Annual Report: 1/1/98 - 12/21/98

A Scalable Distributed Multicomputer For Massively Parallel Applications

DAAH04-96-1-0327

James Griffioen, James Lumpp, Rajendra Yavatkar, Raphael Finkel Dept. of Computer Science, Univ. of Kentucky

Abstract

The Unify distributed shared memory operating system is designed to connect machines that are distant from one another (such as a large intranet or a wide-area internet) into a large-scale multi-computer. Example accomplishments from the past year include the following. We designed and implemented an optimized diff'ing algorithm that attempts to minimize the amount of data transmitted while keeping the computational costs of computing and merging diffs low. For certain applications this can produce a large reduction in the amount of data transferred. We investigated dynamic consistency models where the systems automatically and continuously (on-the-fly) changes the implementation of the consistency mechanisms to match the current reference patterns. Using dynamic consistency, users do not have to specify the consistency model, but rather allow the system to find the best model. We implemented a checkpointing system that records the state of multithreaded applications. The checkpointing system will be used to restart from a globally consistent state following a fault. We also made significant improvements to the prototype which is now available for both Solaris and Linux systems (`http://www.dcs.uky.edu/~unify`). We implemented new test-suite applications to evaluate the ease of programming different models. We also improved support for Treadmarks and DIB applications.

Unify Project: 1998 DARPA Progress Report

1 Publications

The following papers describing research supported in part by this grant were submitted or appeared this past year:

- James E. Lumpp, Jr., Kuppuswamy Sivakumar, Christopher Diaz, James N. Griffioen, "Performance Visualization for Distributed Shared Memory System", *Parallel and Distributed Computing Practices*, volume l, number 3, to appear, March, 1999
- James E. Lumpp, Jr., Shashi K. Mazumdar, Stephen D. Gedney, "Performance Modeling of the Finite-Difference Time-Domain Method on Parallel Systems," *ACES Journal*, vol 13, number 2, pages 147-159, May, 1998.
- Su Wen, James Griffioen and Rajendra Yavatkar. "Integrating Concast and Multicast Communication Models," Proceedings of SPIE's International Symposium on Voice, Video, and Data Communications. Nov. 2-4, 1998.
- Christopher Diaz, James Griffioen, James E. Lumpp Jr., Raphael Finkel, "Implementations of Water-Nsquared with Unify Distributed Shared Memory", In preparation.
- R. Sumanasekera, J. Griffioen, J.E. Lumpp Jr., "A Reliable-On-Demand Dissemination Service", submitted for publication
- Raphael Finkel, Christopher Diaz, Su Wen, "Programming Merge Sort with Unify Distributed Shared Memory", In preparation.
- Michael Rogers, Christopher Diaz, James Griffioen, James E. Lumpp Jr., Raphael Finkel, "Managing and Propagating Diffs in Unify", In preparation.
- William R. Dieter and James E. Lumpp, Jr., "A User-level Checkpointing Library for POSIX Threads Programs," *Proceedings of the Twenty Ninth International Symposium on Fault-Tolerant Computing (FTCS-29)*, to appear, June 15-18, 1999.
- J. E. Lumpp, Jr. and W. R. Dieter, "Checkpointing with Multicast Communication," *Proceedings of the IEEE Aerospace Conference*, volume 4, February 1998, pages 467-479.
- J. E. Lumpp, Jr., K. Sivakumar, C. Diaz, and J. Griffioen, "Xunify – A Performance Debugger for a Distributed Shared Memory System," *Proceedings of the Hawaii International Conference on Systems Sciences*, January 1998.
- P. Zhang, T. W. Wu, R. Finkel, "Parallel computation for acoustic radiation in a sub-sonic nonuniform flow with a coupled FEM/BEM formulation", *Engineering Analysis with Boundary Elements*, in press, 1998.
- Yeshayahu Artsy and Raphael Finkel, "Designing a process migration facility — The Charlotte Experience", in Milojicic, D., Douglis, F., Wheler, R., *Mobility Processes, Computers, and Agents*, Addison-Wesley Longman and ACM Press, January 1999.
- Raphael A. Finkel, "Message-passing systems", article in *Encyclopedia of Distributed Computing*, Partha Dasgupta and Joseph E. Urban, editors, 1999 (to appear).

