Efficient Support for Two-Dimensional Data Distributions in Distributed Shared Memory Systems*

David K. Lowenthal†
Vincent W. Freeh‡
David W. Miller†

Abstract

Despite their clear advantage in scalability, two-dimensional data distributions are not efficiently supported by current software distributed shared memory (SDSM) systems. This is because sharing between nodes occurs on both columns and rows. Sharing in two dimensions is not a good match for SDSM systems, because either a row- or column-major data layout of pages leads to (1) severe thrashing, if a strong memory consistency is used, or (2) exchange of unnecessary data between nodes, if a relaxed memory consistency is used.

This paper examines two alternatives for efficiently supporting two-dimensional data distributions in SDSM systems. We develop two new page consistency protocols for this purpose. One protocol, called Explicit-2D, requires that the user or compiler explicitly identify truly shared elements within *a page; the other, called Implicit-2D, infers such elements implicitly. Knowledge of truly shared elements allows the SDSM, at synchronization points, to send only truly shared data, which reduces* diff *sizes. As the problem size or the number of nodes grows, programs written using a two-dimensional distributions with our new protocols are superior to those using a one-dimensional one. The difference in our tests is as much as 12% for Red-Black SOR, and increases with the problem size and number of nodes.*

1 Introduction

One key to writing efficient parallel programs is choosing an effective distribution of data to processors (nodes). An ideal distribution balances the computational load and minimizes communication. For applications that perform uniform computation on each data element, balancing the load can be effectively done by assigning each node an equal number of data elements. The key factor then becomes minimizing communication overhead. For many applications, a two-dimensional data distribution, where each node is assigned a two-dimensional subgrid, has less communication overhead than its one-dimensional counterpart.

Unfortunately, two-dimensional distributions are not efficiently supported in software distributed shared memory (SDSM) systems [14]. In an SDSM [11, 3, 10, 7], two-dimensional distributions can cause a large amount of excess communication between nodes, even when using relaxed memory consistencies such as eager write shared [3], lazy write shared [10, 7, 8], or a hybrid [1]. The primary problem is that two-dimensional distributions force nodes to share data in two dimensions, but the data is organized in either a row- or column-major layout. Consequently, nodes share both rows and columns, meaning that *every* element in a page must be write-shared, even though typically, in the non-major axis only a small number of values are truly shared. This is true no matter which write-shared variant is used.

This paper introduces two alternatives, *Explicit-2D* and *Implicit-2D* , that extend write-shared protocols to allow two-dimensional data distributions. The first provides a programming interface through which the user (or compiler) can indicate where columns are shared between nodes. The second infers shared columns automatically, obviating specification of the shared data. Both mechanisms provide the information necessary to determine which portions of the page are truly shared. This allows the SDSM to avoid communicating useless data. When the sharing pattern is stable, programs written using a two-dimensional distributions execute 12.3% faster than a one-dimensional version when using 25 nodes on Red-Black SOR. Furthermore, we show that two-dimensional distributions scale better than their one-dimensional counterparts. Finally, we show that Explicit-2D is the appropriate protocol when the sharing pattern is not stable, while Implicit-2D is better when either the sharing pattern is stable or the shared columns cannot be determined statically.

†Department of Computer Science, The University of Georgia, Athens, GA 30602. Contact:dkl@cs.uga.edu

