UPTEC F03 104

Examensarbete 20 p
Januari 2004

Improving DSZOOM´s Run Time System

Niklas Ekström

UPPSALA UNIVERSITET

Engineering Physics Programme
Uppsala University School of Engineering

UPTEC F 03104	Date of issue January 2004
Author Niklas Ekström	
Title (English) Improving DSZOOM's Run Time System	
Title (Swedish) Förbättring av DSZOOMs körtidssystem	
Abstract DSZOOM is a software-based distributed shared memory (DSM) system intended to run on a cluster of computers that communicates through interconnect hardware. In order to run on a multi-system image (MSI) cluster, meaning a cluster where each node has a separate operating system running, there is a need for a run-time system that supports process distribution. In this thesis, we have implemented a run-time system that will distribute processes across the cluster without the assistance of the operating system. This is done using the robust and widely accepted message passing interface (MPI) standard. The run-time system is also optimized to take advantage of UltraSPARC's Block Load/Store instructions. We have also implemented hierarchical lock technology in DSZOOM. The lock algorithm creates node affinity by keeping the lock in a node for some time if there is contention. The micro-benchmark study shows a 500% speed-up on a two node cluster with 20 processors. The hierarchical lock can run on any number of nodes.	
Keywords DSM, MPI, Hierarchical lock, MSI	
Supervisor(s) Zoran Radovic	
Examiner Erik Hagersten	
Project name	Sponsors
Language English	Security
ISSN 1401-5757	Pages
Supplementary bibliographical information	

School of Engineering, Studies Office
Visiting address: Lägerhyddsvägen 1, The Ångström laboratory, Uppsala
Postal address: Box 536, SE-751 21, Uppsala, Sweden
Phone: +46-(0)18-4713003
Fax: +46-(0)18-4713000
E-mail: kansli@uth.uu.se

Improving DSZOOM's Run Time System

21st January 2004

Abstract

DSZOOM is a software-based distributed shared memory (DSM) system intended to run on a cluster of computers that communicates through interconnect hardware. In order to run on a multi-system image (MSI) cluster, meaning a cluster where each node has a separate operating system running, there is a need for a run-time system that supports process distribution.

In this thesis, we have implemented a run-time system that will distribute processes across the cluster without the assistance of the operating system. This is done using the robust and widely accepted message passing interface (MPI) standard. The run-time system is also optimized to take advantage of UltraSPARC's Block Load/Store instructions.

We have also implemented hierarchical lock technology in DSZOOM. The lock algorithm creates node affinity by keeping the lock in a node for some time if there is contention. The micro-benchmark study shows a 500% speed-up on a two node cluster with 20 processors. The hierarchical lock can run on any number of nodes.

Contents

1 Introduction

For a long time, computer engineers have realized that sequential computers have an upper speed limit, determined by several factors, one of them being the speed of light. In order to make computers faster beyond that, one has to make the computer make computations in parallel. Such computers are called parallel computers and work by dividing the computation onto several processors. The amount of speedup that is possible by using a parallel computer is governed by Amdahls law, also known as the law of diminishing returns. What this law states is basically that if a part of a program can not be run in parallel that part wont run any faster no matter how many processors your parallel computer has, and this sequential part will eventually become the limiting factor. It is then up to the programmer of a parallel program to make algorithms that have as little sequential code as possible.

The work of the computer engineers is to make parallel computers that are as efficient as possible in every possible aspect. One big issue in any computer is managing the physical location of the data, so that the executorial units of a processor have the data available when it is going to perform a computation, instead of stalling until the data arrives. This issue only grows bigger in a parallel computer since there can be many processors that want to use a certain datum and therefore the computer has to compromise on the location of the data.

When a program is running in parallel, each of the parallel running sections is called a thread. Up until recently there would be one thread executing on each processor, but with the advent of Simultaneous Multithreading (SMT), there can be a certain number of threads, often two or four, running on each processor.

One other issue with computer designs is the communication and synchronization among these threads. Often one thread may be dependent on the result of one or many other threads to be able to proceed with its computations, and may also have to synchronize with other threads so that the threads proceed in correct order, for example when entering a critical section. On some systems, two threads also have to synchronize to be sure that the communication of a datum is completed. The goal for communication and synchronization is to be as fast as possible, both in transferring many bytes at a time (high bandwidth) and in low setup time for the first byte to arrive (low latency).

There have been many different designs that try to solve these issues and build an efficient parallel computer. A design to build large parallel computers that has become popular is to connect many smaller Shared Memory Processor (SMP) computers in a cluster using fast interconnect communication hardware. This way, many hundreds and possibly thousands of processors can work together. One problem with this approach is that the communication across the interconnect is often relatively slow. An even bigger problem is that cache coherence is often not supported in hardware which makes it impossible to run shared-memory programs. It is still possible to run programs that use message passing, but there are many reasons why one would want to write a parallel program using the shared-memory model. One reason is that many believe the shared-memory model to be the most intuitive to program for. Another is that some algorithms can not be written and run effectively using message passing.

This problem is remedied by implementing memory coherence in software. Such a solution is called a Software Distributed Shared Memory (SW-DSM) system and many different variants have been proposed [3, 8, 9, 12, 15, 14, 16, 19]. Implementing coherency in software has many benefits such as the possibility to implement optimizations that would be too specific to implement in hardware. The downside of a software solution is that there will generally be higher raw latencies and lower raw bandwidth, but using cleaver latency hiding techniques this can be compensated.

DSZOOM [12] is a SW-DSM system that inserts fine-grain access control checks in the program code (called *snippets*) and uses a blocking directory that allows the entire coherence protocol to run in the requesting processor. DSZOOM's coherence protocol is well tested and has previously proven to work well in experimental setups on a system with single-system image capabilities. The goal of this thesis is to make DSZOOM work on a real cluster where the nodes all have different system images (operating systems).

Contributions

1. Simple, portable and robust process creation/distribution using well proven MPI technology (Section 3)
2. Efficient synchronization implementation with hierarchical spin lock technology to create node locality (Section 4)
3. Instrumentation and protocol support that is optimized for UltraSPARC processors Block Load/Store instructions, which are required to work with the interconnect hardware (e.g., Infiniband [7] and/or Sun Fire Link [18]) (Section 5)

2 Run-Time Support

To be able to make the cluster of computers provide a single system image, DSZOOM implements a run-time system that performs operations such as synchronization and process creation across the cluster. The current version of DSZOOM implements the PARMACS parallel execution environment.

2.1 PARMACS Overview

PARMACS is a set of m4 macros that is used to write portable parallel programs. These macros are today considered as a PARMACS Application Programming Interface (API) standard, but are mainly used in academia. By implementing the macros for different architectures, programs written with the PARMACS API can be compiled and executed at those architectures. PARMACS macros are originally developed by researchers at the Argonne National Laboratory [10].

