An Adaptive Heterogeneous Software DSM*

John Paul Walters
Institute for Scientific Computing
Wayne State University
jwalters@wayne.edu

Hai Jiang
Department of Computer Science
Arkansas State University
hjiang@astate.edu

Vipin Chaudhary
Institute for Scientific Computing
Wayne State University
vipin@wayne.edu

Abstract

This paper presents a mechanism to run parallel applications in heterogeneous, dynamic environments while maintaining thread synchrony. A heterogeneous software DSM is used to provide synchronization constructs similar to Pthreads*, while providing for individual thread mobility. An asymmetric data conversion scheme is adopted to restore thread states among different computers during thread migration. Within this framework we create a mechanism capable of maintaining the distributed state between migrated (and possibly heterogeneous) threads. We show that thread synchrony can be maintained with minimal overhead and minimal burden to the programmer.*

1. Introduction

Grid Computing has demonstrated that current computation technologies focus more on collaboration, data sharing, cycle stealing, and other modes of interaction among dynamic and geographically distributed organizations [7]. Studies have indicated that a large fraction of workstations may be unused for a large fraction of time [1]. Collecting and orchestrating these otherwise idle machines will utilize these computing resources effectively and provide common users a virtual supercomputing platform to solve more complex problems. Such dynamically generated virtual supercomputers benefit both users and systems by speeding up application execution and by improving throughput. However, in grids and other open and heterogeneous distributed systems, utilizing computational power adaptively and effectively is still an unsolved problem.

Increasingly, parallel processing is being seen as the only cost-effective method for the fast solution of computationally large and data-intensive problems [8]. How to execute parallel programs within heterogeneous distributed systems is unclear. Multithreading is popular but the mobility of the finer-grained thread becomes the concern. Further, many thread migration packages only work in homogeneous environments with restrictions on thread stacks and memory addresses [5, 9]. Although virtual machine (VM) techniques have been used to hide platform heterogeneity, most VMs work at the system level and have difficulties in distinguishing individual applications. Even if some modified VMs can support heterogeneous thread migration [20], the requirement of pre-installed VMs prohibits them from being the solution to open systems in Grid Computing.

Another major issue of parallel computing in distributed systems is to share global data among threads spread across different machines. Since they have their own disjoint address spaces, common global data sharing is not as straightforward as it otherwise would be. Distributed shared memory (DSM) systems have been deployed for global data sharing. However, flexible heterogeneous DSMs are not popular. And most DSMs require a programmer's assistance, *i.e.*, new primitives have to be inserted manually.

This paper extends a thread migration package, *MigThread*, to overcome common difficulties in supporting adaptive parallel computing on clusters, including fine granularity, globally sharing, adaptivity, transparency, heterogeneity, shareability, and openness. Parallel computing jobs can be dispatched to newly added machines by migrating running threads dynamically. Thus an idle machines' computing power is utilized for better throughput and parallel applications can be sped up by load balancing/redistribution.

*This research was supported in part by NSF IGERT grant 9987598, NSF MRI grant 9977815, and NSF ITR grant 0081696 as well as the Wayne State Institute for Scientific Computing.

Proceedings of the 2006 International Conference on Parallel Processing Workshops (ICPPW'06)
0-7695-2637-3/06 $20.00 © 2006 IEEE

To support global data in parallel applications, a distributed shared data (DSD) scheme is proposed to share common variables among threads no matter where they move. No restriction is placed on platform homogeneity as in most DSM systems. Data copies will be synchronized without explicit primitives in programs. The granularity size of data management is flexible, *i.e.*, the inconsistency detection is handled at the page level whereas data updating is manipulated at the object level. Such a hierarchical strategy can reduce false sharing in page-based DSMs and achieve concurrent updating. Since the DSD is totally transparent to programmers, parallel computing can be ported from self-contained multiprocessors to heterogeneous adaptive distributed systems smoothly.

Our contributions in this paper are as follows:

- **Parallel to distributed applications**: Applications created using traditional threading systems, such as *Pthreads*, can be easily and automatically converted to our distributed threads systems for running on remote heterogeneous (or homogeneous) machines.
- **Consistency**: We provide a mechanism for threads (both homogeneous and heterogeneous) to maintain a consistent global state.
- **Transparency**: Unlike traditional DSMs, our heterogeneous strategy is completely transparent to the end user.

The remainder of this paper is organized as follows: Section 2 gives an overview of the related work. Section 3 introduces the thread migration package, *MigThread*, and our asymmetric data conversion technique. In Section 4 we describe our heterogeneous distributed state mechanism. Performance analysis and some experimental results are shown in Section 5. Finally, our conclusions and continuing work are presented in Section 6.