‡Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556. Contact: vin@nd.edu

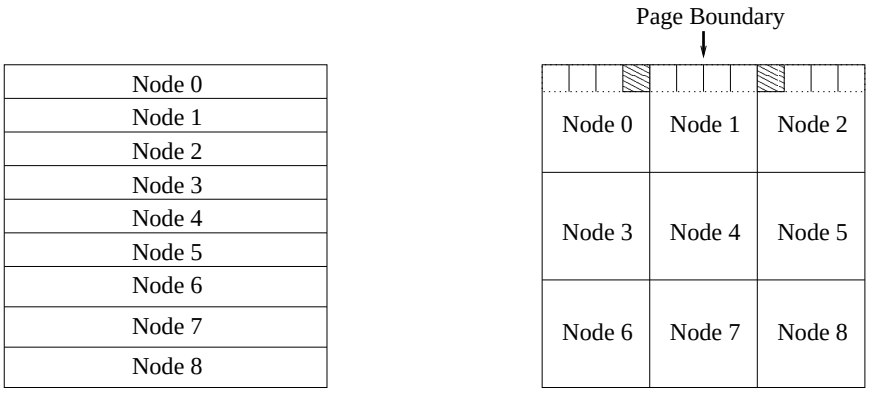

Figure 1. A one-dimensional and two-dimensional data distribution in an SDSM. In the former, each node is assigned a equal number of consecutive rows; in the latter, each node is assigned an equal-sized square. Also, the layout of a row (which is comprised of two pages in this example) in the two-dimensional distribution is shown.

The rest of this paper is organized as follows. The next section gives an overview of the problem. Section 3 describes the implementation, and Section 4 gives performance results. Finally, Section 5 summarizes the paper.

2 Overview

Parallelization of grid applications with uniform workloads and regular, nearest-neighbor sharing patterns are often carried out by assigning each node a contiguous region of the grid. Such an assignment results in a node requiring data from each of its adjacent neighboring nodes. Assuming the dimensionality of the array is at least two, typically one or two dimensions of the grid are distributed. The scalability advantages of two-dimensional distributions have been noted by other researchers [5]. However, current SDSM systems do not efficiently support such distributions. This is despite the significant amount of SDSM research that has focused on ways to reduce consistency-related communication by using *diffs* on shared pages [3, 10, 7, 1]. (See [7] for a full discussion of these techniques.)

No SDSM system we know of supplies the user with a protocol that allows efficient execution of multi-dimensional distributions. Standard write-shared protocols will incur excess communication for even simple two-dimensional distributions on very regular applications; this is because they can only handle false sharing, not true sharing [1]. The left side of Figure 1 is a strips-based distribution, where each node is assigned a contiguous set of rows; in the right part, each node is assigned a square. The right-hand side also shows the first row of the array. In this example, assume the row is split over two pages. Suppose this row is written and the program uses nearest-neighbor communication, and consider the induced communication from the point of view of node 1. If an eager write shared protocol is used, at the end of iteration i, node 1 must create a *diff* for both pages, because it writes to parts of both. This *diff* is sent to both node 0 and node 2. Node 1 must also receive *diffs* from node 0 and node 2. These (entire) *diffs* are sent, received, and applied, even though each node will require only boundary points (shaded in the figure), which often are a small percentage of the modified data within that page.

The primary problem is that while write-shared protocols tolerate false sharing, they force every *element* that is modified on a write-shared page to be communicated, even if that element is not needed by any other node. One solution to this is given in [14] (and was previously discussed but not implemented in [4]). Their solution is to use static analysis to detect this situation; they then have their compiler rewrite the code to duplicate the shared column in private memory and explicitly communicate that data between nodes. While this solves the problem, it requires significant compiler analysis, code rewriting, and additional memory. A final solution is to use a language such as Java where there is an extra level of indirection on object accesses [6], but this relies on using such a language.

Our approach is instead to develop new protocols that allow the user (or compiler) to inform the SDSM system of what data is actually needed by other nodes, so that *only* that data can be communicated. Our technique is implemented in two new SDSM protocols described in the next section.

3 Implementation Alternatives

We experimented with two basic alternatives to achieve efficient support for two-dimensional distributions. Both protocols, which we call Explicit-2D and Implicit-2D, are implemented inside the Filaments SDSM [12], which supports multiple consistency protocols including eager release

consistency. Although the new protocols do not support three-dimensional (or greater) distributions, most parallel programs can typically achieve as much efficiency from two distributed dimensions as from three or more [5].

