Integrating Software Distributed Shared Memory and Message Passing Programming

H.J. Wong and A.P. Rendell

Department of Computer Science, The Australian National University
Canberra ACT Australia

1Jin.Wong@anu.edu.au
2Alistair.Rendell@anu.edu.au

Abstract—Software Distributed Shared Memory (SDSM) systems provide programmers with a shared memory programming environment across distributed memory architectures. In contrast to the message passing programming environment, the SDSM can resolve data dependencies within the application without the programmer having to explicitly specify communication. However, this service is provided at a cost to performance. Thus it makes sense to use message passing directly when data dependencies are easy to solve using message passing. For example, it is not complicated to specify data transfer for large contiguous regions of memory.

This paper outlines how the Danui SDSM library has been extended to include support for message passing. Four different message passing transfers are identified depending on whether the data being sent/received resides in private or globally shared buffers. Transfers between globally shared buffers are further categorized as symmetrical or asymmetrical depending on whether they correspond to the same region of shared memory. The implication of each transfer type on the memory consistency of the global address space is discussed. Central to the Danui SDSM extension is the use of information provided and implied by message passing operations. The overhead of the implementation is analyzed.

I. INTRODUCTION

Software Distributed Shared Memory (SDSM) systems simplify cluster programming by providing shared memory programming across a distributed memory environment. Unlike the message passing paradigm, a shared memory programmer does not need to worry about when, where, and what data to communicate. Instead, these concerns are left to the SDSM that uses a memory model to imply data communication at memory accesses and memory sequence points. This quality makes SDSM systems easier to use for applications with unpredictable access patterns. However, having to imply, and not being able to specify, when, where, and what data to communicate can be counterproductive when the required data transfer is straightforward for the programmer.

SDSM systems make use of the underlying communication infrastructure which, on a distributed memory system, is typically some form of message passing. Additional overheads are incurred by the SDSM in order to correctly determine and execute the required message passing communications. Thus, the performance of SDSM systems can be bettered or at least matched by hand-tuned message passing solutions [1]. After all the SDSM is just one of the message passing solutions.

Given that usability is not always of concern and that the performance of the SDSM can be matched or bettered by message passing, it is desirable to have an integrated shared memory and message passing programming environment that allows the programmer to take advantage of features from both paradigms.

The Message Passing Interface (MPI) is a widely supported standard for message passing [2], [3], [4] that is also available on modern high performance interconnects like InfiniBand [5], [6], [7]. Developers can therefore take advantage of this support to implement portable SDSM libraries.

Silva et al [1] describe DSMPI as an MPI-based SDSM where shared memory is managed as a pool of shared objects. More recently, Ojima et al [8] implemented SCASH-MPI, an MPI-based version of the SCASH SDSM [9]. Both systems employ a daemon to implement one-sided SDSM protocols. DSMPI uses a separate MPI process, while SCASH-MPI uses a background thread through which all communication with other processes must be routed. These design decisions were taken because, at that time, MPI libraries did not have good support for the MPI_THREAD_MULTIPLE mode that allows MPI library calls to be made concurrently from any thread in the MPI process.

We have developed Danui as an ANSI-based SDSM that requires MPI_THREAD_MULTIPLE support. This allows the daemon thread to use blocking communication when waiting for events from other processes because the main thread is able to use MPI directly when sending events to other processes. In contrast, the daemon thread for SCASH-MPI had to employ polling because it needed to serve requests coming from other processes, while also facilitate communication originating from the main thread.

The flexibility of using MPI from any thread places Danui in a unique position to implement an integrated shared memory and message passing programming environment. However, message passing operations access memory and additional bookkeeping work is sometimes necessary to keep shared memory consistent. This paper describes the issues involved, and the key observations that make it possible to support message passing on shared memory buffers correctly and efficiently.

Section II describes how Danui implements the distributed shared memory. Issues and key observations are discussed in
section III, while section IV outlines how the observations are used to extend Danui to support the integrated programming environment. Section V presents the performance of the implementation. Lastly, section VI lists related work before our conclusions in section VII.

