CableS : Thread Control and Memory System Extensions for Shared Virtual Memory Clusters

Peter Jamieson and Angelos Bilas

Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
{jamieson,bilas}@eecg.toronto.edu

Abstract. Clusters of high-end workstations and PCs are currently used in many application domains to perform large-scale computations or as scalable servers for I/O bound tasks. Although clusters have many advantages, their applicability in new areas and especially in areas of commercial applications has been limited. One of the main reasons for this is the fact that clusters do not provide a single system image and thus are hard to program. In this work we address this problem by providing a single cluster image with respect to thread and memory management to programmers. The main limitation of our system is that it does not yet provide file system and networking support across cluster nodes. We implement our system on a 16-processor cluster interconnected with a low-latency, high-bandwidth system area network. We demonstrate the versatility of our system with a wide range of applications. We show that clusters can be used to support applications that have been written for more expensive tightly–coupled systems, with very little effort on the programmer side. Finally, we show that the overhead introduced by the extra functionality of *CableS* affects the parallel section of applications that have been tuned for the shared memory abstraction only in cases where the data placement policy of the system results in improper placement due to operating system limitations in virtual memory mappings granularity.

1 Introduction and Background

Recently there has been progress in building high-performance clusters out of high–end workstations and low-latency, high-bandwidth system area networks (SANs). SANs, used as interconnection networks provide memory–to–memory latencies of under 10μs and bandwidth in the order of hundreds of MBytes/s . For instance, the cluster we are developing at the University of Toronto uses Myrinet as the interconnection network and currently provides one–way, memory–to–memory latency of about 8μs and bandwidth of about 120MBytes/s. Similar clusters are being built at many other research institutions.

Moreover, the shared memory abstraction, is used in an increasing number of application areas. Developers have been writing new applications for the shared

R. Eigenmann and M.J. Voss (Eds.): WOMPAT 2001, LNCS 2104, pp. 170–184, 2001.

address space abstraction and legacy applications are being ported to the same abstraction as well. Finally, most vendors are designing both small–scale symmetric multiprocessors (SMPs) and large–scale, hardware cache-coherent distributed shared memory (DSM) systems, targeting both scientific and commercial applications.

Shared memory clusters are an attractive approach to providing affordable and scalable compute cycles and I/O. For this reason, there has recently been a lot of work on designing efficient shared virtual memory (SVM) protocols for such clusters [23,16,26,13]. These protocols take advantage of features provided by SANs, such as low–latencies for short messages and direct remote memory operations with no remote processor intervention [12,10,9], to improve system performance and scalability [16]. Providing a shared memory programming abstraction on clusters has made it easier to run applications that have been written for more traditional, tightly–coupled multiprocessors (both shared bus and distributed shared memory machines).

Recent work [23,16,26] has shown that the performance of SVM clusters is competitive for wide ranges of applications to more traditional, tightly–coupled multiprocessors. For instance, the authors in [16] find that a 64–processor cluster offers, for most SPLASH-2 [24] applications (after a number of optimizations), at least 50% of the performance offered by a 64–processor SGI Origin2000.

However, despite the many advantages of clusters over more traditional, tightly–coupled scalable servers and the competitive performance they offer, their use is not widespread, especially in areas of commercial applications. One of the reasons is that despite the progress on the performance side, it still is a very challenging task to port existing applications or to write new ones for the shared memory programming APIs provided by clusters.

Although these APIs provide sufficient primitives to write parallel programs, they also impose several restrictions: (i) Processes cannot always be created and destroyed on the fly during application execution. (ii) Programmers allocate shared memory only during program initialization and should not free memory until the end of execution. (iii) In most shared memory clusters the synchronization primitives supported are *lock/unlock* and *barrier* primitives. However, more modern APIs support conditional waits as well as other primitives.

These limitations are not very important for large classes of scientific applications that are well structured. However, they pose important obstacles for using clusters in areas of applications that exhibit a more dynamic behavior, such as commercially–oriented applications. Thus, in many areas, traditional shared memory multiprocessors are used because of their ability to support legacy applications with no or very few changes. Also, development of new applications usually occurs on these architectures, since they provide APIs that pose fewer restrictions to the programmer. In essence, current clusters that support shared memory provide a very limited single system image to the programmer with respect to process, memory management, and synchronization.