3.1 Explicit-2D

In the Explicit-2D protocol, the sharing pattern is specified by the user (or compiler); an API call (`shareCol`) contains information that allows the SDSM to know what columns to send. This allows the SDSM to avoid transmitting entire pages when only a small number of elements on that page are actually needed.

3.1.1 User Interface

The user interface contains only two function calls. The first, `shareCol`, indicates that a column or columns is shared between two nodes; this allows us to specify regular array access patterns. It takes all necessary information, including the sharing nodes, size and number of elements in the column, number of columns, and a stride. The second function, `stableSharing`, indicates that the sharing pattern does not change across iterations of a computational phase [3].

3.1.2 Implementation

For each call to `shareCol`, the Filaments SDSM creates a descriptor containing the above information as well as the starting and ending SDSM page on which the column lies. This will allow the SDSM to efficiently gather and scatter the elements when needed. Each node is sent the sharing information of all other nodes.

There are two ways to implement the Explicit-2D protocol. The first is by detecting and exchanging the columns that are shared between nodes. If elements of a column are not modified between synchronization points, they do not need to be disseminated to other nodes. Write faulting determines which column elements have been modified.

When a node write faults on a shared page, that page is either *marked*, if it falls within any column descriptor, or *cloned*, if it does not. At the next synchronization point, any page that is cloned is handled in exactly the same way as in eager release consistency. However, pages within the range of a column descriptor are handled differently. Specifically, each page in each descriptor is traversed, and a single message per descriptor is created. The message contains each page that is marked as written. This mimics what is done in an explicit message passing program. This message is sent to the destination node, which applies the changes to each page while traversing the descriptor. Finally, all nodes set the permissions on each page in a column descriptor to read-only in preparation for the next iteration.

There are several sources of inefficiency in the above implementation of the Explicit-2D protocol. Specifically, on each iteration, three actions occur: (1) initial write fault to each boundary element, (2) receipt of column(s) (at the next synchronization point), and (3) reset of column pages to read-only. Each of these has significant costs, including fault handlers, reprotection of pages, and message overhead. The cost of these items is substantial (see Section 4). If a stable sharing pattern has been indicated (through a call to `stableSharing`), then Explicit-2D keeps a stable sharing list for each iteration. At the end of an iteration, all *active* descriptors (ones whose columns were written), are placed on a stable sharing list for that iteration.

Then, descriptor lists for earlier iterations are matched against the one from the current iteration; if there is a match, then we have reached a stable sharing point. After we have found stable sharing, the actions taken by Explicit-2D change from using the (costly) detection scheme to simply sending only the needed columns for that iteration to nodes. This avoids overheads due to page faults and protection switches for the remainder of the computation.

If stable sharing is not specified, we can eliminate the above overheads by simply exchanging *all* columns. While this will, in general, send more data,[1] there is no need to keep track of which pages are modified. There is a tradeoff, though; when the number of columns is large and few are modified on each iteration, detection may be more effective.

3.2 Implicit-2D

While the Explicit-2D protocol results in an efficient program, it requires the programmer or compiler to specify explicitly shared data. In some cases this might be straightforward but in others it is difficult; in the general case, it cannot be statically determined. For the latter cases, we have developed the Implicit-2D protocol, which infers shared data using system-level techniques.

3.2.1 User Interface

The basic idea behind Implicit-2D is the following: when a remote access is made, a page fault occurs. However, instead of satisfying the fault by either obtaining a copy of the page or creating a clone, the access is logged, storing the address on which the fault occurred as well as who has the needed data. At the next synchronization point, before the normal barrier or reduction is carried out, all logged addresses are combined into one message (to each node), and the data is returned; then, the (deferred) computation can complete. This way both (1) avoids thrashing and (2) avoids