2 Personnel Supported

The following Masters and Ph.D students were supported during the Fall 1998 semester or will be supported during the Spring 1999 semester.

- Chris Diaz (Ph.D. Student)
- Bill Dieter (Ph.D. Student)
- Billy Mullins (Masters Student)
- Michael Rogers (Ph.D. Student)
- Amit Sehgal (Ph.D. Student)
- Su Wen (Ph.D. Student)

3 Scientific Progress and Accomplishments

3.1 Project Overview

The Unify research project is investigating scalable approaches for designing distributed shared-memory (DSM) multicomputers. Unify's goal is to create a massively parallel multicomputer from geographically disperse machines interconnected by wide-area networks such as large corporate/campus networks, the Internet, or evolving networks like the vBNS or Internet II. Unlike existing software packages such as PVM, MPI and others, which impose a message-passing model on the application programmer, Unify provides a distributed shared memory abstraction that hides all communication from the application programmer. Although it is widely accepted that distributed shared memory is a more convenient programming model than message passing, existing large-scale systems have turned to message passing because the overhead needed to support DSM was considered excessive and thus will not scale to massively parallel systems.

The Unify project is investigating a multicomputer design that will support DSM with minimal overhead. To support DSM across a wide-area network, Unify (1) minimizes communication, (2) hides the high latencies of a wide-area network, and (3) introduces an efficient group communication protocol that reduces traffic and reacts quickly to network errors and congestion.

3.2 Recent Accomplishments

During the past year, we have made significant progress toward achieving the objectives and milestones defined for the project. Our contributions this past year occurred in five basic areas: (1) *enhancements to the unify design and prototype*, (2) *implementation of a monitoring and checkpointing system*, (3) *application development and experimental evaluation of the Unify prototype*, (4) *porting Unify and associated packages to Linux clusters*, and (5) *hardware support for fast synchronization.* The following sections briefly describe our accomplishments in each of these areas.

3.2.1 Enhancements to the Unify Design/Prototype

Over the past year, we continued to make improvements and enhancement to the Unify prototype.

Enhanced Diffing Algorithm: Unify's diffing mechanism now contains an efficient merging algorithm that reduces the size of diffs. We store our diffs in balanced-tree structures. The balanced-tree structures provide faster access to consistency data and the merging algorithm bounds the size of the diffs data structures. Because the size of the diffs data structures are bound, Unify consumes less memory due to diffs and Unify transfers less data over the network than other DSMs. The merging algorithm has very little computational overhead because the algorithm can prune the tree structures while merging the diffs. We have compiled experimental results that show that Unify executes faster and manages memory more efficiently for many applications.

Dynamic Consistency Models: While Unify provides multiple consistency mechanisms, it did not allow a segment's consistency model to change over time. In many applications, a shared segment's access patterns will change as the application runs. For example, one region of the code may access the segment in read-only mode which was accessed repeated in multiple-writer mode by an earlier region of the code. Optimal performance is achieved by dynamically changing the consistency model for a segment as the application runs. However, the user should not be required to specify when the consistency model needs to change. Instead, our objective is to automatically determine when updates to shared data should be transmitted to obtain optimal performance.

We designed a *Dynamic Consistency mechanism* that relieves the user of the burden of analyzing the application's sharing patterns and deciding on an appropriate consistency mechanism. Instead, the DSM itself decides and activates the appropriate consistency mechanism for a process based on the process's access patterns to shared data. Also, if the process's access patterns change, the DSM changes the consistency mechanism on-the-fly.

To date, we have demonstrated that user-specified consistency changes can significantly improve certain applications. We are currently implementing the dynamic consistency mechanism that will automatically monitor access patterns and determine the best consistency model to use at each point in the program.

Reactive Object Support: The reactive objects system provides monitoring functions and adaptation mechanisms that allow Unify applications to respond to the performance being achieved during the execution of the program.

