High-Level Reactive Support for Scalable Distributed Systems

James E. Lumpp, Jr.[‡], James Griffioen[†], Rajendra Yavatkar[†],
Thomas Kay[†], Christopher Diaz[†]

Department of Electrical Engineering[‡]
Department of Computer Science[†]
University of Kentucky
Lexington, KY 40506, USA

[jel,griff,raj,kay,diaz]@dcs.uky.edu

Abstract

To make large-scale distributed systems practical for a broader user base, we are investigating a new framework to support and aid the development of "reactive objects." Reactive objects offer large-scale applications the ability to automatically adapt to the state of the system as well as the behavior of the application. The system automatically gathers, condenses, and provides access to performance statistics needed by the distributed application for adaptation. Reactive objects automatically respond to changes in the system to improve performance. To hide adaptation from the programmer, the system provides default adaptation policies and mechanisms that can be included in reactive objects. By separating policies from algorithms and data structures, the system encourages reuse of parallelized components. Ultimately, our goal is to create a library of common scientific reactive objects that can be directly incorporated into parallel applications.

1 Introduction

Grand Challenge Problems require massive computational resources with sustained rates in the teraflops [NCO96]. Scientific applications such as global climate modeling, quantum chromodynamics, and computational fluid dynamics are highly parallel and require massive computational power and data storage. Massively parallel numerical algorithms that use domain decomposition methods to solve the partial differential equations found in such applications have been quite successful [Cai93]. Such high levels of performance can only be achieved by a system consisting of hundreds or thousands of processors.

However, cost/performance ratios make it unlikely that such power will be achieved by a single, massively parallel architecture. Such systems are typically built from special purpose hardware and suffer from scalability limitations. In contrast, large-scale, loosely-coupled distributed systems consisting of commodity high-performance workstations interconnected by high-speed networks appear quite promising. Scalability is achieved by introducing new workstations or networks of

This research was supported in part by the National Science Foundation under Grants CDA-9502645, CDA-9320179, CCR-9309176 and the Advanced Research Projects Agency under Grant DAAH04-96-1-0327.

workstations to the system. These systems use off-the-shelf components (workstations) that can be easily upgraded to take advantages of rapid changes in technology. Moreover, such systems already exist and are widely available in most business, governmental, and academic settings. For these reasons, we expect that large scale networks of workstations will emerge as the architecture of choice for many grand challenge caliber applications. However, we do not expect thousands of machines to be connected to a single high-speed local area network. Therefore, to obtain the requisite computational power users will need to access machines on several different local area networks connected by wide area networks spanning substantial geographical distances. Finally, we expect that these systems will have dual use, serving as general purpose computing environments and also as powerful massively parallel compute engines.

Unfortunately, the geographic distribution of processing power, memory, and secondary storage complicates, rather than simplifies, the design of applications. Application design is further complicated by the static and dynamic variability of the system. The large-scale systems we envision will inevitably consist of dissimilar machines with widely varying processor speeds, memory sizes, and disk speeds. Similarly, the networks that interconnect the machines will have widely varying bandwidths and latencies. In addition to these static dissimiliarities, dynamic runtime differences will also occur. Transient users performing general purpose tasks will cause the load on certain processors and networks to vary dynamically. Our goal is to develop a programming environment that masks the programming difficulties associated with a dynamic distributed environment but yet makes optimal use of the available and constantly changing resources.

To make large-scale distributed systems usable by the scientific community, we are investigating a new framework to support and aid the development of "reactive objects". Reactive objects offer large-scale applications the ability to automatically adapt to the state of the system as well as the behavior of the application. The system automatically gathers, condenses, and provides access to performance statistics needed by the distributed application for adaptation. Reactive objects automatically adjust to changes in the underlying system, dynamically optimizing the application for the current environment. To hide adaptation from the programmer, the system provides default adaptation policies and mechanisms that can be included in reactive objects. The system encourages reuse of parallelized components by separating policies from algorithms and data structures. Ultimately, our goal is to create a library of common scientific reactive objects that can be directly incorporated into parallel applications. Novice users will use reactive objects as high-performance "building blocks" for developing applications. Experienced users will use services of the monitoring system and predefined reactive object components to develop objects that "plug in" to the reactive object support system.

The remainder of the paper is organized as follows. Section 2 begins by reviewing related work. Section 3 then briefly overviews the Unify distributed shared memory system that serves as the scalable distributed operating system platform for our reactive object research. Having laid a background and description of the environment, section 4 identifies the problems that must be addresses by a large-scale reactive distributed system. Section 5 then presents the architecture and design of our reactive object system and section 6 concludes by describing the status of the reactive object system.