2. Related Work

There have been a number of notable attempts at designing process/thread migration and DSM systems. Most thread migration systems impose many restrictions and only work in homogeneous environments. Arachne [5] supports thread migration by adding three keywords to the C++ language and using a preprocessor to generate pure C++ code. Neither pointers nor the heap are supported here. Ariadne[14] achieves thread context-switching by calling C-library setjmp() and longjmp(). On destination nodes, stacks are scanned for pointer detection which can fail and lead to incorrect results. Many thread migration systems, such as Millipede[9], adopt an "iso-address" strategy. Such a strategy imposes strict restrictions on resources which affect their scalability and make them inappropriate for Grid Computing. JESSICA [20] inserted an embedded global object space layer and implemented a cluster-aware Java just-in-time compiler to support transparent Java thread migration. Since the installation of the modified JVM is required, open systems will face difficulties. Charm++ [13] and Emerald [12] are a new language and compiler designed to support fine-grained object mobility. Compiler-produced templates are used to describe data structures and translate pointers.

Process migration brings mobility to sequential computations. The Tui system [16] is an application-level process migration package which utilizes a compiler support and a debugger interface to examine and restore process states. It applies an intermediate data format, just as in XDR [17]. Process introspection (PI) [6] is a general approach for checkpointing and applies the "receiver makes right" (RMR) strategy [18]. Data types are maintained in tables and conversion routines are deployed for all supported platforms. Programmers must flatten down aggregate data types manually. SNOW [3] is another heterogeneous process migration system which tries to migrate live data instead of the stack and heap data. PVM installation is a requirement and because of this, communication states are supported.

Global Data sharing can be achieved by distributed shared memory/state systems. TreadMarks [2] is a DSM system with several advanced features, such as multiple writers, mirrored pages, and a relaxed memory consistency model, to produce an illusion of shared memory among computers. This page-based approach implies a false sharing problem because of the relatively coarse granularity of pages. Strings [15] is a thread-safe DSM supporting multithreaded applications. Most DSM systems work on homogeneous clusters. Mermaid [19] supports data sharing across heterogeneous platforms, but only for restricted data types. InterWeave [4] supports data sharing on the top of page-based DSM on heterogeneous clusters. Its data conversion scheme is similar to CGT-RMR where data is broken down to fields. However, pointers are not handled. Programmers need to code using new primitives.

3. Background

3.1. *MigThread*: Approach & Overview

MigThread is designed and deployed to distribute/redistribute jobs dynamically so that multithreaded parallel applications can move their threads around according to requests from schedulers for load balancing and load sharing [10]. When new machines join the system, the same applications need to be started remotely after static code/program migration. All reachable threads are activated and blocked for possible coming threads. *MigThread* adopts an "iso-computing" strategy, *i.e.*, threads can

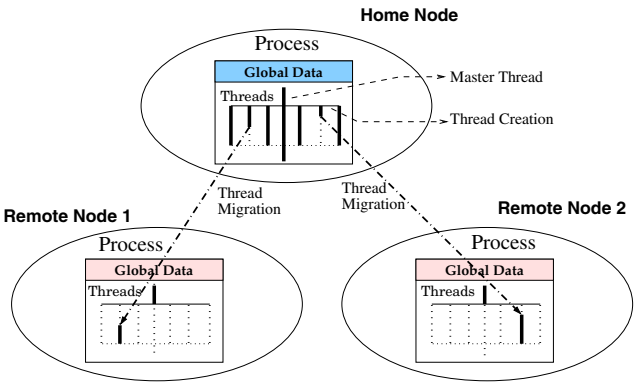

Figure 1. Thread migration with *MigThread*.

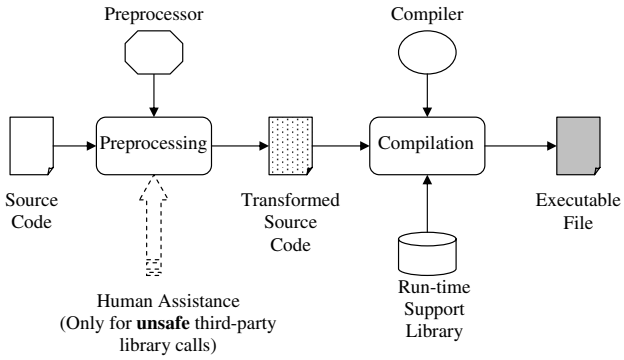

Figure 2. The Infrastructure of *MigThread*.

```
char MThV_heter[60]="(4,-1)(0,0)(4,1)(0,0)(4,1)(0,0)(8,0)(0,0)";
char MThP_heter[41]="(4,-1)(0,0)(4,-1)(0,0)";
```

