The Design of MPI Based Distributed Shared Memory Systems to Support OpenMP on Clusters

H'sien J. Wong [1] and A.P. Rendell [2]

Department of Computer Science, The Australian National University
Canberra ACT Australia
[1]Jin.Wong@anu.edu.au
[2]Alistair.Rendell@anu.edu.au

***Abstract*— OpenMP can be supported in cluster environments by using distributed shared memory (DSM) systems. A portable approach for building DSM systems is to layer it on MPI. With these goals in mind, this paper makes two contributions. The first is a discussion about two software DSM systems that we have implemented using MPI. One uses background polling threads while the other uses processes that are driven only by incoming MPI messages. Comparisons of the two approaches show the latter to be a more scalable architecture that is better suited for the multi-core processors that are becoming commonplace. The second contribution recognizes that a common workaround for sub-team synchronizations in OpenMP is to use the flush directive on shared variables within busy-wait loops. In such a situation, only the flush in the last iteration of the busy-wait loop will result in the conditions necessary for exiting the loop. Thus transfer of the shared value need only be done if there were changes. We implement in our DSM a flush mechanism that eliminates the unnecessary data transfers entirely without any additional support or hints from the programmer.**

I. Introduction

Software Distributed Shared Memory (DSM) systems provide an illusion of shared memory even on systems, such as clusters, where the memory is physically distributed. A motivation for doing this is that shared memory parallel programming is often considered simpler to use compared to message passing. One reason for this is that the shared memory programmer does not need to worry about how information gets from one cluster node to another, while the message passing programmer needs to match every send with a suitable receive; and this can quickly become very complicated when the communication pattern is irregular.

To date, however, parallel programming on clusters has been dominated by message passing, and in particular the Message Passing Interface (MPI). There are many MPI libraries available with prominent examples being MPICH, LAM-MPI, and Open MPI. As a result of this dominance, vendors of cluster network components often invest considerable time and energy in developing high performance MPI implementations that can make best use of their underlying hardware.

Although software DSM systems have been available for some time (e.g. [1], [2], and [3]) they have not been widely exploited on cluster computers. Two major reasons for this have been the lack of a standard interface (the equivalent of MPI), and the time required to port and optimize DSM systems to new hardware. This situation is, however, ripe for change. Notably OpenMP is now widely recognized as the shared memory programming paradigm of choice within the high performance computing community. Recent work that supports the use of OpenMP on clusters have included several implementations that utilize DSM systems to support shared memory programming on clusters (e.g. [4], [5], [6], and [7]). Furthermore, the release of a commercially supported version of OpenMP for clusters by Intel [8] has substantially raised the profile of software DSM systems. In spite of this, portability remains a major issue and there is a continued need for cluster OpenMP DSM systems that are vendor neutral, open source, and amenable to community development and research.

The goal of this work is to further the development of a portable OpenMP DSM system that is able to run on clusters while also exploiting the latest hardware advances. To this end the DSM systems outlined here use MPI to perform the underlying communication operations as this is able to exploit the very best MPI implementation on a given platform. Moreover, using MPI has other advantages, such as the ability to use MPI profiling and analysis tools to examine the performance of the DSM system, and also opens up the possibility of exploring programming models that use both MPI and OpenMP to share data between nodes on a cluster.

The work reported here builds on that previously reported by Ojima et al. who developed SCASH-MPI [9], an MPI based version of the SCASH [10] DSM system. In this implementation, additional polling threads are created that run alongside each MPI process to provide the one-sided memory operations required by the DSM. Requests from remote nodes for memory pages that reside on that node are directed to the communication thread that sits waiting to service incoming requests. In this respect the implementation is very similar to the data server model that has been used to implement Global Arrays [11] and the Distributed Data Interface [12] on some platforms.

The use of OpenMP on clusters is then supported by using the SCASH DSM library as a target platform for the Omni OpenMP compiler [13]. Briefly, Omni translates OpenMP source code into calls to various OpenMP runtimes based on different parallel programming approaches such as Pthread or the Unix `shmem` system calls. As the memory model of SCASH is similar to the "shmem memory model", the DSM is weaved into the OpenMP solution by modifying the `shmem` based runtime to use SCASH functions in place of `shmem` ones. Using the same technique, the runtime is modified to produce an OpenMP solution that is based on our DSM systems.

This paper makes two contributions. First we compare two competing approaches to implementing a DSM system over MPI, one is similar to Ojima et al. [9] in that it uses threads and requires polling, while the other uses only processes and is entirely event-driven. The second contribution involves how the OpenMP flush directive is handled in software DSM systems. Specifically, the flush operation involves bringing some region of locally cached memory into a consistent state with the equivalent region of global memory. As such, a common use of the flush directive is to implement non-standard synchronization between tasks by way of synchronizing shared variables and busy-wait loops. Taking advantage of flush semantics, the DSM is able to apply an invalidation protocol [2] to flushed regions. This reduces the overall impact of the busy-wait loop on network traffic and DSM performance by *eliminating* unnecessary data transfers.

The paper is structured into the following sections. First we describe our DSM system and the two alternative implementations. Next we consider flush operations and an observer approach. In section IV we provide experimental data to evaluate the different approaches, while in section V we give our conclusions.

II. DSM-over-MPI Implementations

The SCASH DSM is page based as well as home based. As a *page based* DSM, shared memory in SCASH is organized into fixed size chunks – called pages – that define the granularity of sharing. The DSM is *home based* in the sense that each of these pages have a home process at which the *master copy* of the page is maintained. If the master copy of a page is held at some process, then that page is a *remote page* from the point of view of *all other processes*. Before a remote page can be accessed (read or write), a copy of the page needs to be fetched from the home process asynchronously. This is done in SCASH through the Remote Memory Access (RMA) functionality provided by the PM [14] communications library. Unfortunately, PM's RMA capabilities are implemented through low level modifications to the host operating system and drivers for the relevant network cards. As such PM is not readily ported between machines, or run on clusters that are not under the administrative control of the user.