In order to overcome the above limitations and to provide a more complete and functional single cluster image to the programmer, we design and implement

a *pthreads* interface on top of our cluster. This allows existing *pthreads* programs to run on our system with minor modifications. Programs can dynamically create and destroy threads on our system, allocate global shared memory throughout execution, and use all synchronization primitives specified by the *pthreads* API. More specifically, our system, *CableS* (**C**luster en**able**d thread**S**), provides:

Support for dynamic node and thread management: *CableS* allows the application to dynamically create threads at any point during execution. Currently, new threads are allocated to nodes with a simple, round–robin policy. On the fly, the system performs all the necessary initialization to support the *pthreads* API.

Support for dynamic memory management: *CableS* addresses a number of issues with respect to memory management. (a) It provides all necessary mechanisms to support different memory placement policies. Currently, *CableS* supports first touch placement, but can be extended to support other policies as well. (b) It provides the ability to allocate global, shared memory dynamically at any time during program execution. (c) It deals with static global variables in a transparent way.

Support for modern synchronization primitives: *CableS* supports the conditional wait primitives.

The main limitation of *CableS* is that, although it provides a single system image with respect to thread management, memory management, and synchronization support, it does not yet include file system and networking support across cluster nodes.

We demonstrate the viability of our approach and the versatility of our system by using a wide range of applications: (a) We run existing *pthreads* applications with minor modifications. (b) We use a public–domain OpenMP compiler, OdinMP [7], that translates OpenMP programs to *pthreads* programs for shared memory multiprocessors and run the translated OpenMP programs on our system. Our system supports the OpenMP programs with no modifications to the OpenMP source and minor modifications to the *pthreads* sources. (c) We provide an implementation of the M4 macros for *pthreads* and we run some SPLASH-2 applications.

We also show that the overhead introduced by the extra functionality affects the parallel section of applications that have been tuned for the shared memory abstraction only in cases where the data placement policy of the system results in improper placement due to operating system limitations in virtual memory mappings granularity. In the SPLASH-2 applications most overhead is introduced during application initialization and termination, whereas the execution time of the parallel section is only affected by the data placement, currently determined by our first touch policy.

The paper is organized in the following sections. Section 2 introduces our platform and current architectural considerations for modern clusters. Section 3 describes the design of *CableS*. Section 4 presents our experimental results. Section 5 presents related work and Section 6 discusses our high level conclusions.

2 Modern Clusters and System Area Networks

Nodes in modern clusters are usually interconnected with low–latency, high–bandwidth SANs that support user–level access to network resources [12,9,5]. By allowing users to directly access the network without operating system intervention, these systems dramatically reduce latencies compared to traditional TCP/IP based local area networks. Moreover, to further reduce latencies in SANs, direct memory operations are usually supported; reads and writes to remote memory are performed without remote processor intervention.

This mechanism provides fast access to remote memory within a cluster. However, there are a number of limitations associated with these operations and with modern SANs in general: (i) Connection establishment requires mapping of remote memory locally, which is an expensive operation. This operation usually sets up some form of page table on the network interface card (NIC) that allows it to access remote memory. (ii) The amount of remote memory that can be mapped is limited. (iii) To provide asynchronous communication primitives, most communication layers use a notification mechanism. Notifications use the interrupt mechanism provided by the operating system and are usually very expensive compared to direct remote operations.

The specific system we use consists of a cluster of 8 2-way PentiumPro SMP nodes interconnected with a Myrinet network. Each SMP is running WindowsNT. The nodes in the system are connected with a low–latency, high–bandwidth Myrinet SAN [5]. The software infrastructure in the system includes a custom communication layer and a highly optimized SVM system.

The communication layer we use on top of Myrinet is a user-level communication layer, Virtual Memory Mapped Communication (VMMC) [2,9]. VMMC provides both explicit, direct remote memory operations (reads and writes) and notification–based send primitives.

The SVM protocol used is GeNIMA [16], which is a home-based, page-level SVM protocol. The consistency model in the protocol is Release Consistency [11]. GeNIMA provides an API based on the M4 macros, which are extensively used for writing shared memory applications in the scientific computing community.

