Accurate Data Redistribution Cost Estimation in Software Distributed Shared Memory Systems *

Donald G. Morris III
Hewlett-Packard Company
19447 Pruneridge Avenue
Cupertino, CA 95014
dmorris@cup.hp.com

David K. Lowenthal
Dept. of Computer Science
The University of Georgia
Athens, GA 30602
dkl@cs.uga.edu

ABSTRACT

Distributing data is one of the key problems in implementing efficient distributed-memory parallel programs. The problem becomes more difficult in programs where data redistribution between computational phases is considered. The *global data distribution* problem is to find the optimal distribution in multi-phase parallel programs. Solving this problem requires accurate knowledge of data redistribution cost.

We are investigating this problem in the context of a software distributed shared memory (SDSM) system, in which obtaining accurate redistribution cost estimates is difficult. This is because SDSM communication is implicit: It depends on access patterns, page locations, and the SDSM consistency protocol.

We have developed integrated compile- and run-time analysis for SDSM systems to determine accurate redistribution cost estimates with low overhead. Our resulting system, SUIF-Adapt, can efficiently and accurately estimate execution time, including redistribution, to within 5% of the actual time in all of our test cases and is often much closer. These precise costs enable SUIF-Adapt to find efficient global data distributions in multiple-phase programs.

1. INTRODUCTION

Large-scale, time-consuming scientific programs are ideal candidates for parallelization on distributed-memory multicomputers. One of the fundamental problems in distributed-memory parallelization is to distribute data to the processors (nodes) so that communication is minimized and the computational load is balanced. The data distribution problem is complicated when scientific applications consist of multiple phases (sections of code between barrier synchronization points). Given a program that consists of a collection of phases, one must determine a *global data distribution*, which is an assignment of one distribution to each phase. It is often the case that better performance can be obtained using optimal per-phase distributions. However, if two successive distributions are distinct, a data redistribution is necessary.

A data redistribution in an explicit message-passing program is performed between phases by inserting send and receive calls on each participating node. While redistribution and computation are cleanly separated, message passing programs are often thought to be complex and error prone.

Software distributed shared memory (SDSM) systems [11, 2, 9] eliminate explicit internode communication by providing the abstraction of shared variables. Unfortunately, in an SDSM model, data redistribution is implicit and is intertwined with the execution of the subsequent phase, making it difficult to determine redistribution cost *accurately*. In particular, it is necessary to take into account the state each page is in after phase i, the type of access that will be performed on that page during phase $i+1$, and the SDSM consistency protocol that is used. Furthermore, because we are interested in applications where an effective global data distribution cannot be determined statically, SDSM redistribution times cannot be determined until run time. This means that any algorithm to estimate redistribution cost must have low overhead.

We are developing an approach to determining accurate redistribution times, with low overhead, in SDSM systems. We focus in this paper on supporting applications with regular communication patterns but possibly unbalanced workloads. Our approach has been implemented within the SUIF-Adapt system [6], which is a combination of the SUIF Stanford compiler toolkit [4] and the Adapt run-time data distribution system [12]. First, we modified the compiler to to generate deferred regular section descriptors (DRSDs), which are similar to regular section descriptors [5] except that they are created partially at run time. The key element of a DRSD is what we call a translation function, which is used by the run-time system to determine dynamically the write and reference sets for each node. We also modified the run-time system to determine redistribution time by (1) using the DRSD to determine which SDSM pages are accessed and then (2) dynamically simulating the state of the SDSM in each phase, using a state machine to determine protocol actions. In particular, this allows SUIF-Adapt to distinguish between page transfers and page invalidations. Furthermore, we develop an accurate model of redistribution between phases i and $i+1$ that takes into account both per-node and per-phase redistribution. Finally, we describe a series of improvements to our analysis that, in total, re-

*Supported by NSF Grant CCR-9733063.

PPOPP'01, June 18-20, 2001, Snowbird, Utah, USA.

duces its running time by an order of magnitude.

Performance results indicate the following:

- SUIF-Adapt estimates total phase time, including potential redistribution, to within 5% of the actual time.
- SUIF-Adapt uses the accurate estimates to determine efficient global data distributions. In particular, these estimates lead to choosing a more effective data distribution in many cases. This is important because the best distribution in our experiments was at least 10% better than the other candidate distributions whenever (1) the computational characteristics were nonuniform or (2) the nodes were not dedicated to the parallel application. Also, as the computation to communication ratio increases, the disparity grows.
- The time for our analysis, which is generally amortized over many iterations of the computation, was less than 2 seconds (out of a total of 50-150 seconds) for a program with several array accesses.

The remainder of this paper is organized as follows. Section 2 describes our framework and explains SUIF-Adapt. Section 3 provides implementation details, and Section 4 presents performance results. Section 5 discusses the applicability of our work to both irregular applications and write-shared protocols, and Section 6 concludes.

2. OVERVIEW

In this section we describe the computational model upon which our system is based and the basic strategy used by SUIF-Adapt.

2.1 Computational Model

Our model of computation is Single Program Multiple Data (SPMD), in which each node executes the same code but references a different subset of the data elements. The applications we currently address use regularly-accessed arrays and are divisible into one or more *phases*, which are sections of application code between two barrier synchronization points. We also consider only data distributions where the first dimension is distributed. This is because in an SDSM system, distributing more than one dimension requires a new SDSM protocol or significant compiler support [18, 3]. Furthermore, we assume the existence of a loop that directly encloses one or more phases; we call this a *phase cycle*. Our model assumes each phase uses either a variable block distribution, where a contiguous (but possibly unequal) set of rows are assigned to each node, or a cyclic distribution, where the rows are assigned to nodes in a modulo fashion. However, in this paper we focus exclusively on variable block distributions (of which `BLOCK` is a degenerate case). Finally, we assume that effective per-phase data distributions cannot be determined statically. A sample application (flame simulation) that has two phases enclosed in a phase cycle is presented in the left-hand side of Figure 1.

