Optimizing Irregular Shared-Memory Applications for Clusters [*]

Seung-Jai Min
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907
smin@purdue.edu

Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907
eigenman@purdue.edu

ABSTRACT

Irregular applications pose challenges in optimizing communication, due to the difficulty of analyzing irregular data accesses accurately and efficiently. This challenge is especially big when translating irregular shared-memory applications to message-passing form for clusters. The lack of effective irregular data analysis in the translation system results in unnecessary or redundant communication, which limits application scalability.

In this paper, we present a Lean Distributed Shared Memory (LDSM) system, which features a fast and accurate irregular data access (IDA) analysis. The analysis uses a *region-based diff method* and makes use of a runtime library that is optimized for irregular applications. We describe three optimizations that improve the LDSM system performance. A *parallel array reduction transformation* reduces overheads in the analysis. A *packed communication optimization* and a *differential communication optimization* effectively eliminate unnecessary and redundant messages. We evaluate the performance of the optimized LDSM system on a set of representative irregular benchmarks. The optimized LDSM executes irregular applications on average 45% faster than the hand-tuned MPI applications.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: [Compilers, Run-time environments]

General Terms

Languages, Performance

Keywords

Compiler Analysis, Runtime Techniques, OpenMP, MPI, Irregular Data Accesses, Performance

[*]This work was supported, in part, by the National Science Foundation under grants No. 0429535-CCF, 0650016-CNS, and CNS-0751153.

```
#pragma omp parallel for private(i, j)
for (i=0; i<N; i++) {
   j = C[i];
   <...> = A[B[j]];
}
```

(a) Multiple levels of indirections

```
#pragma omp parallel for private(i, j, cond)
for (i=0; i<NROWS; i++) {
   for (j=row[i]; j<row[i+1]; j++) {
      if (cond) {
         A[i] = <...>;
      }
      cond = <...>;
   }
}
```

(b) Irregular data access due to control flow

Figure 1: Irregular Data Access Examples

1. INTRODUCTION

The ultimate goal behind the presented work is to extend the ease of shared memory parallel programming to distributed memory platforms, such as clusters [21, 10, 4, 2, 17, 12]. In doing so, we face the big challenge of developing techniques that deal with irregular data accesses. For compilers, such techniques are difficult because they involve the analysis of complex program expressions (such as non-affine subscripts) and control flow; also, there is often insufficient static information.

Figure 1 shows two irregular access patterns commonly found in scientific and engineering applications. The first example has an irregular access due to the multiple levels of indirection arrays, `B` and `C`; in the second example, the shared array `A` is irregularly written because the control flow changes depending on the value of `cond` and `row`, which may not be *loop invariants.* In the presence of irregular data accesses, a compiler is forced to generate messages in a conservative way, which often leads to unnecessary or redundant communication. To address this problem, there have been numerous efforts to optimize irregular applications on clusters [4, 5, 13, 20, 25, 16]. Table 1 summarizes these approaches.

The direct MPI method [4] translates OpenMP [22] shared memory programs into MPI [11] programs. Software distributed shared memory (DSM) systems [2] are runtime systems that provide a shared address space abstraction on a cluster platform. Combined compile-time/runtime approaches integrate compiler support with runtime function-

Table 1: Research on Optimizing Irregular Shared Memory Programs for Clusters (*LDSM is our approach proposed in this paper)

	Implementation Examples	Irregular Data Access Analysis
Direct MPI	OpenMP to MPI [4]	Monotonicity
Software DSM	TreadMarks [16]	Page-based Diff
Compiler + Runtime Libraries	CHAOS [13, 20, 25]	Inspector/Executor
	***LDSM**	Region-based Diff

ality. Our approach falls in this category. A related system is the CHAOS [24] runtime library, which supports data distribution and communication schedule generation for irregular applications. Several efforts have optimized irregular applications using CHAOS [13, 20, 25]. Our approach is named Lean Distributed Shared Memory (LDSM). It borrows concepts from software DSMs; however it uses a thin runtime library layer only and is tightly integrated with the compiler.

A key functionality in our approach is the analysis of irregular data accesses (IDA). Both compile-time and runtime IDA analysis techniques have been proposed by others. In the direct MPI approach [4], a compile-time technique uses monotonicity properties of the involved indirection arrays to refine the compiler analysis. With monotonicity, the message size can be reduced if the irregular data accesses are clustered, not sparse. The disadvantage of this technique is the limited applicability that not all indirection arrays can be proved monotonic by the compiler.

The inspector/executor model [15, 19] is a well known runtime technique. It analyzes indirect array accesses, such as `A[B[i]]`. An inspector loop is inserted into the code before the original loop, where the irregular accesses occur. The inspector loop inspects the value of indirection array elements and finds a set of array indices that will incur non-local memory references [1, 27].

Programming in the inspector/executor model is complex, so is the compiler analysis to automate this process [1, 9, 28]. This is especially true in the presence of multiple levels of indirections [9], complex control flow, or pointer expressions [3].

Page-based diff is an alternative runtime analysis technique, used in page-based software DSM systems [2]. The software DSM system maintains memory consistency. On the first write access to a page, the content of a page is copied to a *twin page*. After the write accesses are complete, the original page is compared with the twin to detect the modified memory ranges. The page-based diff method can precisely detect shared writes without compiler support. A disadvantage of software DSMs is that they may generate excess communication due to false sharing. False sharing is caused by the large memory coherence unit, which is a page. Another disadvantage of the page-based diff method is the memory overhead for allocating twin pages. This memory overhead can be especially large when irregular data accesses are sparse. IDA analysis of our LDSM system is a region-based method, where the data regions are identified by the compiler. In contrast to page-based diff methods, regions can have arbitrary sizes; false sharing can be eliminated.