2 Related Work

Detecting and reacting to changes in the state of the system requires that an application monitor the system's run-time performance. Researchers have investigated both hardware and software instrumentation approaches to observe and record state changes in the system [SKB89]. Hardware instrumentation has the advantage of introducing minimal delay or intrusion to the target system [MLCS90] and can be invaluable for measurements not attainable via software (e.g., cache hit ratios). However, hardware monitors are inflexible, low-level (i.e., can't monitor process-level events), and often prohibitively expensive. Software instrumentation has the advantage of flexibility, high-level event monitoring, and no expensive hardware. As a result, virtually all parallel/distributed debuggers [MH89] and performance evaluation [HML95] tools rely on software instrumentation. Unfortunately, software monitoring can adversely affect performance and correctness. Event-based models of monitoring [Bat89, MLCS90] have been proposed to gather information "on-the-fly" [Sch89, HKMC90, MC91, JK93]. Event-based monitoring systems use *predicates* defined on subsets of system state to recognize state changes [BW83, LCSM90]. Our work builds on past work in event-based, on-the-fly, predicate recognition systems. In addition, our research attempts to identify and only monitor the events of interest to large-scale distributed applications.

Techniques to alter the behavior of running programs in response to changes in the underlying computing system or unforeseen algorithmic problems have been studied in several contexts. The former was studied primarily in the area of migration and the latter in the area of program steering.

Process migration and load balancing algorithms are well-known examples of programs reacting to run-time changes in system load [Smi88]. More general approaches, such as MESSIAHS [CS94], allow users to develop application-specific distributed scheduling algorithms. Process migration involves the transfer of work, data, and process state which is costly and time consuming. Consequently, process migration is typically used in situations where decomposition is coarse-grained and migrations occur infrequently so that costs are amoratized over time. The coarse-grain nature often causes the resulting "balanced load" to be from from truely balanced. Migration of lightweight processes, such as Filaments [FLA94], allows for a more fine-grained work decomposition. Assuming the application can be decomposed into many fine-grained pieces of work, the system can react to system load changes quickly and effectively. Others have proposed modifications to parallel programming languages or compilers to allow dynamic scheduling of application modules to processors [Luc92]. Both approaches require that the application be decomposed into many fine-grained parallelizable components.

To react to algorithmic errors or mismatches between the algorithm and its current input data, several researchers have proposed the use of interactive steering systems [EGSM94, GEK+95, VS95, Sos92]. Interactive steering systems put a knowledgable user in the loop, allowing the user to dynamically alter the algorithm to redistribute the work or "steer" the algorithm to the data. The following briefly describes a few steering systems most closely related to and influencing our work.

The Falcon and Progress systems [EGSM94, GEK+95, VS95] allow a user to monitor and then manipulate arbitrary program variables via *sensors* and *actuators* that control or steer the application at runtime. The Meta toolset developed for ISIS [MW91] takes a similar approach. The state of the system is monitored by inserting probe calls in the application to sense the state of the application where the programmer determines it will be most productive. A separate monitoring thread records and analyzes the state information at the probe events. The application writer can also place actuators at strategic points in the computation where it is safe to steer the application. The monitor process communicates with a GUI that displays the progress of the application and allows the user to dynamically alter the program's state or direction. This technique provides a flexible and powerful framework for developing steerable applications, but requires the application developer to provide all the sensing and actuator code needed to steer the object. The adaptive system that we present in this paper provides a higher-level interface and default monitoring capabilities that could be implemented on the low-level support provided by a system such as Progress or Meta.

The Dynascope system [Sos92] provides a framework for constructing *directors* that steer an application. Portions of the application are interpreted while other portions run on the native machine. Only interpreted code can be directed. The interpreted code sends all events to the director who analyses the current program execution and modifies its behavior. Director access to and modification of application state is only supported at the machine-level. No direct parallel or distributed support is present.

The majority of current steering systems target tightly-coupled systems consisting of uniform dedicated nodes and high-speed interconnection networks. In such an environment, there is little variance or change in the system state, implying that poor performance stems primarily from algorithmic problems and requires steering. Consequently, most of the research efforts focus on monitoring and controlling the program's state rather than monitoring the system's state and conforming to it. In a large-scale distributed system, system state changes are often the primary factor influencing performance. Consequently, distributed systems must provide convenient abstractions to help applications adjust to system state changes. The work presented here provides the necessary framework to obtain information about and react to changes in the system state information.

Although the Mentat system [GSN93, Gri93] does not support interactive steering, it does provide a high-level interface to parallel programming constructs by encapsulating the complexities of parallel programming in parallel objects. Applications invoke the services of high-level mentat objects which hide the parallelism. Mentat statically processes mentat objects to determine the execution order of the objects based on data dependencies. The resulting blocks are then executed asynchronously with data forwarded between objects as required. Our approach is similar to Mentat in that we also encapsulate the complexity of the underlying system in higher level objects. However, we allow the objects to dynamically and continuously adjust to changes in system load or configuration.

Finally, there are several examples of application-specific implementations of adaptation including distributed branch-and-bound algorithms [LM92] and N-body simulations [GKS94]. Others have developed techniques to dynamically change the algorithm or communication model to meet real-time deadlines of a particular real-world environment [LMnS90]. Such techniques can be encapsulated within the reactive objects we describe here.

For high-performance distributed environments, the dynamic nature of the network and hosts can make the overall behavior of the system difficult to predict. A low-level steering system would require the programmer to understand the details of their application, the characteristics of a highly dynamic and complicated distributed system, and the typical workload that is expected from external sources. If the programmer can be provided with some higher level abstractions that will automatically adapt to changes in the distributed system or application, the programmer can focus on solving the problem rather than targetting or reacting to a runtime environment. For example, a programmer should not have to include probes and actuators throughout their program to watch for nodes that become heavily loaded by other users. It is preferable to provide objects that automatically respond to such conditions by redistributing some work to another node or avoiding links that are performing poorly.