DSZOOM implements primarily those parts of PARMACS that are needed by the Stanford Parallel Applications for Shared-Memory (SPLASH-2) benchmark suite [21]. The following macros have been implemented in the DSZOOM system (originally based on the implementation by Artiaga et. al. [2, 1]):

- *MAIN_INITENV()*: this macro initialize the PARMACS environment. This should be the first executable function in the application.
- *MAIN_END()*: this macro terminates the PARMACS environment. It should be the last statement executed by the application.
- *MAIN_ENV*: this macro contains the variables and symbol definitions for the PARMACS environment. It should only appear once in the application, in the beginning of the main source file.
- *EXTERN_ENV*: this macro contains symbol definitions and should be included in the beginning of each source file except the main source file.
- *CLOCK(unsigned long time)*: this macro sets the time variable to the current time.
- *CREATE(void (*proc)(void))*: this macro creates a new process, starting its execution on the proc routine. The new process can access the shared memory and perform synchronization with the other processes. The process is created on a node of the cluster in a round-robin fashion.
- *WAIT_FOR_END(int n)*: this macro blocks the caller until n of its child processes (created with the CREATE macro) has finished.
- *G_MALLOC(int size)*: this macro allocate size bytes of shared memory and return a pointer to it.
- *G_FREE(void *ptr)*: this macro releases the memory allocated with G_MALLOC and pointed by ptr.
- *LOCKDEC(l)*: this macro declares l as a variable of type lock.
- *LOCKINIT(lock l)*: it initializes a variable of type lock. After initialization, the first process calling the LOCK macro must be able to enter the mutual exclusion area.

- *LOCK(lock l)*: this macro enters a mutual exclusion area protected by a lock variable. Just one process can enter the area at a time. If another process has already entered, the caller process blocks until no one else is in the protected area.
- *UNLOCK(lock l)*: this macro exits from a mutual exclusion area. If processes are blocked in the same lock variable (due to a LOCK call), one of them is released.
- *ALOCKDEC(al, int n)*: this macro declares *al* as an array of locks of *n* elements.
- *ALOCKINIT(alock al, int n)*: it initializes an array of locks of *n* elements. The first processes locking each of the elements must be able to enter the mutual exclusion area.
- *ALOCK(alock al, int i)*: this macro enters the mutual exclusion area protected by the *i*-th lock in an array of locks. The macro assumes that the first subscript of the array is 0 (zero). The caller process blocks if another process is currently in the protected area.
- *AULOCK(alock al, int i)*: this macro exits the mutual exclusion area protected by the *i*-th lock in an array of locks. If processes are blocked in the same element of the array of locks (due to an ALOCK call), one of them is released.
- *BARDEC(b)*: this macro declares *b* as a variable of type barrier.
- *BARINIT(barrier b)*: this macro initializes a variable of type barrier. This operation is needed only once at the beginning of the application.
- *BARRIER(barrier b, int n)*: a process which execute this macro is blocked until *n* processes (including itself) call the macro on the same barrier *b*; then, all processes waiting for this barrier are released.
- *GSDEC(gs)*: this macro declares *gs* as a variable of type global_subscript.
- *GSINIT(global_subscript gs)*: this macro initializes a variable of type global_subscript. After exiting a self-scheduled loop, the variable is reset, so GSINIT should be called just once at the beginning of the application.
- *GETSUB(global_subscript gs, int subs, int max_subs, int max_processes)*: this macro obtains the next subscript available in a self-scheduled loop, and returns it into the subs variable. *max_subs* is the maximum legal value for the subscript, and *max_processes* the number of processes working on the same loop. When there is no other subscript available, -1 is returned in *subs* and the process blocks until all the processes working on the same loop also get a -1. The first subscript is always 0 (zero).

- *PAUSEDEC(ev [,int n])*: this macro declares *ev* as an array of events. The first subscript in the array is assumed 0 (zero). If *n* is omitted, *ev* is declared as a single event.
- *PAUSEINIT(event ev [, int n])*: this macro initializes an array of *n* events (or the first one in the array, if *n* is omitted). After initialization, all events in the array are cleared.
- *SETPAUSE(event ev [, int i])*: this macro sets the *i*-th element in an array of events (or the first one, if *n* is omitted).
- *CLEARPAUSE(event ev [,int i])*: this macro clears the *i*-th element in an array of events (or the first one, if *n* is omitted).
- *WAITPAUSE(event ev [,int i])*: the caller process blocks if the *i*-th event of the array of events (or the first one, if *n* is omitted) is cleared. As the event is set, all processes waiting for that event are released.
- *PAUSE(event ev [,int i])*: if the *i*-th event in the array is set, the caller process clears it and goes on; otherwise it blocks. As the event is set, only one of the processes blocked after calling this macro is released, clearing the event again. If *i* is omitted, the macro acts on the first event in the array.
- *EVENT(event ev [, int i])*: if the *i*-th event in the array is cleared, the process sets it and goes on; otherwise, the caller process blocks. When the event is cleared, only one of the processes blocked after calling this macro is released, setting the event again. If *i* is omitted, the macro acts on the first event in the array.
- *//START_ TIME(int mynum, int n)*: this macro is an DSZOOM extension macro that waits until *n* processes has arrived at that mark and then starts the timing of the parallel section. The timing stops when all processes have finished in the WAIT_FOR_END macro, and the time is printed in the console.

PARMACS programs start with a single process and then forks of child processes that carry out parallel work. In the parallel section, processes may want to synchronize with each other using synchronization primitives supported by the run-time system. The processes also communicate through shared memory, which is kept coherent by the DSZOOM system with blocking directory coherence protocol code which is called by the instrumented loads and stores in the program.

In the current DSZOOM implementation, it is assumed that child processes may not perform any I/O operations during parallel execution, except for printing text to the console. This is not explicitly stated in the PARMACS standard, but all SPLASH-2 applications have this behaviour.

A skeleton of one simple PARMACS application is shown in Appendix A.

2.2 Previous DSZOOM Run-Time Implementation

The previous DSZOOM implementation supports the same PARMACS API as this new version. However, the previous run-time system implementation relies on single system image capabilities such as creating processes on remote nodes and reading/writing to remote memories. One machine that has these capabilities and that the previous DSZOOM implementation could run on is the Sun WildFire prototype [5, 11, 6]. Using that setup, the previous DSZOOM implementation was used as a proof-of-concept to try out different coherence protocols, synchronization algorithms, and perform numerous instrumentation experiments.

3 MPI Process Distribution

When a DSZOOM program begins execution it starts out with a single process on one of the nodes of the cluster. When the parallel part begins processes are created and must be distributed to the different nodes to make use of all the processors in the cluster. In addition, before the program is started processes need to be started on each node to set up memory mappings through the interconnect controllers.

To facilitate starting processes on cluster nodes and communicating amongst the processes before the shared-memory system in DSZOOM is set up, we decide to use the Message Passing Interface (MPI) library. This decision was made since the MPI standard is a simple to use, robust, highly performing and widely accepted standard for communication and process creation across parallel platforms such as clusters.