Both CHAOS and LDSM include compiler as well as runtime support. However, there are differences between these two approaches. First, CHAOS supports programs in High Performance Fortran (HPF), where data distributions are specified in the program. The inspector/executor model is used in CHAOS for IDA analysis. By contrast, LDSM uses plain OpenMP programs, and it performs region-based IDA analysis. Second, the compiler support for the inspector/executor model in CHAOS is to instrument the program with inspector loops. By contrast, compiler support in LDSM performs advanced analysis of data regions, the results of which are then passed on to the runtime system. Third, the inspector/executor model uses inspector loops to *pre-inspect* the original loops. By contrast, our LDSM system performs *post-inspection* after the original loop, based on diff mechanism. The advantage of post-inspection is that the analysis is not hindered by complex program patterns, such as multiple indirections or involved control flow, which pose significant challenges for the inspector/executor model.

LDSM uses the proposed region-based IDA analysis for analyzing irregular access patterns in the applications. To improve the performance of LDSM, we present three optimization techniques: An overhead reduction technique for IDA analysis and two communication optimization techniques. The first technique takes advantage of array reduction patterns, which are frequently found in the irregular programs. Array reductions can be transformed in a way that allows the LDSM system to eliminate twin regions (reducing memory overhead) and to take advantage of the precise inspection mechanism for write accesses (reducing message overhead). The second optimization technique reduces communication cost by packing multiple data regions accessed by a sparse computation into a single message. The third optimization is motivated by the observation that similar contents may appear in messages. In such a case, sending only the difference instead of the whole message is more efficient.

We have implemented the proposed optimization techniques in LDSM. The LDSM functions, a set of runtime libraries, are written in C with MPI library calls for communication, which produces a portable implementation. We evaluate our three optimization techniques using five irregular test programs - CG and IS from the NAS Parallel Benchmarks (NPB), EQUAKE from the SPEC OMPM2001 suite, SPMUL, and MOLDYN.

This paper makes the following contributions:

- We propose a new IDA analysis technique, a region-based IDA analysis, which consists of compile-time analysis and runtime inspection.
- We present a parallel array reduction transformation technique that reduces overheads in the IDA analysis.
- We present two optimization techniques - packed communication and differential communication - to reduce communication.
- We evaluate the performance of five representative irregular applications. The proposed optimization techniques improve the performance by 45%, on average, over the hand-tuned MPI versions on 32 processors.

The remainder of the paper is organized as follows. Section 2 describes our region-based IDA analysis and LDSM functions. Section 3 presents techniques to reduce overheads in the IDA analysis. Section 4 shows two communication optimization techniques. Section 5 presents performance evaluation results on five irregular benchmarks. Section 6 discusses additional related work, followed by conclusions in Section 7.

Table 2: The LDSM functions

```
/* The LDSM interface function for the compiler to inform */
/* compile-time analysis results to LDSM */
void ldsm_compiler(id, range_list);

/* The LDSM communication function for barrier operation */
void ldsm_barrier(id)

/* The LDSM communication function for allreduce operation */
void ldsm_allreduce(id)

/* The LDSM communication function for allgather operation */
void ldsm_allgather(id)
```

2. IRREGULAR DATA ACCESS (IDA) ANALYSIS

Our IDA analysis is a combined compile-time/runtime technique. The compiler instruments the code to pass its analysis results to LDSM; LDSM performs the runtime inspection within the compiler-defined regions. This process is accomplished through a set of the LDSM functions, which are inserted by the compiler or application programmer into the program. Table 2 describes a partial list of the LDSM functions.

The `ldsm_compiler` function connects the compiler and the LDSM system. The communication functions include `ldsm_barrier`, `ldsm_allreduce`, and `ldsm_allgather`; they perform runtime IDA analysis, create a communication schedule, and perform communications. In `ldsm_compiler`, the first parameter, `id`, is used to identify the matching communication function in the program. The second parameter `range_list` is a list of compiler-defined memory regions. It is represented as a list of triplets, such as ($saddr_1$, $eaddr_1$, pid_1), ($saddr_2$, $eaddr_2$, pid_2), ..., where *saddr* and *eaddr* are the start address and the end address of the memory region that will be read accessed by the remote processor, *pid*. `ldsm_allreduce` and `ldsm_allgather` are inserted when the compiler recognizes collective communication patterns, such as allreduce or allgather. Recognition of collective communication patterns is described in Section 3.2.

```
#pragma omp parallel for private(i)
for (i=0; i<N1; i++) {
   A[B[[i]] = <...>;
}
#pragma omp parallel for private(i)
for (i=0; i<N2; i++) {
   <...> = A[i];
}
```

(a) An example of input OpenMP code

```
ldsm_compiler(1, range_list);
for (i=lb1; i<ub1; i++) {
   A[B[i]] = <...>;
}
ldsm_barrier(1);

for (i=lb2; i<ub2; i++) {
   <...> = A[i];
}
```

(b) Translated LDSM code

Figure 2: IDA Analysis Example

```
#pragma omp parallel private(lsum)
{  lsum =0;
   #pragma omp for
   for (i=0; i<N; i++) {
      lsum += <...>;
   }
   #pragma omp critical
   sum += lsum;
}
```

(a) Type 1: a transformed parallel reduction idiom

```
#pragma omp for private(ii, i, j, rd)
for (ii=0; ii<ninter; ii++) {
   i = inter[ii, 0];
   j = inter[ii, 1];
   rd = compute_distance(x[i], x[j]);
   if (rd < cutoffSquare) {
      f[i] += <...>;
      f[j] -= <...>;
   }
}
```

(b) Type 2: sequential reduction in a parallel loop

Figure 3: Reduction patterns in OpenMP

Figure 2 shows a simple example code and its translated version to demonstrate how the LDSM functions are used in our IDA analysis. The code in Figure 2 (a) has a parallel loop with irregular write accesses followed by a parallel loop with regular accesses. The translated code in Figure 2 (b) has two LDSM function calls, `ldsm_compiler` and `ldsm_barrier`.