3 Unify Overview

We are currently investigating reactive support for distributed applications in the context of the Unify distributed system [GYF95]. The objective of the Unify project is to develop a scalable multicomputer linking hundreds or thousands of high-performance machines in geographically distant locations. Such large-scale parallelism is necessary to achieve the massive computational power required by today's grand challenge problems [NCO96]. To provide a convenient programming model, Unify's goal is to support a highly scalable distributed shared memory programming paradigm. Conventional DSM approaches are inappropriate in such large-scale geographically distributed environments. To achieve scalability in such an environment, Unify supports new shared memory abstractions and mechanisms that (1) mask the distribution of resources, (2) limit/reduce the frequency of communication and the amount of data transferred, (3) hide the propagation latencies typical of large-scale networks (e.g., by overlapping communication with computation), and (4) support large-scale concurrency via synchronization and consistency primitives free of serial bottlenecks. Ideally the system must provide convenient data sharing abstractions that exhibit performance and scalability similar to that of existing large-scale message passing multicomputers such as PVM [Sun90] and MPI [For94]. The following briefly highlights the salient features of Unify:

Single Address Space: A single virtual address space, shared by all applications, allows applications to conveniently and efficiently share structured address-dependent data (such as trees and linked lists) as well as other address-independent data.

Multiple Memory Types: Our design supports three basic memory abstractions for shared data. Random access memory is directly addressable. Sequential access memory is accessed in a read/front, write/append fashion. Associative memory is accessed via <key,value> pairs. Sequential access and associative memory can often be supported with weaker spatial consistency guarantees (described below) that can be implemented more efficiently than random access memory segments.

Multiple Grades of Consistency: The Unify design supports a set of consistency management primitives that allow an application to select the appropriate consistency semantics from a spectrum of consistency protocols, including "automatic" methods where the operating system enforces consistency and "application-aided" methods where the user defines consistency checkpoints. Several existing DSM systems have demonstrated the benefits of weak application-aided temporal consistency models. In addition, Unify supports weak automatic methods where the memory becomes consistent after some time lag T. This type of automatic consistency is particularly useful for applications that can detect stale data such as Grapevine [BLNS82].

Our design also introduces a new consistency dimension called "spatial" consistency. Spatial consistency determines the relative order of the contents of the replicas of a segment. For many distributed applications that use keyed lookups or sequential access, the order of the data items within a segment is unimportant; only the values of individual data items are important. Spatial consistency allows efficient implementation of such applications.

Scalable Synchronization Primitives: Most DSM designs support locks, semaphores, and/or barriers as the basic synchronization methods. However, these synchronization primitives pose a serious bottleneck, particularly for large systems with high latencies. To provide efficient synchronization in the presence of long and varying latencies, Unify proposes the use of a modified form of event counts and sequencers. For a large class of applications, event counts can result in reduced communication and greater concurrency. In particular, conventional synchronization methods require that all participants observe the synchronization event simultaneously. Event counts allow participants to observe the event at different times, effectively relaxing the communication constraints and allowing greater concurrency. Moreover, conventional synchronization primitives can be implemented via event counts.

Hierarchy of Sharing Domains: We believe that sharing in a large scale distributed multicomputer will follow the principle of locality. To exploit localized sharing and communication, we partition the set of hosts into "sharing domains". Each sharing domain uses a separate multicast group to reduce the cost of intra-domain information sharing. Sharing domains distribute the burden of information retrieval and distribution by allowing any member of the domain to issue or answer inter-domain requests (addressed to multicast groups). Every host consults the hosts in its local sharing domain before going outside the domain for information. Therefore, as soon as one host obtains cross-domain information from a site in a remote domain, the information effectively becomes available to all other hosts in the sharing domain.

Reliable Multicast Support: Although protocols such as ATM and RSVP provide quality of service guarantees, they do not guarantee end-to-end reliability. In fact, classical IP performance over ATM is currently a topic of much research because of TCP/IP's poor performance and high data loss rates. Distributed applications require a reliable multicast mechanism that not only delivers data reliably, but also attempts to synchronously deliver data to all recipients. Distributed applications typically block until the multicast information has been reliably delivered to all participants. Such delays can severely affect performance.

To achieve reliable, scalable, efficient dissemination of shared data or synchronization information, Unify supports reliable multicast via a tree-based multicast transport protocol (TMTP) [YGS95]. TMTP builds on the efficient delivery of IP multicast (possibly via the MBONE) which is widely available. To provide reliability TMTP uses a combination of sender and receiver initiated approaches that constructs a separate control tree to handle error and flow control. As a result, retransmissions are handled locally in a timely fashion avoiding retransmissions to the entire Internet. The use of localized NACKs with NACK suppression ensures quick response to lost messages with minimal overhead. Finally, batched positive acknowlegements reduce Internet traffic and eliminate the packet implosion problem.

The utility and scalability of large scale multicomputers can only be demonstrated by developing and evaluating real-life, large-scale parallel and distributed applications. We have implemented the Unify system as a runtime library on Unix workstations and run tests involving a significant number of hosts. DSM applications link with the library to create shared segments with appropriate memory types and consistency semantics.