II. DANUI SOFTWARE DISTRIBUTED SHARED MEMORY

Danui is an MPI-based Software Distributed Shared Memory (SDSM) system that provides shared memory programming for a selected region of the address space (figure 1). This region is divided into pages and memory consistency is maintained using a Home-based Lazy Release Consistency (HLRC) protocol [10].

The SDSM utilizes the mprotect system call to adjust the level of memory access permissible on each page. These are, in order of increasing accessibility: invalid, read-only, and read-write. Memory access attempts beyond the permission level will throw the segmentation fault signal (SIGSEGV) that is caught by a handler installed by the SDSM. This mechanism allows the SDSM to detect memory access at page size granularities.

From the perspective of a process, a page is local if it is home at the process; otherwise, it is referred to as a remote page. The page locality attribute is combined with the permission access levels to derive the page states used by the SDSM. Local pages are always valid and so are either in the Local Read-only (LRO) or Local Read-write (LRW) states. Similarly, remote pages can be in the Remote Read-only (RRO) or Remote Read-write (RRW) states. In addition, remote pages may also be invalid, in which case the Remote Unmapped (RUM) state is used.

When a RUM page is accessed, the handler will fetch the page from the home process to update the view of the page. The page is then transitioned into the RRO state before returning from the handler. A page fault on a RRO page indicates write access. The SDSM uses twinning-and-diffing [11] to keep track of modifications made to remote pages. Here, a copy of the page, called the twin, is made so that changes made to the page can be determined by comparing the page data with the twin (figure 2 illustrates how twinning-and-diffing works). Diffing is performed at memory sequence points which are defined by memory flush operations.

Flushes may be explicitly invoked or be implied at barrier synchronization points. The objectives of the flush are: 1) to make all local modifications since the last memory sequence point globally visible, and 2) to ensure that all flushed modifications by other processes will be locally visible. The first criteria is satisfied by the diffing phase of the twinning-and-diffing protocol. The second is achieved by invalidating remote pages by transitioning them into the RUM state. This ensures that subsequent access to the page will trigger a page fetch operation thereby updating the view of the page.

Accessing remote pages is costly because updated copies need to be fetched and modifications need to be sent back to the home processes. Thus, if access patterns are regular and consistent, it is better to have data residing in local pages [12]. Home migration [13], [14], [15] can be used at barrier points because processes are synchronized allowing for collective information sharing. In Danui, processes maintain a list of modified pages in a write-set that is shared at the barrier point. This allows each process to individually discover pages that have been modified by one or more processes but not by the home process. Such pages are migrated to one of the writers and all the diffs by the remaining writers will be sent to the new home. In the special single-writer case, where there is only one such writer, no diffs need to be transferred.

A. LIMITATIONS OF THE SDSM MODEL

Limitations of the shared memory model, with respect to message passing, have been identified by Kranz et al [16].
Although the arguments were for a Hardware Distributed Shared Memory (HDSM) system (the MIT Alewife machine), they can be reevaluated for the SDSM:

1) Bulk Data Transfer: To obtain updated values of a large shared array that has been modified by another process, a process would have to fetch pages piecemeal, one page at a time, with each costing a page fault, a page request (network latency cost), and a data transfer (a second network latency cost). Contrast this with message passing where the modified data could be sent in one message; costing one network latency and making better use of the network bandwidth.

2) Known Communication Patterns: The convenience of the SDSM lies in being able to access any part of shared memory, making it ideal for unpredictable access patterns. However, when access patterns are predictable, the model provides little convenience benefit but still retains the same overhead costs. Explicit message passing operations allow finer grained synchronization that exposes more parallelism than coarse grain approaches like the barrier [17].

3) Combining Synchronization with Data Transfer: Pairwise synchronization between producers and consumers of data is also more efficient, and perhaps also more intuitive, using blocking point-to-point message passing. The shared memory equivalent uses a synchronization variable between each pair of processes. In figure 3, the variable is initialized to “false” before the computation, and then the consumer process uses the flush operation in a busy-wait loop until the variable is “true.” For Danui, each flush operation invalidates the page holding the variable and each inspection of the loop exit-condition results in a read fault that fetches a fresh copy of the page. In other words, the consumer process is in a loop that repeatedly fetches the page. Figure 4 shows the message passing equivalent that achieves the same synchronization and data dependency goals with one message.