Figure 3. Tag calculation at run-time.

only be migrated to the corresponding threads on remote machines as shown in Figure 1. For example, the second thread at one node can only be migrated to other second threads on other nodes. In fact, it is the application level thread state, not the thread itself that is transferred over. Once the receiver threads load the incoming states, they can continue the computation and complete the work. Threads can migrate again if the hosting node is overloaded.

MigThread supports home-based parallel computing. Parallel applications are initially started at one node, called the *home node*. Then the default thread, the *master thread*, spawns some slave threads, called *local threads*. When the same applications are restarted at newly joined machines (*remote nodes*), their default thread and slave threads act as *skeleton threads*, holding computing slots for migrating states. Once the state of a *local thread* at the *home node* is transferred, it becomes a *stub thread* for future resource access. The corresponding *skeleton thread* at a *remote node* is renamed to a *remote thread* to finish the rest of work. If the *master thread* moves to a default thread at a *remote node*, the latter will become the new *home node*. Previous *local threads* become *remote threads*, and some slave threads at the new *home node* are activated to work as *stub threads* for new and old *remote threads*. Whether a thread or node is *remote* or *local* is determined by its relationship with the initial *master thread*.

Since many user-level threads are invisible to operating system kernels, an application-level migration scheme is appropriate for both portability and heterogeneity. Thread states typically consist of the global data segment, stack, heap, and register contents. They should be extracted from their original locations (libraries or kernels) and abstracted up to the application level. Therefore, the physical state is transformed into a logical form to achieve platform-independence and reduce migration restrictions, enabling the proposed scheme to be independent of any thread library or operating system. User-level management of both the stack and heap are provided as well.

3.2. Data Conversion Scheme *CGT-RMR*

Since thread states have been abstracted at the language level and transformed into pure data, data conversion is necessary when parallel computing spans across different platforms. A data conversion scheme, called Coarse-Grain Tagged "receiver makes right" (CGT-RMR) is adopted to tackle data alignment and padding physically, convert data structures as a whole, and eventually generating a lighter workload compared to existing standards [11]. It accepts ASCII character sets, handles byte ordering, and adopts IEEE 754 floating-point standard because of its marketplace dominance.

In *MigThread*, tags are used to describe data types and padding so that data conversion routines can handle aggregate types as well as common scalar types.

The preprocessor defines rules to calculate structure members' sizes and variant padding patterns, and inserts **sprintf()** calls to glue partial results together. The actual tag generation takes place at run-time when the **sprintf()** statement is executed. On a Linux machine, the simple example's tags can be two character strings as shown in Figure 3.

A tag is a sequence of *(m,n)* tuples, and can be expressed as one of the following cases (where m and n are positive numbers):

- (m, n) : scalar types. The item "m" and "n" indicates the size and number of this scalar type, respectively.
- $((m', n')...(m'', n''), n)$: aggregate types. The "m" in the tuple (m, n) can be substituted with another tag (or tuple sequence) repeatedly. Thus, a tag can be expanded recursively until all fields are converted to scalar types. The last "n" indicates the number of the top-level aggregate types.

- $(m, -n)$: pointers. The "m" is the size of pointer type on the current platform. The "-n" sign indicates the number of pointers.
- $(m, 0)$: padding slots. The "m" specifies the number of bytes this padding slot can occupy. The $(0, 0)$ is a frequently occurring case and indicates no padding.

These tags, along with the preprocessor, allow *MigThread* to gather thread states into a portable format such that they can be restarted on remote, heterogeneous machines in the same manner as homogeneous machines.

4. Distributed Shared Data

Once a thread has been migrated we require a mechanism in order to ensure that an application's global state is properly maintained. Rather than act as a traditional DSM, we note that multithreaded applications that rely on a dynamic global space also require synchronization points to serialize access to the critical sections. To this end we implemented a release consistency model by extending the **pthread_mutex_lock()/pthread_mutex_unlock()** to a distributed lock/unlock mechanism.