In contrast, the MPI standard is a common standard for message passing that is widely ported to many different platforms. Thus, by layering the DSM interface over MPI, a portable DSM implementation can be achieved. In porting SCASH to use MPI, SCASH-MPI replaces the functionality once provided by PM (both RMA and message passing) with a polling *communication* thread. The main thread which does everything else is referred to as the *computation* thread. Interaction between the computation and communications threads takes place via three shared queue data structures: 1) a message send queue, 2) a message receive queue, and 3) an RMA request queue. This provides the computation thread access to asynchronous message passing as well as RMA operations. The use of these queues is illustrated in figure 1.

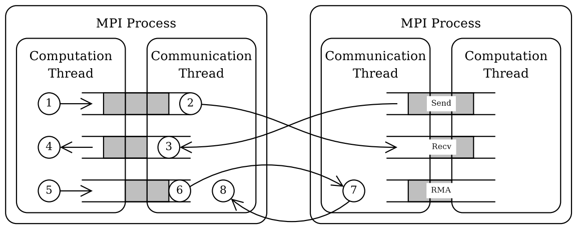

Fig. 1. SCASH-MPI Communications (adapted from figure 2 in [9]). (1) Message is placed on the send queue. (2) Message is picked up from the send queue and sent via MPI. (3) Message is received via MPI and placed on the receive queue. (4) Message is picked up from the receive queue. (5) RMA request is placed on the RMA queue. (6) RMA request is picked up and the correct asynchronous MPI call is issued for data transfer. After which, the RMA request is forwarded to the remote process. (7) The RMA request is received and the matching MPI call is issued for data transfer. (8) The asynchronous data transfer call has completed.

One consequence of dividing responsibilities between the computation and communication threads is that the computation thread can only check for incoming messages when it executes the DSM library code. Especially affected by this is the lock protocol used by SCASH-MPI. In this implementation, the status of each lock – whether free or acquired and by whom – is maintained by a lock master. To obtain a lock, the requester communicates its request to the master and the master should then check the status of the lock and respond accordingly. The fact that the master can only receive, and thus respond to, incoming messages when it executes DSM library code means that a successful lock acquire response can be delayed unnecessarily. For example, when the computation work done by the master process does not result in any page faults, nor does it require the use of any DSM functionality for some extended period, other processes attempting to acquire a lock will be delayed for the length of this period *even if the lock is not held by any process*.

The problem can be avoided if more DSM intelligence is given to the communication thread, allowing the lock response to take place in the background. Thus, in our implementations we take an alternative approach in which responsibility is divided according to whether a thread is fulfilling the *worker* or *DSM* roles. The role of the worker is to perform the computation pertaining to the application that uses the DSM, while the DSM thread performs all the necessary DSM related tasks. The merit of this partitioning is that one-sided DSM protocols can now be executed in the background. Using this role partitioning, we have implemented two DSMs over MPI. The first uses a background polling thread that plays the role of DSM, while the second uses an event-driven approach with separate MPI processes for both the DSM and worker roles.

A. Polling DSM

The polling DSM is similar to SCASH-MPI in that it uses a background thread. However, as previously mentioned, this thread now plays the DSM role rather than just that of a

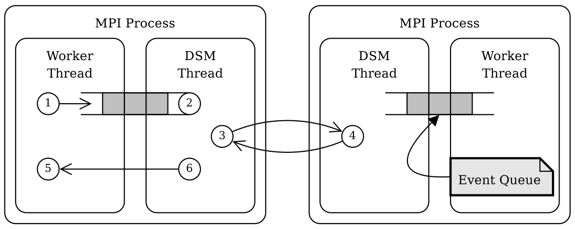

Fig. 2. The polling DSM. (1) An event is placed on the event queue. (2) An event is picked up from the event queue; the event handler is executed. (3) and (4) Inter-process communications that take place in the background. These may be started by a local or remote event or from other conditions which the polling DSM thread is able to monitor in the background. (5) If an event is blocking, the worker thread waits on `pthread_cond_wait` and (6) the DSM thread signals the blocked worker using `pthread_cond_signal`.

communication thread. Figure 2 shows the various kinds of communications that can take place in the DSM and that the three inter-thread queues that exist in SCASH-MPI (figure 1) have been replaced by a single first-in-first-out (FIFO) event queue. It is through this queue that the worker thread is able to request DSM services.

During each iteration of the polling loop, the event queue is checked for events and the various opened `MPI_Requests` are tested (points 2 and 3 of figure 2 respectively). The following pseudo code describes the polling loop that is done while the "polling" state is true.

```
dsm_poll():
  while polling:
    event = dequeue_head(event_queue)
    if event != NULL:
      event.handler(event)
    Check opened MPI asynchronous calls
```

If an event is blocking, then the worker thread waits using `pthread_cond_wait` after putting the event in the queue; resuming only when the event is over and the DSM thread calls `pthread_cond_signal` (points 5 and 6 in figure 2). Although the event queue is a shared data structure that is manipulated by both the worker and DSM threads, a lock-less implementation can and has been used as there is only one producer (worker thread) and one consumer (DSM thread) per queue. The sequence diagram in figure 3 illustrates the initialization, event and message polling, and finalization of the DSM.