The next section highlights the problems that arise from integration and what additional information can be obtained from the use of message passing operations.
be used, the flushed pages would also be invalidated, forcing the read access by the send operation to fetch fresh copies of the flushed pages. This would negate the benefit of bulk transfer integration, since the receiver could have fetched the pages directly from the home processes; i.e. the data transfer between sender and receiver would be rendered pointless. The only significance of the send and receive in this case would be its synchronization aspect; in which case an empty acknowledgement message would have been sufficient.

Another issue arises from the write effect on shared memory whenever data is received. In the alternate situation below, the intention is for the producer to make writes to \( A \) at W1 and W2. Send and receive operations are used to synchronize the writes with respect to the read by the consumer at R1 such that R1 should only witness the modifications at W1, but not W2. The first send-receive communication also doubles as the medium for transferring the values in \( A \) produced at W1.

```
Producer                      Consumer
A[... ] ← ... → W1
Send(A)                      Recv(A) → W3
                             ... ← A[... ] → R1
Recv(ack)                    Send(ack)
A[... ] ← ... → W2          Barrier
```

Assuming that the consumer begins the example with a valid copy of \( A \), the write effect from the receive operation at W3 will result in creation of twins based off this copy; which would be stale because the producer would have had sent diffs prior to the send operation as discussed in the previous issue. Thus, at the next barrier, the producer would make diffs for modifications at W2, while the consumer would make diffs for “modifications” at W3. These diffs from the producer and consumer would, therefore, result in a race condition.

The solution for this problem is to recognize that the write effect at W3 is for updating the consumer’s view of \( A \), and that it is not an actual modification to shared memory. This situation is similar to the writes to shared memory when a page is validated via a page fetch from the home process and transitions into the read-only state; not read-write. By taking this approach, no diffs for \( A \) will be erroneously created by the consumer at the barrier.

In the next part of this section, we describe the additional information that is normally not available using SDSM mechanisms alone that can be obtained from message passing operations. Section IV then describes how the new information can be used to extend the Danui SDSM to resolve the issues described above.

### B. Information Provided and Implied by Message Passing

The mechanisms that the SDSM uses to gather information about memory accesses has limitations. For starters, the memory protection and page fault system only enables the detection of access violations and not memory access. That the SDSM is unaware of read accesses to pages which already have read-only or read-write permissions illustrates this drawback. Secondly, when a page fault is triggered, the SDSM only knows where the fault address is, and not the size of the actual memory region that would be accessed (aside from that one byte!). Lastly, twinning-and-diffing can only provide information about which bytes have changed and not which bytes have been written to. Consider the example of incrementing a 4-byte integer from zero (0x00000000) to one (0x00000001). Although four bytes would have been written, twinning-and-diffing would have noticed only a change in the least significant byte\(^1\) because the same byte values as the twin copy were written to the remaining three bytes (see silent-writes, figure 2 panels 6 and 7).

By comparison, message passing operations provide a wealth of information to the SDSM. For example, send operations resolve to read accesses regardless of the states of the underlying shared pages. Thus, where previously the SDSM would only know of read accesses when a page fault occurred for invalid pages, it can now also know about read accesses on read-only and read-write pages. The SDSM also knows that the memory access applies to the region covered by the message buffer, a far cry from a single byte at some fault address. Moreover, should the access type be a write (i.e. at the receiver end of data transfer between asymmetrical buffers), the SDSM knows that every byte in the message buffer will be written; effectively sidestepping the silent-write problem described above.

There is also additional temporal information that is derived from when the memory accesses might occur. A memory read on a message buffer, for example, is guaranteed to have completed when a blocking MPI_Send returns. If this buffer is in shared memory, then the programmer must ensure that no other processes write to the same region of shared memory, otherwise a data race would occur. Using arguments of this nature, each SDSM process can make assumptions about the behaviour of other processes based only on information that is local to that process. This adds the following additional derived information from message passing operations on message buffers in shared memory:

1. If the message passing operation results in a read access on the message buffer, then no other process should be writing to the buffer; and
2. If the message passing operation results in a write access on the message buffer, then no other process should be reading or writing to the buffer.

