount:1$\rightarrow n$    $\leftarrow$ control loop
  $\overrightarrow{v_i} \leftarrow \overrightarrow{v_i} + \overrightarrow{a_i}\Delta/2$ for all $i$    data ↻
  $\overrightarrow{r_i} \leftarrow \overrightarrow{r_i} + \overrightarrow{v_i}\Delta$ for all $i$    data ↻
  Compute $\overrightarrow{a_i}$ as a function of $\{\overrightarrow{r_i}\}$ for all $i$    data ↻
  $\overrightarrow{v_i} \leftarrow \overrightarrow{v_i} + \overrightarrow{a_i}\Delta/2$ for all $i$    data ↻
endfor
```

Figure 1: Pseudocode example of the velocity-Verlet implementation of a molecular dynamics simulation illustrating the central control loop with multiple data access loops.

Though this pseudocode does not seem challenging to implement in an executable form, useful science requires the execution of this algorithm on datasets that challenge the performance of today's computers. Simulating a one molar concentration of molecules would require storage of over 5.4×10^{24} floating point numbers and more time than the age of the universe given an execution speed of one million instructions per second.

Though molecular science discoveries have shown that simulation of this large a number of molecules in not needed to accurately simulate physical phenomena, realistic simulations running in reasonable execution times still require parallel supercomputers, leading the developer not only to program the psuedocode in Figure 1, but to also consider parallel data access, local caching (assuming a distributed memory machine), communications of boundary conditions, and a host of other issues specific to a parallel version of this simulation. In the next section, we will explore how these issues can be addressed in a high performance software architecture, specifically focusing on the services which should be provided by this architecture's connectors.

3. DEVELOPING AN ARCHITECTURE

Based on the regular structure of scientific programs, we can easily establish the means of utilizing component-based software development for high performance scientific applications. Because a transformational algorithm is applied at multiple timesteps, there is usually a dependence between control loop executions, but internal data access loops can often be encapsulated in a component wrapper and replicated in order to elicit parallelism in the resulting software system [17].

While connector elements are typically described as the locus of communications, holding a secondary position to components in traditional software architectural development, they are much more than system service wrappers in high performance computing. In [9], Mehta, et. al. showed that a connector can have an arbitrarily rich internal structure supporting advanced services such as caching and priority-based delivery. In order to establish the connector "services" which should be provided for high-performance scientific application architectures, it is first important that we analyze communications support in concurrent programming.

3.1 The Evolution of the Connector

Since the advent of modern computers, developers have used concurrent programming to build everything from real-time controllers to time-sharing systems. When computer scientists first systematically explored the notion of concurrent program execution, researchers such as Dijkstra quickly established the need for synchronization of shared variables [3]. Shared variables with protected access via synchronization primitives like locks and semiphores were the first form of communications between concurrent processes.

An evolution of synchronization technology came with the concept of monitors [6, 7]. Rather than rely on a collection of primitives for safe communication, monitors handle access of shared data by allowing only one concurrent process to execute a monitor's methods at any given time, guaranteeing deterministic access to shared data (see Figure 2). In many ways, monitors are implicit software connectors allowing concurrently-executing components to communicate via shared memory. Much like software buses or blackboard systems, monitors allow for component interaction free from race conditions and other low-level memory management issues.

An interesting modern evolution of the high performance connector is transactional memory systems [13]. With transactional memory, the monitor concept is refined to a single *commit* method which allows for non-blocking (i.e., speculative) execution. In the next section, we will explore other services in addition to communications and synchronization a high performance connector could ideally support.

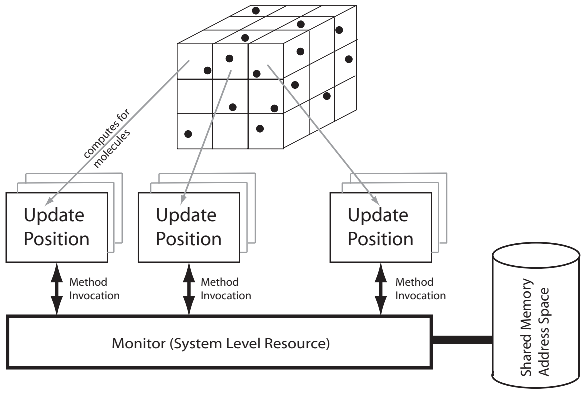

Figure 2: Multiple components calculate the trajectories of molecules in a segmented 3-D space using monitors for communications.

3.2 Connector Services

The evolution of the software connector suggests that within the domain of high-performance computing, connectors are the paramount architectural elements, subsuming components as the focal point for software engineering-oriented development support. This begs the question: what are the services which high performance connectors must provide? We have identified three key services:

- **Communications** – Efficient point-to-point communication is predominant in all software connectors, including high-performance connectors. Underlying delivery issues such as reliability and network topology should be obscured as much as possible from the developer.
- **Separation of Concerns** – Just as the component provides a wrapper around the transformational algorithms at the heart of scientific computing, connectors should encapsulate all communication.
- **Synchronization** – High performance connectors should guarantee safe access to data, arbitrating between components in such a way as to provide deterministic performance.

4. CONCLUSION

In this paper, we have explored the ability of software architectures to provide an abstraction for parallel programming paradigms often encapsulated in either domain-specific languages or parallel library routines. Unlike domain-specific languages, software architectures do not require investment in new languages by the developer, reducing the risk of non-acceptance. Additionally, because it is a high-level abstraction, software architecture does not cause undue code obfuscation but rather provide the developer with a more concise model with which to aid development.

The evolution of parallel programming paradigms from simple synchronization primitives to implicit connectors like monitors and transaction commits suggest that explicit high-performance connectors are a logical progression of existing parallel constructs. We have begun the study of complex connectors suitable for this problem domain.

5. REFERENCES

[1] D. Cann. Retire fortran?: a debate rekindled. *Commun. ACM*, 35(8):81–89, 1992.

[2] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[3] E. W. Dijkstra. *Cooperating sequential processes*. Technological University, Eindhoven, The Netherlands, 1965.

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Programming with the Message Passing Interface*. MIT Press, 1999.

[5] S. Z. Guyer and C. Lin. Broadway: A software architecture for scientific computing. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, pages 175–192, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[6] P. B. Hanson. *Operating System Principles*, chapter Class Concept, pages 226–232. Prentice Hall, Englewood Cliffs, NJ, 1973.

[7] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[8] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[9] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM Press.

[10] D. E. Perry and A. L. Wolf. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes*, October, 1992.

[11] D. E. Post and L. G.Votta. Computational science demands a new paradigm. *Physics Today*, 58(1):35–41, 2005.

[12] D. Rapaport. *The Art of Molecular Dynamics Simulation*. Cambridge U. Press, Cambridge, UK, 1995.

[13] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[14] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[15] B. Stroustrup. The pivot - a brief overview. Presentation. Workshop on Patterns in High Performance Computing. Urbana-Champaign, Illinois, May 2005.

[16] W. Tracz. Dssa (domain-specific software architecture) pedagogical example. *ACM SIGSOFT Software Engineering Notes*, July, 1995.

[17] D. Woollard, N. Medvidovic, W. Yamada, and T. Berger. Software engineering for neural dynamics: A case study. In *Proceedings of the First International Workshop on Software Engineering for High Performance Computing System Applications*, Edinburgh, Scotland, May 2004.