A traditional DSM relies on the **mprotect()** system call in order to trap writes and propagate those changes through the DSM system. In a basic DSM a signal handler is installed to trap *SEGV* signals. When a *SEGV* is raised, the handler makes a copy of the page that triggered the *SEGV* then allows the write to continue by unprotecting the page in question. Eventually (depending on DSM optimizations, consistency models, etc.) the copied original pages (*twins*) are compared to their current page. A *diff* is taken between the *twin* and the current page. These differences can be propagated through the DSM system and applied directly to nodes owing to the fact that nodes are homogeneous to one another.

One major problem with this strategy is in heterogeneity. Since the typical DSM relies on a *twin/diff* strategy, it is unable to handle changes in page size, endianness, etc. In this case of a grid-type scenario, where many different machines are in use with little control over their architecture, this can be a major limitation.

Our solution addresses the problem of heterogeneity by abstracting all data to the application level. Like a traditional DSM, our strategy relies on detecting writes through the **mprotect()** system call. Since a machine is always homogeneous to itself, a *twin/diff* strategy will suffice for detecting writes. However, because our solution is designed specifically for heterogeneity, we cannot rely on the *twin/diff* strategy alone for detecting writes. Instead, we employ a *twin/diff* followed by a mapping stage where we abstract each page difference to an application-level index.

The application-level index is not, however, the complete tag as discussed in Section 3.2. Instead, a table is built upon application start-up that contains the tag information. Since the *MigThread* preprocessor collects all global data into a single structure, *GThV*, we need only maintain a table for *GThV*. Each row in the table represents an element from the *GThV* structure.

```
struct GThV_t{

        void * GThP;
        int A[237*237];
        int B[237*237];
        int C[237*237];
        int n;

} *GThV;
```

Figure 4. Example of source structure used to generate index table.

	Datatype	
Address	Size	Number
0x40058000	4	-1
0x40058004	0	0
0x40058004	4	56169
0x4008eda8	0	0
0x4008eda8	4	56169
0x400c5b4c	0	0
0x400c5b4c	4	56169
0x400fc8f0	0	0
0x400fc8f0	4	1
0x400fc8f4	0	0

Table 1. Index table generated from Figure 4.

A sample *GThV* structure, along with its corresponding index table, is shown in Figure 4 and Table 1 respectively. In Table 1, we keep track of the base address for each element in the *GThV* structure. Where *GThV* contains an array, the address in the table is the address of the first element of the array, and the number of array elements are noted in the *Number* column. A negative value is used for the *Number* field if the element in the corresponding row is a pointer. The *Size* column of the table contains the size of its corresponding element. The size used is that of the machine on which the table resides.

It is important to note that the table is architecture independent. Thus, while the data-type sizes may differ within the tables (depending on the architecture), the indexes of each element will remain the same. With each index then, it is straightforward to map the index to a memory address and vice-versa.

Once a *twin/diff* has been abstracted to an index, it can be formed into a tag along with the raw data and propa-

gated throughout the DSM system. The index mapping can be done very rapidly and adds very little overhead to the standard *twin/diff* method (see Section 5). Data conversion is done on an as-needed basis with homogeneous machines performing a simple **memcpy()** and heterogeneous machines performing byte-swapping, etc.

Typical DSMs can optimize the standard *twin/diff* method through optimizations at the page level. When differences exceed a certain threshold, for example, it is common to send the entire page rather than to continue with the *diff*. Since we seek a completely heterogeneous solution, we cannot perform optimizations at the level of the page. Instead, we take advantage of additional information contained in the table used for mapping *diffs* to indexes. Arrays can be easily identified, and we can transfer and convert**/memcpy()** large arrays quickly by dealing with them as a whole. In fact, this saves time and resources both in converting the data and in forming the tags used to identify heterogeneous data.

Our basic solution consists of four major functions:

- **MTh_lock(index, rank)**: Thread *rank* requests mutex *index*. Upon acquiring the lock, any outstanding updates are transferred to thread *rank* before **MTh_lock()** completes.
- **MTh_unlock(index, rank)**: Thread *rank* informs the base thread that mutex *index* should be released. Updates made by the remote thread (*rank*) are propagated back to the base thread at this time.
- **MTh_barrier(index, rank)**: Thread *rank* enters into barrier *index*. We provide barrier constructs to ease the programming burden as well as to speed up barrier processing. In so doing, programmers need not use the distributed mutex directly for barrier synchronization.
- **MTh_join()**: Each remote thread calls **MTh_join()** immediately prior to thread termination. This informs the base thread that it too should terminate, allowing the program to end gracefully via a call to **pthread_join()**.

We now discuss both the lock and unlock mechanisms in greater detail.