The key for a good data distribution is to minimize the maximum execution time of any node. In our work we are investigating multi-phase applications, so we seek a series of *local* distributions that will provide good performance for the entire phase cycle. This involves a tradeoff between good intra-phase performance and inter-phase redistribution cost.

We use a shared-memory programming model, which is provided through a software distributed shared memory, or SDSM. Hence, application programs can be written as conventional shared-memory programs as opposed to more complicated message-passing ones. When a non-local reference is made, a page fault occurs, and the SDSM system implicitly resolves the reference by sending a message to the node that owns the page. Note that we use a multi-threaded SDSM, so multiple page faults can be outstanding. Further information on SDSM systems can be found in, for example, [7]. A key issue involved in building an SDSM system is solving the *memory coherence problem*, which is typically done through a *page consistency protocol*; in this paper, we use the write-invalidate protocol [11], where each page has either a single writer or multiple readers.

2.2 SUIF-Adapt Strategy

Below we describe the aspects of SUIF-Adapt that pertain to this paper. The basic SUIF-Adapt strategy is described in detail in [6].

For a phase cycle with N phases, SUIF-Adapt constructs an $(N+1) \times (N+1)$ directed graph that we call the run-time data distribution graph, or RDDG. An example of an RDDG is shown in the right-hand side of Figure 1. This graph is similar to the one introduced by Kennedy and Kremer in their compiler-only solution to the global data distribution problem [10], which is similar to the "communication graph" originally described by Anderson and Lam [1]. For phase i, vertex (i, i) is the best local distribution for that phase. The other N vertices in row i consist of the best local distributions for all other phases in the phase cycle, plus a sequential distribution (which considers whether it is better to avoid parallelization of a phase). The last row is a copy of the first one, indicating that control returns from phase $N-1$ to phase 0. Edges in the RDDG are weighted to represent the execution time for the source phase in the original distribution plus the time spent to redistribute data (implicitly via SDSM page faults) into the target distribution. Adapt keeps track of time per row for each phase and so can predict the completion time for any one-dimensional distribution (e.g., `BLOCK`- and `CYCLIC`-based distributions).

Once the graph is constructed, SUIF-Adapt uses a greedy heuristic that finds shortest paths between the first and last rows; the minimum shortest path is an effective global distribution for the phase cycle. The redistribution is effected by modifying the loop ranges in a similar manner as [8]; this causes the necessary SDSM page faults.

Figure 1 shows the RDDG for flame simulation. In the first phase (numbered 0), the workload is uniform and the communication is nearest neighbor, so a `BLOCK` distribution for all arrays is desired; our modified SUIF compiler can infer this. Hence, vertex $(0, 0)$ is `BLOCK`. The best distribution for the second phase is unknown statically because the workload of *AdaptiveSolver* depends on the value of $x[i]$; Adapt will determine an appropriate variable-block distribution by measuring at run-time the computation time for each row and then dividing the rows between the nodes such that the load is balanced. For brevity, we denote this distribution as `VARBLOCK`, which becomes vertex $(1, 1)$. In our example (for 2 nodes), nodes 0 and 1 are assigned 1/4 and 3/4 of the data, respectively. From this, the RDDG is constructed by placing each distribution in each row (along with a vertex for sequential). For example, in Figure 1, `VARBLOCK` was found to be best in phase 1 and so is considered a candidate in phase 0, even though `BLOCK` is better in phase 0.

```
for time := 1 to timesteps {
  for i := 1 to N
    for j := 1 to N
      x[i,j] := x[i,j] +
        F(y[i-1,j],y[i,j],
          y[i+1,j],z[i+1,j])
  for i := 1 to N
    for j := 1 to N
      z[i,j] := AdaptiveSolver(x[i,j])
}
```

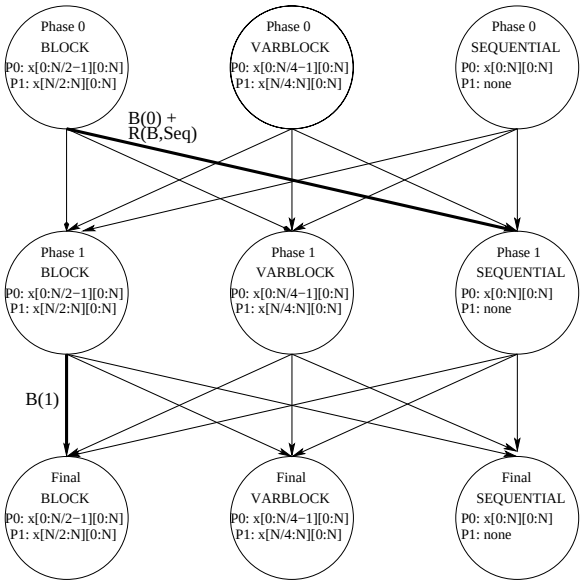


Figure 1: Flame simulation outline and resulting RDDG. On the left-hand side is the code for the flame kernel; each (entire) `for i` loop constitutes a phase. On the right-hand side is the resulting RDDG (assuming two nodes). In this example, only the distribution of array x is shown. We assume that the run-time measurements taken by Adapt find that to balance the load in phase 1, node 0 does $1/4$ of the work and node 1 does $3/4$. For brevity, we denote this distribution as `VARBLOCK`. Two edges are highlighted; $B(1)$ denotes the time to execute phase 1 in a `BLOCK` distribution (there is no redistribution cost). The other edge must include $R(B, Seq)$, the time to redistribute from `BLOCK` to a sequential distribution.