B. Event-driven DSM

The previous DSM needed to do polling because it had to serve requests from an event queue as well as messages from remote DSM threads. The event-driven DSM avoids the need to do polling by transmitting all requests to the DSM as MPI messages. To do this only MPI processes are created that are then assigned either worker or DSM roles during DSM initialization. Figure 4 shows the communications that can take place in the event-driven DSM. Figure 5 depicts a sequence diagram involving a pair of worker and DSM processes. As shown, the DSM process never returns from the

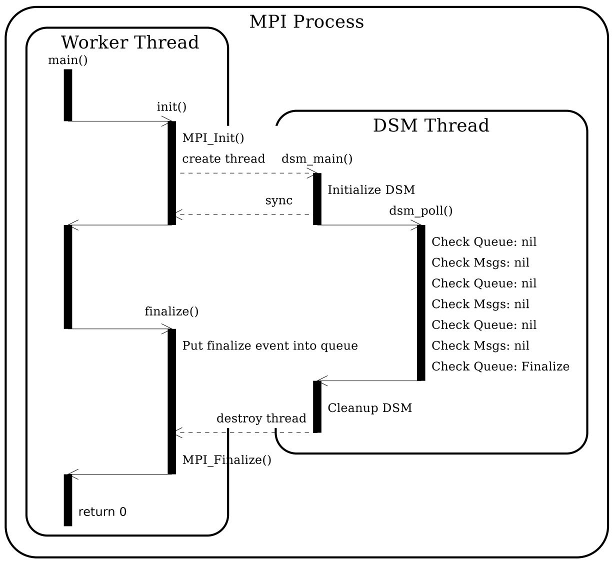

Fig. 3. Initializing and finalizing the polling DSM. The polling DSM thread is created during the initialization of the DSM. Its first task is to prepare the necessary DSM data structures (e.g. page table). After which it synchronizes with the worker thread using Pthread's conditional signal; this ends the `init()` call of the worker thread. From this point on, the event queue can be used. During polling, the DSM thread has to check the event queue and for other MPI calls as part of the various DSM protocols (e.g. page server, locks, and flushes). When the Finalize event is received, the DSM thread exits the polling loop. This returns execution to the rest of the `dsm_main()` function where various finalization tasks are performed before the DSM thread is destroyed.

call to the initialization function. Instead, it enters the infinite event loop and only terminates when it executes the Finalize event handler. A simple example of an event loop follows (the function pointer of the event handler is sent as part of the event message).

```
event_loop:
  forever:
    MPI_Recv(event)
    event.handler(event)
```

It should be noted that an event often gives rise to a number of subsequent operations and events. For example, a read fault for a remote page might play out as follows:

1) Worker process sends the Read Fault event to its local DSM process (the one in the same cluster node), and then blocks on `MPI_Recv` waiting for the page to be fetched.
2) On receipt of the Read Fault event the local DSM process posts an `MPI_Irecv` to receive the page *data* from the relevant remote DSM process. It then sends a Fetch event to that remote DSM process requesting a copy of the page.
3) The remote DSM on receipt of the Fetch event identifies the relevant page and sends it using a synchronous `MPI_Send` call that matches the `MPI_Irecv` given in 2. When this transfer is complete the remote DSM process sends a Page Ready event to the local DSM process.
4) The local DSM process receives the Page Ready event,

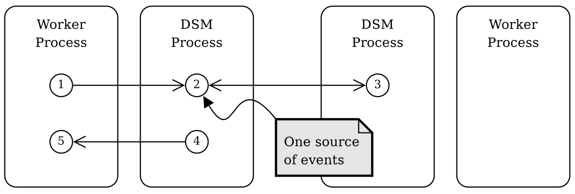

Fig. 4. The Event-driven DSM. (1) Events are sent by worker processes to DSM processes using MPI. (2) and (3) DSM processes receive events from both workers and other DSM processes at the same point. (4) If an event is blocking, the DSM process sends a synchronization message. This may also be a result such as a memory address. (5) If an event is blocking, worker processes wait for a synchronization message or result.

updates the page table, and issues an MPI_Send to match the receive call given in 1 above.

5) On completion of the MPI_Recv the worker process checks the state of new page, adjusts its memory protection to the appropriate setting, and then resumes execution.

It is important to note that the Fetch and Page Ready events given in step 2 and 3 above must be sent using asynchronous send calls otherwise the DSM system may deadlock. Thus, a background event sending mechanism has been incorporated into the DSM's event loop. This allows event handlers to enqueue asynchronous send events. In order to send and receive events simultaneously while still maintaining the event-driven characteristic, the event loop uses the MPI_Waitsome function as follows:

```
event_loop:
  MPI_Irecv(event)
  forever:
    MPI_Waitsome(on Irecv and Isend (if any))
    if Irecv completed:
      event.handler(event)
      MPI_Irecv(event)
    if Isend completed:
      dequeue head of background event queue
    if background event queue is not empty:
      MPI_Isend(background event)
```

III. OpenMP Flush, an Observer Model

The OpenMP flush directive is used to ensure that a thread's local view of memory is consistent with global memory [15]. Flush is used to ensure that a write by one thread is made visible to all other threads. For this to happen, the thread modifying its local view of memory memory needs to call flush so that global memory will be updated with these changes, meanwhile another thread interested in those changes must call flush to ensure that those changes are propagated into its local view of global memory. And, from the perspective of global memory the reader's call to flush must happen after the writer's call to flush.

The OpenMP flush allows simple busy-wait synchronization schemes to be implemented. The LU benchmark in NPB-3.2.1 [16], for example, uses the flush directive to implement synchronization between neighbouring threads in order to achieve a pipelined solution in its computation.

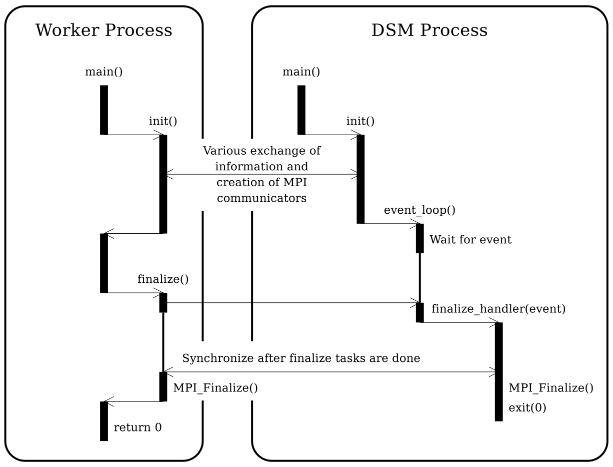

Fig. 5. Event-driven DSM Sequence Diagram showing initialization and finalization. During the init() function negotiations take place that determines if a process should play the Worker or DSM roles. Those that become DSM processes never return the init() call; instead they enter the event_loop() and wait for events in an infinite loop. The DSM process terminates only within handler for the Finalize event.

```
INITIALIZE
flag = 0

PRODUCER THREAD      CONSUMER THREAD
data = ...           do
!$omp flush (data)     !$omp flush (flag)
flag = 1             while (flag .eq. 0)
!$omp flush (flag)   !$omp flush (data)
                     ... = data
```

Fig. 6. An example of how a busy-wait loop synchronization can be constructed using the OpenMP flush directive (reproduced from [17] p.165).