4.1. MTh_lock()

In order to ensure that a thread has an accurate view of the global space, an effective strategy must be employed that will propagate any outstanding updates to the thread acquiring the distributed lock. We note, however that there may be multiple threads operating in the global space at once, resulting in differing views of the *GThV* structure. In this case, we rely on the programmer to ensure that there are no race conditions. This is true for any multithreaded program.

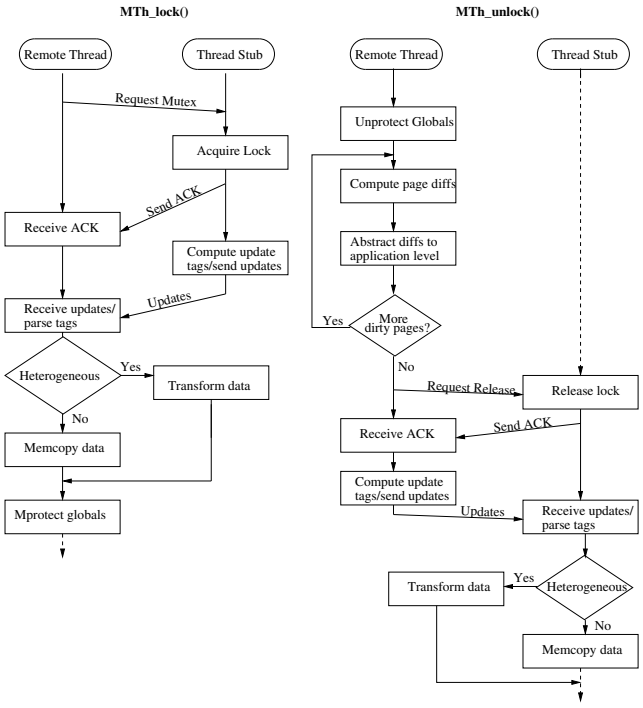

Figure 5. Overview of the lock/unlock mechanism.

The remote thread receives updates in the form of a series of tags and raw data. The tags indicate to the remote thread what the raw data represents and is also used in determining whether the remote thread and the home thread are homogeneous to one another. If both the home and remote threads are homogeneous to one another, a simple **memcpy()** can be used to copy the raw data into the appropriate memory locations. However, even in the case of two homogeneous threads, the remote thread must still parse through each tag to determine the correct memory location into which the raw data should be **memcpy()**'d.

In the event that the remote thread and the home thread are heterogeneous to one another the raw data must be converted to the native format using the CGT-RMR technique described in Section 3.2. The tags sent by the home thread will indicate the endianness of the host system as well as the size of each data type in the raw data. The remote thread can then compare the data sizes, endianness, etc. and convert the data appropriately.

The basis for our distributed locking mechanism lies in our ability to accurately detect writes to the global variables contained with the *GThV* structure. In order to detect writes, we use the **mprotect()** system call and a signal handler to trap writes to the *GThV* structure. Upon writing to a page in the *GThV* structure, a copy of the unmodified page is made and the write is allowed to proceed. This minimizes the time spent in the signal handler as subsequent writes to

the same page will not trigger a segmentation fault, but will instead go through directly.

4.2. MTh_unlock()

MTh_unlock() functions similarly to **MTh_lock()** (but opposite) with respect to the propagation of updates to and from the base node. **MTh_unlock()** however, is also responsible for mapping the detected writes to their actual memory locations/tags before requesting that the home thread release the lock.

After making a call to **MTh_unlock()** the remote thread must go about detecting the individual writes to each dirty page, mapping them to their base memory location, and then finally mapping the base memory location to the application-level tag that will be used in the actual data conversion upon updating the home thread. As we will discuss in Section 5, this process can become quite time-consuming for large updates as each byte on the dirty page must be compared to its corresponding byte on the original page.

After detecting the writes, the remote thread must release the distributed lock and propagate any outstanding changes back to the home thread. The process for propagating such changes is exactly the same as the **MTh_lock()** case, with the remote thread and the home thread switching places. See Figure 5 for a diagrammatic overview of the **MTh_lock()** and **MTh_unlock()** processes.

5. Performance Evaluation

We tested our system on a combination of Sun and Intel/Linux machines. Our Sun machine is a 4 CPU Sun Fire V440 (1.28 GHz) with 16 GB RAM. Our Linux system is a 2.4 GHz Pentium 4 with 512 MB RAM.