The effective global distribution is determined by finding the minimum of the three shortest paths (`BLOCK`, phase 0 to `BLOCK`, final phase; `VARBLOCK`, phase 0 to `VARBLOCK`, final phase; and sequential, phase 0 to sequential, final phase).

This paper focuses on computing precise redistribution estimates. There are two primary difficulties. The first is producing accurate per-node redistribution costs. This requires ensuring that SUIF-Adapt can distinguish between page transfers and (much cheaper) page invalidations. Otherwise, we cannot precisely determine costs due to differences in data access sets between phases. Precise estimates are determined through a combination of compile-time analysis, where deferred regular section descriptors (DRSDs) are generated, and run-time analysis, where the bounds of the DRSDs are computed, and the SDSM is dynamically simulated in each phase. It also requires "unrolling" the RDDG to represent explicitly multiple iterations of a phase cycle (Figure 2). In other words, the first two rows of the graph are copied and placed directly below the current first two rows, and the final row is placed last. A path through this graph would represent *two* iterations of a phase cycle. Otherwise, the edge costs can contain errors due to inaccuracies in the SDSM state upon entering the first phase in a phase cycle.

The second difficulty is producing accurate per-phase redistribution costs. Specifically, SUIF-Adapt needs to take into account both redistribution and computation. Otherwise, it is unclear what is the exact per-phase cost; the most likely choice would be to use the maximum redistribution cost of any single node, which is inaccurate (see Section 3).

The process of determining accurate per-node and per-phase redistribution costs is nontrivial. The next section

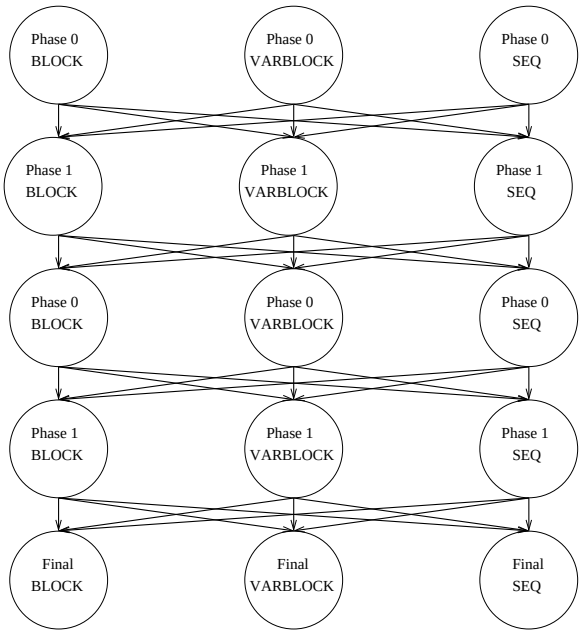


Figure 2: Unrolled RDDG. The first two rows represent the first iteration of the phase cycle, and the second two rows represent the second iteration. As before, the last row represents a return to the first phase.

will explain how we address these issues.

3. IMPLEMENTATION

We have developed our analysis to obtain accurate redistribution times within the SUIF-Adapt system [6]. Of note is that SUIF-Adapt is implemented on top of the Filaments package [13], which provides a multithreaded software distributed shared memory (SDSM). As described above, we focus on the write-invalidate protocol; furthermore, we assume that the pages are *padded* when necessary to avoid false sharing. Section 5 discusses possible extensions to SUIF-Adapt to handle most write-shared protocols. In this section, we describe in turn the key compiler modification and the key run-time modifications.

3.1 SUIF Modifications

As discussed in Section 2, a model for precise page access determination is needed for the run time system to accurately estimate redistribution costs. From the compiler side, this requires informing the run-time system about each array expression in each phase.

To carry out this task, we modified SUIF to generate what we call Deferred Regular Section Descriptors (DRSDs). DRSDs extend Regular Section Descriptors (RSDs) [5]; both express an array reference in a simple, compact form of a start, end, and step in each dimension of the array. One DRSD is generated for each array reference in each phase.

The difference between DRSDs and RSDs is that the former defers computation of the start and end, in exactly *one* of the dimensions, until run time. This is done through a *translation function*, which is a function that is exactly the first index expression that contains the phase loop variable. In this way the local loop bounds for each node are determined. This allows correct determination of changes in page ownership. For example, $A[\alpha_1 i + \beta_1]$ and $A[\alpha_2 i - \beta_2]$ provide different ownership and access ranges for the same loop (where $\alpha_1 \neq \alpha_2$ or $\beta_1 \neq \beta_2$). Regular RSDs for such references would allow Adapt to know that *some* node referenced the array; the translation function present in the DRSD allows the run-time system to calculate exactly which node performs which accesses. Without a translation function, it would be necessary to perform static analysis to determine read and write sets for each array reference (similar what is done in Fortran D [17]). However, this would be problematic, because Adapt modifies loop bounds at run time to balance the load, which would cause the sets to become stale. An example of the code generated by our SUIF frontend is shown in Figure 3; it includes one DRSD. Note also the dynamic instantiation of the loop bounds in each phase.

3.2 Adapt Modifications

As described in Section 2, SUIF-Adapt must label each RDDG edge with an accurate redistribution cost. To obtain this cost, we first must calculate accurate *per-node* redistribution costs. These costs can be used to calculate *per-phase* redistribution costs.