[1] This scheme sends exactly the same amount of data as the detection implementation in the case when the application *requires* that all elements of all columns are exchanged.

```
regular( ) {
    N_c := y_e − y_w + 1
    for i := startrow to endrow
        for j := startcol to endcol
            S_c := j − y_w
            A'[(N_c + j + S_c)/N_c][i][j] := B'[(N_c + j + 1 + S_c)/N_c][i][j + 1] + ...
            checkFaultOccurred(i,j);
}

update_deferred( ) {
    while (!deferred_updates.empty( )) {
        (i,j) = deferred_update.remove( )
        // data must now be present, so use R/W shadow copy
        A'[1][i][j] := B'[1][i][j + 1] + . . .
    }
}
```

Figure 2. Transformed Jacobi code. If a fault occurs, the update is logged, not actually computed. (The user code specifies exactly what should be logged.) After all necessary data has been acquired (which occurs at the next synchronization point), function `update_deferred` is called. It uses the copy of the page that is readable and writable.

sending useless data: It infers at run time the actual data that is needed and its location.

The user interface for Implicit-2D allows sharing to be determined dynamically. This requires user code modification, which we made by hand, although the required user code changes could be generated by a preprocessor using only standard compiler techniques as well as existing optimizations (see below).

The first modification involves using what we call *shadow variables*. Given a variable with a virtual address A_v that maps to a physical address A_p, a shadow variable is a distinct virtual address $A_v{}'$ that also maps to A_p. In our system, for each array in SDSM space, the user code must allocate *three* shadow arrays (described below). In practice, to make the code simpler, the three arrays are grouped into one, with an additional dimension added to distinguish between them. Our convention is that $B'[1][N][N]$ is a copy owned by the currently executing node, while $B'[0][N][N]$ and $B'[2][N][N]$ are owned by the left and right neighbors, respectively.

The next modification is to replace, for each array B, references $B[i][j]$ with $B'[F(i,j)][i][j]$. In particular, two-dimensional array accesses are changed to three-dimensional ones. Function F always evaluates to either 0, 1, or 2. A value of 1 should be returned for any access in that lies within the rectangular subarray owned by a node. We designate a node's owned subarray by coordinates (x_n, y_w) [upper left] and (x_s, y_e) [lower right]. When $i < x_n$ or $i > x_s$, the $B[i][j]$ is remote and is handled by current write-shared techniques for one-dimensional distributions. On the other hand, when $x_n \leq i \leq x_s$, this node owns points $B[i][y_w]$ to $B[i][y_e]$. However, $B[i][y_s + 1]$ is remote. Hence, in the above transformation from $B[i][j]$ to $B'[F(i,j)][i][j]$, we must have $F = 0$ if $j < y_w$, $F = 1$ if $y_w \leq j \leq y_e$, and $F = 2$ if $j > y_e$. For the access $B[i][j]$, we determine F via the formula $(N_c + j - S_c)/N_c$, where N_c is $y_e - y_w + 1$ and S_c is y_w. Intuitively, the column index (j) is divided by the number of column elements owned (N_c). The N_c in the numerator is to designate 1 as ownership (and 0 and 2 as left and right neighbors), and S_c is subtracted to shift the expression when it does not start at 0.

In general, we support any access of the form $B[ai + b][cj + d]$, where a, b, c, and d are constants. This access is converted to $B'[\frac{N_c+cj+d-S_c}{N_c}][ai + b][cj + d]$. The values N_c and S_c are determined inside the Implicit-2D protocol. The added array indexing overhead can be reduced by using known compiler techniques (e.g., [2]), which is orthogonal to our work.