Our test programs consisted of a simple matrix multiplication and LU-decomposition code with square matrices of size 99x99,138x138, 177x177, and 255x255. Each test consisted of three threads, two of which were migrated while the third was not. Our system was tested for both homogeneous and heterogeneous cases. We characterize the homogeneous aspects of our system through the matrix multiplication example. The greatest amount of time is spent in the data conversion portion of our system, so we give a performance analysis for data conversion in both LU-decomposition and matrix multiplication.

When components of parallel applications are spread among multiple machines, a penalty is paid for data sharing. The penalty can be classified as follows:

$$C_{share} = t_{index} + t_{tag} + t_{pack} + t_{unpack} + t_{conv} \quad (1)$$

where t_{index} is the time required to map writes to the protected global space into indexes that will ultimately be converted into application-level tags, t_{tag} indicates the time

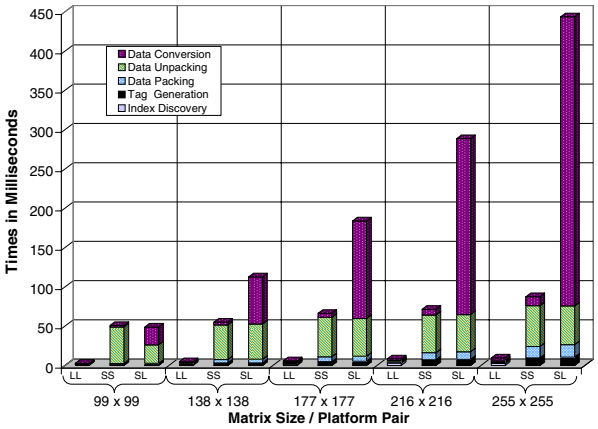

Figure 6. Data sharing overhead breakdown.

to generate tags from the indexes, t_{index}, while t_{pack} and t_{unpack} show the data packing/unpacking costs. t_{conv} is the data conversion time to update the copy at home node.

The extra data sharing costs of running the matrix multiplication application on clusters are shown in Figure 6. Platform pairs "LL", "SS", and "SL" represent Linux/Linux, Solaris/Solaris, and Solaris/Linux, respectively. When the sizes of matrices are increased, the overall cost and each individual cost also grow proportionally. Similar to all other distributed computing applications, our system faces communication and synchronization overheads. Among them, the costs of packing/unpacking, t_{pack} and t_{unpack}, are comparatively small. Therefore we primarily focus our discussion on t_{conv}, t_{tag}, and t_{index}.

Figure 7 summarizes our results for the matrix multiplication example. In this case, we show each component of our system as a percent of the total execution time. Notice that in the heterogeneous case, the data conversion portion quickly overtakes all other components as the matrix size increases, as is to be expected. In the homogeneous cases, the data conversion phase remains relatively low.

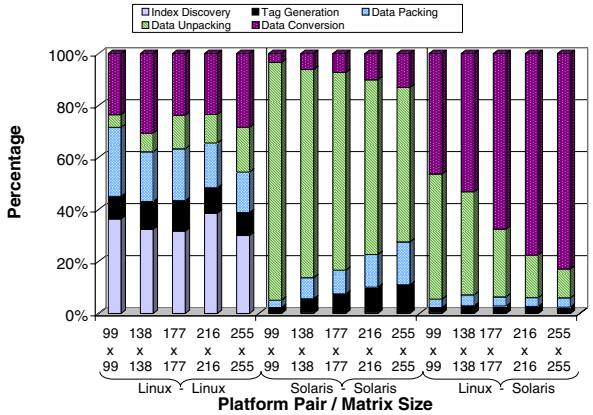

Figure 7. Costs as a percentage of total time.

In Figure 8 we examine the time required to map writes

to the protected global space into indexes that will ultimately be converted into application-level tags (t_{index}). This metric is a measure of the performance of the system on which the unlock takes place. It is possible that a series of updates can build up at the home node, resulting in a rather large batch update being transferred to a remote thread. In Figure 9 we see a spike for matrix size 216 resulting from just such a case. In the future, we hope to improve this worst-case performance.

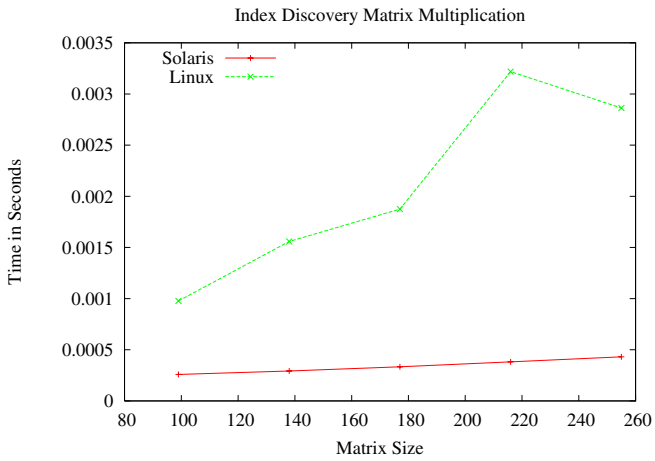

Figure 8. Mapping writes to their application-level indexes for the matrix multiplication sample code.