Along with the MPI library we are using, which is the Sun MPI Library, follows the Sun HPC Cluster Runtime Environment (CRE). The CRE must be installed in advance by a system administrator on all nodes that will make up the cluster. When the cluster runtime environment is set up, processes can be started remotely on any node within the cluster.

Like all programs compiled with the MPI library our program is started using the mprun command. The number of processes and on what nodes they are going to be started at can be given as arguments to mprun. For example, if you wish to run a program on the three cluster nodes "simba," "tembo" and "duma," you would write:

```
mprun -np 3 -l "simba, tembo, duma" <program filename>
```

Shown in Figure 1 is a flowgraph that visualizes the entire system described below. The processes started with mprun are called startup processes in our system, and they all start executing the same program. Using the *#pragma* *init* directive, the first function that is called is the *dszoom_ startup()* that is part of the run-time code. First MPI is initialized with a call to *MPI_ Init()* and then a call to *MPI_ Comm_ get_ parent()* is made to see if the process is a startup process or a *worker process* (we discuss them further down in this section). If the returned value is *MPI_ COMM_ NULL* it is a startup process and *dszoom_ startup_ process()* is called.

Function calls to *MPI_ Comm_ size()* and *MPI_ Comm_ rank()* gives the number of processes (which in our case equals the number of nodes) and the rank for each process which is a unique identifier that tells the process apart. The process that gets rank=0 becomes the *master process* and the other processes with rank≠0 becomes *slave processes*. All the startup processes performs initialization, such as allocation of memory, mapping of memory through the interconnect controller and initialization of the coherence state, but only the master process goes on to execute the actual program. The slave processes only wait until the program finishes and then performs clean up actions such as freeing the allocated memory.

Before leaving the *dszoom_ startup_ process()* function the master processes will gather the hostnames of all nodes, which will be used later on when the master process are going to distribute worker processes to the nodes. After this the slave processes will enter *dszoom_ slave_ process()* where they will wait for the master process to signal them to proceed. The master process on the other hand will return from its init section and enter the *main()* function where it is the single process that will start to execute the actual application.

The first PARMACS macro that will appear in the program is *MAIN_ INITENV* (see Appendix A for a framework of a PARMACS program). This macro will make a call to the *dszoom_ main_ initenv()* function, which will signal the slave processes so that all startup processes does initialization of the DSZOOM coherent shared-memory system. This is done with the functions *dszoom_ export_ create()*, *dszoom_ import_ map()* and *dszoom_ coherence_ init()*. When the initialization is done the master process initializes the run-time data in *dszoom_ runtime_ init()* and then returns to the program. All slave processes wait for the master process to signal them when the program is about to exit.

The program will now do general initialization, allocate shared memory and initialize synchronization primitives. Synchronization and shared-memory macros will in general translate into function calls into the run-time system. Most macros are straight forward implementations of commonly known algorithms, except the hierarchical spin lock described in section 4.

After the initialization the program enters the parallel section by executing the *CREATE()* macro several times to create child processes that do parallel work. The *CREATE()* macro translates into a call to the *dszoom_ create()* function where a child process is started with the *MPI_ Comm_ spawn()* call. The relevant arguments to this function are:

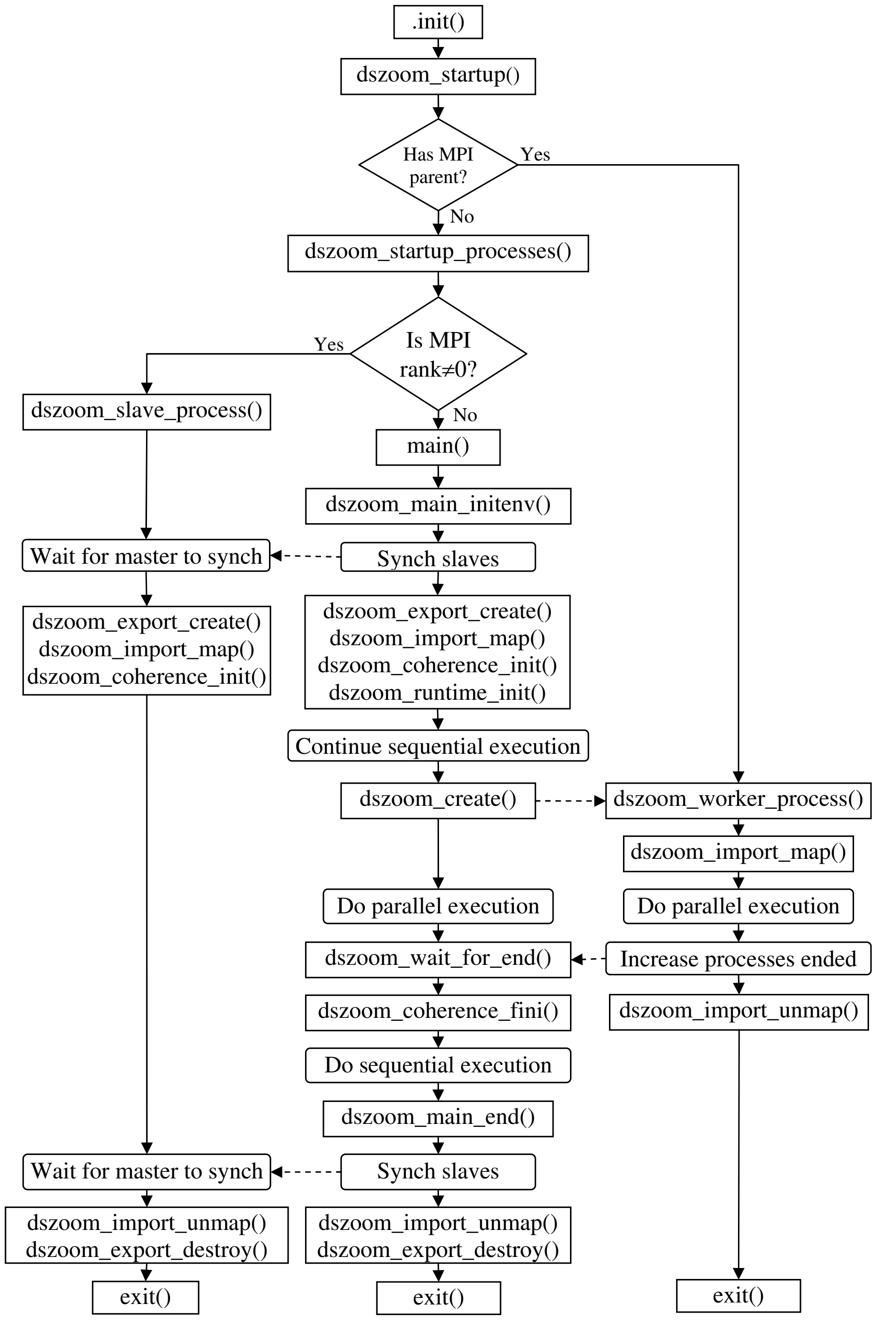

Figure 1: The MPI-based DSZOOM run-time system.