For explanation purpose, we assume that the example code is executed on two processors, processor 0 and processor 1. Assuming the parallel loops in the translated code are block partitioned and the upper bound of the second parallel loop, `N2`, is 2000, the new loop bounds of the second parallel loop, `lb2` and `ub2`, will be (0, 1000) for processor 0 and (1000, 2000) for processor 1. To instruct LDSM which memory regions to inspect, `range_list` in `ldsm_compiler` will contain (&A[1000], &A[1999], 1) for processor 0 and (&A[0], &A[999], 0) for processor 1. When the program execution reaches `ldsm_compiler`, the runtime system allocates *twin* data structures of the same size as the original regions specified in the `range_list`; it then copies the contents of the original memory regions to the corresponding twins. After the irregular writes are executed in the first parallel loop, `ldsm_barrier` is invoked to compute the difference between the twins and the original regions. If the computation yields multiple disjoint data ranges, they are packed into a single message and sent to the receiving processor.

For irregular read accesses, the IDA analysis is performed by the compiler only. In some cases, the compiler may need to overestimate the accessed regions in a conservative way.

3. IRREGULAR DATA ACCESS ANALYSIS OVERHEAD REDUCTION TECHNIQUE

As described in the previous section, the proposed IDA analysis has an advantage in analyzing irregular write accesses, because our IDA analysis is precise on irregular writes, whereas irregular reads can be overestimated. Since overestimated analysis for irregular reads can incur unnecessary communication overhead, it is beneficial to convert remote read accesses into remote write accesses, whenever applicable. The parallel array reduction transformation (PART)

```
#pragma omp parallel for private(j, k)
for (j=0; j<NROWS; j++) {
  y[j] = 0.0;
  for (k=row[j]; k<row[j+1]; k++) {
    y[j] = y[j] + A[k]*x[col[k]];
  }
}
```

(a) Original OpenMP code

```
MPI_Allgather(x, size, displ, ...);
for (j=lb; j<ub; j++) {
  y[j] = 0.0;
  for (k=row[j]; k<row[j+1]; k++) {
    y[j] = y[j] + A[k]*x[col[k]];
  }
}
```

(b) MPI version using MPI_Allgather

```
ldsm_compiler(id, range_list);
for (j=0; j<NROWS; j++) {
  l_y[j] = 0.0;
  for (k=row[j]; k<row[j+1]; k++) {
    if (lb <= col[k] && ub >= col[k]) {
      l_y[j] = l_y[j] + A[k]*x[col[k]];
    }
  }
}
ldsm_allreduce(id);
```

(c) Baseline version of the parallel array reduction transformed code

```
/* d_A, d_row, and d_col are the distributed
 * versions of A, row, and col, respectively. */
ldsm_compiler(id, range_list);
for (j=0; j<NROWS; j++) {
  l_y[j] = 0.0;
  for (k=d_row[j]; k<d_row[j+1]; k++) {
    l_y[j] = l_y[j] + d_A[k]*x[d_col[k]];
  }
}
ldsm_allreduce(id);
```

(d) Optimized version of the parallel array reduction transformed code

Figure 4: Three SMVP translations: an MPI version of SMVP code and two different versions of the parallel array reduction transformed SMVP codes

achieves this goal by rearranging data accesses, so that remote irregular reads are replaced by local accesses. Instead, remote irregular writes are introduced. PART also reduces the memory overhead by enabling our IDA analysis to perform its diff operation without twin memory. The following subsections provide details of the PART technique.

3.1 Array Reduction Recognition

In OpenMP, reduction computations can be expressed with the reduction clause, `omp reduction`. However, the reduction clause supports only scalar reductions, not array reductions. Array reductions are often implemented in the form of transformed parallel reduction idioms using an OpenMP parallel loop, followed by an `omp atomic` statement or an `omp critical` section that updates global reduction memory. Converting reduction idioms to the transformed parallel reduction idioms has been discussed in the context of parallelizing compilers [23].

Figure 3 (a) shows the transformed parallel reduction idiom, which we will call `type 1` reduction. In our benchmarks, reductions expressed in `type 1` form operate on either scalar variables or small arrays. Hence, the effect on the total execution time is minor.

Most of the significant array reductions used in the NPB3.0-OMP and the SPEC OMP2001 benchmarks are of `type 2`, where the reduction computations are performed sequentially in each processor, as shown in Figure 3 (b). Our compiler recognizes `type 2` sequential reductions by identifying the commutative associative operators in a parallel loop and converts them into a parallel reduction form, as follows.

3.2 Parallel Array Reduction Transformation

The parallel array reduction transformation (PART) technique translates the program section with the sequential irregular array reduction into the parallel irregular array reduction form with the LDSM functions.

If the reduction has irregular reads due to the indirection arrays, PART converts remote reads into local reads by distributing the indirection arrays. Instead, PART introduces remote write accesses, because the local reduction array needs to be communicated to produce the global reduction results, which is accomplished by the `ldsm_allreduce` function, after the reduction loop. The `ldsm_compiler` function is inserted before the array reduction loop to inform the LDSM system of the local reduction array, which is used for accumulating partial reduction results.