3 System Design

CableS supports a full *pthreads* (POSIX threads IEEE POSIX 1003.1 [1]) API which enables legacy shared memory applications written for traditional, tightly coupled, hardware shared memory systems to run on shared memory clusters. Within the *pthreads* API, *CableS* addresses the following issues: (i) It supports the *pthreads* API. (ii) It provides support for dynamic global memory management.

3.1 Thread Management

Thread and node management is the core of the API in which a thread can be created and administrated. The API also includes mechanisms to kill threads, cancel threads, and store thread private data.

In a distributed environment, threads of execution need to be started and administered on remote systems. For this purpose, *CableS* needs to maintain and manage global state that stores location and resource information about each thread in an application. Thus, *CableS* uses per application global state, called the application control block (ACB). This state is updated by all nodes in the system via direct remote operations as well as notification handlers.

CableS' communication library, VMMC, provides communication primitives that allow one node to perform reads and writes to another node's memory without interrupting the remote processor. *CableS* maintains the most up to date system information on the first node where the application starts (master node). To ensure consistency of the ACBs, updates are performed either by the master node through remote handler invocations, or by node update regions in which the system guarantees that the node is the exclusive writer.

The thread management component of the *pthreads* library is hinged around thread creation. Thread creation in *CableS* involves one of three possible cases: (i) Create a thread on the local node. (ii) Create a thread on a remote node that is not used by this application. This operation is called attaching a remote node to the application. (iii) Create a thread on an already attached remote node.

Local thread creation is equivalent to a call to the local operating system to create a thread. *CableS* creates remote threads by a combination of direct memory operations and remote handler invocation; thus, remote thread creation is expected to be a fairly expensive operation.

When *CableS* needs to attach a new node to the application, the master node M creates a remote process on the new node N. Node N, starts executing the initialization sequence and performs all necessary mappings for the global shared memory that is already allocated on M. N then retrieves global state information from M including shared memory mappings and sends an initialization acknowledgment back to M. M broadcasts to all other nodes in the system that N exists and that they can establish their mappings with N. At the end of this phase, node N has been introduced into the system and can be used for remote thread creations.

The remaining thread management operations involve mostly state management, mainly, through direct reads and writes to global state in the ACB.

3.2 Synchronization Support

The *pthreads* API provides two synchronization constructs *mutexes* and conditions. Current SVM APIs that mostly target compute-bound parallel applications provide two other synchronization primitives, locks and barriers.

Since mutexes and locks are very similar, we use the underlying SVM lock mechanism to provide mutexes in the *pthreads* API. For performance reasons,

locks are implemented in SVM as spin locks, and we maintain this implementation in *CableS*.

The *pthreads* condition is a synchronization construct in which a thread waits until another thread sends a signal. As with mutexes, conditions can be implemented either by spinning on a flag or by suspending the thread on an operating system event. Although implementations that use spinning consume processor cycles, they are more common in parallel systems to reduce wake–up latency. For this reason, our first implementation of *pthreads* conditional wait primitives uses spinning.

Global synchronization (barriers) can be implemented in *pthreads* with mutexes (or conditions). However, to support legacy parallel applications efficiently we extend the *pthreads* synchronization to support a barrier operation.

3.3 Memory Subsystem

Current System Limitations. Modern SANs that support direct remote memory operations, such as remote read and write operations require some form of memory registration to avoid remote processor intervention and interrupts. In these mechanisms, a node maps one or more regions of remote memory to the local network interface card (NIC) and it performs direct operations on these regions without requiring processor intervention on the remote side. This mapping operation is called registration and usually requires work at both the sending as well as the receiving NIC. SVM systems on clusters interconnected with SANs take advantage of these features to reduce the overhead associated with propagation and obtaining updates of shared data [23,16]. For this purpose, they perform all necessary registration operations at initialization time. This results in a number of limitations:

All shared memory has to be allocated at initialization time, since all memory registration operations happen at initialization.