Argument	Description
command	Name of program to be spawned
argv	Arguments to command
info	A set of key-value pairs telling the runtime system where and how to start the processes
comm	Intracommunicator containing spawned process

The filename of the program to be spawned is the same as that of the master process, which is available in the *main()* function as *argv[0]*. Since the child process does not use any arguments, we give *MPI_ARGV_NULL*. The info argument needs to be created with calls to *MPI_Info_create()* and *MPI_Info_set()*, and later freed with *MPI_Info_free()*. The only key we use is the "sun_hosts" key whose corresponding value tells on what node to start the spawned process. This is a platform dependent key-value pair, but there should be a similar key on all platforms. When *MPI_Comm_spawn()* returns, the intercommunicator pointed to by the comm argument is the group with the spawned process in it.

The semantics of the *CREATE()* macro is not clearly documented, but from [2] we conclude that the *fork()* call contains at least the semantics of *CREATE()*. The semantics of *fork()* is basically that it creates a child process and then copies the entire state of the process to the child process. There is quite a lot of state belonging to a process and not all needs to be copied to the child process in our *CREATE()* macro implementation.

First of all, since we are going to call a routine right away when the process is started, we do not need to copy the contents of the processor registers or the stack. Since it is reasonable to assume that programs are not allowed to do I/O in the parallel section, and this in fact is the case with the SPLASH-2 applications, we do not copy the state needed to do that, such as file descriptors. We also does not copy the memory allocated on the heap using *malloc()* since global memory is allocated using *G_MALLOC()*, and this is in fact the way SPLASH-2 applications behave. So the only state that will actually be copied is the initialized and uninitialized data segments, the `.data` and `.bss` segments.

When the child process is started, it enters *dszoom_startup()* where it executes *MPI_Comm_get_parent()*. This MPI call will return an intercommunicator to the parent process, which is the master process. The child process now knows it is a worker process and calls *dszoom_worker_process()* where it will receive the data segments from the master process. This is done by the master process executing a *MPI_Send()* using the intercommunicator it got from *MPI_Comm_spawn()* and the worker process calling *MPI_Recv()* using the intercommunicator it got from *MPI_Comm_get_parent()*. The new data segments will be copied to replace the data that was there before. After this, no more MPI calls can be made since data belonging to MPI in the data segment has been overwritten. This can be fixed by not overwriting these certain pieces of data, but as of now there is really no need to do that.

The worker process will now call *dszoom_ import_ map()* to set up the memory mappings for the DSZOOM coherent shared-memory system. Then the routine that was specified in the *CREATE()* macro will be called to perform the parallel work.

One of the first things that will happen in the parallel section is that all processes will execute the *//START_ TIME()* macro. It waits for all processes to arrive at this point using a barrier; and the master process then stores the time when the processes are released from the barrier. This will be used to get the total time of the parallel section, which is used when benchmarking DSZOOM applications.

When the worker process is done with its parallel work it will atomically increase a run-time data counter called *processes_ ended*, which keeps track of how many worker processes has ended. It will then unmap the memory mappings with a *dszoom_ import_ unmap()* call and finally terminate with a call to *exit()*.

The master process will in general also enter the parallel section and perform parallel work. When it is done, it will return from the parallel routine and execute the *WAIT_ FOR_ END()* macro. This macro will call three functions; *dszoom_ wait_for_ end()*, *dszoom_ time_ stop()* and *dszoom_ coherence_fini()*.

The *dszoom_ wait_for_ end()* function will wait until the *processes_ ended* counter is equal to the number of child processes, which means that all child processes are done. After this, the *dszoom_ time_ stop()* function will take the current time and then subtract the time that was stored when the processes was released in the *//START_ TIME()* macro, thus getting the total execution time of the parallel section. Finally, the *dszoom_ coherence_fini()* function will do any final coherence work needed, such as moving all dirty cachelines to the master process memory (this function is used for master-centric I/O operations because we do not instrument any system calls).

When the worker processes have ended, the master process will usually do something with the data that was computed in the parallel section, such as saving it to disk or presenting the result at the command-line. It will then free up any resources previously allocated. As the very last sentence in the program the *MAIN_ END()* macro will be executed. This calls the *dszoom_ main_ end()* function that finalizes the DSZOOM run-time system in the following steps. First, the total time of the parallel execution is printed on the command line. Then the slave processes are signaled using MPI so they know that they should finish up. All startup processes (the master process and the slave processes) will then execute *dszoom_ import_ unmap()* and *dszoom_ export_ destroy()* to un-map the memories and free the allocated memory. After this, the MPI library will finalize with a call to *MPI_ Finalize()*, the processes will call *exit()*, and the execution is finished.

4 Hierarchical Spin Lock

A lock is one of the most important synchronization primitives. Locks are used in many places to protect a piece of code or data so that only one thread can access the data at any time. When a thread acquires a lock and wishes to access the data associated with it, the data has to move from the cache where it was last used to the cache of the processor on which the thread runs. Thus, we can say that the data moves where the lock moves.

If the system used is a non-uniform communication architecture (NUCA) machine [13], the time it takes to move the data is dependent on how the caches are interconnected, e.g. in a cluster it takes shorter time to move data between caches in the same node than between caches in different nodes. A hierarchical lock takes advantage of this knowledge and tries to keep the lock within a node for some time to create data locality.

There is also the aspect of wasted bandwidth when a processor repeatedly uses the atomic instruction to acquire the lock in a remote memory. This is another issue that the hierarchical lock solves by only having one processor from each node go to the remote memory and try to obtain the lock.

To this end Radovic and Hagersten have proposed a hierarchical lock [13] that has been implemented in DSZOOM as part of this thesis. It works by having a blocking directory that keeps track of where the global lock currently is, much like how the blocking directory coherence protocol in DSZOOM works. Using this technology this hierarchical spin lock can run on clusters with any number of nodes, unlike the RH lock [13] that can only be used on a two node cluster.

In this implementation there is a data structure on each node with the following fields:

```
struct hier_lock {
  volatile char l; // local lock bit
  volatile char g; // global lock bit
  volatile char h; // home node
  volatile char d; // directory
};
```

When a lock is initialized, a home node is designated, currently in a Round-robin fashion, and written into the *h* field on every node. This field never changes and is only there so that it is fast and easy to see what the home node is for every lock. The *d* field is the directory entry and indicates in which node the global lock is currently placed. The directory is only used in the home node and is therefore only there as padding in all nodes except the home node. When the lock is initialized the global lock is placed in the home node, so the *d* field is equal to the home node number, and the *g* bit is true in the home node and false in every other node. Initially the *l* bit is false in all nodes.