In Figure 9 we measure the time taken to convert the indexes measured in Figure 8 into the proper application-level tags (t_{tag}). In order to avoid the creation of tags for every array element that may have been modified between the lock/unlock, our system attempts to group consecutive array elements into a single tag. Thus, in many cases we can distill many (hundreds, perhaps thousands) indexes into a single tag. This allows for greater efficiency in the actual data conversion phase of the update. It also considerably reduces the time necessary to create tags as fewer calls to **sprintf()** are required. This, in turn, allows us to send fairly large batch updates to and from the home node with a minimal number of socket writes.

In Figure 10 we show the performance of the actual data conversion (t_{conv}). As we noted earlier, this is the most expensive portion of the distributed state process. In this case, we must take into account whether the system is actually homogeneous or heterogeneous. In the case of the homogeneous systems (Solaris/Solaris and Linux/Linux) we can clearly see that the data conversion time is quite minimal, even in the case of large updates. This is due to the fact that we can simply perform a **memcpy()** on the new data. Comparing Figure 10 with Figure 11 we notice that in the homogeneous case the timings are roughly similar, despite the fact that the LU-decomposition example transfers more

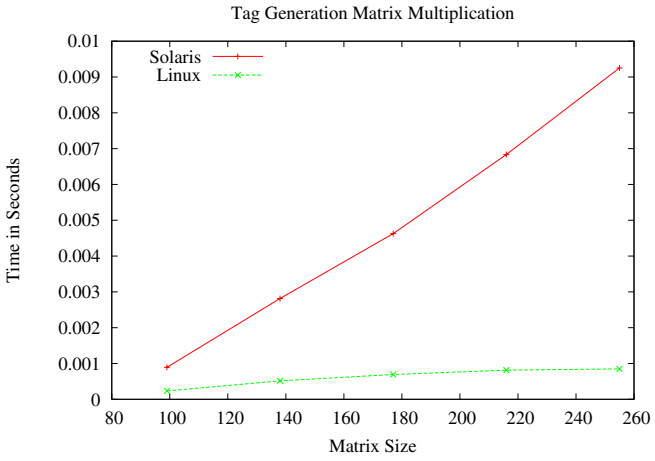

Figure 9. Forming application-level tags from the indexes, matrix multiplication example.

data per update than the matrix multiplication example. In the heterogeneous case, however, the size of the updates quickly becomes apparent as we note the cost of heterogeneity.

The primary reason for this great performance difference between the homogeneous and heterogeneous cases is that we are unable to perform a simple **memcpy()** in the heterogeneous case. Instead, we must (potentially) convert each byte of data in order to ensure program correctness. This requires not only byte swapping, and sign extension, but also greater interaction with the tags (a string comparison to ensure identical tags, as in the homogeneous case, is no longer sufficient). We are optimistic that the overhead due to heterogeneity can be improved, particularly by lessening our reliance on string operations with the tags.

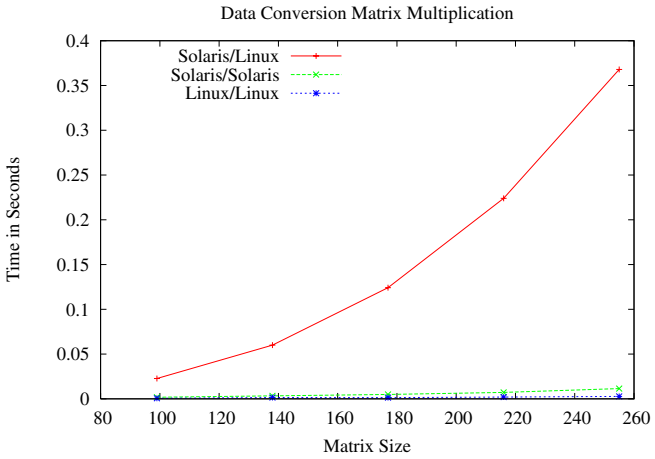

Figure 10. Data conversion for matrix multiplication.

