

An Architectural Style for High-Performance Asymmetrical Parallel Computations

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

Nenad Medvidovic
University of Southern California
Computer Science Department
Los Angeles, California 90089
nen@usc.edu

ABSTRACT

Researchers with deep knowledge of scientific domains are becoming more interested in developing highly-adaptive and irregular (*asymmetrical*) parallel computations, leading to development challenges for both delivery of data for computation and mapping of processes to physical resources. Using software engineering principles, we have developed a new communications protocol and architectural style for asymmetrical parallel computations called ADaPT.

Utilizing the support of architecturally-aware middleware, we show that ADaPT provides a more efficient solution in terms of message passing and load balancing than asymmetrical parallel computations using collective calls in the Message-Passing Interface (MPI) or more advanced frameworks implementing explicit load-balancing policies. Additionally, developers using ADaPT gain significant windfall from good practices in software engineering, including implementation-level support of architectural artifacts and separation of computational loci from communication protocols.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures

Keywords

High-Performance Computing, Asymmetrical Parallel Computations, ADaPT.

1. INTRODUCTION

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. A growing number of physicists, biologists, chemists, and computer scientists have developed highly-adaptive and irregular parallel applications that are characterized by compu-

tational intensity, loosely-synchronous parallelism and dynamic computation. Because the computation time of each parallel process varies significantly for this class of computation, we shall refer to them as *asymmetrical* parallel computations. Adaptive mesh refinements for the simulation of crack growth, combinatorial search applications used in artificial intelligence, and partial differential equation field solvers [2] are examples of asymmetrical computations.

While supercomputing platforms available to us continue to increase performance, our ability to build software capable of matching theoretical limits is lacking [8]. At the same time, researchers with significant depth of knowledge in a scientific domain but with limited software experience are confounded by the interface bloat of libraries such as the Message-Passing Interface (MPI), which has 12 different routines for point-to-point communications alone [5].

Would-be practitioners of high-performance computing are introduced early to the mantra of optimization. The myth that high-level concepts inherent to software engineering principles, such as “separation of concerns,” result in inefficiencies at the performance level has caused these researchers to eschew best practices of traditional software development in favor of highly-optimized library routines.

We contend that a sound software engineering solution to asymmetrical parallel computations provides decoupling of connectors from computational loci and reduces the complexity of development for the programmer while still providing an efficient solution both in terms of load-balancing and message-delivery. In this paper, we present such a solution.

In the next section, we will discuss our motivations for creating the ADaPT protocol and architecture, including the load-balancing inefficiencies of “optimized” communications libraries when computing asymmetrical parallel computations. We will then present ADaPT, a communications protocol and associated software architecture for asymmetrical computations. Additionally, we will present analysis which shows ADaPT’s ability to outperform both MPI and other load-balancing frameworks using traditional work-sharing strategies. We conclude with an overview of future research opportunities afforded by ADaPT.

2. MOTIVATION

This work has been motivated by our experience with two key classes of existing approaches: use of optimized communications libraries such as MPI [4], and message-passing frameworks which implement load-balancing strategies based on work sharing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE’06, May 20–28, 2006, Shanghai, China.

Copyright 2006 ACM 1-59593-085-X/06/0005 ...\$5.00.

2.1 Message-Passing Interface

High-performance communications libraries such as MPI are optimized to reduce the bandwidth needed to communicate a large amount of data to subprocesses. In order to accomplish this reduction, collective calls in MPI are synchronous, causing barriers at data-distribution points in the software. When used to compute uniform parallel computations barriers are unobtrusive. In asymmetrical computations, however, an effective mapping of processes to physical resources contributes more significantly to wall-clock time to completion than efficient communications. For these computations, asynchronous communications are needed, despite increased bandwidth.

To illustrate this phenomena, let us consider a mapping of a large normalized population of computation times with a high level of variance onto a significantly smaller number of physical nodes (a strategy known as *overaggregation*).

The MPI library offers developers efficient use of bandwidth via collective scatter and gather commands. While bandwidth is conserved using these collective calls, analyses made by Gropp, et. al. and Skjellum [10, 4] suggest that most implementations of MPI’s scatter are built on top of MPI’s *rendezvous* protocol and result in a synchronous barrier at each subsequent distribution of data.

Since each process has variable computation time, a number of subprocesses will remain idle until the longest process completes during each of the scatters. In [1] we have shown that the smallest contribution to overall wall-clock time to completion made by this idle time is given as $n \times \bar{\mu}$, where n is the number of subprocesses and $\bar{\mu}$ is the mean of the computation times. In comparison, the wall-clock time saved using the collective calls to reduce bandwidth is negligible.