The lock works so that the lock is acquired when both the l- and the g-bit is true, and the g bit can be true in only one node at any time. The directory must point to the node where the g bit is true. The lock is released by clearing the l bit. To modify the `hier_lock` struct on any node, the l bit on that node must first be set. This is done using an atomic instruction. So, to acquire a lock that is currently on another node, the following steps have to be taken:

1. Spin lock the l-bit on the local node using an atomic instruction.
2. Spin lock the l-bit on the home node using an atomic instruction.
3. Spin lock the l-bit on the remote node that the directory points to using an atomic instruction.
4. Clear the g-bit and the l-bit on the remote node.
5. Change the directory to point to the local node, and clear the l-bit on the home node.
6. Set the g-bit on the local node. The lock is now acquired!

This is the worst case scenario where the data structure on three nodes has to be touched to acquire the lock. The best case scenario is if the global lock is currently in the local node, when only the local lock bit has to be locked using an atomic instruction.

If there is a lot of contention on the lock it makes sense to use this hierarchical lock since it is then more likely that there is another thread in the local node that also wants to lock the lock and can do so very quickly when the lock is released. One problem that can occur is if a lock is always handed over to another thread in the same node, and never to a thread in another node. Then the threads in the other nodes will be stalled and can not do any real work. This is sometimes called starvation, and can be avoided by introducing a fairness factor, which is treated thoroughly in [13].

Below follows acquire/release C code for the hierarchical lock:

```
void hier_lock_acquire( struct hier_lock *lock )
{
  int home_node;
  int remote_node;
  struct hier_lock *home_lock;
  struct hier_lock *remote_lock;

  // lock local lock
  local_spin_lock( &lock->l );
  if ( lock->g == 0 )
  {
      home_node = lock->h;
      home_lock = (struct hier_lock *)
        SET_NODE_ACCESS( lock, home_node );

      // lock directory lock
      if ( home_node != local_node )
        remote_spin_lock( &home_lock->l );

      // clear global bit in remote node
      remote_node = home_lock->d;
      if ( remote_node != home_node )
      {
        remote_lock = (struct hier_lock *)
          SET_NODE_ACCESS( lock, remote_node );
        remote_spin_lock( &remote_lock->l );
        remote_lock->g = 0;
        remote_lock->l = 0;
      }
      else
        home_lock->g = 0;

      // update directory value
      home_lock->d = local_node;

      // unlock directory lock
      if ( home_node != local_node )
        home_lock->l = 0;

      // make local lock the global lock
      lock->g = 1;
    }
  }

  void hier_lock_release( struct hier_lock *lock )
  {
    lock->l = 0;
  }
```

4.1 Contention-Sensitive Hierarchical Spin Lock

If there is high contention on locking a lock, the hierarchical spin lock performs well since it has very low hand-over time to local threads (threads in the same node), and also because the data that is guarded by the lock can stay in the node, thus creating locality. However, the hand-over time to a remote thread is quite high, which is a clear disadvantage if the probability of handing over the lock to a local thread is low. If this probability, which we denote P_{local}, is low (below some threshold) it is more beneficiary to use a normal spin lock instead of a hierarchical spin lock.

P_{local} is dependent on several factors, the two most important being:

- The amount of contention over the lock. More contention means there is greater probability of handing over the lock to a local thread, because of the way the hierarchical spin lock works.
- The ratio between number of processors and number of nodes. P_{local} will be higher if there are fewer nodes with many processors per node (few fat nodes) instead of many nodes with few processors per node (many thin nodes).

So because of this there are several cases when a normal spin lock is better then a hierarchical. However, there are ways to combine the benefits of a spin lock with the benefits from a hierarchical spin lock. This is done by letting the lock start out as a normal spin lock, but if a thread senses contention it turns it into a hierarchical lock.

The data structures used are precisely the same as for a hierarchical spin lock. The only thing that is modified is that the pointers that normally points to the node-local lock structure now all point to the lock structure in the home node. As long as there is no contention the lock works exactly like a normal spin lock, except that the code also checks so that the g-bit is set, which is true after initialization.

If a thread fails to lock the lock, it will start to count the number of retries it does until it acquires the lock. If the count goes above some specified threshold it determines that there is contention over the lock and that it should turn into a hierarchical lock. This is done by changing the pointer to point to the node-local lock structure. After this, the lock acts purely as a hierarchical spin lock.

When the lock later moves from the home node to a different node, other threads will notice this and will also change their pointers and start acting like a pure hierarchical lock.

5 Instrumentation and Protocol Implementation Optimizations

Global coherence in the DSZOOM system is resolved by coherence protocols (which are triggered by the inserted access control checks) implemented in C, that copies data to the node's local memory with UltraSPARC processor's Block Load/Store operations from the remote memory (which is locally mapped in every node). Performance is dependent on three components:

Block load	`ldda [reg_addr] imm_asi,` $freg_{rd}$
Block store	`stda` $freg_{rd}$`, [reg_addr] imm_asi`

Table 1: Assembly language syntax for Block Load/Store

1. Instrumented machine code fragments, so called *snippets*.
2. Trampoline routine that is called by snippets.
3. Coherence protocol itself, implemented in C.

This thesis improves on all of the three aspects given above. Most fundamentally the code is optimized to utilize the Block Load/Block Store instructions more efficiently.

5.1 Block Load/Store

The UltraSPARC processors implements two instructions called *Block Load* and *Block Store*, which are used to load or store a 64-byte block between the memory and eight floating-point registers. The reason we are interested in these instructions is primarily because these instructions are the only ones we can use to communicate through our interconnect hardware.

Block Load and Block Store are somewhat different from normal instructions since they do not obey Total Store Order (TSO) [20], or even obey the processor's consistency rules. So in order to be certain that all the data has been read from memory into floating-point registers, a `membar #Sync` instruction must be issued after a Block Load. The instructions also do not allocate into the L2-cache, to avoid pollution. Finally, the memory addresses accessed by the instructions must be 64-byte aligned, and the floating-point registers need to be aligned on an eight double-precision register boundary.

The assembly language syntax for the instructions is shown in Table 1. The `imm_asi` is the Address Space Identifier (ASI) that identifies what kind of load/store operation we wish to perform. We will be using `ASI_BLK_P (0xf0)` which is an unprivileged Block Load/Store from/to primary address space.