We have implemented several common DSM applications including matrix multiplication, SOR, MP3D, and Water and observed impressive speedups in our local environment. The library is currently available to and in used by our scientific colleagues with computational complex problems. We are currently working to provide them with a high-level reactive object support system that will provide maximal performance despite the highly dynamic changes in the runtime environment.

4 Conforming Applications to Networks of Workstations

Programming distributed shared memory applications for large-scale networks of workstations is complicated by several factors that are not present or significant in multiprocessor environments. Multiprocessor systems do not exhibit the variability in processor speed, memory capacity, network loads or network routes that a loosely-coupled system experiences. Each time an application executes, it may find itself in a new environment. The environment could even change while the application is running. Processor loads can change as other applications come and go, and network bandwidth, latency, packet loss, and routes can vary dynamically in response to other traffic in the system. To make effective use of the computational power in the Unify environment, the system must help the application adapt and conform to changes in the environment.

Our experience with the Unify system has shown us that there are many reasons why an application may not achieve the desired speedup. In almost all cases, the poor performance stems from a mismatch between the algorithm and the distributed environment where the algorithm is executed. For example, assigning spatially related partitions of a matrix to non-neighboring machines can result in an excessive number of high-latency messages. The second and less common source of poor performance arises from a mismatch between the algorithm and the data. A common example of this type of mismatch is the use of fixed partitions on a sparse matrix. In both cases described above, slight runtime modifications to the algorithm will often rectify the problem and boost performance.

Our goal is to develop the infrastructure needed to recognize the characteristics of the runtime environment that influence performance and monitor those characteristics during program execution. The system must then provide the necessary hooks for distributed applications to procure information about the environment and change the algorithm or the data structure's implementation to conform or react to changes in the environment.

To determine which environmental characteristics we need to monitor, we must identify the environmental changes that can negatively affect performance. Roughly speaking, there are two environmental factors that limit the speedup a distributed application can achieve: *computational overload* and *non-optimal communication*.

Computational overload occurs when the work assigned to a machine exceeds the machine's capabilities. Computation overload is common because most distributed computing systems consist of heterogeneous workstations with widely varying computational power. This disparity arises from the incremental growth of a system over several years and the need for heterogeneity to run certain software packages. The uneven distributation of computational power is complicated by the fact that transient users can dynamically increase or decrease the load on certain machines. The base-line differences and dynamic variability in compute power means that an algorithm will rarely partition the work correctly and thus must continually re-partition the work over the lifetime of the application. Machines will also have different amounts of physical memory, and the amount available to the application will change dynamically. Even if all machines have identical processors, insufficient memory at some nodes will lead to paging that degrades the node's performance.

Non-optimal communication occurs when closely related logical nodes are assigned to distant physical machines or if inappropriate DSM primitives are used. Even if domain decomposition is calculated such the correct amount of work is allocated to each machine, the interactions between domains may not match the underlying network topology or network routes. Unlike multiprocessor machines, the network topology and current network routes are rarely known and can change dynamically. This makes it difficult to ensure that spatially related domains are located on neighboring machines. If frequently interacting domains are placed on distant machines, excessive latency, packet loss, and low bandwidth can occur. Non-optimal placement of processes on machines may also mean that efficient mechanisms such as link-layer multicasting cannot be used. To combat this, the system must provide information about the current topology and network performance to aid in the assignment of decomposition domains to machines. Finally, the high-latencies of wide area systems can have an enormous impact on DSM synchronization costs which tend to be latency dominated. In many cases, synchronization events can be traded for larger granularity. In a wide area environment, performance may actually be optimized by reducing the number of synchronization events and increasing the size of shared regions. Modifying the way in which the application uses the DSM system can produce significant speedups.

The following sections describe our reactive system architecture. The system monitors the environmental changes described above and provides default and user-specific mechanisms to react to the changes.

5 A Reactive Object System

To provide distributed applications with the information necessary to effectively map the application onto the available distributed resources, we propose a reactive object system architecture pictured in Figure 1.