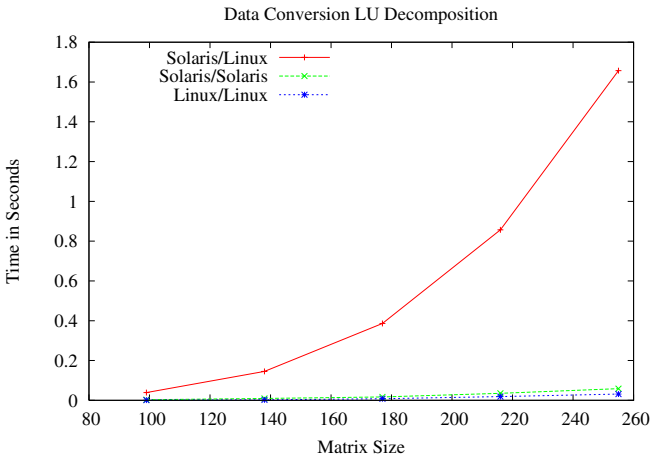

Figure 11. Data conversion for LU decomposition.

6. Conclusions and Future Work

In this paper we described our heterogeneous distributed shared memory system built on top of *MigThread*. We demonstrated that heterogeneity can be achieved by utilizing traditional DSM techniques and abstracting system-level data to the application-level for portability.

We have further shown that parallel threaded applications can be converted directly to a distributed system through the use of our pre-processor, allowing for the use of compute resources beyond the bounds of an individual workstation. Our distributed state primitives map easily to their *Pthreads* counterparts, providing a straightforward mechanism for porting parallel applications to distributed applications.

Work continues on improving and optimizing the heterogeneous portion of our distributed state mechanism. We hope to further reduce the time necessary to convert the data for even the largest of updates. Additional work, such as supporting file I/O migration and socket migration also continues as both will be necessary for a truly portable heterogeneous system.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The Utility of Exploiting Idle Workstations for parallel Computation. In *Proc. of the Conference on Measurement and Modeling of Computer Systems*, 1997.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2), 1996.

[3] K. Chanchio and X. H. Sun. Data Collction and Restoration for Heterogeneous Process Migration. In *Proc. of 21st International Conference on Distributed Computing Systems*, 2001.

[4] D.-Q. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. Scott. Multi-level Shared State for Distributed Systems. In *Proc. of International Conference on parallel Processing*, 2002.

[5] B. Dimitrov and V. Rego. Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):459–469, 1998.

[6] A. Ferrari, S. Chapin, and A. Grimshaw. Process introspection: A checkpoint mechanism for high performance heterogenesous distributed systems. Technical Report CS-96-15, University of Virginia, 1996.

[7] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. In *IEEE Computer*, 2002.

[8] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to parallel Computing*. Addison Wesley, 2 edition, 2003.

[9] A. Itzkovitz, A. Schuster, and L. Wolfovich. Thread Migration and its Applications in Distributed Shared Memory Systems. *Journal of Systems and Software*, 42(1):71–87, 1998.

[10] H. Jiang and V. Chaudhary. Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems. In *Proc. of the 37th Hawaii International Conference on System Sciences*, 2004.

[11] H. Jiang, V. Chaudhary, and J. Walters. Data Conversion for Process/Thread Migration and Checkpointing. In *Proc. of the International Conference on Parallel Processing*, 2003.

[12] E. Jul, H. Levy, N. Hutchinson, and A. Blad. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, 1998.

[13] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[14] E. Mascarenhas and V. Rego. Ariadne: Architecture of a Portable Threads system supporting Mobile Processes. Technical Report CSD-TR 95-017, Purdue Univ., 1995.

[15] S. Roy and V. Chaudhary. Design Issues for a High-Performance DSM on SMP Clusters. *Journal of Cluster Computing*, 2(3):177–186, 1999.

[16] P. Smith and N. C. Hutchinson. Heterogeneous Process Migration: The Tui System. *Software Practice and Experience*, 28(6), 1998.

[17] R. Srinivasan. XDR: External Data Representation Stndard. RFC, 1995.

[18] H. Zhou and A. Geist. "Receiver Makes Right" Data Conversion in PVM. In *Proc. of the 14th Int'l Conf. on Computers and Communications*, 1995.

[19] S. Zhou, M. Stumm, M. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Trans. on Parallel and Distributed Systems*, 3(5), 1992.

[20] W. Zhu, C.-L. Wang, and F. C. Lau. JESSICA2: A Distributed java Virtual Machine With Transparent Thread Migration Support. In *In Proc. of the CLUSTER*, 2002.