The final user code modification is to the application code, as shown in Figure 2. The system will catch all references to neighboring nodes, i.e., references to $B'[0]$ and $B'[2]$. The user code must be modified to record (with assistance from the SDSM) which iterations reference data owned by neighbors. Because access permissions on neighbor shadows are protected, an access generates a page fault. The `checkFaultOccurred()` function, which is application specific, is inserted into the application kernel to record this information and queue the iteration so that it can be executed later. A second version of kernel code (*update_deferred*) is needed, which executes the deferred iterations. The deferred kernel uses the local shadow for all

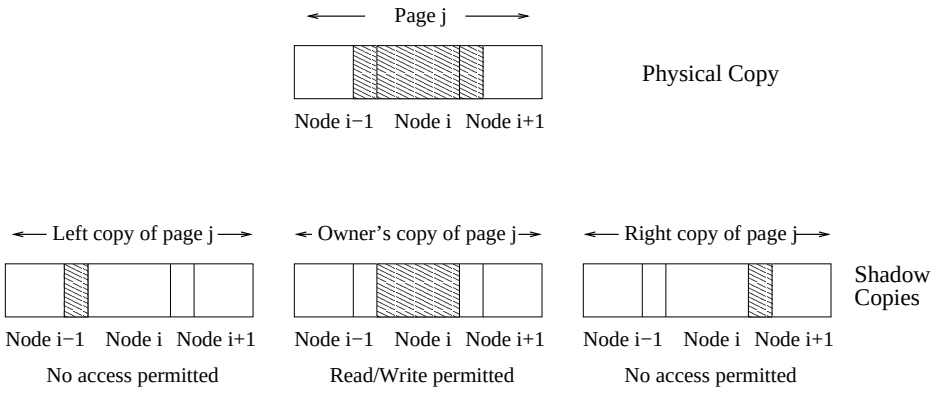

Figure 3. Picture of the page mapping scheme used by Implicit-2D. For each page j, three shadow copies of the page are mapped. The middle copy is readable and writable, whereas the other two represent data not owned and are hence not access permissible; accesses to these copies will be logged and re-executed at the next synchronization point.

references because it is executed after all remote data has been obtained.

3.2.2 Implementation

The basic idea behind the implementation of Implicit-2D is to allow several different views of a single page. This is similar to the work pioneered by the Millipede system [9]. On node i, we create three views for each page using `mmap`; one is owned by node i and allows reads and writes to complete uninterrupted. The other two are logically "owned" by nodes $i-1$ and $i+1$, assuming that node i is an interior node. These views are protected with `PROT_NONE`, which means that any accesses to them will be trapped. Figure 3 shows the situation. Accesses to an element owned by a left or right neighboring node are automatically redirected to the other copies by changing the value of the first dimension, which was described above.

Upon access to a view that is protected with `PROT_NONE`, the Implicit-2D handler code logs the faulting address as well as the specific values of the loop variables (which are passed in through the user code). It enqueues this information on a list (denoted *deferred_updates* here) that is also visible to the user. The offending page is then re-protected with `PROT_NONE` in case there are other addresses on it that are accessed. Most accesses will likely be to a view that allows reads and writes; these accesses are not affected.

At the next synchronization point, all needed data is determined by the addresses on the *deferred_updates* list. The request is combined into a single message for each of the left and right neighbors (assuming a node has both). Upon receipt of such a request, a node will respond with the needed data, which will result in an upcall to the user code, causing the deferred iterations to be executed.

After determining what elements are truly shared, the Implicit-2D protocol performs checks to see if the elements represent a column. If so, metadata in the message can be removed; furthermore, if stable sharing exists, we can achieve performance equivalent to the Explicit-2D protocol, as we describe next. The only requirement is to switch the function pointer to a version of the code that uses two-dimensional arrays.

4 Performance

This section reports the performance of two programs: Red-Black SOR and Jacobi iteration. For each application we executed three programs using using two-dimensional distributions with the Explicit-2D protocol. The different programs are (1) exchanging only the necessary columns (*Explicit-Precise*), (2) exchanging all columns (*Explicit-All*), and (3) stable sharing (*Explicit-Stable*). We also compare to programs using Implicit-2D both without and with stable sharing (*Implicit* and *Implicit-Stable*). For comparison purposes, we executed an eager write-shared protocol that used a one-dimensional distribution (*WS*) as well as an explicit message-passing program using MPICH [13] that used a two-dimensional distribution.