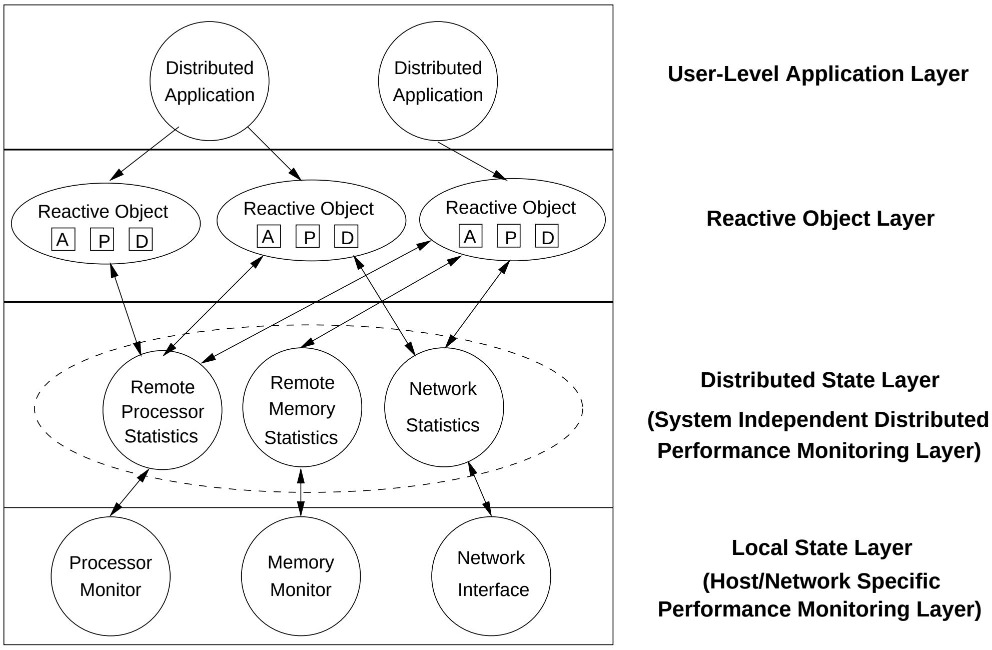

Figure 1: Organization of the Reactive Objects System.

The objective is to provide a high-level interface for writing distributed applications that can dynamically react to the environment. The support system is organized into three layers: a host monitoring layer, a distributed state monitoring layer and a high-level reactive object layer. In general, users' applications will interact directly with objects in the reactive object layer. The reactive object layer supports parallelized objects that hide all adaptation from the user. For certain applications, the application developer may need to create new reactive objects. Only in rare cases do we envision users dealing with or modifying components in the monitoring layers.

The goal of the monitoring layers is to dynamically gather performance information and feed it to the reactive object layer where decisions about adjustments will be made. Monitoring of system state is divided into two layers, one responsible for observing the state of the local machine and one responsible for the state across machines. These layers provide state information to the reactive objects layer which responds to the state of the system to provide maximum performance to the target application. In general, users will not modify the monitoring layers.

5.1 Local State Layer

The *local state layer* monitors the state of the local machine and presents the collected information to the upper layers. The performance information required by the distributed state and reactive object layers is specified as *predicates* defined in terms of local state introduced in our previous work [LCSM90]. For example, "load average > 2" or "change in load average" would be specified as partial predicates used to determine when load balancing modifications are required. When the specified events occur on the local host, the predicates will evaluate true and trigger actions [MLCS90] that notify the upper layers or pass state information to the upper layers. Local layer objects on each machine measure processor statistics, memory system behavior and network utilization. Processor statistics include the amount of processor time, typically measured in instructions per second, that the application has been receiving. Processor statistics also monitor application run time, sub-divided into application compute time, communication (system) overhead time, and missed time (time used by other applications). Memory measures include delays arising from paging within the the application. Memory performance can be useful in adjusting both data structures or algorithms. The network interface object records information about the bandwidth and latency of current connections as well as retransmission and dropped packet rates which can be used to infer information about the network topology and link performance.

5.2 Distributed State Layer

The distributed state layer supports predicates that can be local, non-local, or global. Reactive objects present the distributed state layer with local, non-local or global predicates. Local predicates are passed through to the local state layer. Non-local predicates specify states spaning more than one machine (e.g., "the load of the local processor 50% greater than the load on a neighbor processor") while global predicates specify overall system state (e.g., "all processors idle"). The objects that comprise the distributed state layer communicate between machines to evaluate predicates and notify the reactive objects when events occur.

In addition to event recognition, distributed state layer objects also tabulate and maintain state information that reactive objects can query. For example, a reactive object may request network topology or interconnection information such as the latency recently experienced between two nodes of the system. The network statistics object probes other hosts in the system to determine the expected latency, loss rate, or possible bandwidth to these machines. Given expected network performance, the system constructs a pseudo-topology indicating which machines are distant and which machines are neighbors and comprise a Unify sharing domain. A reactive object might use this information to request the four processors with the least total latency among the fully connected set. The reactive object can then use the four selected machines to run a communication intensive sub-task.

Although the distributed state layer knows nothing about the application, it does understand a limited number of common high-level DSM concepts such as shared memory regions, synchronization events, data transfer events, sharing domains, page fault rates, processor speeds, etc. Consequently the distributed state layer can separate things such as communication costs into synchronization communication costs and data transfer communication costs.

Given the ability to obtain performance information about the environment in which an application finds itself, we must develop a application interface to simplify interaction with the monitoring system. The Reactive Object Layer provides the application with self-adapting distributed objects.

5.3 Reactive Object Layer

The monitoring layers provide all the information and notification mechanisms necessary for an application to conform to the current environment or react to changes in the environment. However, these capabilities are at an inappropriate level of abstraction to support a computational scientist developing an application. Such users require higher-level support to aid them in the development of a reactive program. To that end, the system includes a high-level reactive object layer that hides performance monitoring and adaptation from the application developer.