Figure 6 illustrates an example consisting of three processes and a shared buffer (\( X \)). Prior to the first SDSM barrier, the buffer is assigned a value of 7 (“W(X)” means write the value of 7 into \( X \)). Following the barrier, \( X \) is sent from process 0 to process 1. Data transfer for this is indicated by the solid arrow connecting the send operation at process 0 to the receive operation at process 1. Additional dotted arrows and gray boxes indicate the implied restrictions on memory accesses at

\(^1\)Differing granularity may differ for other SDSMs. e.g. TreadMarks use a 4-byte granularity.
Fig. 6. Memory accesses by processes limit what other processes can correctly perform. Arrows with dotted lines connect memory access actions that imply restrictions on other processes. The “W(X)7” notation means “write the value 7 into X”, while “R(X)7” means “read the value 7 from X”.

the other two processes. Notice that the “W(X)8” operation at process 1 is legal because the memory accesses at process 1 after the receive operation are synchronized with respect to the accesses at process 0 up to, and including, the send; i.e., the write happens after the send. On the other hand, as there is no such synchronization between process 0 and process 3, the restrictions from the send continue to apply even when the receive at process 1 has completed. This however ends when, transitively, the completion of the receive at process 3 implies the completion of the send at process 0.

IV. EXTENSIONS FOR SUPPORTING MESSAGE PASSING

Section II described the implementation of the Danui SDSM, and section III presented the challenges of integration as well as the additional information that message passing operations makes available. This section describes how the additional information has been used to extend support for MPI in Danui. The following MPI operations are currently supported:

1) Blocking: MPI_Send and MPI_Recv.
2) Non-blocking: MPI_Isend, MPI_Irecv, and the related MPI_Test, MPI_Wait, MPI_Waitall, and MPI_Request_free functions.
3) Collective: MPI_Bcast.

There are two points to note before proceeding with the implementation details. Firstly, functions that interact with message buffers are renamed by adding either an “s” or an “a” prefix to differentiate between symmetrical and asymmetrical communication respectively. For example, a symmetrical shared-to-shared data transfer can be achieved using sMPI_Send and sMPI_Recv. We believe that this is simple for the programmer to determine and so should not hinder usability.

Secondly, the implementation restricts non-blocking calls by requiring them to complete before the next memory sequence point. This is because the algorithms for the SDSM flush and barrier operations assume that no local memory accesses would take place, thus it might be problematic if memory accesses from non-blocking calls interfere.

A. Infrastructure

The general strategy used for supporting MPI is to wrap MPI calls between actions that modify the bookkeeping information maintained by the SDSM so that memory consistency is maintained. That is:

1) Bookkeeping action,
2) MPI library call, and
3) Bookkeeping action.

However, returning from a non-blocking MPI library call does not indicate the completion of the data transfer. Thus, bookkeeping actions that need to be executed after the data transfer have to be delayed. Danui implements an infrastructure for tracking MPI_Request objects and associating them with call-back actions and parameters. Next, the MPI_Test, MPI_Wait, and MPI_Waitall routines are overloaded 3 to check for the completion of tracked requests so that the outstanding bookkeeping actions can be performed by invoking the associated call-back routine. Using this infrastructure, the modified strategy for non-blocking operations is as follows:

1) Bookkeeping action,
2) Non-blocking MPI library call, and
3) Register call-back function and parameters.

In cases where there is no need to keep the request handle, MPI permits the programmer to discard the request by using MPI_Request_free. This presents a challenge to the callback infrastructure. How is it possible to track a request that has been freed? To handle this, Danui does not free the request, instead the overloaded MPI_Request_free function moves the tracked request into a set of “freed” requests together with its associated call-back function and parameters. Polling is then used for requests in this set to check for completion; this is strategically performed only at events that might imply the completion of one or more non-blocking communications. In general these are routines that can be used to receive synchronization messages. In Danui, polling is inserted at the following points:

- after MPI_Recv,
- after MPI_Wait.

3While the write operation “W(X)8” at process 1 would also imply that process 0 must not read X after the send completes, that is not considered to be new information from message passing and, hence, restrictions of this nature are not indicated in the figure.