The PART technique provides an opportunity to reduce overheads incurred by LDSM's twin mechanism. Parallel array reductions make use of an array data structure to hold partial reduction result on each processor. For `sum` reductions, this array is initialized to zero, making it easier for our IDA analysis to detect modified elements. When it is known to the LDSM system what data value the reduction array will be initialized, such as zero in sum reduction and one in product reduction, it is unnecessary to allocate a twin memory for diff operation. Twin-less diff in IDA analysis for irregular array reductions brings two advantages; it not only decreases the memory usage, but also reduces the runtime overhead of the diff computation, because performing the diff operation on the reduction array against a known constant, such as zero, is faster than performing diff comparison between the reduction array and its twin array.

In Figure 4, Sparse Matrix Vector Multiply (SMVP) is used as an example to illustrate the PART process. SMVP is often expressed in the form of $y \leftarrow A{\cdot}x$, where A is a sparse input matrix and x, y are dense column vectors, called the input and output arrays. Most of the sparse input matrices are so large that they are usually stored in a compressed format. The `row` and `col` arrays are used to locate the position of nonzero elements before compression. The original OpenMP code in Figure 4 (a) has an irregular read access to input array `x` due to the multiple levels of indirection arrays, `col` and `row`. Therefore, each processor needs to broadcast its portion of array `x` to other processors before the irregular read access happens. An MPI programmer can recognize this collective communication pattern and optimize this task by using `MPI_Allgather` collective communication routine. The MPI version of this example is shown in Figure 4 (b).

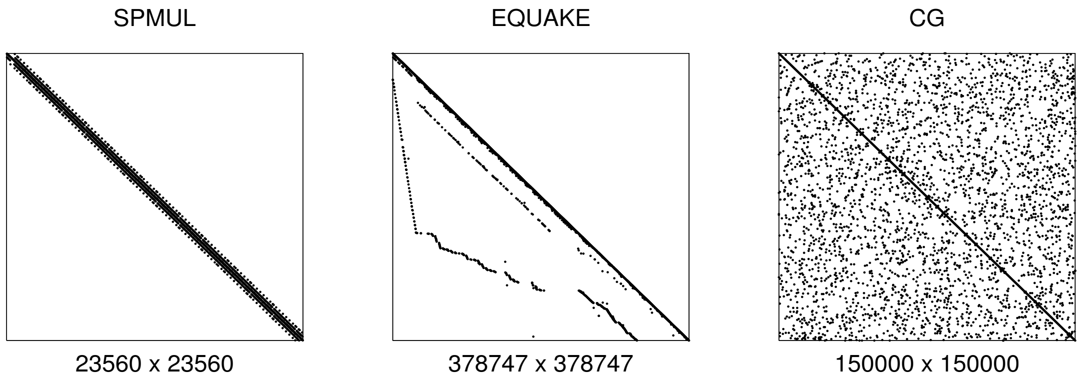

Figure 5: Nonzero Distribution of Sparse Input Matrix in SPMUL, EQUAKE, and CG: the percentage of nonzero elements is less than 1% in all three sparse input matrices. In SPMUL and EQUAKE, the output of the sparse matrix-vector multiplication in each processor is also sparse, however, CG produces a dense output due to its random nonzero distributions in the input matrix.

However, the original OpenMP code has a `type 2` reduction, which can be translated into the parallel reduction form. Figure 4 (c) and (d) show two different versions of the transformed reductions, the baseline version and the optimized version, respectively. The baseline version eliminates the non-local reads by inserting a conditional statement to selectively access only the local elements of input array `x`. However, the conditional statement within a loop can cause performance degradation. The optimized version removes the non-local reads without the conditional execution; it distributes the sparse input matrix `A` and generates the distributed version of `row` array and `col` array accordingly, so that the irregular read access to the array `x` is always local. The `ldsm_allreduce` function performs the IDA analysis to the local reduction array, `l_y`, and communicates the modified ranges within `l_y` to obtain the global reduction output.

We have evaluated representative irregular applications from various fields. They include a sparse-matrix vector multiplication, an integer sorting algorithm, and a molecular dynamics simulation. We found that most irregular array accesses in these benchmarks have a `type 2` reduction; PART can be applied to a broad range of irregular applications.

4. COMMUNICATION OPTIMIZATION TECHNIQUES

We present two communication optimization techniques to eliminate redundant and unnecessary communication and improve the communication performance of irregular applications on distributed memory systems.

4.1 Packed Communication Optimization

Irregular accesses often touch only a small fraction of the data in a given range. In such cases, sending the whole ranges will consume excessive bandwidth. *Packed Communication Optimization* (PCO) uses region-based IDA analysis for detecting sparsely modified data ranges and packs them into a single message. For sparse irregular applications, PCO considerably reduces message sizes. However, PCO comes with the overheads of packing/unpacking data ranges and extra bookkeeping. To predict the profitability of PCO, we measure *total_count* (the total size of the modified data ranges) and *range_count* (the number of modified data ranges). Each modified data range needs an offset and a size information to represent the original memory location before packing. We apply PCO when the new message size, $total\_count + 2 \times range\_count$, is less than 10% of the original message size. For example, sparse matrix-vector multiply (SMVP) is an important class of sparse irregular applications. Once the array reduction of the SMVP computation is transformed into the parallel reduction form, as described in Section 3, the local reduction array on each processor will be communicated to perform the global reduction.