A simpler but similar example is shown in figure 6. In the example, the producer thread first writes to *data*, flushes it, updates *flag* and then flushes *flag*. The consumer thread sits in a busy loop, where it flushes *flag* during each iteration. This continues while *flag* is equal to zero. When *flag* is not zero, the implication is that *data* has been updated by the producer. Hence, it can get the value of *data* by flushing *data* and then reading it.

Flush is an expensive operation for home-based DSMs. If a similar approach to some hardware caches is used on a copy of a globally shared page, then flush involves diffing the page, sending those diffs to the home location, and then invalidating the copy (leaving a new copy to be retrieved if and when that page is next accessed). In the context of the busy-wait loop case, this ultimately means sending an empty diff, and re-fetching the page during each iteration. A slightly better approach is to refresh only the region of the page that is being flushed (see figure 7). However, this approach still suffers from the same problem because when used in the busy-wait loop most of the refreshes would be redundant as all but the last call to flush would contain the same information; that it should not exit the loop.

To solve the redundant refresh problem we implemented an observer flush model in our event-driven DSM. The model is

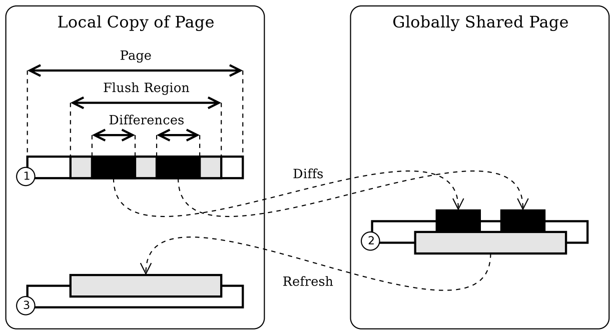

Fig. 7. A region specific flush involves: (1) sending the changes made to the region for (2) application on the home node followed by (3) refreshing the local copy from the updated global shared page. Only the flush region is refreshed.

so called because each time a refresh is requested from a DSM process, the requesting process becomes an "observer" of that region. Thus, the next time there is a change to the region, observers of the region can be notified of the change. From the observer's perspective, a region only really needs to be refreshed on a flush if a change notice was received for that region.

To implement the observer model two bookkeeping data structures are used. On the observer side, a set of subscribed regions is maintained; each with a `has_changes` boolean field that indicates if the remote region has changes. This field is set to true when the region is initialized because the observer would not have received change notification for changes that take place between fetching the remote page and flushing the region (and thus becoming an observer). Meanwhile at where the master copy of the region resides, a set of observed regions is maintained, each of which contains a list of observers and an `ack` counter (initialized to zero).

There are five different flush communication patterns that can arise. All of them begin with a flush request from a worker process. This is because the OpenMP memory model only requires a thread's (OpenMP thread) local view of memory to be consistent after a flush. That is, a DSM process only needs to update its local view of memory when a call to flush is made.

CASE-1: A remote region is flushed and differences were detected.

- The encoded diff data is sent to the home process using `MPI_Isend`, and an `MPI_Irecv` is posted to receive the refreshed data. Next, the `has_changes` field is set to false, and a Diff event is sent to the DSM process where the modified data resides.
- When the Diff event arrives at the relevant DSM process, the encoded diff data is received by posting an `MPI_Recv` that matches the `MPI_Isend` issued by the flushing process in the previous step. The diff is then decoded and applied. When this is done, the refreshed data is sent by issuing an `MPI_Send` to match the `MPI_Irecv` call made earlier by the flushing process. Next, Change Notice events are sent to all observers of the region (except for the flushing process) and the `ack` counter incremented by one as each notice is sent. All notified observers are also removed from the observer list leaving the flushing process as the only observer of the region. If `ack` is zero, then a Refresh is sent to the flushing process immediately, skipping the notification and acknowledgement handshake protocol described in the subsequent two steps.
- Upon receipt of a Change Notice, the Observer updates the `has_changes` field of the region, for which the notice is intended, to true. Once this is done, an Ack event has to be sent back in reply to complete the handshake.
- Back at the DSM process that is home to the flush region, the `ack` counter is decremented for every Ack event that is received. On receiving an Ack event that decrements `ack` to zero, the Refresh event can be sent back to all waiting flushing processes. There may be more than one flushing process waiting for handshakes to complete as more flushes to the region may have begun during the handshake process. Flushes 3 and 6 of figure 8 shows how CASE-1 and CASE-5 (described below) may intertwine and have to complete together at the end of both handshakes.
- When the Refresh event arrives at the flushing process, `MPI_Wait` is used to ensure that both asynchronous MPI calls from the first step have completed. When this is done, the flushing DSM process can synchronize with its worker.

CASE-2: A remote region is flushed and no differences were detected. However the `has_changes` field is true.

- In this case, the flushing process has no diffs to send. Thus, it only needs to post an `MPI_Irecv` call to receive the refresh data. When this is ready, the `has_changes` field is set to false and a refresh is requested from the relevant DSM process.
- When the request arrives at the relevant DSM process, the refresh data is transferred using an `MPI_Send` that matches the `MPI_Irecv` above. As there are no changes, this does not trigger any change notices. Thus, the refresh event can be sent to the flushing process immediately. The flushing process is also recorded to be an observer of the region.
- Handling the receipt of the refresh event is as in CASE-1.