Monitoring functions have been implemented to collect information about process utilization and memory statistics on local machines. The system automatically gathers and condenses the local statistics from each machine. We designed and implemented a simple predicate grammar that Reactive Objects submit to the local monitoring layer to query the gathered information. We designed and implemented reactive matrix objects that multiply matrices and average matrices. These matrices reactively adjust themselves if the computation time on one or more machines is significantly larger than the average. Initially, the reactive matrix distributes the the computation evenly among workers by rows and the workers begin computing their portion. After each worker has finished a given number of iterations, it waits at a barrier. After the barrier, the workers adjust the allotment of work based on their computation times and continue the computation. We built test applications using the matrices that show significant performance improvements over static allocation methods.

Enhancements to SEClib: A stand alone package called *SEClib* has been developed that provides synchronization using unreliable multicast. A second version that supports piggy-backing of false sharing information is being used in Unify. Both versions are publicly available in the distribution.

3.2.2 Performance Evaluation and Debugging Support

Monitoring and Performance Evaluation:
Xunify is a performance monitoring and debugging tool for Unify. The tool helps the programmer visualize the performance of an application while it is running and can save trace information for postmortem visualization. Xunify is unique in the fact that it provides performance information at three different levels: (1) the DSM programming-model level, (2) the logical-message send and receive calls, and (3) the low-level transport protocol details. The three levels allow application writers to quickly identify inefficiencies due to inappropriate constructs or algorithms at the programming-model level, or alternatively to identify unique bottlenecks that arise from the particular network/runtime

environment. We have enhanced Xunify to monitor the new features of Unify as they are implemented. In addition, a major rework of the internals of Xunify has begun to improve the real-time display performance of the tool. Xunify has been made available in the various releases of Unify.

A Checkpointing System:

To facilitate checkpointing in Unify, we have implemented a user-level library that can checkpoint multi-threaded programs. A multi-threaded program's state can be divided into private state and shared state. A thread's *private state* includes its program counter, stack pointer, and registers. Its *shared state* includes everything common to all threads, such as the address space and open file state. A multi-threaded checkpointing library must save and recover both the program's shared state and each thread's private state.

User-level thread libraries are implemented outside the kernel. The library uses timers to preempt threads when their time slice is over. User-level threads cannot take advantage of a SMP because the kernel is not aware of threads. Thus it cannot schedule them to run concurrently on separate processors. Implementing a checkpointing library for a user-level threads package is a straightforward extension of a single-threaded checkpointing library. Because user-level threads do not really have private state, the checkpointing library only has to save the state of the timers and the timer signal handlers in addition to the rest of the state that it contained in the application's address space.

Checkpointing kernel-level threads presents a bigger challenge. Because the kernel maintains each thread's state, the checkpointing library must ask the kernel to restart threads during recovery in addition to restoring each thread's private state and the program's shared state to the state it was in before the checkpoint.

We solved each of these problems for POSIX threaded programs. The code is publicly available and we have written a paper describing the library. The current version includes memory exclusion and call-back functions to call user code before and after the checkpoint and during recovery. The call-backs are useful for tasks such as reopening connections to other processes.

Unify's checkpointing system uses the multi-threaded checkpointing library to checkpoint individual Unify workers. To coordinate checkpoints across Unify applications the system uses a stop-and-sync algorithm to force all the workers take checkpoints at the same time. The current version can coordinate any number of workers to take a consistent global checkpoint of any number of multi-threaded workers. The recovery code is still being developed, recovery with multiple workers has been demonstrated but the code is still in the early alpha stages.

3.2.3 Application Development and Experimentation

To gain experience with the system, We implemented several versions of the "merge sort" algorithm that evaluate Unify and compare Unify with the commercial Treadmarks package. The major difference between the various versions is the way in which data is merged. Merge sort algorithms are implemented with Put_Updates() (static), or Get_Updates() (dynamic and linear) models, and single or multiple shared segments. Events and sequencer are used for synchronization. The merge-sort applications have provided information on the ease or difficulty of implementing common algorithm constructs on Unify, as well as performance comparisons with Treadmarks. We are currently preparing a paper describing the implementations and the performance comparisons we have made.