In this section, we first describe how SUIF-Adapt determines per-node costs and then how it determines per-phase costs. Finally, we describe two optimizations that we implemented in SUIF-Adapt, one to improve the accuracy (unrolling the RDDG) and one to reduce the overhead ("fast-forwarding").

```
translationFunc(int i) {
  return i+1
}

adpCreateDRSD() {
  D = ConstructDRSD("z", 2)
  D->phase_loop_var_dimension = 1
  D->func = translationFunc
  D->dim[1].step = 1
  // evaluation in dim. 1 of start, end deferred
  D->dim[2].start = 1
  D->dim[2].end = n
  D->dim[2].step = 1
  adpAddDRSD(0, D, READ)
}

adpPhaseCycle(2)  // contains 2 phases
adpCreateDRSD()   // for reference z[i+1][j]

for time := 1 to timesteps {
  adpGetLoopBounds(0, &start, &end, &step)
  for i := start to end by step
    for j := 1 to N
      x[i,j] := x[i,j] +
        F(y[i-1,j],y[i,j],
          y[i+1,j],z[i+1,j])

  adpGetLoopBounds(1, &start, &end, &step)
  for i := start to end by step
    for j := 1 to N
      z[i,j] := AdaptiveSolver(x[i,j])
}
```

Figure 3: A portion of the output of our SUIF frontend for the Flame code. A DRSD is shown for the array reference $z[i+1][j]$. Note that the evaluation of start and end in the first dimension is deferred (because i is the loop variable). Note also that the loop bounds are determined dynamically.

3.2.1 Per-Node Redistribution Costs

To determine the per-node redistribution costs precisely, SUIF-Adapt uses the DRSDs generated by the compiler to calculate precise lists of SDSM pages accessed during phase execution; then, page state transitions are computed. Figure 4 shows a DFA for page state transitions using the write-invalidate protocol. The page state in phase i and whether the page is read or written in phase $i+1$ determines the type of communication needed. For example, suppose a page ϕ is held by node 0 in *exclusive* mode in a source distribution in phase i, and is read by node 1 in phase $i+1$. SUIF-Adapt adds the cost of a page transfer from 0 to 1 to the total redistribution cost incurred by node 1 as well as the cost of a page service to the cost incurred by node 0. Furthermore, the target distribution is updated so that page ϕ is *read shared* between nodes 0 and 1 after phase $i+1$. Note that the preceding example assumes that node 1 knows the owner of page ϕ; if not, SUIF-Adapt adds in the cost of forwarding page requests.

The procedure above is used on *each* page in *each* DRSD. The problem becomes how to efficiently obtain all pages

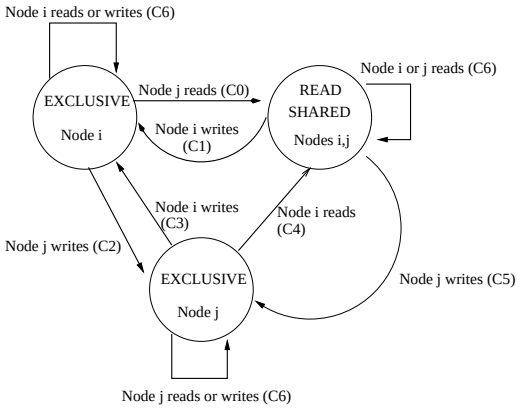


Figure 4: DFA showing page state transitions and resulting communication costs for the *write invalidate* protocol. Note that page forwarding is not included, though it is handled by SUIF-Adapt.

accessed from each DRSD.

A straightforward algorithm for determining which pages are accessed is to simply generate each *element* in each DRSD. However, this is unnecessarily expensive, because the granularity of communication in an SDSM is a page. Hence, our initial approach to obtain page accesses from a DRSD was to generate elements from the DRSD only when a new page is accessed. To accomplish this, we kept track of the number of elements needed to fill a page. For example, if a two-dimensional array fits the entire second dimension on a single page, only the first dimension needs to be traversed. Similarly, if half of the second dimension fits on a page, the DRSD is traversed by half the size of that dimension. The original step of the DRSD is still used if it is larger than the number of array elements per page.

The above algorithm invokes the redistribution cost algorithm once per DRSD per edge. This means that each page in each DRSD is traversed for each edge in the RDDG, with potentially many DRSDs covering large arrays. In an $N \times N$ RDDG, if phase $i+1$ entails a total of Φ page *accesses*, recalculation at every edge for the transition between phases i and $i+1$ requires $N^2\Phi$ calls to determine the next page.

The key observation is that in the RDDG, a row of nodes represents a single phase of program execution. Because all nodes execute the same code, the union of all SDSM pages touched by the nodes in a phase is the same, regardless of the distribution of the data. The ownership of the pages will change with different distributions, leading to corresponding changes in communication costs. However, we are guaranteed that pages are accessed by *some* node, independent of the particular distribution. Hence, we modified our algorithm so that we determine the pages accessed *once* per phase and then compute N^2 source/destination pairs (including communication costs) for that phase. This reduces the number of calls for the transition between phase i and phase $i+1$ to Φ, for a savings of $N(N-1)$ calls. Note that the state of each page still must be updated each time; in other words, for each different source/destination pair, we must recompute the new SDSM state.

3.2.2 Per-Phase Redistribution Costs

Given accurate per-node redistribution costs, we need to find accurate per-phase redistribution costs. This requires consideration of redistribution *and* computation.

Action	T_0	T_1	Total Time
$R^k_{i,i+1}$	8	2	—
Phase $i+1$ (comp. only)	10	15	15
Phase $i+1$ (comp. + $R^k_{i,i+1}$)	18	17	18
$R_{i,i+1}$	–	–	3

Table 1: Redistribution time example. T_i is the time incurred on node i for a particular action. In this example, we assume that node 0 takes 8 time units to redistribute and 10 units to compute; node 1's values are 2 and 15. $R^k_{i,i+1}$ is the time for node k to redistribute from phase i to $i+1$. The two middle rows show times for phase $i+1$ both excluding and including redistribution time. Although the largest local redistribution time for a node is 8 time units, the global redistribution time ($R_{i,i+1}$) is only 3.