A distributed application will typically interface with the reactive system by invoking the services of predefined reactive objects. Each reactive object implements a reactive parallelized version of some commonly used abstraction. Examples of common abstractions might include a matrix abstraction with methods to perform LU decomposition, successive over-relaxation, Cholesky factorization, averaging, and min/max element identification, an image processing matrix abstraction with methods to perform contrast enhancement, thresholding, segmentation, correlation, and edge detection, or a queryable storage abstraction with insertion, deletion, and query methods that might be implemented with a hash table, B-tree, linked list or array depending on which is the most efficient given the current state of the system. Given a library of predefined reactive objects, the application developer can concentrate on the problem to be solved rather than reacting to changes in the system. In the worst case, a developer may need to implement new application-specific reactive objects. Even in this case, the reactive object layer provides reusable policies and data structures to simplify the coding of a user defined reactive object.

The reactive object layer responds to method invocations from user-level applications and also communicates with the underlying global state monitoring system. Communication with the monitoring system occurs in one of two ways. A reactive object can register with the distributed state layer to be notified when an event occurs. This is analogous to a "performance interrupt" where the reactive object is informed that the monitored condition has occurred. For example, an object may register via a predicate specification to be informed of an excessively slow host. Once notified of this condition, the reactive object can redistribute work to avoid that host. The second method of communication is that of polling the distributed state layer for the information it has gathered and tabulated. For example, a reactive object may request processor performance summary information between iterations of a concurrent loop to determine if changes in the distribution of work between processors would be beneficial.

Figure 2 illustrates the internal structure of a reactive object. The object consists of three major components: *adjustable data structures*, *adaptive algorithms*, and *reactive policies*. The reactive object organization separates policy from mechanism, thereby allowing existing adjustable algorithms and data structures to be reused and combined with appropriate (possibly user-defined) policies. Moreover, the separation of algorithm from data structure allows adjustable data structures to be used by many different algorithms that can dynamically manipulate the structure to meet the algorithm's current needs.

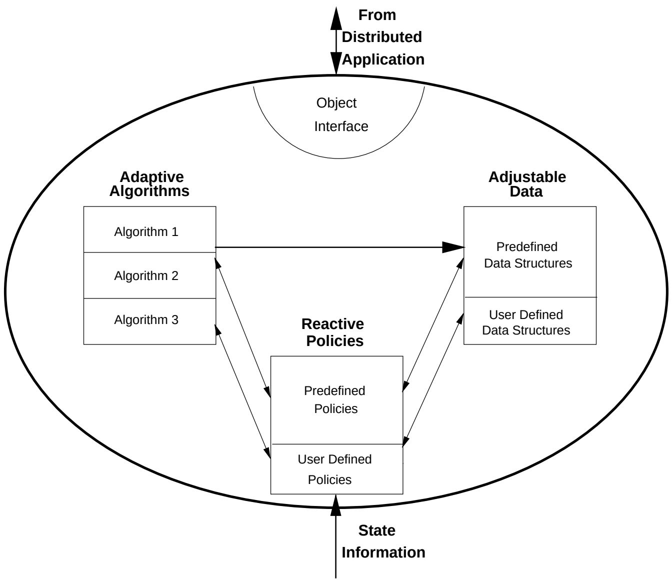

Figure 2: Reactive Object Template.

An *adjustable distributed data object* provides a configurable storage abstraction that algorithms can use. For example, a `reactive matrix object` with successive over-relaxation or averaging methods might make use of an adjustable `neighbor-exchange-matrix structure`, where the neighbor-exchange-matrix structure is a matrix storage abstraction developed specifically to support parallelized access and exchange of information between neighboring partitions. Although the abstraction understands decomposition (i.e., how the matrix is partitioned and the host currently working on each piece), it does not decide the decomposition. Instead, it provides methods that the algorithm or policy component can invoke to tell the object how to change the decomposition. Given a new decomposition description, the object can adjust itself (shift the boundaries and redefine the shared segments used) to accommodate the new description. Other methods might allow the algorithm to switch the object's implementation from Unify's random segments to sequential segments. In many cases, the object must provide an operation by which the policy or algorithm can inform the object that is it safe to adjust [GEK+95]. Our experience indicates that these types of parallelized storage structures are used repeatedly in a wide range of applications and will incur heavy reuse.

The *policy component* represents the heart of the reactive object, monitoring state changes and modifying the adjustable data objects and adaptive algorithms as needed. The policy component registers predicates with, and responds to notifications from, the distributed state layer.

In many cases, a predefined policy can be used in conjunction with a predefined data structure to control adaptation. For example, a predefined load balancing policy may monitor global load information by registering predicates with the distributed state layer. When the load becomes imbalanced the policy code will be invoked that will in turn invoke the adjustment methods of a neighbor-exchange-matrix structure to change the partitions of the matrix. Alternatively, a policy to conform to spatial locality might request the topology from the distributed state layer and invoke the adjustment methods of the matrix object to reassign spatially related partitions to neighboring hosts. Finally, another policy might monitor high-level state information such as the synchronization to data transmission ratio and then modify the algorithm to use large or small granularity sharing with more or less synchronization.

An *adaptive algorithm*, or set of algorithms, can change the structure of the computations as well as change or adjust implementations of the data structures it uses. Changes to the algorithm will typically be initiated by the policy component that will directly modify the algorithm or adjust the data component which in-turn will be detected by the algorithm. When the algorithm is directly altered it may be modified or an entirely new (more appropriate) algorithm will be substituted. If, instead, the data component is modified, the algorithm will detect changes in the data structure and adjust accordingly (e.g., between two iterations of a loop the algorithm may find it has a larger or smaller partition of the problem space).