CASE-3: A remote region is flushed and no differences were detected. The `has_changes` field in this case is false.

- Since there are no diffs to send and no changes have been made to the region, nothing needs to be done. This is where the observer model makes substantial savings over the traditional approach to implementing flush operations.

CASE-4: A local region is flushed and no changes have been made locally.

- Nothing needs to be done in this case. However, a penalty has been incurred by the local DSM process in that it has to check for differences. In the busy-wait scenario however, this cost is incurred by the process that is waiting in the busy-loop (i.e. the home of the

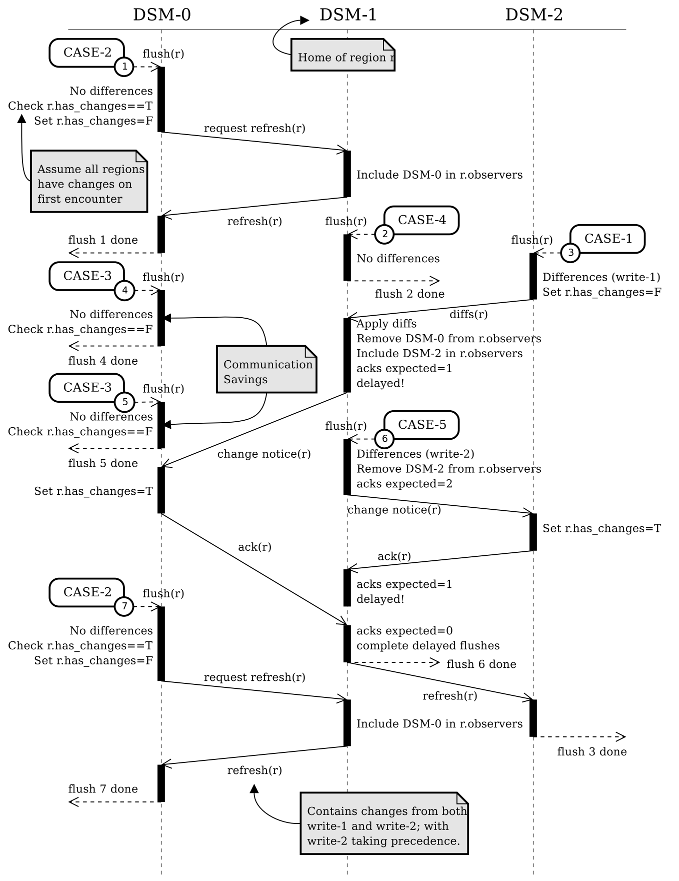

Fig. 8. Example communication sequence from the observer flush model showing all five cases. At the end of the sequence, DSM-0 and DSM-1 can see both write-1 and write-2. DSM-2, however, has not called flush since receiving the change notice and so only sees write-1 (which it contributed). Note that the full-line arrows represent event communication and the dotted arrows are synchronization messages within the cluster node. These do not represent when the diff or refresh data are communicated (see text for details).

synchronizing variable is the busy-waiting process). Thus, its impact on the overall performance is minimal if any.

CASE-5: A local region is flushed and there have been changes made to the region.

- In this case, change notices are sent to all observers of the region. As in CASE-1, the flush can only end if `ack` is zero.

The sequence diagram in figure 8 shows a possible interaction of all five flush cases. The example revolves around the flushing of region r, whose home is DSM-1. In the diagram, flushes 4 and 5 represent the communication cost savings achieved by the observer model when redundant refreshes are avoided.

It is also interesting to note that although the change notice received by DSM-0 was a result of flushing write-1, it gets a refresh that includes both write-1 and write-2 (from flushes 3 and 6 respectively). This explains why there only needs to be one change notice between the flush that registers the DSM as an observer and the one that requests for the refresh. This is why we can safely remove observers from the observers list upon sending a change notice.

The use of acknowledgements prevents a writer from completing its flush before all change notices have been received. For a busy-wait loop, the significance of this cannot be more important. Acknowledgements are needed so the writer knows that the reader has received the change notice for the data it has written. In the example in figure 6, this ensures that the change notice for *flag* does not get ahead of those for *data*. This is possible without the handshake because MPI only guarantees the order of messages sent from one process to another, and not the order of messages sent from two processes to one [18]. Since if the region homes are on different processes, notices may arrive in different orders. If that does happen then the reader may exit the busy-wait, and because the notice for *data* has not arrived, skips the refresh for *data* thinking that it is up-to-date.

IV. Evaluation

The previous sections introduced two alternative DSM-over-MPI implementations. The first takes a polling approach by running a thread in the background that performs the DSM tasks. The second requires the user to launch additional MPI processes, with a subset of these becoming DSM related processes. In this second model since MPI is used to handle all inter-process communication events it is possible to avoid polling and instead develop an event-driven approach. In addition we have also detailed a new observer model for implementing OpenMP flush that reduces the cost of these operations and is likely to be beneficial in situations where flush is used in combination with busy-wait loops to provide inter-thread (OpenMP thread) synchronization.

In order to evaluate our DSM implementations, we modify the Omni OpenMP compiler to also target our DSM interface (both DSMs use the same interface). The Omni OpenMP compiler is capable of targeting several different shared memory models. For instance, it can compile OpenMP into a Pthread implementation or into one that uses `shmem` system calls. The latter is implemented in a runtime library called OMPSM. The library contains preprocessor commands to switch between using `shmem`, SCASH, or SCASH-MPI. Taking advantage of this, we add our DSMs to this list of compile options. As our DSMs share the same interface, this work only has to be done once.

A. Cluster Specifications

Our experimental environment is a cluster of six AMD Athlon dual-core processors. These are connected via a gigabit ethernet network. Full details are given in table I.