3The original MPI functionality is still available from within the overloaded functions via the PMPI profiling interface.
Fig. 7. Messages buffers (shaded areas) may span multiple pages (boxes) and do not have to be aligned with SDSM pages. Buffer A is a small buffer that fits entirely within the page, it only partially overlaps with the page. Buffer B is aligned and occupies three pages fully. Buffer C occupies two pages partially and five pages fully. For bookkeeping, the extent to which buffers occupy pages is a factor for consideration.

- after a successful MPI_Test,  
- after MPI_Bcast,  
- after MPI_Barrier, and  
- before SDSM flush and barrier operations.

For the SDSM flush and barrier operations, the polling routine must additionally indicate at its completion that there are no outstanding incomplete non-blocking communications because non-blocking operations are not permitted to overlap with SDSM memory sequence points.

B. Bookkeeping

Bookkeeping actions are considered at page granularity because Danui is a page based SDSM. Consequently, the state of the page is a factor for consideration when deciding on the necessary bookkeeping actions.

However, message buffers might not be aligned to SDSM pages. Figure 7 shows three message buffers of different sizes. Buffer A is a small buffer that fits entirely within the page, so it only partially overlaps with the page; buffer B is aligned and occupies three pages fully; and buffer C occupies the first and last pages partially and the five internal pages fully. Whether a page fully or partially overlaps with the message buffer is referred to as the extent of the access. This also plays an important role in deciding on bookkeeping actions.

The type of memory access is also a factor that determines the correct bookkeeping action. Where the effect of the MPI function is to send shared data from the calling process, a read access will be performed. And when data is received into shared memory, the access type may either be:

- a write – if the message buffer is asymmetrical with respect to the sender (cases 2 and 4a of figure 5), or  
- an update – if the message buffer is symmetrical with respect to the sender (case 4b of figure 5).

This section describes the bookkeeping actions that are needed for each combination of access type, page state, and extent of the access. The discussion will focus on read accesses, on write accesses next, and then update accesses.

1) Bookkeeping for Read Accesses: Read accesses occur when message passing is used to send data from shared memory. The necessary bookkeeping for read accesses are shown in table I. Information about the page are listed in the first two columns, while the necessary bookkeeping actions to be done before and after the MPI call are listed in the next two columns.

<table>
<thead>
<tr>
<th>State</th>
<th>Extent</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRO</td>
<td>Full</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRW</td>
<td>Full</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUM</td>
<td>Full</td>
<td>RUM-Fault</td>
<td>RUM-Fault</td>
</tr>
<tr>
<td>RRO</td>
<td>Full</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRW</td>
<td>Full</td>
<td>Claim page</td>
<td>Flush partial page</td>
</tr>
</tbody>
</table>

The first requirement for read access is that pages being read must be valid before the message passing operation. For the Danui SDSM, only remote pages can be invalid, thus the RUM-Fault handler is executed for the pages in the RUM state before the MPI call. The handler fetches the page from the home process and transitions the page into the RRO state, making it ready for the read access. It is necessary to explicitly invoke the fault handler before the MPI call because the MPI library may abort when a segmentation fault is caught (e.g. Open MPI does).

The second requirement, as noted in the survey of the issues in section III, is that all modifications made prior to the data transfer have to be made globally visible before the message passing operation. This requirement only affects RRW pages because modified local pages in the LRW state are already globally visible. Two different actions are prescribed for RRW pages depending on the extent of the access: “flush partial page,” and “claim page.”

Flush partial page: When only part of the RRW page overlaps with the message buffer, then the SDSM needs to flush the partial page. This is however done without relying on the SDSM flush operation. Instead, it is noted that no modifications to this part of the page by other processes is possible because this read access imposes write restrictions on other processes (figure 6). Thus the SDSM only needs to determine if the portion has been modified by checking the twin. This is necessary even though the page is already in the RRW state because it may have gotten into that state due to modifications to other parts of the page that are not in the message buffer. If there are differences, the part of page that overlaps with the message buffer is sent to the home process as is (i.e. not in an encoded form). In addition, the same data is also used to update the twin so that it matches the global view at the home process; allowing for diffs to be correctly created at the next memory sequence point.