5.2 Snippet Improvements

The snippets have been improved by making them shorter and somewhat simpler, which gives less code expansion. The main reason for them being shorter is the inclusion of a trampoline, which keeps common code from being included in the snippet code. Here is an example of a floating-point load snippet:

```
#
FloatLoad
#
    $I                        ! perform the original load
*1  fcmps   $F, $3, $3        ! do DEADBEEF test
*2  fcmpd   $F, $3, $3
    fbe,pt  $F, .LQ$L         ! if not DEADBEEF => load is done
    add     $1, $2, %g3
    srl     %g3, 28, %g2      ! do range check
    sub     %g2, 8, %g2
    brnz,pt %g2, .LQ$L        ! if outside range => load is done
    mov     %g3, %g2
    save    %sp, -192, %sp    ! get new register window
    call    dszoom_load_trampoline
    mov     %o7, %l7          ! call trampoline code to
    restore                   ! perform coherence protocol
    $D      [%g2], $3         ! perform corrected load
    stb     %g4, [%g3]        ! set new directory bits
.LQ$L:
#
```

5.3 Trampoline Routines

As shown above, the trampoline is called by the instrumented code and executed before the protocol code is run. Since the trampoline code is stored in only one place it keeps the code expansion down and also does not use unnecessary space in the instruction cache. The trampoline stores away important registers and also makes room in the stack frame and saves the floating-point (f-p) registers there. This way the protocol code has free f-p registers and does not have to bother with saving the f-p registers when using the Block Load/Store instructions.

```
dszoom_load_trampoline:
    add     %fp, -64, %l0     ! make room for 64 bytes + what
    and     %l0, -64, %l0     ! is needed for alignment in the
                              ! stack frame
    mov     %y, %l1           ! save global registers in
    mov     %g1, %l2          ! registers that are local to
    mov     %ccr, %l3         ! this register window
    mov     %fprs, %l4
    mov     %g5, %l5
    call    dszoom_load_protocol    ! call protocol code
    stda    %f0, [%l0] 0xf0 ! save f-p regs
    mov     %l1, %y           ! restore registers
    mov     %l2, %g1
mov     %l3, %ccr
mov     %l4, %fprs
mov     %l5, %g5
ldda    [%l0] 0xf0, %f0 ! restore f-p regs
jmpl    %l7 + 8, %g0    ! return to instrumented code
membar  #Sync           ! sync memory in delay slot
```

5.4 Protocol Implemantation Optimizations

The protocol code has been somewhat simplified and straighten out. The major improvement is that the protocol uses only simple Block Load/Store-based routines to copy data, which are quite fast since there is no more need to spill the f-p registers. Now, coherence routines are also ready to be used with interconnect hardware.

```
//##############################################################
//## dszoom_load_protocol
//##
//## INPUT:   %g3 = effective address of the load
//##
//## OUTPUT:  %g3 = &(dir->dir_tag)
//##          %g4 = dir->dir_tag
//##############################################################
void dszoom_load_protocol( void )
{
  unsigned int effective_addr;
  unsigned int cache_line_number;

  unsigned char bits;     // presence bits
  unsigned char mask;     // this node's presence bit

  int home_node;          // home node of cache line
  int remote_node;        // node that holds data
  int nodes_valid;        // number of nodes with valid data

  dszoom_dir_t *dir;
  dszoom_dir_t *remote_mtag;

  effective_addr = dszoom_get_reg_g3();

  cache_line_number = ( effective_addr -
    DSZOOM_G_MEM_START_ADDR ) >>
    DSZOOM_CACHE_LINE_SHIFT;
```

```
home_node = cache_line_number % dszoom_nodes;

mask = 1 << dszoom_node;

dir = (dszoom_dir_t *)
  ( DSZOOM_DIR_START_ADDR_ALIAS + home_node *
    DSZOOM_G_MEM_SIZE );
dir += cache_line_number;

bits = dszoom_spin_lock_ldstub( &dir->dir_tag );

if ( bits == 0 )        // home node is exclusive
  bits = 1 << home_node;

// safety check guarding against false deadbeef
if ( ( bits & mask ) == 0 )
{
  dszoom_get_node_count( bits, &remote_node, &nodes_valid );

  if ( nodes_valid == 1 && remote_node != home_node )
  {
    // remote node's mtag needs to be locked and downgraded
    remote_mtag = (dszoom_dir_t *)
      ( DSZOOM_DIR_START_ADDR_ALIAS + remote_node *
        DSZOOM_G_MEM_SIZE );
    remote_mtag += cache_line_number;

    dszoom_spin_lock_ldstub( &remote_mtag->dir_tag );
    remote_mtag->dir_tag = 1;       // not exclusive
  }

  // copy cache line
  dszoom_block_load(
    (void *) ( DSZOOM_G_MEM_BIG_START_ADDR +
               remote_node * DSZOOM_G_MEM_SIZE +
               cache_line_number *
               DSZOOM_CACHE_LINE_SIZE ) );

  dszoom_block_store(
    (void *) ( DSZOOM_G_MEM_START_ADDR +
               cache_line_number *
               DSZOOM_CACHE_LINE_SIZE ) );
}

// return &( dir->dir_tag ) via %g3
dszoom_set_reg_g3( (unsigned int) &dir->dir_tag );
```

```
  // return dir->dir_tag via %g4
  if ( bits == mask && dszoom_node == home_node )
    dszoom_set_reg_g4( (unsigned int) 0 );
  else
    dszoom_set_reg_g4( (unsigned int) ( bits | mask ) );
}
```

The Block Load and Block Store functions are simply wrappers for the assembler instructions (which can easily be in-lined instead):

```
dszoom_block_load:
        ldda    [%o0] 0xf0, %f0
        retl
        membar  #Sync

dszoom_block_store:
        retl
        stda    %f0, [%o0] 0xf0
```

6 Performance Study

6.1 Experimental Setup

All the experiments were performed on a 2-node Sun WildFire machine [5, 6] built from two Sun Enterprise E6000 [17] nodes connected through a hardware-coherent interface. Each E6000 node has 16 UltraSPARC II 250 MHz processors and 4 GB shared memory. The system is configured as a traditional cache-coherent, non-uniform memory access (CC-NUMA) architecture with its data migration capability activated while its coherent memory replication (CMR) has been kept inactive. The WildFire interconnect has a raw bandwidth of 800 Mbyte/s in each direction and it has a latency of 1700 ns for remote memory accesses.

The following software was used:

- Slightly modified Solaris 2.6 operating system
- mprun version 3.1 as part of Sun HPC CRE 1.1
- Sun MPI 4.1
- cc: Sun WorkShop 6 update 2
- SPARC Assembler Instrumentation Tool (SAIT) [4]

Program	Problem Size
Barnes	29k particles
Cholesky	tk29.O
FFT	1M points
FMM	32k particles
LU-c	1024×1024 matrices, 16×16 blocks
LU-nc	1024×1024 matrices, 16×16 blocks
Ocean-c	514×514
Ocean-nc	258×258
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10
Radix	4M integers, radix 1024
Raytrace	car
Water-Nsq	2197 molecules, 2 iterations
Water-Sp	2197 molecules, 2 iterations

Table 2: SPLASH-2 benchmark suite.

The benchmarks used are from the Stanford Parallel Applications for Shared Memory (SPLASH-2) suite [21]. Only unmodified benchmarks from the original Stanford distribution are used (Table 2). The applications are representative for scientific, engineering, and graphics computing fields. All benchmarks were run five times and then the shortest execution time was taken.