While these collective calls only consider bandwidth optimizations, it is clear that in asymmetrical parallel computations, process load-balancing across subprocesses is a more important optimization to pursue.

2.2 Load-Balancing Frameworks

Attempts to develop message-passing frameworks that can assist computational scientists in the development of asymmetrical parallel computations can be divided into two groups: *static* load-balancing frameworks and *dynamic* load-balancing frameworks. Because *a priori* knowledge of the computation involved in asymmetrical parallel computations is required of static load balancers, such frameworks are inapplicable to this class of problems.

Unlike static load balancers, dynamic load-balancing frameworks do not require information a priori and are able to re-deploy balanced distributions of data during program execution. Notable examples of parallel development frameworks which provide dynamic load-balancing are PREMA [2] and Charm++ [6]. Unfortunately, these frameworks often incur significant performance losses due to the introduction of barriers for load-balancing. Additionally, these frameworks do not provide explicit support for consistency of structure and development.

A software architectural solution can provide a number of benefits in addition to load balancing. Employing a sound software engineering principle, the separation of communication from computational elements shields the developer from the need to optimize communications and provides enforcement of architectural constraints. An added benefit is that architectural components reified as explicit implementation-

level artifacts allow for easy reconfiguration of software in principle. We will revisit this point below.

3. A NOVEL PROTOCOL

Recalling the motivating example in Section 2, we see that there are a number of keys to optimizing overaggregated asymmetrical parallel computations in addition to providing the developer with architectural constructs. Two overlooked aspects of performance optimizations that must be addressed in order to provide a truly efficient solution are *asynchronous load-balancing* and *event pattern optimization*. In addition to simply providing a load-balanced distribution, asynchronous load-balancing provides a best effort redistribution of processes without introducing a barrier to computation.

Event pattern optimization suggests that a protocol is capable of utilizing the predictability of future messages given analysis of past messages. During overaggregated parallel computations, a number of computations need to be distributed to each of the subprocesses over the course of the parallel computation, causing a pattern to emerge.

In order to incorporate each of these optimizations into a high-performance communications protocol, we have developed ADaPT, an Addaptive Data-parallel Publication/Subscription Transport protocol and software architecture. The thesis of ADaPT is that it is possible to exploit the sequence that emerges from sending multiple messages to each parallel process in order to reduce the overall wall clock time to completion of the computation while still making a best-effort to avoid sending messages to each subprocess to quickly for the process to buffer.

3.1 ADaPT Defined

We feel that for the purposes of this paper it is most helpful to define ADaPT’s protocol, architectural elements, and implementation.

3.1.1 Protocol

ADaPT views each parallel process as an independent software component (Worker) residing on a physical node capable of performing computations on data. Each Worker initiates parallel computation by subscribing to the main process (Master). The sequence that emerges from the sending of multiple messages to each Worker in the case of over-aggregation can be exploited to reduce the overall processing time at each physical node.

An important distinction between ADaPT and traditional publication/subscription systems is that unlike traditional pub/sub systems, ADaPT does not duplicate messages to service multiple downstream requests. Rather, it distributes messages uniquely from a queue in a round-robin fashion.

Upon receipt of a subscription, the Master publishes a message to the Worker. There is another divergence from traditional pub/sub systems at this point. The Master waits for another request from the subscribed Worker before publishing another message to that Worker. Using data from each subscribed Worker on its computation time, or μ , the Master tracks an average processing time, or $\bar{\mu}$.

Because the protocol is adaptive, when a predetermined number of messages have been sent to the Workers and a $\bar{\mu}$ has been calculated, the Master switches from this conservative phase to an aggressive phase during which it sends messages of the requested type to the process at the regular