Below, we present the results of runs on either 9, 16, or 25 UltraSparc 5s, each with a 360 MHz processor and 8K page size, connected by a 100Mbs Fast Ethernet. All programs used `gcc` with the `-O` flag. All programs were run when no other user was on the machine, and all times reported are the median of three test runs.

4.1 Scalability

We evaluated the scalability of one- and two-dimensional distributions using Red-Black. We ran a 9-

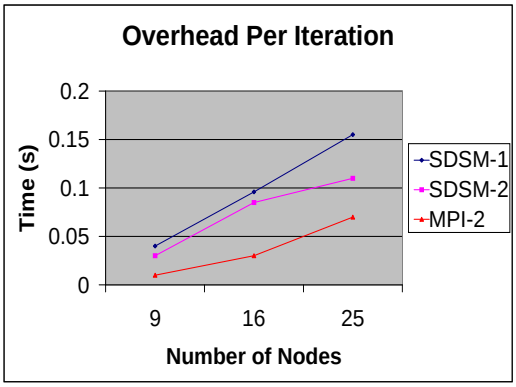

Figure 4. Graph of absolute overhead per iteration. Each program used a grid sized such that each node was assigned a 1024×1024 subgrid. The increase in overhead is much less with a two-dimensional distribution.

node Red-Black iteration test using a 3072×3072 grid, a 16-node test using a 4096×4096 grid, and a 5120×5120 grid using 25 nodes. The grid is scaled so that each node was assigned the same amount of data in each test. An increase in the problem size when using a two-dimensional distribution does not affect the communication due to data exchange (except for boundary nodes), whereas the communication increases with the one-dimensional distribution. There will always be an increase in total overhead due to barrier synchronization, which scales as $\mathcal{O}(\log \mathcal{P})$. Figure 4 shows that when moving from 9 to 16 nodes, the overheads in both the one- and two-dimensional programs increase at about the same rate. However, the rate increases much more in the one-dimensional program when moving from 16 to 25 nodes. This suggests that the difference will become more pronounced as the number of nodes increases. As a comparison, the results for the MPI two-dimensional distribution are also shown.

4.2 Red-Black SOR

For Red-Black SOR, we used arrays of size of 5120×5120 and 25 nodes. The tests were run for 100 iterations. We ran both one- and two-dimensional tests; the one-dimensional ones used a standard eager write-shared protocol, and the two-dimensional ones used either the Explicit-2D or Implicit-2D protocol. Each node was assigned a 256×5120 strip in the one-dimensional tests and a 1024×1024 square in the two-dimensional tests. Also, for Explicit-2D, we use the *stride* part of our user interface to avoid sending useless data when columns are exchanged (because in each phase, half the points are unneeded).

The performance of Red-Black is shown in Figure 5. The left graph shows the components of the overall time for a single iteration, including the time for computation, protection changes, faulting, copying, and releasing and applying *diffs*. The seven different programs described above were tested. It is important to note that for the stable sharing versions, we measured an iteration *after* stable sharing was found. The right graph shows overall execution time; this includes *all* iterations, including ones where stable sharing has not yet been detected.

From Figure 5, it is clear that the stable sharing tests (as well as the version that exchanges all columns) are superior to the one-dimensional write-shared test. Explicit-Precise suffers from significant overheads due to faulting, copying, and (especially) protection changes. Implicit suffers from significant per-iteration computational overhead to access three-dimensional arrays with a complicated index expression. The program that uses a one-dimensional distribution performs somewhat better than the two-dimensional detection version, but is inferior to the stable sharing version due to the increased size of the message exchange at the synchronization point. The result is that the performance of the two-dimensional distribution (with stable sharing) is 12.3% better than the one-dimensional distribution. It is also important to note that to make the one-dimensional tests as efficient as possible, we had to increase the socket receive buffer size. This reduces the number of message retransmissions necessary. The two-dimensional tests send a fixed amount of data, independent of the number of nodes. Because the amount of communicated data increases in the one-dimensional tests, eventually there is a point where the buffer size cannot be increased (although this point was not reached in our tests).