Figure 5 illustrates the distribution of nonzero elements of sparse input matrices used in SPMUL, EQUAKE, and CG, where nonzero elements are shown in black dots. The percentage of nonzero elements in SPMUL, EQUAKE, and CG is 0.09%, 0.004%, and 0.16%, respectively. In SPMUL and EQUAKE, the SMVP computations with such sparse input matrices produce sparse local reduction arrays. However, CG outputs a dense local reduction array, because of its random nonzero distribution pattern in the sparse input matrix. Among the irregular benchmarks, the profitability test shows that SPMUL and EQUAKE are good candidates for the PCO technique. However, PCO is not applied to CG, IS, and MOLDYN, because these benchmarks do not exhibit enough sparsity in the communication data to pass the profitability test.

4.2 Differential Communication Optimization

In some iterative applications, the contents of messages do not change significantly from one iteration to the next. Sending differential messages can be more efficient than sending the whole data. In this section, we introduce our novel communication optimization technique, *Differential Communication Optimization* (DCO), which effectively removes redundant data in messages.

DCO computes the difference between the current data and the previously sent data. Instead of the original message, the difference data are packed and sent to the receiving processor. The receiving processor reconstructs the original message by adding the received difference data to the previously reconstructed message. The DCO technique is applied selectively, because computing the difference data takes time and memory. The profitability test for DCO is similar to the

Table 3: Average data change rate of `inter` for different update periods in MOLDYN

Update Period	1	5	10	20
Avg. Data Change Rate	0.22%	0.95%	1.68%	2.15%

one used in PCO, except that the test is applied to the difference data, instead of the original communication data.

For example, the MOLDYN benchmark simulates the interaction between the particles that are within the cutoff distance from each other. The connectivity information contains whether the two particles are within the cutoff distance or not, which is stored in the array, `inter`. The `inter` array is periodically updated in parallel and is communicated by the participating processors to build the global connectivity information. When communicating the `inter` array between processors, we measured how much of the `inter` array has been changed from previous communication. In Table 3, *Update Period* of *N* indicates that `inter` is updated after every `N` iterations and *Data Change Rate* denotes the difference between the current `inter` data and the previously sent `inter` data. This experiment shows that only a small fraction of `inter` is changed and the data change rate is smaller, as the update is more frequent. According to this, DCO will greatly reduce communication in MOLDYN.

5. PERFORMANCE EVALUATION

We evaluate the performance of three LDSM optimization techniques – PART, PCO, and DCO. We compare the performance of the irregular applications optimized by the LDSM system with the performance of their hand-tuned MPI counterparts.

5.1 Experimental Setup

We selected five representative irregular applications, which consist of three applications that contain sparse matrix-vector multiplications (SPMUL, EQUAKE, and CG), the IS integer sorting application, and the MOLDYN molecular dynamics simulation. CG and IS are from the NAS Parallel NPB3.0-OMP benchmark suite; we used the *Class C* data set. EQUAKE is from the SPEC OMPM2001 suite and is evaluated with the *ref* data set. The hand-tuned MPI versions for the CG and IS benchmarks are part of the NPB2.4 suite. The SPMUL, EQUAKE, and MOLDYN do not have existing MPI versions. For applications without MPI versions, we created the hand-tuned MPI versions from the OpenMP versions of these applications and we tried to express the available parallelism as best as we could. The level of optimizations we applied to these hand-tuned MPI codes are similar to the level of optimizations used in NPB2.4-MPI applications, except that we didn't change the algorithm, while some NPB2.4-MPI versions do. The time and effort to create these MPI versions were greater than the programming effort to create their OpenMP versions.

The performance is shown in terms of execution time and scalability. We define scalability as T_{serial}/T_n where T_n is the execution time of the benchmark on n processors and T_{serial} is the execution time of the corresponding serial version. We performed our experiment on a cluster of thirty-two P4 3.2 GHz Linux nodes, with 4 GB memory per node,

Table 4: Runtime overhead of the LDSM system on 32 processors

Benchmarks	Total Exec. Time	Runtime Overhead	Percentage
SPMUL	0.42 sec	2.75 msec	0.65 %
EQUAKE	116.59 sec	40.64 msec	0.04 %
CG	30.69 sec	8.44 msec	0.03%
IS	4.11 sec	354.14 msec	8.62 %
MOLDYN	12.11 sec	1667.61 msec	13.77 %

connected with InfiniBand. The MPI library used for these platforms is MPICH2 on a Linux platform and the back-end compiler is the Intel *icc* compiler version 9.0.32 with optimization level O3.

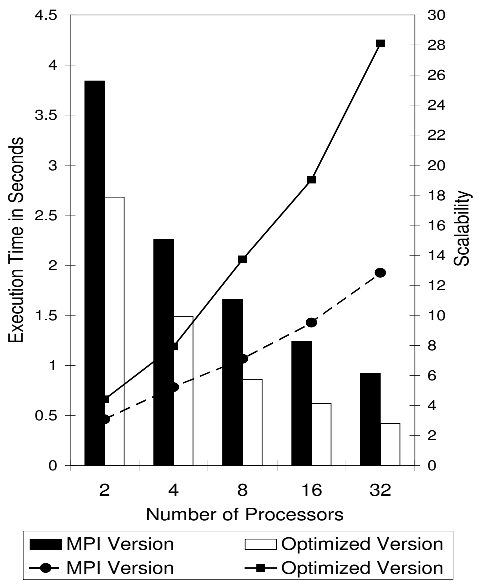

Figure 6: SPMUL Performance

5.2 Runtime Overhead of LDSM System

We measured the total runtime overhead of our LDSM system, such as the runtime inspection time of the region-based IDA analysis, packing and unpacking the modified ranges for PCO, and computing difference data for DCO. The overhead does not include inter-processor communication time. Table 4 shows the runtime overhead of LDSM system for irregular applications running on 32 processor nodes. SPMUL, EQUAKE, and CG are static irregular applications that exhibit highly repetitive communication patterns. The compile-time analysis to detect repetitive communication patterns in the context of a software DSM has been presented [18]. We have applied this technique to the LDSM system. For applications with repetitive communication patterns, LDSM performs IDA analysis only in the first iteration; the created communication schedule is reused for the remaining iterations. IS and MOLDYN are adaptive irregular applications where the communication patterns change over time. These applications show higher overhead than static irregular applications, because the communication schedules are calculated frequently.