B. Experiment #1 – Polling vs. Event-Driven

The hypothesis is that the polling DSM will require more overheads as the DSM thread has to run continuously. However if the number of DSM events is high, this may not be an issue as the worker is likely to spend a significant fraction of the total time waiting for the DSM anyway. In this scenario,

TABLE I

CLUSTER SPECIFICATIONS

HARDWARE	
CPU	AMD Athlon^{TM} 64 X2 Dual Core Processor 4200+
Clock	2.2GHz
Cores	2
No. of nodes	6
Network	Gigabit Ethernet
SOFTWARE	
Operating System	Linux
Kernel	2.6.11
MPI	MPICH2-1.0.5p4
GCC	3.3.5

there will be less polling and, therefore, less of a difference in performance.

To test this, we use the Embarrassingly Parallel (EP) benchmark from NAS Parallel Benchmarks (NPB-3.2.1) [16] to represent the case with light DSM load. For the heavy DSM load, the Block Tridiagonal (BT) benchmark is used. To increase the DSM workload, the BT benchmark is not optimized in any way. Also, a round robin page placement is used so that a page access has a $n-1$ out of n chance of resulting in a page fetch; where n is the number of different page homes. Both benchmarks are run using class size A.

These benchmarks are run under two different load conditions. The first is the normal dual-core set up, while the second is with a “busy-core” environment in which a “core filler” process that executes a small group of instructions in an infinite loop is used to keep one of the cores busy.

By comparing the performance under dual-core and busy-core conditions, we can determine to what degree each DSM implementation is affected by the loss of one core. This tells us how dependent an implementation is on background processing. Furthermore, as we are comparing each DSM with itself, this cancels out the effects of the other minor differences in the implementation.

Table II lists the timings for the light DSM load benchmark for the polling and event-driven DSMs. The dual-code and busy-core performances of each DSM is listed with ratios between these times calculated. A ratio that is close to 1.0 tells us that the performance has not been affected much by a reduction to one usable core. Conversely, a higher ratio indicates the higher dependency of the system on extra computing cycles. It is calculated as:

$$ratio = \frac{\textit{busy-core time}}{\textit{dual-core time}}$$

From the light DSM load ratios listed in table II, we can tell that the polling DSM has been greatly affected by the loss of one usable core. The event-driven DSM, however, does not see much of a change in performance with ratios that are very close to 1.0.

With the heavy DSM load, table III shows that the event-driven DSM is now a little more affected by the lost of one core. Other than the case of one OpenMP thread (which really is a low DSM usage case), the ratios for the polling DSM are higher than the event-driven DSM, but not by much. The reason for this is that with the heavy DSM load, the DSM is handling more events and does much less polling. With the DSM role overwhelmed by incoming events, the runtime of the benchmark is likely to be dominated by the DSM overhead rather than the computation work; i.e. the worker spends much of its time waiting for the DSM. More critically, however, the overall time for the event driven approach is significantly less than for the polling approach.

TABLE II

LIGHT DSM LOAD – RUNTIMES OF EP

No. of	**Polling DSM**			**Event-Driven DSM**		
OpenMP	(seconds)		$(\frac{\text{Busy}}{\text{Dual}})$	(seconds)		$(\frac{\text{Busy}}{\text{Dual}})$
Threads	**Dual**	**Busy**	**Ratio**	**Dual**	**Busy**	**Ratio**
1	31.63	64.06	**2.03**	31.77	31.85	**1.00**
2	15.79	31.16	**1.97**	15.94	16.04	**1.01**
3	10.58	23.57	**2.23**	10.61	10.75	**1.01**
4	7.97	18.08	**2.27**	7.98	8.05	**1.01**
5	6.42	14.99	**2.33**	6.46	6.50	**1.01**
6	5.38	12.37	**2.30**	5.34	5.38	**1.01**

TABLE III

HEAVY DSM LOAD – RUNTIMES OF BT (UNOPTIMIZED)

No. of	**Polling DSM**			**Event-Driven DSM**		
OpenMP	(seconds)		$(\frac{\text{Busy}}{\text{Dual}})$	(seconds)		$(\frac{\text{Busy}}{\text{Dual}})$
Threads	**Dual**	**Busy**	**Ratio**	**Dual**	**Busy**	**Ratio**
1	154	339	**2.20**	224	226	**1.01**
2	790	822	**1.04**	603	638	**1.06**
3	875	951	**1.09**	605	628	**1.04**
4	810	876	**1.08**	522	546	**1.05**
5	765	845	**1.10**	475	497	**1.05**
6	727	805	**1.11**	435	456	**1.05**

Not surprisingly the overall conclusions of these tests is that the cost of polling depends on how busy the DSM is; an overworked DSM that is handling event after event does little polling. In such situations the event-based DSM will have similar performance to the polling DSM.

Although the cost of polling can be hidden by using a second core, this does not make good use of computing resources. The poor performance of the polling DSM when running the light DSM load benchmark with a busy second core indicates that the polling thread is likely to reduce the effectiveness of any optimizations made to reduce the number of DSM requests (e.g. better page placement).

C. Experiment #2 – Observer Flush (event counts)

This experiment is designed to demonstrate the operation and effectiveness of the observer flush implementation as compared to the “basic flush” illustrated in figure 7. A synthetic benchmark is used to obtain deterministic event counts from our event-driven DSM. The benchmark is divided into four phases – numbered 0 to 3.