We now describe how we find the per-phase redistribution cost (denoted $R_{i,i+1}$) between two RDDG vertices in different phases. This assumes that there are P nodes and that communication can proceed in parallel between different pairs of nodes. First, find the maximum of the execution times of the nodes in phase $i+1$, $T_{exec} = \max_{k=0}^{P-1}\{T^k_{i+1}\}$, where T^k_{i+1} represents the execution time for node k in phase $i+1$. Next, find the maximum of the total per-node times (the sum of execution time and redistribution time), $T_{total} = \max_{k=0}^{P-1}\{T^k_{i+1} + R^k_{i,i+1}\}$, where $R^k_{i,i+1}$ is the time spent due to redistribution from phase i to $i+1$ for node k. Recall that we described how to compute $R^k_{i,i+1}$ in the previous section. Then $R = T_{total} - T_{exec}$ is the correct weight of the edge, as it represents the difference between when the computation *does* finish, given the redistribution, and when it *would have* finished, if no redistribution were done. An example is shown in Table 1.

3.2.3 Optimizations

We make two main improvements to the procedures described above. First, for accuracy, we unroll the graph once. This avoids inaccuracy due to a different SDSM state upon entry to a phase cycle than upon return at the end of the phase cycle. In general, phase cycles are executed many

```
for step := 1 to numsteps {
  for i := 1 to N
    for j := 1 to N
      y[i][j] = 0.25 * (x[i-1][j] + x[i+1][j] +
                        x[i][j-1] + x[i][j+1]);
  for i := 1 to N
    for j := 1 to N
      x[i][j] := y[i][j];
}
```

Figure 5: Jacobi iteration outline.

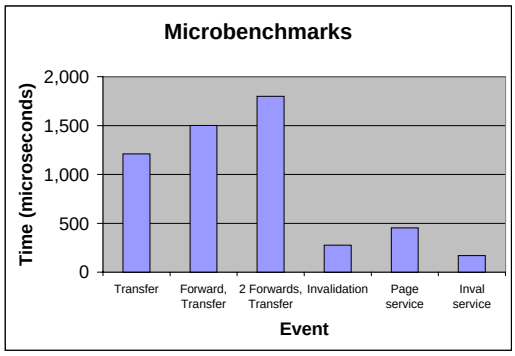


Figure 6: Time, in microseconds, for a single page transfer (with and without forwarding), page invalidation, page service, and invalidation service. Each test measured 512 events and took the average time per event. To obtain these numbers, we executed microbenchmarks in which one node requests several pages, which are served by another node; this is followed by one node invalidating the pages. (For forwarding, we used one or two intermediate nodes, explicitly setting the probable owner field to ensure the request would be passed on.)

times, so the SDSM state at the first phase in a phase cycle is, except for the first time it is executed, the one after the last phase in the phase cycle. We could unroll further, but we obtained good results by unrolling just once (see Section 4).

Second, for efficiency, we can in some cases avoid generating redundant page accesses. For example, given accesses such as $B[i+\beta_1][j]$ and $B[i+\beta_2][j]$, where $\mid \beta_1 - \beta_2 \mid$ is small, the associated DRSDs will have significant overlap. We remove the redundancy by adding a bit per page between each row of the RDDG. When a page is accessed for the first time, the corresponding bit is marked. During a traversal of a DRSD, SUIF-Adapt checks to see if a page is unmarked before reporting it accessed. If a page is already marked, the algorithm "fast forwards" to either the next unmarked page or the first page owned by a different node, whichever occurs first. This method allows us to proceed through the entire DRSD descriptors without accessing any pages other than boundary ones, if all the pages have previously been generated. It is more general than a compiler scheme to generate code to "factor out" all repeats; such a scheme is limited by what can be inferred statically. For programs such as Jacobi iteration and flame simulation, this reduces the page set determination cost substantially (see Section 4).

4. PERFORMANCE

This section discusses the performance of our accurate redistribution cost measurement in SUIF-Adapt. We perform three kinds of experiments, to determine: (1) how expensive our analysis is, (2) how well our analysis accurately models redistribution time, and (3) how much payoff we get due to accurate redistribution times. We use two test programs. Figure 5 shows Jacobi iteration, a well-known benchmark with two phases, a uniform amount of work in each, and nearest-neighbor communication. The other program is a flame simulation benchmark; we use the kernel that was shown in Figure 1. To simulate different amounts of work, we inserted parameterizable delay loops into each phase. The version of flame simulation that we tested had large amounts of work in both phases, though the second phase had an unbalanced load clustered in the top quarter of the grid.

All tests were run on 2, 4, and 8 200 MHz Pentium Pros connected by a switched 100Mbs Fast Ethernet[1]. Both the SUIF-Adapt system itself and the test programs were compiled under `gcc` with the -O2 flag. Both programs used matrices of size 1024×1024.

We used isolated experiments to determine the times for page transfer, page servicing, page invalidation, invalidation servicing, and page forwarding. These are the overheads that occur in a write-invalidate SDSM. In our microbenchmarks (see Figure 6), one node requests several pages, which are served by another node; this is followed by one node invalidating the pages. The total times are measured and divided by the number of events, which are then used by SUIF-Adapt to estimate the cost per event. Recall that our SDSM is Filaments [13], which can execute in multithreaded mode (allowing simultaneous outstanding page requests). This cuts the average page transfer time during redistribution approximately in half compared to when it executes in single-threaded mode. Furthermore, we had to modify our training set numbers to take into account that the time to service a page fault or invalidation is affected by page requests and servicing actions of the servicing node, which slows service time. For example, node i may request pages from node j, which would normally take time t_1, but if node j has to either request pages or serve other nodes itself, it will take node i time $t_2 > t_1$ to obtain its pages. This is not reflected in the figure, although it is implemented internally in SUIF-Adapt.