Claim page: As noted above, this read access imposes write restrictions on other processes. Therefore when the whole page is read, it can be assumed that no conflicting writes to the page have taken place at any other process. Applying this observation to read accesses on full RRW pages, it follows that the process performing the read access is the only writer to the page. In other words, a single-writer scenario has been presented to the SDSM, and the page can be migrated to this
process. Migrating the page home eliminates the need to flush modifications because changes made at this process would be globally visible after the home migration. However, unlike the home migration in the barrier operation, home migration in this case has to be initiated in a one-sided fashion. The “claim” protocol is used to perform this one-sided home migration.

Danui implements a home-tree for each page to support one-sided home migration. A home-tree is formed by the home pointers at each process (referred to as the “forwarding pointer” approach by Fang et al [15]). Initially, all home-trees will have a height of one and be rooted at the home process (which points to itself, figure 8 panel 1). A page is claimed by sending a claim request up the home-tree, forwarding the request at intermediate processes if necessary, until it arrives at the home process. At the root of the tree the home pointer is updated to the claiming process, and an acknowledgement message is sent directly to that process (figure 8 panel 4).

Intermediate processes on the path along which the request was forwarded will also update their home pointers to the claiming process (figure 8 panel 5). The page transitions to the RRO state at the old home process, and transitions to the LRW state at the new home.

Claims for multiple pages to the same home process are packed into a single message to lower overheads. In a multi-page claim, a list of pages to claim are sent with the request. However, as home pointers might not point at the root of the home-tree, the list may be split into parts that are either acknowledged at the process or forwarded to other processes. Acknowledgement for multiple pages is sent in one message that contains the number of pages acknowledged. The claiming process waits for acknowledgement messages until the sum of the page counts adds up to the number of pages claimed.

Lastly, home-trees need to be collapsed at the start of the SDSM barrier because home-migration algorithms in the barrier expect home-trees of height one. This is done by maintaining two arrays for storing home pointers: $H$ and $R$. The first stores the actual home pointers such that $H[p]$ always points to the parent in the home-tree for page $p$. The second array, $R$, is used by each process to only store pointers to itself (i.e. it is the root of that home-tree), leaving the rest of the array zeroed. Since pointers to self are always accurate, the array of collapsed home-trees can be calculated by using the MPI_Allreduce collective operation to aggregate the $R$ arrays at all processes into $H$.

2) Bookkeeping for Write Accesses: Write accesses occur when message passing is used to receive data into a buffer that is asymmetrical with respect to the sender’s buffer. Table II lists the actions necessary for write accesses. The objective is to bring pages into a writable state. This is achieved by calling the fault handlers for pages that are not already writable before the MPI call (i.e. the LRO, RUM, and RRO states). For the remote pages, this allows twins to be created before the write allowing for consistency to be maintained at the next memory sequence point.

An exception is made for remote pages that fully overlap with the message buffer. For such pages, the MPI call will be allowed to proceed and then the page will be claimed. This claim is justified because the write access here would imply read/write restrictions on the whole shared page at other processes. Thus, this is also a single-writer situation which the SDSM can use to initiate home migration.

<table>
<thead>
<tr>
<th>State</th>
<th>Extent</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRO</td>
<td>Full</td>
<td>LRW-Fault</td>
<td>LRW-Fault</td>
</tr>
<tr>
<td>LRW</td>
<td>Partial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUM</td>
<td>Full</td>
<td>Partial</td>
<td>RUM-Fault, RRO-Fault</td>
</tr>
<tr>
<td>RRO</td>
<td>Full</td>
<td>Partial</td>
<td>RRO-Fault</td>
</tr>
<tr>
<td>RRW</td>
<td>Full</td>
<td>Partial</td>
<td>Claim page</td>
</tr>
</tbody>
</table>

Fig. 8. The home-tree enables one-sided home migration outside of SDSM barrier operations. Each page has its own home-tree that is formed by the home pointers at each process for that page. These pointers are represented as solid arrows in the diagram. Note that the pointer at the root of the home-tree always points to itself. Bold arrows are used to highlight pointers that have changed from the previous panel, while dotted arrows represent communication between processes. Where there are multiple communication arrows in one panel, numbers indicate the order in which they occur. Panel 1 shows a tree rooted at $P_0$ of height one. Without home migration, the home-tree will remain in this form. $P_1$ claims the page from $P_0$ and then forwarded to $P_1$.