In most SANs today [8,12,9] there is a limit (a) on the number of memory regions and (b) on the total amount of memory that can be registered on the NIC. Each page in the working set of each process should be placed, for performance reasons, on the node where the process runs. The registration limitations conflict with this requirement.

On today's systems there exist three possible solutions to this problem: (i) Shared pages could be grouped in regions and mapped together to solve registration limitations. In this case, pages in the working set of a process will have their primary copies in remote nodes resulting in excessive network traffic and performance degradation. (ii) Place the primary copies of pages in the working set on the node where the process runs. In this way the registration limitations may be violated since there will be a large number of non-contiguous memory regions that have to be registered. (iii) The many non-contiguous regions could be registered in one operation, including the gaps between regions. However, this results in registering essentially all the shared address space. This is not feasible due to the total amount of memory which can be registered.

In most SVM systems today, global static variables are not included in the shared address space. These are global variables that are declared statically in the user program. In the threads programming model, these variables are visible to all threads; however, this is not true in most SVM systems. The compiler/linker automatically allocates these variables to a designated part of the virtual address space. Since this part of the address space is not under the control of the shared memory protocol, static global variables can not be shared across nodes. This imposes additional challenges in the process of porting existing shared memory applications to clusters.

Effects on System API. The above limitations impact the API provided to users in many ways. To support a global shared address space on a cluster without hardware support, most systems today perform the following steps:

1. Start and initialize all the nodes that will be used to run the application at the same time.
2. Allocate a region of the virtual address space on each node. This step is usually fairly simple, since it involves using a system call to reserve part of the application virtual address space.
3. Determine which node will maintain the primary copy of each portion of the shared memory.
4. Establish communication–layer mappings between the primary copy of this region and the same region on all other nodes. These mappings are established by filling in the necessary information in the NIC page table.
5. Provide the application on each node with a convenient interface to the shared address space. This must consider the current restrictions on the usage of the shared address space: applications cannot use static global variables, nor allocate/deallocate shared memory after thread creation.

Proposed Solution. Existing systems deal with these issues by imposing API limitations that make it easy to avoid the related problems. The result is inflexible systems that are not easy to program. *CableS* deals with most of the issues above as follows.

Shared memory allocation and registration: Initially, one contiguous part of the physical address space in each node is used to hold the primary copies of shared pages that will be allocated to this node. This part of the physical address space is always pinned (can't swap out of RAM), since it will be accessed remotely by other nodes. The primary copies are mapped twice to the virtual address space of the process. One mapping is to a contiguous part of the virtual address space and is used only by the protocol . The second mapping is used by the application to access the shared data. For this mapping, the home pages are divided in groups of fixed size (in the current system 64 K-Bytes) and are mapped to arbitrary locations in the virtual address space of the process. It is important to note that these locations are not necessarily contiguous.

As the application requires more shared memory, it first allocates a region in the global virtual address space. Then, it determines which node will hold the primary copies of these pages according to some placement policy (currently first touch).

When a home page is touched: (a) The home node extends the home pages section and registers the additional pages with the NIC. Then, it maps the virtual memory region to the newly allocated home pages. As the primary copies of shared pages are placed in different nodes, the home pages portion of the physical address space is mapped to non-contiguous regions of the shared virtual address space in the home node. (b) Every other node in the system, registers the newly allocated virtual memory region with the NIC so that each node can fetch updates from the primary copies and rely on the OS to allocate arbitrary physical frames for these pages.

First touch policy: Implementing a first touch policy requires that the system delays binding of virtual addresses until the region is first read or written. *CableS* maintains information about each memory segment allocated in the global directory. During execution, when a node touches the segment, it uses the global directory to identify if the segment has been touched by anyone else. If it has, then the segment is registered with the NIC and is mapped to the corresponding region on the home node. If this is the first touch to the region, then the node becomes the home by updating the global information and by appropriately mapping the physical pages to its shared virtual address space so that the application can use it. Synchronization of the global information and ordering simultaneous accesses to a newly allocated region is facilitated through system locks.

Global static variables: CableS deals with global static variables in a transparent way. In current SVM systems and related APIs, such as M4, global static variables can only be pointers, allocated explicitly by the programmer. Explicit allocation greatly simplifies management of these variables, since the system can allocate them in designated areas of the shared address space [1]. However, this imposes a burden on the programmer, since they need to allocate each global variable explicitly. Moreover, these variables greatly hinder porting of existing *pthreads* applications to clusters.