6 Status

We are currently completing the addition of the probes and system monitors needed by the local and distributed state layers in Unify. The local state layer tracks network statistics including packet loss, network congestion, and network bandwidth achieved. To demonstrate the utility of reactive objects, we performed experiments implementing a successive over-relaxation (SOR) algorithm and the SPLASH "water" benchmark [SWG92] with machine assignments that ignored network topology and then with assignments that tried to group communicating processes together.

Figure 3a shows the execution time of a distributed SOR application working on a 512 X 512 matrix, while Figure 3b shows the execution times of the water application on a 4096x4096 problem size. The graphs show the execution time for 2, 4, 6, and 8 workstations. Each workstation resides on one of two subnets interconnected by the campus backbone. With the exception of the machine location, the system is completely homogeneous. All workstations are Sun Sparc 20 Hypersparcs

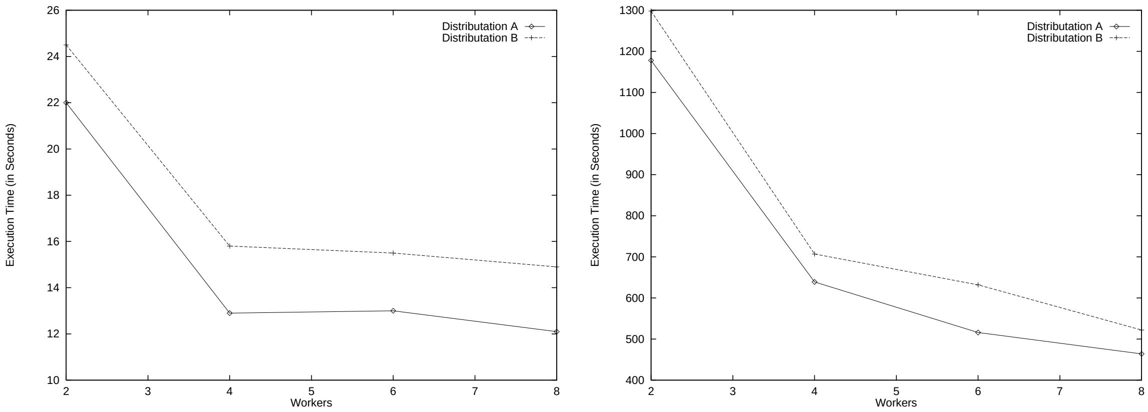

Figure 3: Example variation in run times resulting from two different mappings of decomposed domains to machines. The graphs show Unify execution times for (a) SOR and (b) the SPLASH water benchmark using 2, 4, 6, and 8 machines.

with 64 Mbytes memory, identical disks, and 100 Mbps Ethernet connections. Distribution B represents a distribution of work across machines where machines are selected without regard to network considerations. For Distribution A, the logical problem partitions that required the most communication bandwidth were placed in the same subnet, thereby minimizing communication between subnets. The figures for both applications show a 10-20% increase in performance on identical machines when the distribution of work can adapt to the network topology.

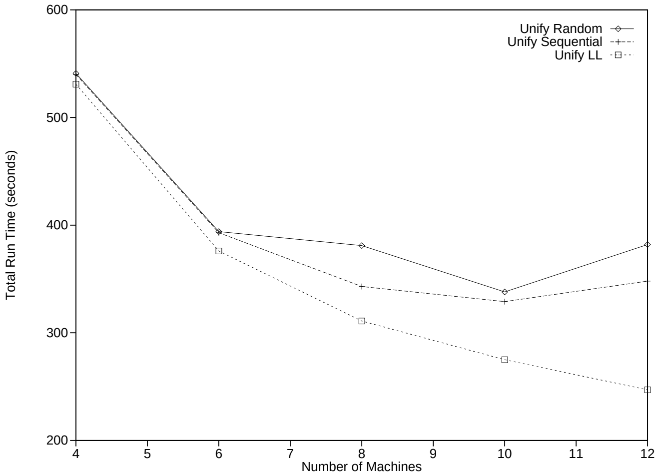

Figure 4: Performance comparison between multiple implementations of the SPLASH water application.

Figure 4 illustrates the performance improvements that result from selecting the appropriate algorithm, memory types and synchronization schemes for the current environment. The graph shows three different implementations of the water problem when executing on a single local area network of homogeneous machines. The "Random" curve represents a conventional implementation using Unify's random segment type. The "Sequential" curve uses Unify's sequential segments to implement the program's data structures. Finally, "Unify LL" shows the execution time when smaller sequential segments and a local locking mechanism are used together to reduce synchronization overhead. As the number of processors increase the version with local locking and sequential segments shows a dramatic performance improvement.

7 Summary

For high-performance distributed environments, the dynamic nature of the network and the nodes can make the overall behavior of the system difficult to manage. Reactive objects provide users with high-level abstractions that automatically adapt to the state of the distributed system or application. The system automatically gathers, condenses, and provides access to performance statistics needed by the distributed application for adaptation. For novice users, reactive objects can provide high-performance "building blocks" for developing applications. Experienced users are provided with default polices, a library of prefined adjustable data structures, and interfaces to the monitoring system that allow them to develop new reactive objects that can be "plugged in" to the reactive object support system. Initial analysis of the potential performance gains attainable with the a reactive architecture are promising.