Phase 0 is an initialization phase to setup the state of shared memory correctly. During this phase, a shared variable x that resides at DSM-HOME is allocated and initialized to 0. The barrier at the end of the phase ensures that the state of the local page (at DSM-HOME) will be reset to read-only.

```
----- PHASE-0 -----
Allocate memory for integer x on DSM-HOME
if DSM-HOME:
  x = 0
barrier()
```

In phase 1, the remote DSM performs 10000 flushes on x, reads the value of x into a local variable y and then flushes x for another 10000 iterations. Event counts (table IV) for the basic flush implementation are 10000 Flush events from the remote worker to its DSM, 10000 Diffs (empty), and 10000 Refreshes. The observer model fairs much better, encountering only 10000 Flush events, 1 Refresh Request and 1 Refresh of data (1 CASE-2 and 999 CASE-3). Both models ignore flushes on regions that are in unmapped pages. Thus, the first 10000 flushes do not result in any action. The event counts only occur after the page is fetched by the `y=x` assignment statement that triggers a read fault. The `sync()` function at the end of this phase synchronizes the worker processes only. It is a light-weight version of the `barrier()` which does not deal with memory consistency.

```
----- PHASE-1 -----
if DSM-REMOTE:
  for 10000 iterations:
    flush(x)
  y = x /* y is local */
  for 10000 iterations:
    flush(x)
sync(1)
```

In phase 2, the DSM-HOME increments x 10000 times. Each time, the new value of x is flushed. In this phase, the basic model does nothing since the flushed region is in a local page. The observer model has more overhead, requiring 10000 flush events, but since these occur within a box this communication can happen very quickly. The only inter-DSM communication is the change notice that is triggered by the flush in the first iteration (CASE-5 communication pattern).

```
----- PHASE-2 -----
if DSM-HOME:
  for 10000 iterations:
    x = x + 1
    flush(x)
sync(2)
```

In phase 3, the remote DSM enters a while-loop with the condition being `x==y`. Given that x has been changed in the previous phase, only one iteration and therefore one flush will be executed here. Total event counts for both models are equal in phase 3. Both incur one within-the-box communication – the Flush event; and two between-boxes communication – Diff and Refresh for the basic model, and Request and Refresh for the observer model.

```
----- PHASE-3 -----
if DSM-REMOTE:
  while x == y:
    flush(x)
```

The final totals in table IV shows that the observer model results in more worker-to-DSM communication than the basic model. This is because the DSM has to check if there are changes to notify observers about even for local flushes; whereas the basic model has no such requirements. When we consider the many redundant refreshes suffered by the basic model however, the conclusion is that the observer model is a success in this respect.

TABLE IV
FLUSH EVENTS RECEIVED

	– Basic –		**– Observer –**	
Received Event	**Home**	**Remote**	**Home**	**Remote**
		PHASE-1		
Flush		10000		10000
Diff	10000			
Refresh Data		10000		1
Refresh Request			1	
		PHASE-2		
Flush			10000	
Change Notice				1
Notice Ack			1	
		PHASE-3		
Flush		1		1
Diff	1			
Refresh Data		1		1
Refresh Request			1	
Total	10001	20002	10003	10004
Total-inter-DSM	10001	10001	3	3

D. Experiment #3 – Slowdown Caused by Busy-wait Flushes

When flushes are used within the context of busy-wait loops, it is not the slowdown of the loop that we are interested in improving. Processes within busy-wait loops are afterall waiting! Instead, it is the slowdown induced by the busy-wait loop on the processes doing useful computation work – which the rest are waiting for – that is of interest. A slowdown is induced by flooding the DSM process at which the useful work is being done with flush related events so that the DSM requests from the local worker process (that is performing the useful work) get delayed. To test this, we design an experiment (see figure 9) with the following components: *useful work*, *page server*, and *busy-wait*.

The useful work component is a finite-difference approximation of the 2D Laplace Equation that uses a 5-point stencil operation. The memory access profile of this calculation is a series of writes to one grid and reads from another grid. In short, the useful work here is a memory touching exercise with some calculation.

The second component is the page server. All memory allocated for the grid is mapped directly onto a separate cluster node. Thus, as our useful work is being done, it will require the services of the DSM subsystem to fetch its pages as well as perform the necessary record keeping tasks (e.g. twinning). To ensure that the useful work will require this service throughout its computation, the Laplace calculation is only performed for one iteration. We then adjust the size of the grid to give longer running times. Also the initialization phases of the Laplace are skipped to prevent pages from being fetched during initialization instead of during the main computation (as there is only one writer, the pages will not be invalidated at a

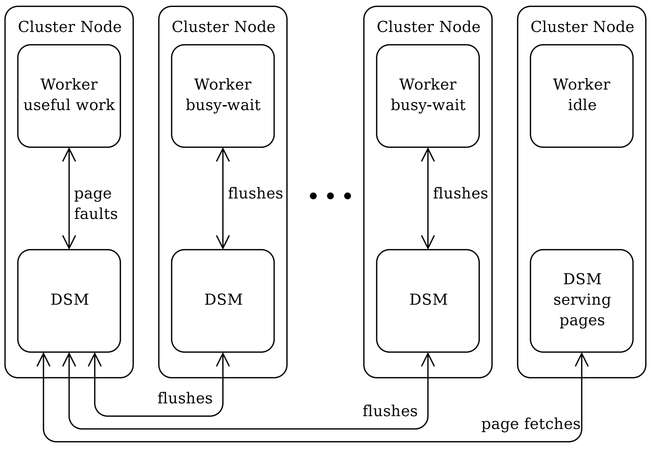

Fig. 9. Experiment #3 setup. One worker does useful work, one DSM serves pages, and the other worker/DSM pairs busy-wait and issue flush related events to the first DSM.

TABLE V
EFFECT ON PERFORMANCE DUE TO REMOTE BUSY-WAIT FLUSHES

– DUAL-CORE –		
Average time in seconds	Basic	Observer
No busy-wait	87.76	87.76
With busy-wait	95.91	88.67
Slowdown	9.29%	1.04%
Average flush related event counts	Basic	Observer
Diff (empty)	1506221	
Refresh request		8
Notice Ack		4
Total	1506221	12
– BUSY-CORE –		
Average time in seconds	Basic	Observer
No busy-wait	90.87	90.87
With busy-wait	105.44	91.97
Slowdown	16.03%	1.21%
Average flush related event counts	Basic	Observer
Diff (empty)	1642811	
Refresh request		8
Notice Ack		4
Total	1642811	12

barrier).

The last component of our experiment is the busy-wait loop. All other processes, that are not doing useful work or serving pages, will busy-wait on a shared synchronizing variable. This variable will be mapped to the node at which the useful work is being done so that the busy-wait loop may have a chance to induce a slowdown.