TABLE II
BOOKKEEPING FOR RECEIVING INTO AN ASYMMETRICAL BUFFER (WRITE ACCESS).
TABLE III
BOOKKEEPING FOR RECEIVING INTO A SYMMETRICAL BUFFER (UPDATE ACCESS).

<table>
<thead>
<tr>
<th>State</th>
<th>Extent</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRO</td>
<td>Full</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRW</td>
<td>Full</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUM</td>
<td>Full</td>
<td>Partial</td>
<td>Transition-RRO</td>
</tr>
<tr>
<td>RRO</td>
<td>Full</td>
<td>Partial</td>
<td>Transition-RRO</td>
</tr>
<tr>
<td>RRW</td>
<td>Full</td>
<td>Partial</td>
<td>Copy to twin</td>
</tr>
</tbody>
</table>

3) Bookkeeping for Update Accesses: Update accesses occur when message passing is used to receive data into a buffer that is symmetrical with respect to the sender's buffer. Table III lists the actions necessary for update accesses. Bookkeeping is required only for remote pages. The general objective is to bring remote pages into the RRO state. This is done using the RUM fault handler for a partially overlapping RUM page. The effect is to obtain a full page update from the home process, and then have the overlapping part of the page updated again via the MPI call.

Pages in the RUM and RRW states that are fully within the message buffer can safely transition into the RRO state after the MPI call completes. On the other hand, RRW pages that only partially overlap the message buffer remain in the RRW state because the twin might be required for modifications in the other part of the page. For the updated part of the page, however, the twin must be updated by copying from the updated portion of the page. This ensures that any differences between the original twin and the updated data will not be mistaken as modifications at the next memory sequence point.

V. PERFORMANCE

The integrated shared memory and message passing programming model makes it possible to use bulk data transfer communications on shared memory to replace process synchronization and flushes. However additional bookkeeping work is required on top of the actual data transfer to maintain memory consistency. This section reports the performance of the integrated programming environment (SDSM+MP) against pure message passing (MPI) and equivalent shared memory only (SDSM) programs.

The Ohio State University MPI Benchmarks (OMB version 3.1.1) [7] is used to compare the bandwidth of the three implementations. Data dependencies need to be injected into the message passing benchmark code to induce data transfer in the SDSM implementation, and bookkeeping in the SDSM+MP implementation. This was done in the latency benchmark because the bandwidth benchmark would have resulted in data races when converted into the shared memory form. Thus the results reported are based off modified versions of the latency benchmark.

Latency is measured using two processes in a “ping-pong” style of communication where a process sends a message, waits for a response, and then repeats the process for several iterations. Two approaches were taken to inject data dependencies into this communication pattern. In the first form illustrated in figure 9, each process is issued a pair of buffers. During an iteration, processes will modify one of their buffers and read from the buffer modified by the other process during the previous iteration. Read/write conflicts are avoided by swapping the buffers before starting the next iteration. This approach exhibits good locality for the writers because each buffer is written to by only one process. The SDSM version is implemented by replacing the data exchange with the SDSM barrier operation (figure 10).

Steps for the second approach is shown in figure 11. In this version, only one buffer is used. Processes take turns to modify this buffer before sending the modified contents to the other process. This variation exhibits poor locality for the writers because both processes are writing to the same buffer. The SDSM version is implemented by replacing the send and receive calls with SDSM barriers (figure 12). Unlike the first
approach, two barriers are required in this case.

The overhead of the dependency injections are minimized by only accessing one byte per page. Thus the data access overhead will be insignificant compared to the overheads from the page faults, page fetches, and various other bookkeeping tasks.

The modified benchmarks are executed using two nodes connected by Gigabit Ethernet. Each node has a Core2 Quad-core Q6600 2.4 GHz processor and 4 GB in memory. The MPI library used is Open MPI 1.2.8.

Figure 13 plots the results of the first approach that exhibits good locality for writers. Performance of the SDSM+MP implementation is similar to the SDSM for buffers which are smaller than 4KB, which is the SDSM page size. For buffer sizes that are 4KB or bigger, the SDSM+MP implementation performs as well as the pure MPI implementation. This is expected because writers only need to claim their buffers once. In the iterations that follow, pages in the large buffer will remain in the LRW state so modifications do not trigger page faults. At the send operation, the read access on local pages do not require any bookkeeping work (see table I). Thus the only overheads are the data transfer and the update access on remote pages. The latter also involves no bookkeeping work because those remote pages would already be in the RRO state from the first access (see table III).