4.1 Cost of Analysis

Figure 7 shows the times to determine redistribution cost for the three different algorithms (described in Section 3). As can be seen, fast forwarding performs quite well (almost a factor of 2 improvement over the "by row" algorithm), whereas the original algorithm ("by edge") does not. Given that the redistribution time is amortized over many iterations, we believe that fast forwarding performs quite ac-

[1]Although the machines are relatively slow by current standards, the key issues we investigate also arise on faster machines.

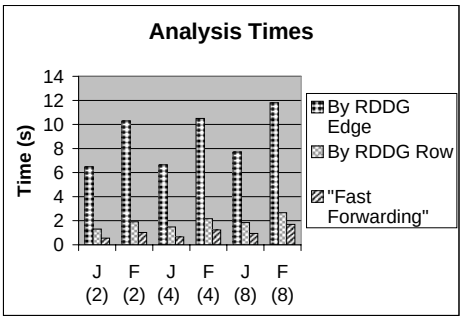


Figure 7: Time for execution of our analysis for Jacobi iteration (J) and flame simulation (F) in seconds for two, four, and eight nodes. By edge means that the set of pages accessed was determined once per RDDG edge, and by row means that the set was determined once per RDDG row. "Fast-forwarding" is the optimization that avoids processing a page multiple times.

ceptably. The execution time of our algorithm depends on number of array accesses per phase, as well as how much of each array is accessed. In Jacobi, there are 5 array references in one phase and 2 in the other; those numbers are 6 and 2 for Flame. In both, the entire array is accessed. While this is a moderate amount of array references, it is important to note that if the number of array references increases, the time taken by the computation itself will surely increase; hence, our analysis time should remain a small percentage of the overall computation. The analysis cost scales linearly with the number of nodes, because we check each node to determine whether or not it accesses each page. However, the cost per check is low, and in the case that few nodes actually access the page, little overhead is added. The cost also scales linearly with the number of unique pages accessed (as a result of array accesses).

Note that this analysis is not executed on each iteration; it is executed only (1) after the first iteration of a phase cycle or (2) when conditions change in the middle of a program. The latter can occur, for example, when a subset of the nodes become busy executing other processes.

4.2 Accuracy of Analysis

The accuracy of our algorithm is quite good, as shown in Table 2 (next page). The estimated and actual phase completion times are quite close, which is evidence that we have accurately modeled the major aspects of the SDSM. Note that these numbers were obtained with the RDDG unrolled once (see below). We attribute the small difference to experimental error caused by daemon processes, network delays, etc. In particular, our system precisely determined the number of invalidations and page faults caused by data redistribution (we instrumented our SDSM with counters for verification), which is critical because a page fault is five times more expensive than an invalidation.

The next test we ran was with flame simulation. We measured the effect of unrolling the RDDG; Figure 8 (previous

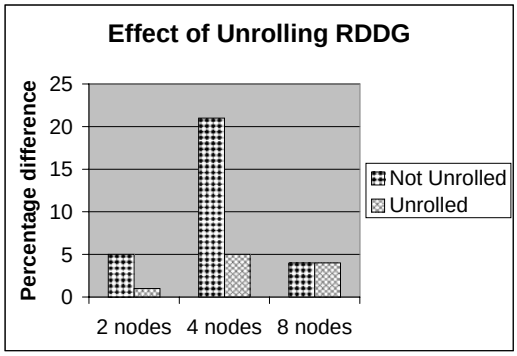


Figure 8: Comparison of accuracy between the RDDG when unrolled and not unrolled for flame simulation. The percentage difference refers to the difference between the estimated and actual execution times. It is clear that unrolling the graph makes a significant difference when there is redistribution (on eight nodes SUIF-Adapt chooses not to redistribute).

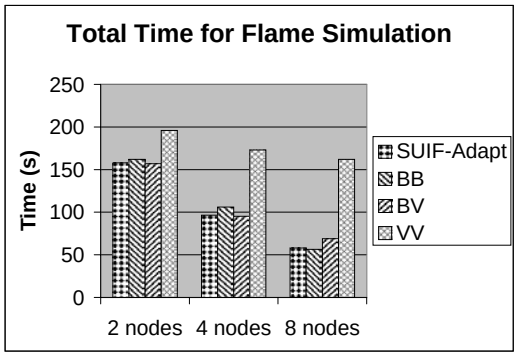


Figure 9: Flame simulation times in seconds. Letters (B/V) correspond to statically chosen distributions (`BLOCK`/`VARBLOCK`) for a phase. Note that SUIF-Adapt is competitive with the best hand-coded distribution in all cases. This is due to the accurate estimates produced by SUIF-Adapt. Without distinguishing between invalidations and page transfers, BB would be chosen in all cases. If redistribution time itself is not considered, BV will be chosen in all cases.