Table V lists the timings obtained when the test is run with six cluster nodes – one doing useful work, one serving pages, and four busy-waiting. Shared memory is allocated for two 8192×8192 grids of `double` datatype quantities. The page size used is the system page size of 4096 bytes. Thus, both grids will require 262144 pages in total. However as the edge quantities of the grid being written to are not accessed, the total number of page fetches is $262112 = (8192 + 8190) \times (16\ \textit{pages per row})$. Note that the timings for the "no busy-wait" case is identical regardless of flush model as it does not involve any flushing. These timings are done for both dual-core and busy-core (see Experiment #1 in section IV-B for details) cases. For each set of "with busy-wait" timings we include the average flush-related event counts.

Table V shows that the observer model has successfully reduced the number of flush related events that the working node has had to deal with to a deterministic amount (2 refresh requests and 1 notice acknowledgement from each busy-wait process). In stark contrast, the DSM in the working node under the basic model has to field 1.5 million flush related events on the average. This difference explains the larger slowdowns we see in both the dual-core and busy-core timings.

V. CONCLUSIONS

The study on MPI based DSMs compares the performance of polling and event-driven DSMs under different DSM loading and shows that the event-driven DSM is a more scalable model. While both architectures perform equally under heavy DSM load, the decrease in light DSM load performance of the polling DSM when tested under busy-core situations indicates that the background polling thread would reduce the effectiveness of optimization efforts such as proper page placement. The polling approach will therefore not be ideal for effective utilization of the multi-core processors that are becoming commonplace. In contrast, the light DSM load performance of the event-driven DSM demonstrates that it would be a scalable architecture with respect to optimization efforts.

The second part observed that if the OpenMP flush is implemented as a simple flush-and-refresh protocol in the DSM, a large volume of network communication would arise when the OpenMP flush is used in a busy-wait loop. Worst of all, most of the refreshes would contain the same information and are therefore redundant. Under the observer model described, the redundant refreshes are effectively eliminated by having the process holding the master copy notify observers only when a change has been made to the region. The experiment shows that the amount of communication is reduced from something that is dependent on the wait duration to one that is deterministic.

ACKNOWLEDGMENT

This work is funded in part by Australian Research Council Linkage Grant LP0669726 and Alexander Technology. We are grateful to Y. Ojima, M. Sato and T. Boku for providing us with a copy of SCASH-MPI and for helpful discussions. APR is grateful to Japanese Society for the Promotion of Science for the award of a short-term fellowship.

REFERENCES

[1] K. Li, "IVY: A shared virtual memory system for parallel computing," *Proceedings of the 1988 International Conference on Parallel Processing, Vol. II Software*, pp. 94–101, Aug. 1988.
[2] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. 2nd ACM SIGPLAN Symp. Principles and Practices of Parallel Programming*, Seattle, Mar. 14-16 1990.

[3] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel, “TreadMarks: Distributed memory on standard workstations and operating systems,” in *Proceedings of the 1994 Winter Usenix Conference*, 1994, pp. 115–131.

[4] M. Sato, H. Harada, and A. Hasegawa, “Cluster-enabled openMP: An openMP compiler for the SCASH software distributed shared memory system,” *Scientific Programming*, vol. 9, no. 2-3, pp. 123–130, 2001.

[5] S. Karlsson, S.-W. Lee, and M. Brorsson, “A fully compliant OpenMP implementation on software distributed shared memory,” in *High Performance Computing - HiPC 2002, 9th International Conference, Bangalore, India*. Springer, Dec. 2002, pp. 195–208.

[6] Y.-S. Kee, J.-S. Kim, and S. Ha, “ParADE: An openMP programming environment for SMP cluster systems,” in *SC*. ACM, 2003, p. 6.

[7] L. Huang, B. M. Chapman, and Z. Liu, “Towards a more efficient implementation of OpenMP for clusters via translation to global arrays,” *Parallel Computing*, vol. 31, no. 10-12, pp. 1114–1139, 2005.

[8] J. P. Hoeflinger, “Extending OpenMP to clusters.” [Online]. Available: http://www.intel.com/

[9] Y. Ojima, M. Sato, T. Boku, and D. Takahashi, “Design of software distributed shared memory system using MPI communication layer,” in *Proc. The 4th International Workshop on OpenMP: Experiences and Implementations WOMPEI 2005*, Tsukuba, Japan, Jan. 2005, pp. 18–25.

[10] H. Harada, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, and Y. Ishikawa, “SCASH: Software DSM using high performance network on commodity hardware and software,” in *Eighth Workshop on Scalable Shared-memory Multiprocessors*. ACM, May 1999, pp. 26–27.

[11] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, “Global arrays: A portable “shared-memory” programming model for distributed memory computers,” in *Proceedings of Supercomputing’94*. Washington DC: IEEE, Nov. 1994, pp. 340–349.

[12] R. M. Olson, M. W. Schmidt, M. S. Gordon, and A. P. Rendell, “Enabling the efficient use of SMP clusters: The GAMESS/DDI model,” in *SC*. ACM, 2003, p. 41.

[13] K. Kusano, S. Satoh, and M. Sato, “Performance evaluation of the omni openMP compiler,” in *ISHPC*, ser. Lecture Notes in Computer Science, M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, Eds., vol. 1940. Springer, 2000, pp. 403–414.

[14] H. Tezuka, A. Hori, and Y. Ishikawa, “PM: A high-performance communication library for multi-user parallel environments,” RWC, Technical Report TR-96015, Nov. 1996.

[15] *OpenMP Application Program Interface version 2.5*, OpenMP Architecture Review Board, May 2005. [Online]. Available: http://www.openmp.org/drupal/mp-documents/spec25.pdf

[16] H. Jin, M. Frumkin, and J. Yan, “The OpenMP implementation of nas parallel benchmarks and its performance.” [Online]. Available: http://www.nas.nasa.gov/News/Techreports/1999/PDF/nas-99-011.pdf

[17] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. Academic Press, 2001.

[18] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI – Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. The MIT Press, 1999.