CableS uses a type quantifier to allocate these global variables in a special area within the executable image. At application initialization, the first node in the system becomes the primary copy for this region. All necessary mappings are established to other nodes as they are attached to the application. Thus, static global variables of arbitrary types can be shared among system nodes.

[1] This assumes that the designated part of the virtual address space for static variables is the same in all processes. This is true in most systems today (or can be easily enforced).

3.4 Summary

CableS provides a shared memory programming model that is very similar to a *pthreads* programming model for tightly–coupled shared memory multiprocessors, such as SMPs and hardware DSMs. To run any *pthreads* program on *CableS*, the following modifications are required:

1. Determine if the program will perform correctly under a Release Consistency memory model.
2. Add the *pthread_start* and *pthread_end* library calls.
3. Prefix all static variables that will be globally shared with the *GLOBAL* identifier.
4. (OPTIONAL) Optimize the code so that threads touch data they require.
5. Link with *CableS*' library.

4 Results

In this section we present three types of results: (i) We demonstrate that legacy *pthreads* programs written for traditional hardware shared memory multiprocessors can run with minor modifications on *CableS*. (ii) As an extension of (i) we show that OpenMP programs can be run by translating them to *pthreads* programs by using a public domain OpenMP compiler, OdinMP [7]. (iii) We provide an implementation of the M4 macros for *pthreads* and run a subset of the SPLASH-2 applications.

4.1 Legacy *pthreads* Programs

We use the five simple steps, outlined above, to convert three publicly available *pthreads* programs for *CableS*. The programs are: (i) Prime number (PN), which, as indicated by its name, computes all prime numbers in a user specified range. (ii) Producer–consumer (PC), a producer–consumer program which runs with two threads. (iii) Pipe (PIPE), which creates a threaded pipeline where each element stage consists of a calculation. Next, we use OdinMP to compile to *pthreads* three SPLASH-2 applications that have been written for OpenMP: FFT, LU, and OCEAN.

Table 1 shows the *pthreads* programs which were run on *CableS*, and the *pthreads* calls each of the programs make, along with the average execution time of each *pthreads* functions. Note that PC only uses two threads; therefore, this program runs on only one node.

Performance-wise, PC shows the approximate cost of local API operations. PN, PIPE, and the OpenMP programs provide an indication of the average execution time of remote operations in *CableS*. We see that local operations are about three orders of magnitude faster than remote operations. With respect to synchronization operations, conditional waits and mutex lock operations include the cost of communication and the application wait time. For example, in PIPE the condition is used by each stage of the pipe to wait for work. Therefore, a stage

Table 1. Shows pthread programs with their respective pthread function calls and execution times (in ms) for the basic API operations.

PROGRAM	C	J	L	Co	Ca	K	G	Cr	Lo	Un	Wa	Si	Br	Sp
PN	•	•	•	•	•		•	2254	23	2	6154	-	1	15677
PC	•	•	•	•			•	1.1	0.05	0.005	17	0.042	-	-
PIPE	•		•	•			•	1008	52	3	527	12	-	11249
OMP_FFT	•		•	•		•	•	1235	54	0.52	1382	0.146	1.1	12302
OMP_LU	•		•	•		•	•	1247	133	1	327	0.134	0.401	12412
OMP_OCEAN	•		•	•		•	•	1312	49	2	494	0.293	0.606	14222

LEGEND: **C** = pthread_create, **J** = pthread_join, **L** = mutexes, **Co** = conditions, **Ca** = thread cancel, **K** = thread specific information, **G** = program uses static global variables **Cr** = create, **Lo** = mutex locks, **Un** = mutex unlock, **Wa** = condition wait, **Si** = condition signal, **Br** = condition broadcast, **Sp** = spawn time.

in the pipe is dependent on the execution time of the previous stage. Condition signals and broadcasts are much faster since these involve sending only small messages to activate threads in remote nodes.