The best SDSM version (Explicit-Stable) is still 10.4% slower than the MPI program. A small amount of the overhead is due to general SDSM overheads that are not present in message-passing programs, such as page faults. However, while much of the programs are similar in structure, the messaging subsystems are different. Our SDSM uses UDP with reliability built on top, whereas MPI uses TCP. The SDSM programs have more variability than the MPI ones, because of costly message retransmissions. It is important to note that this overhead is not inherent to our SDSM; in fact, in test runs where there were very few retransmissions, the SDSM time was within 4% of MPI.

4.3 Jacobi Iteration

The results for Jacobi iteration are shown in Figure 6. Jacobi iteration is a similar program to Red-Black; however, it differs in three important ways: one (not two) barrier per iteration, two arrays (not one), and all column points are

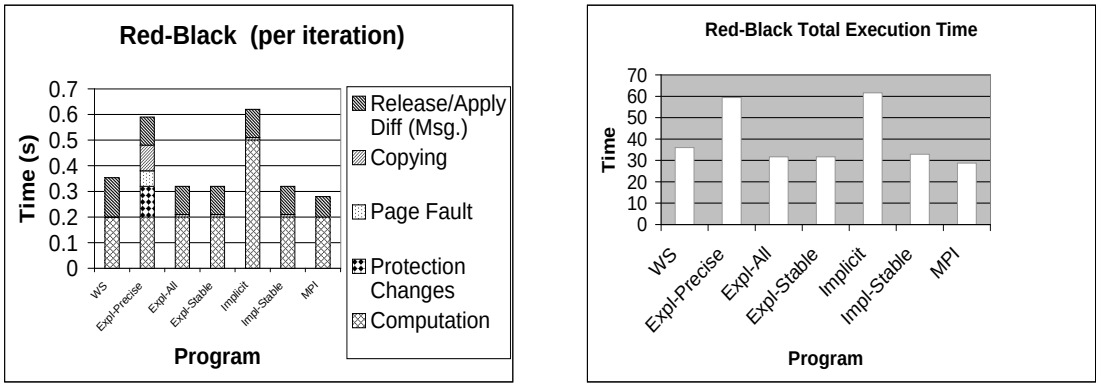

Figure 5. Results from several different Red-Black SOR tests on 25 nodes, one using a one-dimensional distribution (*WS*), and the others using a two-dimensional distribution. The left figure shows the per-iteration cost, and the right figure shows the total time taken for 100 iterations.

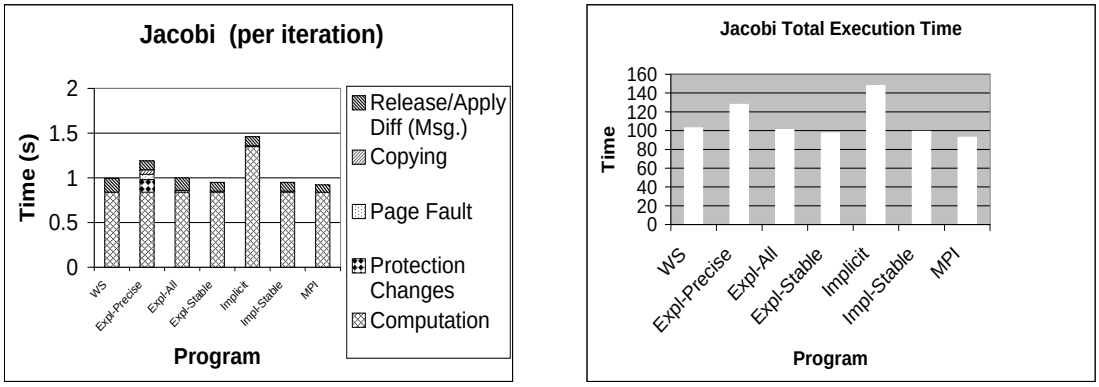

Figure 6. Results from several different Jacobi iteration tests on 16 nodes, one using a one-dimensional distribution (*WS*), and the others using a two-dimensional distribution. The left figure shows the per-iteration cost, and the right figure shows the total time taken for 100 iterations.

needed per iteration.