6.2 Sequential Performance

We wish to see how long time the instrumented instructions take so that we can quantitatively compare different versions of code snippets. To do this, we ran the SPLASH-2 applications with just one processor and with only one kind of instruction instrumented at a time. When we run the applications sequentially, the code will never need to do any coherency work and will therefore never enter the trampoline- or protocol code. The SPLASH-2 applications were compiled and run with no instrumentation (*none*), integer loads instrumented (*intld*), floating-point loads instrumented (*fpld*), with stores instrumented (*st*), and with full instrumentation (*all*). The compiler optimization level used was *-xO0* in Table 3 (Figure 2) and *-fast* in Table 4 (Figure 3). The raw times are shown in Table 3 and 4, and in Figure 2 and 3 the relative times of instrumented integer loads, floating point loads and stores are shown. On average, the instrumentation overhead for *-xO0* is around 62% and for *-fast*, around 147%.

6.3 Hierarchical Lock Evaluation

To evaluate the hierarchical lock implementation we used two kinds of benchmarks. First we run a micro-benchmark that consists of a small loop that

Benchmark	*none*	*intld*	*fpld*	*st*	*all*	Overhead
fft	14.24	14.36	15.95	25.91	27.28	1.92
lu-c	66.34	66.38	82.78	140.81	155.49	2.34
lu-nc	80.11	80.22	97.02	154.01	168.99	2.11
radix	30.78	31.49	32.07	34.34	36.71	1.19
barnes	55.87	57.01	72.52	63.69	75.23	1.35
cholesky	20.40	20.74	24.45	35.07	38.2	1.87
fmm	117.98	123.64	134.78	124.63	143.3	1.21
ocean-c	46.52	47.77	57.60	69.42	82.14	1.77
ocean-nc	18.32	19.77	21.13	24.48	28.72	1.57
radiosity	28.77	30.16	31.17	29.79	32.27	1.12
raytrace	178.45	198.7	252.54	195.81	312.17	1.75
water-nsq	131.88	143.14	155.59	145.59	184.11	1.40
water-sp	34.87	37.14	41.94	38.34	50.07	1.44

Table 3: Raw data from sequential performance benchmarks (-*xO0*).

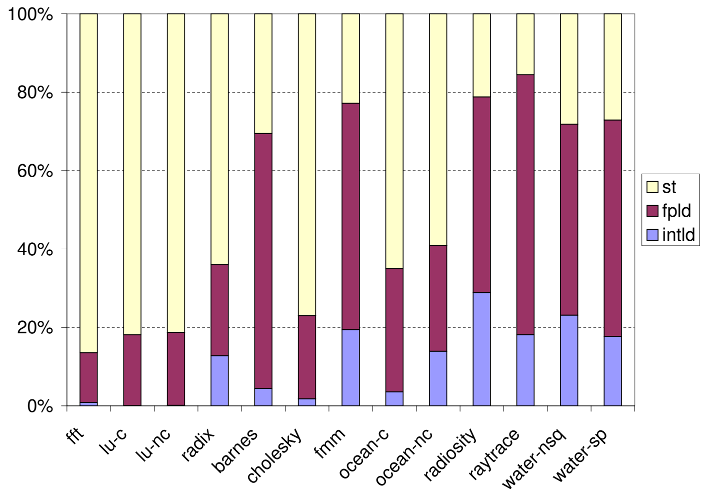

Figure 2: The relative execution times of the instrumented instructions (-*xO0*).

Benchmark	*none*	*intld*	*fpld*	*st*	*all*	Overhead
fft	3.24	3.26	4.51	9.34	10.51	3.24
lu-c	14.91	15.16	24.9	51.47	58.89	3.95
lu-nc	20.81	19.32	28.90	56.70	64.34	3.09
radix	6.37	7.07	6.35	10.52	11.09	1.74
barnes	13.93	15.24	16.59	16.99	21.84	1.57
cholesky	3.59	3.66	5.90	10.84	13.56	3.78
fmm	21.91	21.67	34.95	25.39	34.81	1.59
ocean-c	14.53	14.59	19.95	26.77	31.88	2.19
ocean-nc	3.85	3.79	5.56	6.50	8.32	2.16
radiosity	11.26	11.66	14.17	11.96	15.33	1.36

Table 4: Raw data from sequential performance benchmarks (-*fast*).

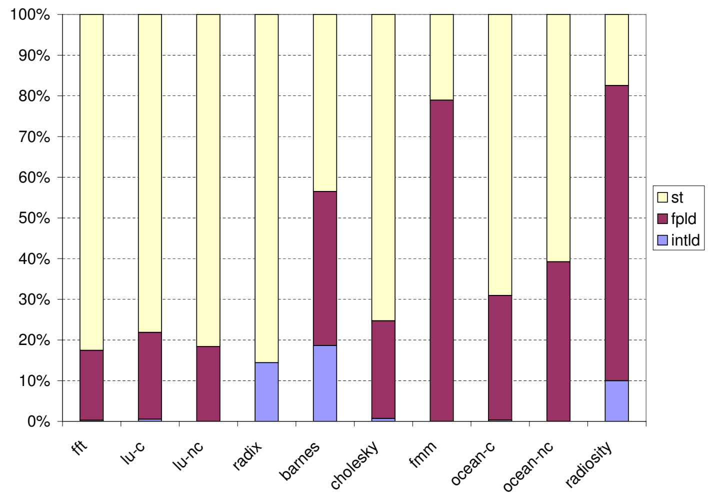

Figure 3: The relative execution times of the instrumented instructions (-*fast*).

```
for ( i = 0; i < 1000000; i++ )
{
  LOCK( Global->idlock );
  Global->counter++;
  UNLOCK( Global->idlock );
}
```

Figure 4: The main loop of the micro-benchmark.

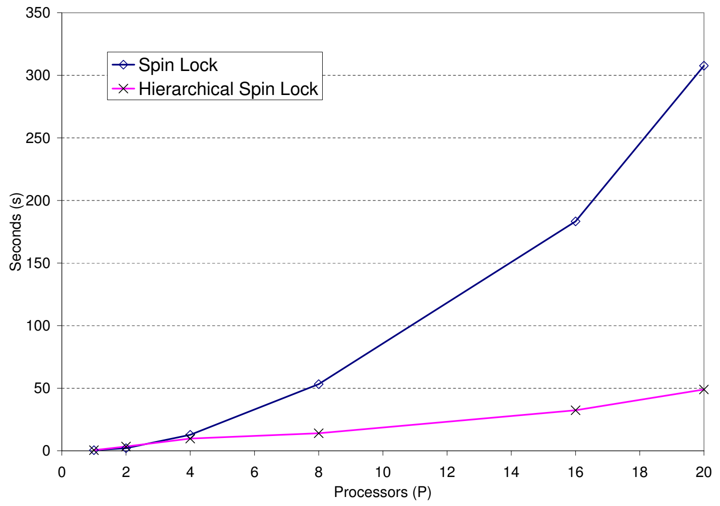

Figure 5: The execution times of the micro-benchmark.

atomically increment a global counter a million times (Figure 4).

The micro-benchmark was run with a normal spin lock and with the hierarchical spin lock on one to twenty processors distributed evenly on the two nodes (e.g., ten processors on each node for the total of twenty processors). The benchmark was run five times with each number of processors, and the shortest execution time was used. The results are shown Figure 5.