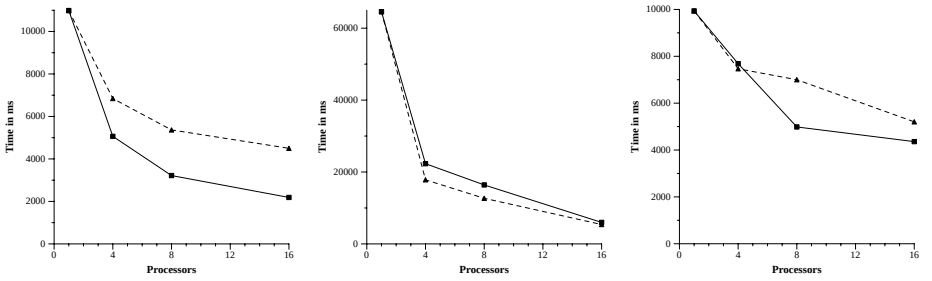
Table 2. Speedups for the three SPLASH-2 OpenMP programs

PROGRAM	4 processors	8 processors	16 processors
FFT	1.61	2.05	2.44
LU	3.17	3.71	7.10
OCEAN	1.33	1.43	1.92

Table 2 shows the speedups of the three OpenMP SPLASH-2 applications. We do not directly compare these results with our M4 results since OdinMP introduces overheads when translating OpenMP programs into *pthreads*. Also, we have not modified the resulting *pthreads* programs for optimal data placement.

4.2 SPLASH-2 Applications

To investigate the overhead *CableS* introduces in applications which have been tuned for the shared memory abstraction, we provided an implementation of the M4 macros on *CableS* and run a subset of the SPLASH-2 applications on two configurations: The original, optimized SVM system that we started from, GeNIMA [16], and *CableS*. In *CableS* we use the earlier introduced *pthreads* barriers, as opposed to a mutex-based implementation of barriers which only uses native *pthreads* calls. The motivation behind this extension is that the *pthreads* was not designed for parallel applications which frequently require global synchroniza-



(a) FFT (b) LU (c) Ocean

Fig. 1. SPLASH-2 M4 vs M4-pthread executions with 1, 4, 8 and 16processors. Solid line is the M4 executions, and dashed line isM4-pthread executions

tion. For a better comparison, specific knowledge provided by the SPLASH-2 applications about global synchronization should be exploited in both systems.

The applications we use are: FFT [4,24,25], LU [24,25], and OCEAN [6,22, 17]. Their common characteristic is that they are optimized to be single-writer applications; meaning, a given word of data is written only by the processor to which it is assigned. Given appropriate data structures, these applications are single-writer at page granularity as well, and pages can be allocated among nodes such that writes to shared data are almost all local. The applications have different inherent and induced communication patterns [24,14], which affect their performance and the impact on nodes.

Figure 1 shows the execution times of each application in both system configurations for 1,4,8, and 16 processors. We see that the current implementation of the first touch placement in *CableS*. Although this implementation results in similar speedup curves, it increases the absolute execution time in applications where the 64-KByte mapping granularity imposed by the operating system results in improper data placement. In these applications, although the granularity of sharing is still one page (4 KBytes), data is placed in nodes in chunks of 64 KBytes. This may result in additional diff computations, with more expensive page faults and synchronization.

FFT incurs higher data wait time on the first node as shown in Figures 2-a and 2-b; this leads to higher barrier synchronization time. Although the main data structures in FFT are placed properly by *CableS*, there are smaller data structures that reside on node 0. The original system, however, allocated these data structures in a round robin fashion. The reason for this difference is that *CableS* performs first touch allocation, and since node 0 initializes these data structures, all other nodes fetch from node 0. Each node in the system fetches about 5200 pages while node 0 only fetches 3600. Given that in the current VMMC implementation the incoming path has priority over the outgoing path, remote requests for shared pages create contention in the I/O and memory bus of node 0. This means page fetches incur higher delays resulting in high data

wait times. Finally, the additional traffic on the memory bus of node 0 increases memory overhead and affects compute time. One way to address this issue would be to distribute all application data structures properly.

Given the large granularity in LU (Figures 2-c and 2-d), the 64-KByte mapping granularity is not an issue. In fact, the performance of the parallel section is almost identical between the two configurations.