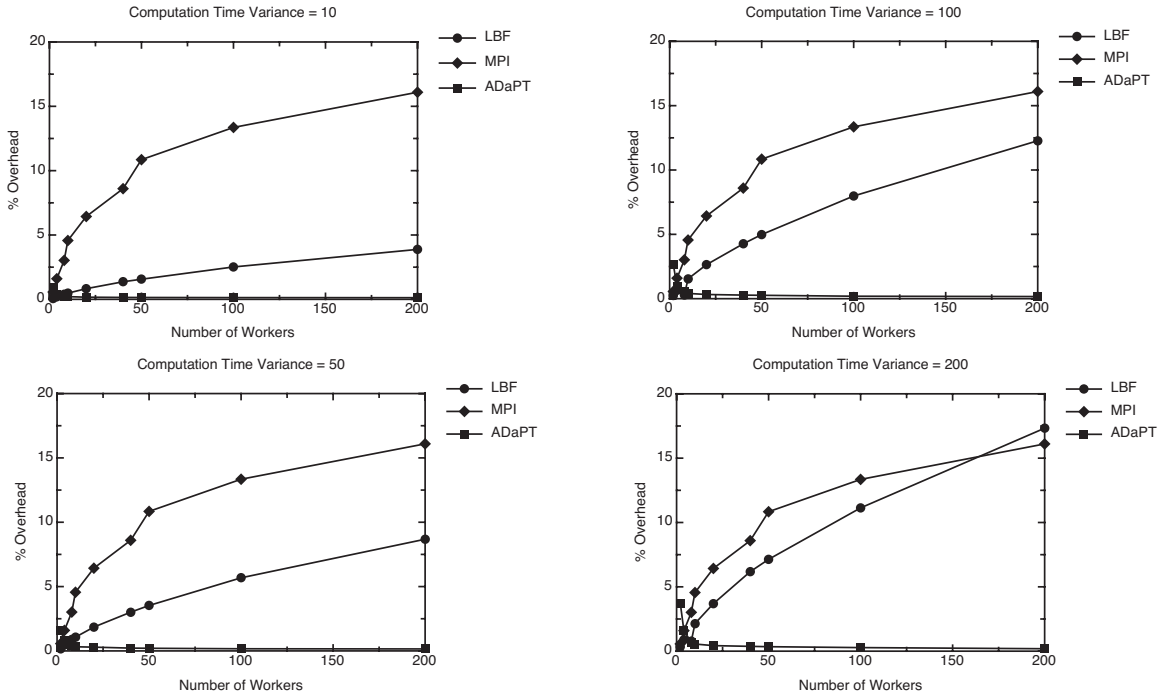


Figure 1: Monte Carlo simulations of overhead for asymmetrical computations exhibiting multiple variances.

interval dictated by $\bar{\mu}$.

Similar to MPI’s *eager* protocol, this phase of ADaPT can be too aggressive, flooding the process’s buffer (datasets in high-performance computing tend to be very large causing memory limitations to surface frequently). If the number of messages in the Worker’s buffer reaches a maximum, the Worker unsubscribes from the Master. After the Worker has computed each of the messages in its buffer, it re-subscribes to the Master, starting once again with the conservative phase of delivery as described above.

3.1.2 Architectural Model and Implementation

We have further codified ADaPT in a software architectural style [9]. In addition to Master and Worker components, the ADaPT connector utilizes an adaptive dispatcher to deliver messages to each subscribed Worker using the ADaPT protocol. The dispatcher uses a priority-based round-robin algorithm which utilizes the calculated $\bar{\mu}$ and attempts to saturate each Worker’s computation load without flooding the Worker’s buffer. This handler automatically switches between the conservative and aggressive phases. The key contribution of this connector is the encapsulation of underlying protocols, allowing the developer to focus instead on the computations to be performed.

Similar to the C2 software architecture [3], messages triggering computation travel downstream from one or more *Masters* to the ADaPT connector. Messages typed as results originating at *Workers* travel upstream through the ADaPT connector back to the *Masters*.

We have implemented these architectural rules through extensions to the Prism framework [7]. Prism-MW, a middleware designed to enforce architectural rules at the level of software artifacts, is a light-weight event-based framework consisting of a core set of functionality with handles

to extensible components, connectors, and event handlers. Topological rules for each architectural style are also enforced through overloaded methods for connecting artifacts.

3.2 Performance Analysis

In analyzing ADaPT’s performance in comparison to load-balancing frameworks as well as synchronous scatters and gathers using MPI, it is first important to define a base metric with which to compare protocols. This metric, the “natural rate” of parallel computation, is the sum of all individual computations to be completed divided by the number of nodes in the parallel computation. In this section we will present comparisons of protocols as measured by percentage overhead (calculated as the wall-clock time for the parallel process to complete minus the natural rate, divided by the natural rate).

In order to properly compare ADaPT’s ability to reduce message traffic as well as to efficiently map asymmetrical computations to physical resources, we developed a Monte Carlo simulation in which a normalized population of computations was delivered to virtual processors via three different communications policies/architectures and the percentage overhead was calculated for each. All message-passing costs were uniform across the network for each policy implemented.