5.3 SPMUL

SPMUL is a kernel benchmark that performs sparse matrix-vector multiplications. The sparse input is a 23560×23560 banded diagonal matrix with less than 0.1% nonzero elements. Due to the sparsity of its input matrix, the performance of SPMUL is mainly enhanced by PCO, and the banded diagonal structure of the input matrix causes each processor to communicate with at most two of its neighbors. In Figure 6, the MPI version uses `MPI_Allgather` collective communication; the optimized version is translated for parallel reductions, followed by PCO. Compared to the hand-tuned MPI version, the optimized version performs 120% faster and achieves 118% higher scalability.

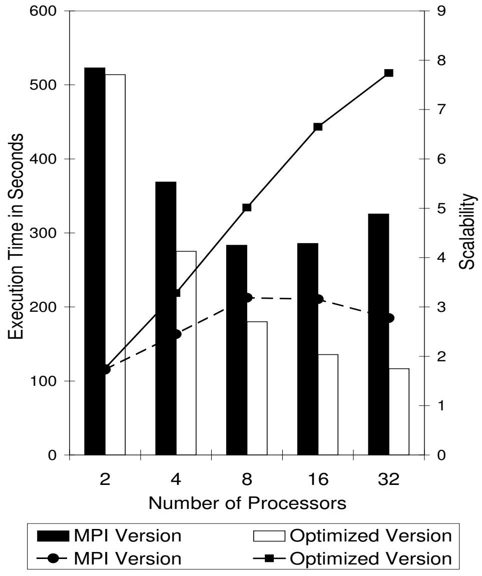

Figure 7: EQUAKE Performance

5.4 EQUAKE

EQUAKE is an application from the SPEC OMPM2001 benchmark suite. It simulates seismic wave propagations in earthquakes. The most time-consuming part of EQUAKE is the *smvp* subroutine, which computes the product of a sparse matrix and a vector. The sparse input matrix is a symmetric matrix, of which only the lower triangle is stored to reduce the memory space. This memory storage structure makes the *smvp* subroutine complex, which prevents our compiler from applying PART technique to EQUAKE. However, the sparse input matrix has less than 0.01% nonzero elements, which enables our LDSM system to apply PCO technique. The optimized version runs 2.8 times faster than the hand-tuned MPI version.

5.5 CG

CG implements a conjugate-gradient method. A sparse matrix-vector multiplication is the most time-consuming part. Although the sparse input has a low nonzero percentage of 0.16%, PCO is not applied, as it fails the runtime profitability test. For performance evaluation, the hand-tuned MPI version uses the NPB2.4-MPI version (NPB3.0-MPI is the same as NPB2.4-MPI) and both the unoptimized and optimized versions are translated from NPB3.0-OMP source. The optimized version is transformed for parallel array reductions. The hand-tuned NPB2.4-MPI CG version distributes the sparse input matrix using 2-D block partitioning, whereas our translation scheme distributes the input matrix using simple 1-D block partitioning. The 2-D block partitioning of the input matrix has better communication performance, as the number of processors increases. The optimized version is 30% faster than the unoptimized version; however, it is 27% slower than the hand-tuned MPI version, because of the difference in distributing the input matrix among processors. Once we upgrade our input matrix distribution scheme to support 2-D block partitioning, we expect the optimized version to run as fast as the MPI hand-tuned version.

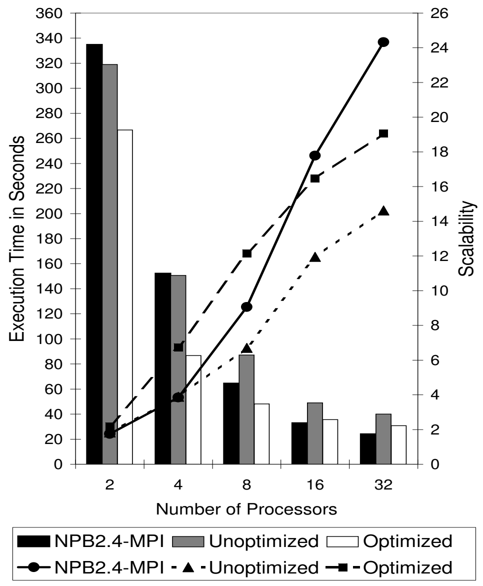

Figure 8: CG Performance

5.6 IS

The IS NAS benchmark performs integer sorting. NPB2.4-MPI IS, the hand-tuned MPI version of IS, uses bucket-sorting with 1024 buckets, whereas the NPB3.0-OMP version of IS performs integer sorting, based on ranking without bucket-sorting. In Figure 9, the single-processor execution time of the MPI version is much faster than our versions, because the different sorting algorithms are used. For the scalability calculation, we used different sorting algorithms to evaluate the serial version of NPB3.0-SER IS. For the hand-tuned MPI version, we evaluated the serial version of IS with the `BUCKET_SORT` option; for our optimized version, we turned off `BUCKET_SORT`.