OCEAN (Figures 2-e–2-f) incurs higher overheads due to the higher granularity of data placement. The 64-Kbyte chunk size is a major data placement issue since 16 contiguous pages must have the same home node. OCEAN does not have large contiguous regions of data and suffers from misplaced pages. This causes contention within the network which increases synchronization overheads by about 139the average for locks and 106

5 Related Work

This work provides a *pthreads* API for a cluster interconnected with a SAN similar to DSM-Threads [21]. This work targets the implementation of a *pthreads* API on clusters of workstations. *CableS*, however, deals thoroughly with outstanding memory issues. The *pthreads* standard is defined in [1]. Most other related work in the area has focused on the following four directions:

To improve the performance of SVM on clusters with SANs. There is a large body of work in this category [23,18,16,26] . Our work relies on the experiences gained in this area and builds upon it to extending the functionality provided by today's clusters.

To provide OpenMP implementations for clusters. Relatively, little work has been done in this area. The authors in [19] provide an OpenMP implementation based on TreadMarks. They convert OpenMP directly into TreadMark system calls. They then compare the OpenMP programs to the native versions of the same applications.

To provide a *pthreads* interface on hardware shared memory multiprocessors, either shared–bus or distributed shared memory. Most hardware shared memory system and operating system vendors provide a *pthreads* interface to applications [20]. In many systems, this is the preferred API for multithreaded applications due to the portability advantages.

To provide a single system image on top of clusters. These projects focus on providing a distributed operating system on top of clusters. The focus is on providing an operating system that can manage all aspects of a cluster in multitasking environments and not on parallel applications. Also, the authors in [3] provide a Java Virtual Machine on top of clusters. This work focuses on Java applications and uses the extra layer of the JVM to provide a single cluster image. Our work is at a lower layer. For instance, a JVM written for the *pthreads* API, such as Kaffe [15] could be ported to our system.

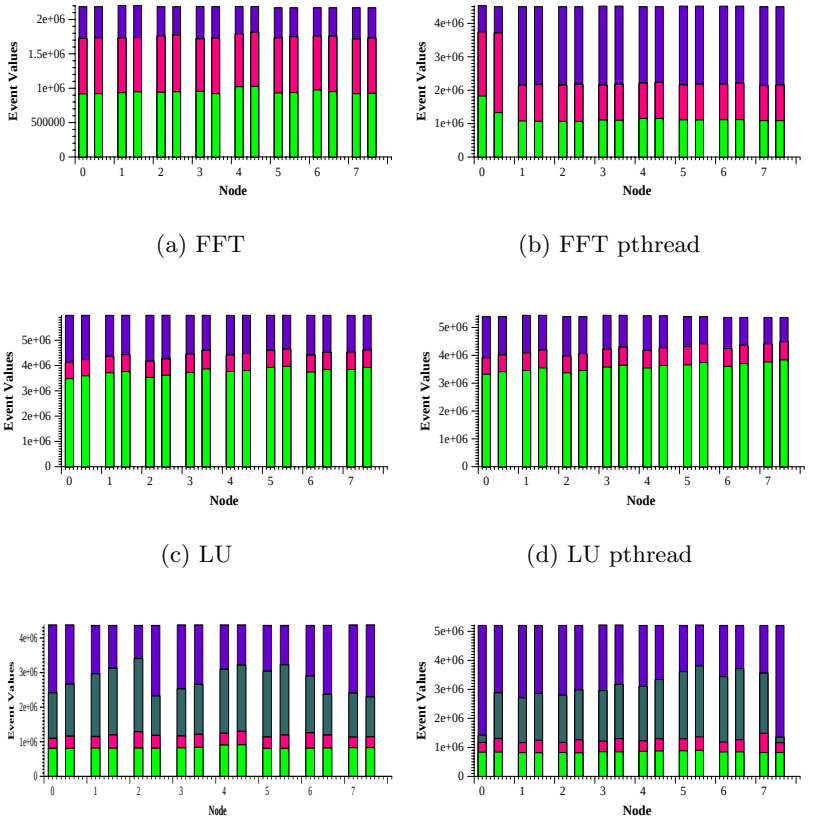


Fig. 2. SPLASH-2 M4 and M4-pthread execution breakdowns on16 processors. For (a), (b), (c), and (d) the lower boxes show compute time, the middle boxes show data time, and the top boxes show barrier time. For (e) and (f) the lower boxes show compute time, the next boxes show data time, the 2nd highest boxes show lock time, and finally, the top boxes show barrier time