MPI (collective calls) - Costs of synchronous scatters and gathers using MPI were modeled using equations from [10, 4]. In this policy each worker receives a computation via a *scatter* and returns via a *gather* before scattering the next subset until all computations are completed. This process is known as a multi-part scatter [1].

Load-balancing framework - The Monte Carlo simulation of the load-balancing framework uses work-sharing methods. All events are delivered to workers before they be-

gin processing and a barrier is periodically introduced. At this barrier, the workload is redistributed evenly between all processors. In order to idealize load balancing, the cost of this calculation was treated as negligible.

ADaPT implementation - Using the routing policies of ADaPT, this implementation assumes that workers are capable of buffering only two events and each worker is homogeneous. We made each of these assumptions in order to conservatively profile ADaPT's performance.

In each of four simulations, a normalized population of 1000 computations was generated with a mean computation time of 100 milliseconds and a variance of 10 milliseconds², 50 milliseconds², 100 milliseconds², and 200 milliseconds², respectively. For each simulation, the aggregation of the parallel computation (i.e, the ratio of workers to computations) was varied from 1:500 to 1:5.

Results of this comparison are shown in Figure 1.

4. DISCUSSION

4.1 Analysis and Evaluation Results

It can be seen in these plots that while ADaPT performs uniformly at all aggregations smaller than 1:100 (i.e., ≥ 10 workers), MPI collective commands and load-balancing frameworks decrease performance as the aggregation is reduced. For load-balancing frameworks, this is due to the increased volume of messaging required in order to re-balance the load across all processors at each barrier. In the presence of load-balancing, the idle time is significantly reduced, but the cost of rerouting messages to new processors makes ADaPT the better performer especially in high variance computations.

From these initial results, we feel that our implementation of ADaPT outperforms collective calls via MPI as well as load-balancing frameworks employing a full worksharing scheme for significantly varied aggregations and computation time variances. In Monte Carlo simulations, ADaPT produced a better mapping of computations to resources, reducing computational overhead to under 1% for aggregations less than 1:100. In the simulations of aggregations greater than 1:100, ADaPT does not perform as well as MPI or other load-balancing frameworks due to the increased percentage of time each worker's buffer remains empty before another event is pushed to the Worker at the calculated rate of computation. This situation seems of little consequence, however, in that data sets are seldom overaggregated to this extreme.

ADaPT offers a significant decrease in overhead for event delivery in parallel computations and also outperforms established load-balancing techniques for use with asymmetrical parallel computations. Additionally, ADaPT, through its implementation in Prism, offers developers architectural artifacts at the level of implementation, clear division between the computation loci (in the form of extensible Workers) and communications algorithms, and reduction of communications knowledge needed by the developer in order to implement asymmetrical parallel computations.

4.2 Future Work

While ADaPT is clearly an applicable architectural style to high-performance computing, we make no claim as to its monopoly of the field. In future work, we hope to build a more substantial architectural framework for high-performance computing which provides more underlying protocol choices

and further assists developers in code migration to new platforms including SMP and other shared-memory machines. We hope to demonstrate the ease of system design and implementation via architectures to the high-performance community without serious performance degradation, as is currently the prevalent though.

Further enhancements to the ADaPT protocol and architecture will include refinement of its topological constraints to encapsulate both data-parallel stages of computation and higher-level workflow stages using multiple layers of masters and workers connected between more advanced ADaPT connectors (themselves perhaps distributed across multiple physical nodes). Also, we hope to further investigate the tradeoffs associated with alternate unsubscription policies and the effects of "pumping" the parallel computation by modifying delivery rates to be faster than average computation rates.

This work was supported by the NSF 0312780 grant. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. The authors also wish to thank the anonymous reviewers for their helpful comments.

5. REFERENCES

- [1] D. Woollard et. al. Adapt: Event-passing protocol for reducing delivery costs in scatter-gather parallel processes. In *Proceeding of the Workshop on Patterns in High Performance Computing*, Urbana, Illinois, May 2005.
- [2] K. Barker et. al. A load balancing framework for adaptive and asynchronous applications. *Parallel and Distributed Systems, IEEE Transactions on*, 15:183–192, 2004.
- [3] R. Taylor et. al. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, June, 1996.
- [4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Programming with the Message Passing Interface*. MIT Press, 1999.
- [5] S. 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] L. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.
- [7] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, March, 2005.
- [8] D. Post and L. Votta. Computational science demands a new paradigm. *Physics Today*, 58(1):35–41, 2005.
- [9] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [10] A. Skjellum. High performance mpi: Extending the message passing interface for higher performance and higher predictability, 1998.