IS performs integer sorting 10 times with different input integer lists to sort each time. Since the input integer list, `key_array`, is used as an indirection array and updated on every iteration, IS is an adaptive irregular benchmark. IS has an irregular array reduction for ranking integer numbers and PCO is not applied because the local reduction array is not sparse enough to pass the profitability test. However, the data change rate of `key_array` is less than 1%, which makes IS a good candidate for DCO.

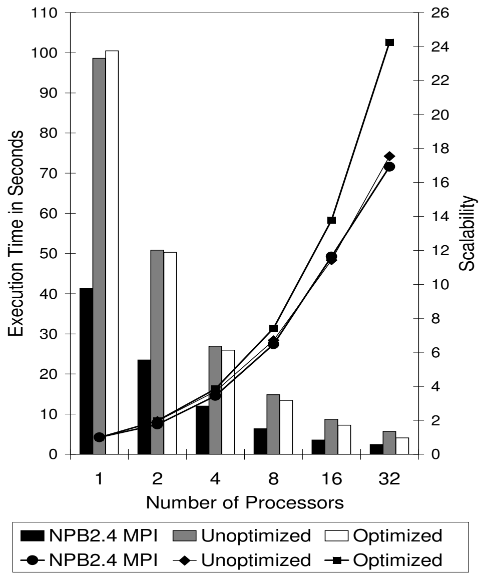

Figure 9: IS Performance

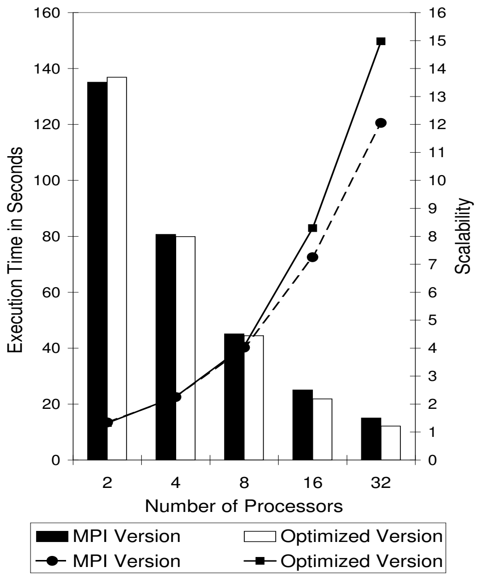

Figure 10: MOLDYN Performance

Figure 9 shows the performance of the IS benchmark. Our optimized version runs 38% faster than the unoptimized version, but runs 68% slower than the hand-tuned MPI version. However, our optimized version achieves higher scalability than the hand-tuned MPI version, if we use the corresponding serial versions as reference points.

5.7 MOLDYN

MOLDYN is a molecular dynamics simulation. Its computational structure resembles the non-bonded force calculation in CHARMM [7]. As explained in section 4.2, DCO is applied but PCO is not, because the nonzero percentage is too high. With transformed parallel reductions, DCO improves the scalability of MOLDYN by 24% over the hand-tuned MPI version.

5.8 Performance Optimization Summary

Table 5 summarizes the optimization techniques used in the irregular benchmarks. The parallel array reduction transformation (PART) is applied to all irregular benchmarks, except EQUAKE. Packed Communication Optimization (PCO) is successfully used in SPMUL and EQUAKE to compress the size of the communication messages. Differential Communication Optimization (DCO) effectively reduces the communication overhead of the IS and MOLDYN benchmarks.

Table 5: Summary of Optimization Techniques

	Parallel Array Reduction Transformation	Packed Communication Optimization	Differential Communication Optimization
SPMUL	Y	Y	N
EQUAKE	N	Y	N
CG	Y	N	N
IS	Y	N	Y
MOLDYN	Y	N	Y

6. RELATED WORK

The key differences between our work and related work have been introduced in Section 1. Here we discuss several additional related contributions.

The Region Trap Library [6] introduced a region-based diff method to reduce false sharing, caused by the page-granularity memory coherence unit. Their work makes use of the page fault mechanism for detecting data accesses to regions. There is no involvement of an operating system in our region-based IDA analysis.

Compiler support for irregular applications on software DSM was proposed in the context of page-based software DSM [16]. Their approach was to use the compiler to identify indirection arrays; the runtime system prefetches the pages that will be accessed through indirection arrays to avoid page fault overhead. Another approach [3] applied an inspector/executor to software DSM. The inspector/executor calculates the list of pages that needs to be prefetched in irregular applications. Both approaches amortize the communication overhead of software DSM by prefetching pages in a single message. Our communication optimizations are performed on data regions, not on fixed-size pages.

The RT-DSM system [29] presented a new write detection method in software DSM, which relies on the compiler and runtime system to detect writes to shared data without invoking an operating system. In their system, each memory store operation is instrumented by the compiler. The disadvantage with this scheme is that the overhead of write detection occurs on every write. Our IDA analysis performs post-inspection, which does not incur overheads on every write access.

The problem of selecting the best communication method for irregular applications was addressed in the context of the Titanium [12] language. This work [26] uses an inspector/executor model to analyze indirect array accesses and implements a packing mode to optimize sparse matrix-vector kernels. Our packed communication optimization takes advantage of the region-based IDA analysis, which is not limited to indirect array accesses.

An MPI message compression technique is proposed to reduce the communication latency [14]. The technique applies compression algorithms to the message content. Our differential communication optimization does not apply compression algorithms. Instead, it sends the difference between the current message and the previous message. To our knowledge, this optimization technique has not been explored before.