We also evaluated the *water* program from the SPLASH benchmark suite, analyzing performance differences between Unify and Treadmarks. Experimentation with various problem sizes and number of processes has shown that Unify provides significantly better performance in wide-area network environments than Treadmarks.

We also developed a library that allows programs written for DIB (Distributed Implementation of Backtracking) to run on Unify. DIB is a general-purpose package that supports distributed execution of a wide range of tree-navigation applications. Dib uses user-supplied routines to generate a problem tree. DIB then transfers data and information among the computers involved in the computation and combines results computed from different parts of the tree.

We decided to implement DIB on Unify so that it would be possible to use all the preexisting applications on Unify without modification. Example DIB applications that we have run include *trivial*, *sales* and *queens*. The *trivial* application is a simple tree-search program, that counts the number of leaves in a tree of specified depth and fanout. The *sales* application implements the traveling salesman problem to find the minimum-cost path visiting each site once. The *queens* application implements the N-queens problem of positioning N queens on a NxN board where no two queens lie in the same horizontal, vertical or diagonal line. We are now looking into improving the performance of Unify-based DIB. The Unify-based DIB version provides a simple way of incorporating more applications into Unify and thus allowing Unify to be tested over a wide range of applications.

3.2.4 Linux port of Unify

Given Linux's growing popularity, particularly as the OS-of-choice for low-cost clusters, we ported Unify to execute on Linux clusters. Unify's ability to run efficiently across both wide and local area networks makes interconnecting Linux clusters and attractive way to construct multicomputers.

Although large parts of the unify code ported without modification, other problems we encountered were:

- byte-swapping problems between the Intel and Sparc architectures
- incompatibilities between Linux and Solaris such as system calls that take different parameters or behave differently. For example, if you send a UDP packet to a port and no one receives the message the sender gets an error when the receiving hosts sends an ICMP packet back. Solaris does not do that so we added code to handle the error return code.
- to support memory mechanisms required by Unify, we had to write a patch to Linux so that the signal handling code passes the address of a segmentation fault to the signal handler.
- dealing with substantial differences between Solaris' and Linux's POSIX Thread implementations (despite the fact that it is a standard!).

The Linux port of Unify has been successfully completed and is currently being used for performance experimentation on our Linux cluster.

3.2.5 Hardware Support for Unify on Clusters

We have developed a low-latency hardware interconnect to enhance the synchronization and communication between Unify workers when Unify is run on a cluster of machines (machines physically close to one another). Cluster based systems are becoming popular alternatives to more expensive, more tightly coupled, parallel machines. The Kentucky Fast Cluster (KFC) Project is developing low cost interconnection hardware that allows processes running on clusters to synchronize in a few microseconds, several orders of magnitude faster than conventional networks. The hardware, based on the Purdue Adapter for Parallel Execution and Rapid Synchronization (PAPERS), costs only a small fraction per-node compared to standard Ethernet and can scale to hundreds or even thousands of machines. Development of hardware accelerated versions of the Unify library routines has just begun, however, the potential for performance improvement is substantial. More information on the KFC hardware can be found at `http://www.dcs.uky.edu/~kfc`.

3.2.6 Unify Public Releases

We packaged up the prototype system including the Unify libraries, the Xunify visualization tool, SEClib, the Treadmarks emulation library, and a variety of example applications. The release is publicly available at `http://www.dcs.uky.edu/~unify` for Solaris and Linux systems.

3.3 Research Plan for 1999

The following lists some of our research and implementation goals for the next reporting period:

- continue to experiment with and debug the prototype system to make it more robust
- develop more real-world applications to stress test the system and evaluate its performance
- implement dynamic consistency models
- complete the checkpointing recovery mechanism
- clean up the distribution and improve the documentation and examples
- develop the Unify libraries that make use of the KFC hardware

4 Technology Transfer

In addition to publishing our findings in conference proceedings, journals, and technical reports, we have made our prototype implementations and source code available on the web. We have improved the documentation and our set of example applications. We have also established an email list for Unify users to ask questions and provide helpful insights to other Unify users.