The second benchmark we use is the SPLASH-2 benchmarks run with the normal spin lock and with the hierarchical spin lock on 16 processors (8 on each node). The results are shown in Figure 6.

From the micro-benchmark it is fairly clear that the hierarchical spin lock introduces locality that keeps the data from constantly moving between the nodes, which is what happens for the normal spin lock. However we can not clearly observe this behaviour in the SPLASH-2 benchmark results (except for Radiosity, where we observe an improvement of 24%). This is not totally unexpected

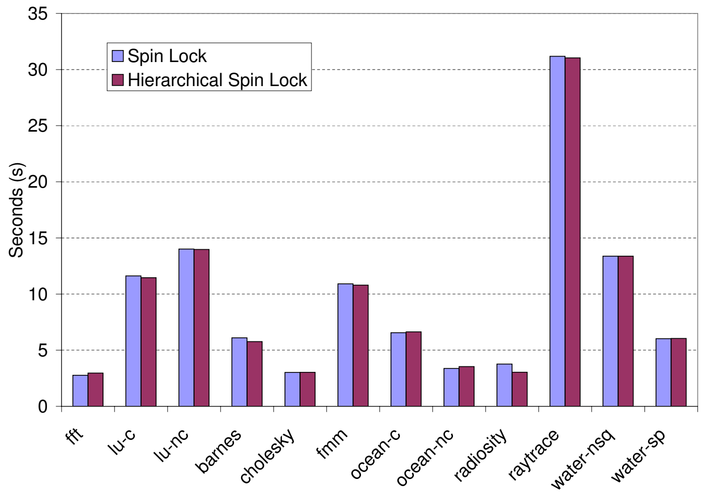

Figure 6: Execution times of SPLASH-2 suite with normal and hierarchical spin locks.

since there is probably not enough contention in the SPLASH-2 benchmarks with only 16-processor runs for the hierarchical lock to keep the lock in a node for long enough time to create locality. There is also quite a high cost in moving the hierarchical lock between the nodes which also makes the hierarchical lock less ideal when there is low contention. This is why the contention sensitive hierarchical lock was proposed (section 4.1). Even though the hierarchical lock does not shine when used with low contention we see that the hierarchical lock is always almost as good as the normal spin lock.

7 Conclusions

A new run-time system for DSZOOM has been created that supports process distribution across the cluster using robust MPI technology. A hierarchical spin lock synchronization primitive has been proposed and implemented that can be used on a cluster with any number of nodes and gives higher performance in tasks that has a lot of contention. In a micro-benchmark we have shown a 500% speed-up on a two node cluster with 20 processors. The snippets and protocol code have been rewritten and trampoline routine code added, to facilitate the use of real interconnect hardware so that DSZOOM can run on a real multi-system image cluster.

Future improvements to the hierarchical lock could include making the lock even more unfair by adding exponential backoff, thus increasing node locality.

Appendix A

A skeleton of a parallel PARMACS program.

```
#include <stdio.h>

MAIN_ENV

struct GlobalMemory {
  int id;
  LOCKDEC( idlock )
  /* ... */
} *Global;

#define DEFAULT_P 1     /* use one processor as default */
int P = DEFAULT_P;      /* number of processes */

void SlaveStart( void );

int main( int argc, char *argv[] )
{
  /* Parse command line arguments to find
   * out desired number of processes and
   * assign the number to P
   */

  // ... P = ...; ...

  MAIN_INITENV();

  /* Allocate global memory (G_MALLOC) and
   * initialize synchronization primitives
   * (LOCKINIT, BARINIT, ...)
   */

  Global = (struct GlobalMemory *)
    G_MALLOC( sizeof ( struct GlobalMemory ) );
  LOCKINIT( Global->idlock );
  Global->id = 0;
  // ...

  /* Create child processes */
  for ( i = 1; i < P; i++ ) {
    CREATE( SlaveStart );
  }
  /* Master process does parallel work aswell */
  SlaveStart();

  /* Wait until all child processes has finished */
  WAIT_FOR_END( P - 1 );

  MAIN_END();
  return 0;
}

void SlaveStart( void )
{
  int MyNum;

  /* Each process acquires a unique id number */
  LOCK( Global->idlock ); // enter CS
  MyNum = Global->id;
  Global->id++;
  UNLOCK( Global->idlock );       // leave CS

  /* Wait for P processes to arrive here */
  //START_TIME( MyNum, P );

  /*
   * Perform parallel work
   * using shared-memory and
   * synchronization primitives
   */

  // ...

  /* When all work is done, just return from
   * the function and the process will terminate */
}
```

References

[1] E. Artiaga, X. Martorell, Y. Becerra, and N. Navarro. Experiences on Implementing PARMACS Macros to Run the SPLASH-2 Suite on Multiprocessors. In *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, January 1998.

[2] E. Artiaga, N. Navarro, X. Martorell, and Y. Becerra. Implementing PARMACS Macros for Shared-Memory Multiprocessor Environments. Tech-

nical Report UPC-DAC-1997-07, Department of Computer Architecture, Polytechnic University of Catalunya, January 1997.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*, pages 152–164, October 1991.

[4] O. Grenholm, Z. Radović, and E. Hagersten. Latency-hiding and Optimizations of the DSZOOM Instrumentation System. Technical Report 2003-029, Department of Information Technology, Uppsala University, May 2003.

[5] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, February 1999.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 3rd edition, 2003.

[7] InfiniBand(SM) Trade Association, InfiniBand Architecture Specification, Release 1.0, October 2000. Available from: http://www.infinibandta.org.

[8] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[9] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[10] E. L. Lusk and R. A. Overbeek. Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors. Technical Report ANL-84-51, Rev. 1, Argonne National Laboratory, June 1987.

[11] L. Noordergraaf and R. van der Pas. Performance Experiences on Sun's Wildfire Prototype. In *Proceedings of Supercomputing '99*, November 1999.

[12] Z. Radović and E. Hagersten. Removing the Overhead from Software-Based Shared Memory. In *Proceedings of Supercomputing 2001*, Denver, Colorado, USA, November 2001.

[13] Z. Radović and E. Hagersten. Efficient Synchronization for Nonuniform Communication Architectures. In *Proceedings of Supercomputing 2002*, Baltimore, Maryland, USA, November 2002.

[14] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. Technical Report 97/3, Western Research Laboratory, Digital Equipment Corporation, February 1997.

[15] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174–185, October 1996.

[16] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, C. E. Lucas, S. S. Mukherjee, S. K. Reinhardt, E. Schnarr, and D. A. Wood. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[17] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of IEEE Hot Interconnects IV*, pages 41–52, August 1996.

[18] S. J. Sistare and C. J. Jackson. Ultra-High Performance Communication with MPI and the Sun Fire Link Interconnect. In *Proceedings of Supercomputing 2002*, Baltimore, Maryland, USA, November 2002.

[19] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating System Principle*, October 1997.

[20] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual, Version 9.* PTR Prentice Hall, Englewood Cliffs, New Jersey, 2000.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.