Tolerating communication latency using computation and communication overlap has been used before. One approach [5] reorders the loop iterations to increase the overlap in irregular applications. To reorder a loop with irregular accesses, an inspector/executor scheme is applied. Another approach [8] presents a runtime framework that automatically schedules data transfers to achieve overlap in UPC [10]. Overlapping communication with computation is a complementary technique to our communication optimizations, which can further improve the performance of irregular applications.

7. CONCLUSIONS

In this paper, we have presented a Lean Distributed Shared Memory (LDSM) system that effectively executes irregular applications on clusters. LDSM includes two parts, a compile-time analysis and a runtime system. The compiler analyzes data regions to be inspected by the runtime system. The integrated compile-time/runtime mechanism allows the LDSM to be lean, compared to a software DSM system, such as TreadMarks [2].

To improve the performance of LDSM, we have introduced a new IDA analysis and presented three optimization techniques. Our region-based IDA analysis is accurate and efficient in detecting irregular data accesses. The *parallel array reduction transformation* enables our IDA analysis to eliminate the need for twin data regions, saving both memory and execution time. The *packed communication optimization* effectively compresses sparse message data and the *differential communication optimization* reduces the message size by sending only a differential message.

We evaluated the performance of our optimization techniques on LDSM using five irregular benchmarks. The optimized LDSM improves scalability (execution time relative to the serial version) on average by 67% over the hand-tuned MPI programs, on 32 processors.

8. REFERENCES

[1] G. Agrawal and J. Saltz. Interprocedural compilation of irregular applications for distributed memory machines. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 48, New York, NY, USA, 1995. ACM Press.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R.Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, and J. Labarta. Runtime Address Space Computation for SDSM Systems. In *The 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006)*, pages 330–344, 2006.

[4] A. Basumallik and R. Eigenmann. Towards Automatic Translation of OpenMP to MPI. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 189–198, New York, NY, USA, 2005. ACM Press.

[5] A. Basumallik and R. Eigenmann. Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 119–128, New York, NY, USA, 2006. ACM Press.

[6] T. Brecht and H. Sandhu. The Region Trap Library: Handling Traps on Application-Defined Regions of Memory. In *ATEC'99: Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, pages 7–7, Berkeley, CA, USA, 1999. USENIX Association.

[7] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[8] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick. Automatic Nonblocking Communication for Partitioned Global Address Space Programs. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 158–167, New York, NY, USA, 2007. ACM.

[9] R. Das, P. Havlak, J. Saltz, and K. Kennedy. Index Array Flattening through Program Transformation. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 70, New York, NY, USA, 1995. ACM Press.

[10] T. El-Ghazawi, W. Carlson, and J. Draper. UPC Language Specifications, v1.1.1, 2003.

[11] M. P. I. Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, 1994.

[12] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium Language Reference Manual. Technical report, Berkeley, CA, USA, 2001.

[13] Y. Hwang, B. Moon, S. Sharma, R. Das, and J. Saltz. Runtime Support to Parallelize Adaptive Irregular Programs, 1994.

[14] J. Ke, M. Burtscher, and E. Speight. Runtime Compression of MPI Messages to Improve the Performance and Scalability of Parallel Applications. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 59, Washington, DC, USA, 2004. IEEE Computer Society.

[15] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):440–451, 1991.

[16] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and Software Distributed Shared Memory Support for Irregular Applications. In *PPOPP '97: Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–56, New York, NY, USA, 1997. ACM Press.

[17] S.-J. Min, A. Basumallik, and R. Eigenmann. Optimizing OpenMP programs on Software Distributed Shared Memory Systems. *Int. J. Parallel Program.*, 31(3):225–249, 2003.

[18] S.-J. Min and R. Eigenmann. Combined Compile-Time and Runtime-Driven, Pro-active Data Movement in Software DSM Systems. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–6, New York, NY, USA, 2004. ACM Press.

[19] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of Runtime Support for Parallel Processors. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 140–152, New York, NY, USA, 1988. ACM Press.

[20] B. Moon, M. Uysal, and J. Saltz. Index Translation Schemes for Adaptive Computations on Distributed Memory Multicomputers. Technical Report CS-TR-3428, 1995.

[21] R. W. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[22] OpenMP Forum. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. Technical report, October 1997.

[23] B. Pottenger and R. Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 444–448, New York, NY, USA, 1995. ACM.

[24] J. Saltz, R. Ponnusamy, S. D. Sharma, B. Moon, Y.-S. Hwang, M. Uysal, and R. Das. A Manual for the CHAOS Runtime Library. Technical Report CS-TR-3437, 1995.

[25] S. D. Sharma, R. Ponnusamy, B. Moon, Y. S. Hwang, R. Das, and J. Saltz. Run-time and Compile-Time Support for Adaptive Irregular Problems. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 97–106, New York, NY, USA, 1994. ACM.

[26] J. Su and K. Yelick. Automatic Support for Irregular Computations in a High-Level Language. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 53.2, Washington, DC, USA, 2005. IEEE Computer Society.

[27] M. Ujaldon, S. D. Sharma, J. H. Saltz, and E. L. Zapata. Run-Time Techniques for Parallelizing Sparse Matrix Problems. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 43–57, 1995.

[28] R. von Hanxleden and K. Kennedy. GIVE-N-TAKE — a balanced code placement framework. *SIGPLAN Not.*, 29(6):107–120, 1994.

[29] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for a Distributed Shared Memory. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 8, Berkeley, CA, USA, 1994. USENIX Association.