Number of Nodes	2		4		8	
	Predicted	Actual	Predicted	Actual	Predicted	Actual
Jacobi, Phase 0	.184	.182	.094	.098	.050	.054
Jacobi, Phase 1	.097	.095	.048	.048	.025	.028
Flame, Phase 0	11.7	11.8	6.07	6.38	2.85	2.79
Flame, Phase 1	3.75	3.78	2.86	2.94	2.77	2.77

Table 2: Accuracy of Jacobi iteration and flame simulation. All times are in seconds. Our analysis produces estimated program execution times that are close to the actual program execution time. Note that in Flame, different distributions are used on four and eight nodes; the four-node tests *include* redistribution cost in phase 0, whereas the eight-node tests use `BLOCK` in both phases and hence do not redistribute. (The *overall* speedup is not superlinear.)

page) shows the results. Clearly, unrolling the graph produces a much more accurate estimate. The reason for this is that upon entry to the phase cycle, each node exclusively owns an equal-sized block of each array (the chosen initial distribution). Hence, if the graph is not unrolled, SUIF-Adapt will overestimate some redistribution edges because page transfers appear necessary for all pages written by a different node in the next phase. However, the final row of the RDDG represents a return to the first phase; at this point, one of the arrays has pages that are read-shared between two nodes. In the steady state, this actually will cause only invalidations when one of the sharing nodes writes. Note that unrolling makes no difference on eight nodes because `BLOCK` (the initial distribution) is chosen for both phases.

4.3 Utility of Analysis

The above numbers show that SUIF-Adapt computes accurate redistribution costs. However, this is only of use if *better* decisions can be made when SUIF-Adapt uses this analysis. Figure 9 (previous page) shows results from our flame simulation kernel. On two and four nodes, a redistribution is chosen by SUIF-Adapt; on eight nodes, `BLOCK` is chosen in both phases. This does indeed match the best hand-coded distribution. It is important to note that the hand-coded distribution uses prior knowledge of application behavior, which is not available in general. Interestingly, if the RDDG is not unrolled, the assumed page transfers described above cause SUIF-Adapt to choose `BLOCK` for both phases for not just eight nodes, but also two and four. This leads to a degradation of around 10%.

While choosing the correct distribution in the Flame program led to a modest improvement, it is important to note that it can be much larger. We ran Jacobi iteration for 100 iterations, starting an unrelated competing process (implemented as a tight loop) on half of the nodes. Significant changes had to be made to Adapt to accomodate non-dedicated environments; further details are contained in [14]. Other work for non-dedicated environments that integrates the compiler, run-time system, and OS is described in [16].

We started the competing process on either the 20th, 60th, or 90th iteration. For two, four, and eight nodes, we ran both (1) a SUIF-Adapt version where the decision to redistribute is made dynamically and (2) a comparable hand-coded version that was identical except that it made the *opposite* decision concerning redistribution. For example, if SUIF-Adapt chose to redistribute, then the hand-coded version did not redistribute. Figure 10 shows that due to the accurate measure of redistribution time, SUIF-Adapt in general adjusts the workload only when it is beneficial. When the competing processes start on the 20th iteration, the redistribution time is amortized. This is correctly determined by SUIF-Adapt. If redistribution time is overestimated, as it is if either (1) the graph is not unrolled, or (2) page invalidations are counted as page transfers, SUIF-Adapt would have potentially chosen not to redistribute. On the other hand, SUIF-Adapt does *not* redistribute when it would be harmful; if the competing processes start on the 90th iteration, the redistribution time cannot be amortized. If redistribution time is not considered or is underestimated, a redistribution may occur.

When the competing processes are started on the 60th iteration, the execution times with and without redistribution are extremely close on four and eight nodes; in other words, the time saved by data redistribution on future iterations is almost equal to the cost of redistribution. SUIF-Adapt chose the correct distribution on four nodes; however, on eight nodes, it chose to redistribute when it would have been better to avoid redistribution. However, the difference between the two versions, excluding the SUIF-Adapt analysis time, is around 2%, which is the closest between the versions in all of our tests.

While SUIF-Adapt typically makes the proper decision when redistributing data, the improvement in execution time in our version of Jacobi iteration is relatively small (around 10%). However, the improvement grows with the computation to communication ratio. This ratio can increase in several ways, such as an increase in problem size or when the amount computation between synchronization points increases. For example, the latter occurs in Jacobi if we use a 9-point stencil instead of a 4-point one. Figure 11 shows the savings per iteration for different computation to communication ratios.

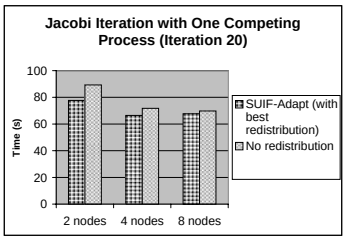
5. DISCUSSION

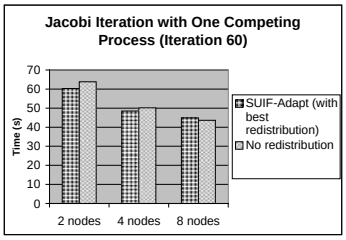
Our work shows that it is possible to accurately estimate redistribution costs in write-invalidate SDSM systems. This section first discusses applying our analysis to programs with irregular communication patterns. Next, we discuss whether or not write-shared protocols such as Munin [2] and Treadmarks [9] could be used.

Irregular Applications

Our current framework supports applications with a regular structure. We utilize this assumption by restricting our attention to `BLOCK`- and `CYCLIC`-based distributions as well as efficiently estimating alternative distributions without actually executing the program in those distributions.

Applications with arbitrary access patterns pose a rather





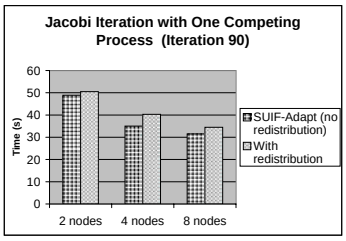


Figure 10: Jacobi iteration with one competing process. In this experiment, we started executing Jacobi iteration as the only application process on the machine. At the iteration specified (20, 60, or 90), we introduced a competing process (on half the nodes) that repeatedly executes a tight loop. This figure shows that SUIF-Adapt almost always makes the correct decision on whether to redistribute; when it does not, the difference is almost negligible. Times are in seconds; note that the SUIF-Adapt times include redistribution analysis.