The key difference between Jacobi and Red-Black is that the presence of two arrays in the former causes the computation to be much greater. Hence, the difference in total execution time is small because a significant amount of computation is performed—in fact, the computation dominates the overhead. This is why the two-dimensional stable sharing programs show only about a 5% improvement over the one-dimensional version. Also, Explicit-All performs slightly worse in this case, because it needs to send and receive twice as many columns (it sends columns from *both* arrays, even though one array is not modified). However, it is still significantly faster than Explicit-Precise. The MPI version is only slightly faster than the best SDSM version, again because most of time in this program is spent in computation.

This section has demonstrated several important points. First, two-dimensional distributions are superior to their one-dimensional counterparts as the problem size or number of nodes increases. Second, Implicit-2D is the better protocol from the perspective of the user or compiler, but its performance is acceptable only if the sharing pattern is stable; otherwise, the extra computational overhead is unacceptable, and Explicit-2D should be used if possible. Third, an increase in computation time relative to the amount of data communicated results in a relative improvement in one-dimensional distributions.

A few other points concerning Implicit-2D are worth mentioning. First, the computational overhead increases if the same nonlocal element is accessed several times; this is because a fault will be generated on *every* access. Second, it is possible that the sharing pattern is not known until run time. In such cases, Explicit-2D cannot be used, while Implicit-2D can. We are currently investigating such applications.

5 Conclusion

This paper has discussed the design and implementation of two new software distributed shared memory (SDSM) protocols, Explicit-2D and Implicit-2D, that support two-dimensional data distributions. They are extensions to standard write-shared protocols. In our tests, the Explicit-2D protocol performs favorably (as much as 12.3% better) compared to one-dimensional data distributions using a standard write-shared protocol. Furthermore, two-dimensional distributions are more scalable than their one-dimensional counterparts, so we expect that the difference will increase along with increases in both the data set size and the number of nodes.

References

[1] C. Amza, A. Cox, L. Lin, S. Dwarkadas, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proceedings of IEEE*, pages 467–475, Mar. 1999.

[2] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, pages 112–125, June 1993.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium On Operating Systems*, pages 152–164, Oct. 1991.

[4] S. Chandra and J. R. Larus. Optimizing communication in HPF programs on fine-grain distributed shared memory. In *Sixth Symposium on Principles and Practice of Parallel Programming*, pages 100–111, June 1997.

[5] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techiniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, Mar. 1992.

[6] Y. C. Hu, W. Yu, D. Wallach, A. Cox, and W. Zwaenepoel. Run-time support for distributed sharing in typed languages. In *Fifth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 89–95, May 2000.

[7] L. Iftode. *Home-Based Shared Virtual Memory*. PhD thesis, Princeton University, June 1998.

[8] L. Iftode, J. Pal Singh, and K. Li. Scope consistency: a bridge between release consistency and entry consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[9] A. Itzkovitz and A. Schuster. Multiview and Millipage – fine-grain sharing in page-based DSMs. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[10] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, Jan. 1994.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4), Nov. 1989.

[12] D. K. Lowenthal, V. W. Freeh, and G. R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, 37:41–54, Nov. 1996.

[13] MPI: A message passing interface specification. June 1995.

[14] K. Zhang, J. Mellor-Crummey, and R. J. Fowler. Compilation and runtime optimizations for software distributed shared memory. In *Fifth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 83–88, May 2000.