6 Conclusion

In this work we design and implement a system that provides a single system image for SVM clusters. Our system supports the *pthreads* API and within this API, provides dynamic thread and memory management as well as all synchronization primitives. We show that this system is able to support *pthreads* applications written for more tightly–coupled, hardware shared memory multiprocessors. We use a wide suite of programs to demonstrate the viability of our

approach to make clusters easier to use in new areas of applications, especially in areas that exhibit dynamic behavior.

Our results show that existing applications can run on top of *CableS*, and applications tuned for performance on shared memory systems incur additional overhead only when the 64-KByte granularity of mapping physical to virtual memory results in a deviation from the first touch allocation and an improper data placement. The rest of the overhead introduced by *CableS* is limited to the initialization and termination sections of these applications.

Acknowledgments. We would like to thank Jeffrey Tang for his help with fixing problems in the VMMC firmware and Reza Azimi for providing help with extending the VMMC driver. Also, we would like to thank Anna Thelin for her OpenMP SPLASH-2 code, and Mats Brorsson for his help in obtaining OdinMP and OpenMP SPLASH-2 resources. Finally, Paul McHardy and Alexis Armour for their insights on the paper.

References

1. International standard iso/iec 9945-1: 1996 (e) ieee std 1003.1, 1996 edition (incorporating ansi/ieee stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995) information technology – portable operating system interface (posix) – part 1: System application program interface (api) [c language].
2. J. S. A.Bilas, C Liao. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proceedings of the The 26th International Symposium on Computer Architecture*, Atlanta, Georgia, May 1998.
3. Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. A high performance cluster jvm presenting a pure single system image. In *ACM Java Grande 2000 Conference*, 2000.
4. D. H. Bailey. FFTs in External or Hierarchical Memories. *Journal of Supercomputing*, 4:23–25, 1990.
5. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.
6. A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390, April 1977.
7. C. Brunschen and M. Brorsson. Odinmp/ccp - a portable implementation of openmp for c. *The 1st European Workshop on OpenMP*, 1999.
8. D. Cohen, G. G. Finn, R. Felderman, and A. DeSchon. The use of message-based multicomputer components to construct gigabit networks. *ACM Computer Communication Review*, 23(3):32–44, July 1993.
9. C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, Aug. 1997.
10. D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Proceedings of Hot Interconnects V Symposium*, Stanford, Aug. 1997.

11. K. Gharachorloo, D. Lenoski, and et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.
12. R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, Feb. 1996.
13. L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.
14. L. Iftode, J. P. Singh, and K. Li. Understanding application performance on shared virtual memory. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, May 1996.
15. T. T. Inc. Wherever you want to run java, kaffe is there.
16. D. Jiang, B. Cokelley, X. Yu, A. Bilas, and J. P. Singh. Applicaiton scaling under shared virtual memory on a cluster of smps. In *The 13th ACM International Conference on Supercomputing (ICS'99)*, June 1999.
17. D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.
18. P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, Jan. 1994.
19. H. Lu, Y. C. Hu, and W. Zwaenepoel. Openmp on networks of workstations. In *Proceedings Supercomputing*, 1998.
20. F. Mueller. A library implementation of posix threads under unix. In *Proceedings of the USENIX Conference*, pages 29–41, Jan. 1993.
21. F. Mueller. Distributed shared-memory threads: Dsm threads. *Workshop on Run-Time systems for Parallel Programming*, pages 31–40, April 1997.
22. J. P. Singh and J. L. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experiences, results, implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.
23. 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 *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, Oct. 1997.
24. S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, May 1995.
25. S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating message-passing in cache-coherent multiprocessors. In *Proceedings of Architectural Support For Programming Languages and Operating Systems*, 1994.
26. Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, Oct. 1996.