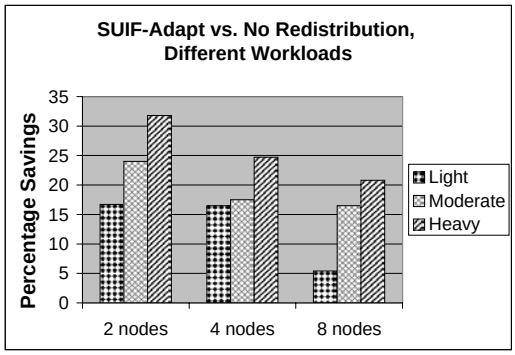


Figure 11: Percentage savings per iteration for Jacobi with SUIF-Adapt when there are competing processes. The benefit of the redistribution selected by SUIF-Adapt compared to a hand-coded version (without redistribution) is shown for two, four, and eight nodes. For each, we tested a light, medium, and heavy computational load. The amount of sequential computation for each is 0.75 seconds, 1.44 seconds, and 3.51 seconds, respectively. The improvement per iteration when using SUIF-Adapt grows with the amount of computation.

difficult problem; one way to handle such programs would be to update an auxiliary data structure to keep track of which elements are accessed. This would entail a great deal of overhead. However, many irregular scientific applications use indirection arrays, where the array subscript is itself an array reference. In this case the indirection array is often a regular section. Support for aggregating communication for these kind of applications through compile- and run-time analysis was presented in [15]. Our goal is to determine efficient data distributions. For applications where a `BLOCK` or `VARBLOCK` distribution is called for, such as some versions of non-bonded force interactions [15], our analysis could be applied. We would need to inspect the indirection array and write a bit vector corresponding to its values; we could then determine the accessed pages.

Write-Shared Protocols

The write-invalidate page consistency protocol (PCP) is insufficient when there are multiple nodes that access a single page, and at least one of them writes. This is known as false sharing, which usually leads to thrashing. Write-shared PCPs tolerate false sharing by *cloning* a page that is being falsely shared; at a later *acquire* point, a single consistent copy of the page is obtained.

To support write-shared protocols, our DRSDs could be used to determine not only which pages are falsely shared, but also which locations within those pages are written. Then, the DFA describing page states would need to be modified to describe write-shared behavior. For example, writing to a read-shared page no longer causes an invalidation. The exact DFA modifications depend on the write-shared

update strategy [7]. We will discuss three such strategies: eager, lazy, and home based.

The easiest version of write-shared to support would be home based, because it is conceptually the simplest. Using the DRSDs, SUIF-Adapt could know which pages are sent to the home and run isolated experiments to determine the cost of the diff (based on its size) performed at the home. An eager scheme is more complicated than a home-based one, but as long as each node exchanges its diffs with all other nodes in its copyset, a DFA can be developed that could capture communication. The lazy scheme would be most difficult to support because protocol actions span phases; write sharing in phase i will cause *no* action; if the page is read in phase $i + 1$, then diffs are accumulated and a consistent copy is obtained. This would cause a circularity, because the distribution chosen greedily by SUIF-Adapt in phase $i+1$ depends on that chosen in phase i. Furthermore, periodic garbage collection may occur with the lazy scheme; this complicates the matter further.

The above discussion assumed that barriers are the only form of synchronization. In some cases, locks give our approach problems, because we cannot determine in what order processes will acquire and release them. This presents a problem in the case that the amount of communication incurred by processes depends on the order of lock acquisition; this is not common, but possible. However, if reductions are used instead, communication only occurs at synchronization points, which makes our analysis possible.

6. CONCLUSION

This paper has described our approach to estimating data redistribution cost accurately in software distributed shared memory (SDSM) systems. We have implemented the approach in SUIF-Adapt. We extended the compiler to generate deferred regular section descriptors (DRSDs). We also modified the run-time system to use the DRSDs along with the knowledge of consistency protocol actions to develop an algorithm to accurately determine data redistribution time. This allows us to in turn accurately estimate total phase time. Through optimizations, the performance of our algorithm was made quite acceptable.

Performance results showed that our estimate of total phase time are within 5% of the actual time. This accuracy allowed us to choose effective global data distributions. In particular, the distributions chosen by SUIF-Adapt lead to a performance benefit of over 10% in some of our tests. Furthermore, in general the benefit can be much more.

7. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[5] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[6] G. M. Howard and D. K. Lowenthal. An integrated compiler/run-time system for global data distribution in distributed shared memory systems. In *Second Workshop on Software DSM*, May 2000.

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

[8] S. Ioannidis and S. Dwarkadas. Compiler and run-time support for adaptive load balancing in software distributed shared memory systems. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-Time Systems for Parallel Computing*, pages 107–122, May 1998.

[9] 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.

[10] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM TOPLAS*, 20(4):869–916, 1998.

[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 and G. R. Andrews. An adaptive approach to data placement. In *Proceedings of the 10th International Symposium on Parallel Processing*, pages 349–353, Apr. 1996.

[13] 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.

[14] D. K. Lowenthal and F. Lowenthal. Supporting regular data distributions on nondedicated parallel machines (submitted to Supercomputing '01). May 2001.

[15] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and distributed shared memory support for irregular applications. In *Sixth Symposium on Principles and Practice of Parallel Programming*, pages 48–56, June 1997.

[16] U. Rencuzogullari and S. Dwarkadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In *Eighth Conference on Principles and Practice of Parallel Programming (to appear)*, June 2001.

[17] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, Jan. 1993.

[18] 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.