Figure 14 plots the results of the second approach that exhibits poor locality for writers. Similar performance is observed for buffers which are smaller than 4KB. For buffer sizes that are 4KB and bigger, the SDSM+MP implementation performs better than the SDSM implementation but poorer than the pure MPI implementation. This cost is attributed to the cost of the page claiming protocol and the cost of making twins for each page it writes to. The twins are never actually used because the process discards them when it sends the modified data and claims the pages. Even so, the performance of the SDSM+MP implementation surpasses that of the SDSM implementation, which also has the advantage of home migration because barriers are used, but has to fetch pages as it encounters them.

VI. RELATED WORK

Inspiration for this work largely originates from the MIT Alewife machine project that integrates message passing primitives into the Hardware Distributed Shared Memory (HDSM) interface [16]. The Danui SDSM builds on this by applying the motivations for an integrated environment to the SDSM context. Memory consistency issues are also greatly different because of the different approaches to caching, and also because the SDSM uses a relaxed model.

This form of integrated message passing and shared memory programming is different from hybrid MPI+OpenMP programming [18], [19], [20], [21], [22] because shared memory is extended across the message passing peers. In the MPI+OpenMP models, shared memory programming is limited to within each MPI process. Thus in those cases MPI does not need to be concerned with memory consistency across message passing peers.

Mirchandaney et al [23] extends the TreadMarks SDSM with routines such as DSM_send and DSM_recv amongst others. However, these were designed to facilitate propagation of diffs in a mixed eager/relaxed memory consistency environment; DSM_send eagerly sends diffs to the receiver while DSM_recv requests for those diffs before page faults on the region occurs.

Zhu et al [24], [25] combines MPI programming with the TreadMarks [26] SDSM. The approach uses compiler analysis to classify arrays as private, distributed, and shared data. The SDSM only needs to manage shared data. Distributed data is allocated outside of shared memory, and their data dependencies resolved using MPI communication calls inserted by the compiler. It was identified, as a problem, that arrays with varying usage patterns may shift between distributed and shared classification during the execution. In such cases, the compiler will default to the shared data classification. Our integrated SDSM+MP programming approach differs in that MPI can be used on shared memory, so it will be useful for data with varying classifications.
VII. CONCLUSIONS AND FUTURE WORK

Shared memory programming is convenient for unpredictable memory access but overheads can be unnecessarily expensive if data dependencies can be solved easily using message passing. We have shown that message passing can be efficiently integrated into a HLRC SDSM programming model. To achieve this, message passing operations were translated into read, write, and update accesses. Although this information is local to a process, it can be used to deduce some of the memory access characteristics at other processes. This technique allowed the Danui SDSM to identify single-writer situations: where the process is the only writer for a region of memory. This in turn provided opportunity for initiating sided-homogeneous migration to avoid flush costs and improve data locality.

Bandwidth performance measurements using good and poor locality tests show that the SDSM+MP implementation outperforms the SDSM approach in both cases. When data locality is poor, bookkeeping overheads dampen the performance of the SDSM+MP but still allows the implementation to achieve a 0.7 Gb/s bandwidth on the Gigabit Ethernet network; pure MPI achieved a bandwidth of 0.9 Gb/s for the same test. But when data locality is good, performance of the SDSM+MP implementation can match that of pure MPI.

The bandwidth tests bring to light the potential for bulk data transfer using integrated message passing to significantly improve performance. We are currently working on producing some application examples to demonstrate use of the model, and also to carry out scalability tests. Where required, the implementation can also be extended to include more collective MPI operations by applying the appropriate memory read, write, or update accesses on the message buffers. For example, the MPI_Bcast operation was implemented by applying a read access at the root process, and write/update accesses at the other processes, depending on whether buffers are asymmetrical or symmetrical.

ACKNOWLEDGEMENTS

This work was part funded by Australian Research Council Grant LP0669726 and Intel Corporation.

REFERENCES