References

[Bat89] Peter Bates. Debugging Heterogeneous Distributed Systems using Event-Based Models of Behavior. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging published in ACM SIGPLAN Notices*, 24(1):11–22, January 1989.

[BLNS82] A. D. Birrel, R. Levin, R.M. Needham, and M.D. Schroeder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4), 1982.

[BW83] P. C. Bates and J. C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software*, 3(4):255–264, April 1983.

[Cai93] X. C. Cai. An optimal two-level overlapping domain decomposition method for elliptic problems in two and three dimensions. *SIAM J. Sci. Comp.*, 14, 1993.

[CS94] Steve J. Chapin and Eugene H. Spafford. Support for Implementing Scheduling Algorithms Using MESSIAHS. *Scientific Programming*, pages 325–340, 1994.

[EGSM94] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon - Toward Interactive Parallel Programs: The On-line Steering of a Molecular Dynamics Application. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing*, pages 26–34, August 1994.

[FLA94] V. Freeh, D. Lowenthal, and G. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *First Symposium on Operating Systems Design and Implementation*, 1994.

[For94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Computer Science Department CS-94-230, University of Tennessee, Knoxville, TN, 1994.

[GEK+95] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, and Jeffery Vetter. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, February 1995.

[GKS94] A. Grama, V. Kumar, and A. Sameh. Scalable Parallel Formulations of the Barnes-Hut Method for n-Body Simulations. In *Supercomputing*, 1994.

[Gri93] A. S. Grimshaw. Easy to Use Object-Oriented Parallel Programming with Mentat. *IEEE Computer*, pages 39–51, May 1993.

[GSN93] A. S. Grimshaw, W. T. Strayer, and P. Narayan. Dynamic Object-Oriented Parallel Processing. *IEEE Parallel and Distributed Technology: Systems and Applications*, pages 33–47, May 1993.

[GYF95] James Griffioen, Rajendra Yavatkar, and Raphael Finkel. Unify: A Scalable Approach to Multicomputer Design. *IEEE Computer Society Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(2):24, July 1995.

[HKMC90] Robert Hood, Ken Kennedy, and John Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Supercomputing '90*, pages 74–81, November 1990.

[HML95] J. K. Hollingsworth, B. P. Miller, and J. E. Lumpp, Jr. Techniques for Performance Measurement of Parallel Programs. In F. Plasil T. L. Casavant P. Tvrdik, editor, *Parallel Computers: Theory and Practice*, Los Alamitos, CA, 1995. IEEE Computer Society Press,.

[JK93] Y.K. Jun and K. Koh. On-the-fly Detection of Access Anomalies in Nested Parallel Loops. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 107–117, May 1993.

[LCSM90] J. E. Lumpp, Jr., T. L. Casavant, H. J. Siegel, and D. C. Marinescu. Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 476–483, June 1990.

[LM92] R. Luling and B. Monien. Load Balancing for Distributed Branch-And-Bound Algorithms. In *Sixth International Parallel Processing Symposium*, 1992.

[LMnS90] T. J. Leblanc, E. P. Markatos, and nd Software. Operating system support for adaptable real-time systems. In *Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems*, pages 1–10, May 1990.

[Luc92] S. Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *In the Proceedings of the SIGPLAN '92 Conference*, June 1992.

[MC91] J. M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing '91*, pages 24–33, November 1991.

[MH89] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[MLCS90] D. C. Marinescu, J. E. Lumpp, Jr., T. L. Casavant, and H. J. Siegel. Models for monitoring and debugging tools for parallel and distributed software. *Journal of Parallel and Distributed Computing*, 9(2):171–184, June 1990.

[MW91] Keith Marzullo and Mark Wood. Tools for constructing distributed reactive systems. Technical Report TR 91-1193, Cornell University, 1991.

[NCO96] NCO. High Performance Computing and Communications: Foundation for America's Information Future. Technical Report http://www.hpcc.gov/blue96/index.html, National Coordination Office for High Performance Computing and Communications (NCO), 1996.

[Sch89] E. Schonberg. On-the-Fly Detection of Access Anomalies. *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation published in ACM SIGPLAN Notices*, 24(7):285–297, June 1989.

[SKB89] M. Simmons, R. Koskela, and I. Bucher. *Instrumentation For Future Parallel Computing Systems*. ACM Press, 1989.

[Smi88] J. M. Smith. A Survey of Process Migration Mechanisms. *Operating Systems Review*, pages 28–40, July 1988.

[Sos92] Rok Sosic. Dynascope: A Tool for Program Directing. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 12–21, July 1992.

[Sun90] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[VS95] Jeffery Vetter and Karsten Schwan. Progress: a Toolkit for Interactive Program Steering. Technical Report GIT-CC-95-16, Georgia Institute of Technology, 1995.

[YGS95] R. Yavatkar, J. Griffioen, and M. Sudan. A Reliable Dissemination Protocol for Interactive Collaborative Applications. In *The Proceedings of the ACM Multimedia '95 Conference*, November 1995.