Adaptive Runtime Tuning of Parallel Sparse Matrix-Vector Multiplication on Distributed Memory Systems

Seyong Lee and Rudolf Eigenmann
School of ECE, Purdue University
West Lafayette, IN, 47907, USA
{lee222,eigenman}@purdue.edu

ABSTRACT

Sparse matrix-vector (SpMV) multiplication is a widely used kernel in scientific applications. In these applications, the SpMV multiplication is usually deeply nested within multiple loops and thus executed a large number of times. We have observed that there can be significant performance variability, due to irregular memory access patterns. Static performance optimizations are difficult because the patterns may be known only at runtime. In this paper, we propose adaptive runtime tuning mechanisms to improve the parallel performance on distributed memory systems. Our adaptive iteration-to-process mapping mechanism balances computational load at runtime with negligible overhead (1% on average), and our runtime communication selection algorithm searches for the best communication method for a given data distribution and mapping. Actual runs on 26 real matrices show that our runtime tuning system reduces execution time up to 68.8% (30.9% on average) over a base block-distributed parallel algorithm on distributed systems with 32 nodes.

Categories and Subject Descriptors

J.2 [**Physical Science and Engineering**]: Physics; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance

Keywords

runtime tuning, sparse matrix, process mapping

1. INTRODUCTION

The importance of sparse matrix-vector (SpMV) multiplication as a computational kernel has led to a large number of research contributions to optimize its performance. Many contributions have improved SpMV multiplications on single-processor or shared memory systems [10, 21]. There has been a focus on architecture-oriented techniques, such as register blocking, cache blocking, and TLB blocking. While these techniques may be applied on distributed systems to tune individual nodes, they do not propose parallel distributed optimizations.

On distributed parallel SpMV multiplication, load balancing and communication cost reduction are two key issues. To address these issues, many graph-partitioning-based decomposition algorithms [3, 5, 4, 14] have been proposed. Due to the complexity of these algorithms, they are generally applied as preprocessing steps, rather than as runtime optimizations. For runtime tuning, several decomposition heuristics have been suggested [12, 22, 13, 2, 16]. They generally aim at distributing non-zero elements as evenly as possible. However, these static allocation methods do not capture dynamic runtime factors affecting the performance. One approach proposed a framework for dynamic optimization of parallel SpMV operations, on top of which load-balancing techniques can be added [11]. The approach aims at only high-level parallelism, however.

In this paper, we propose runtime tuning mechanisms that can be applied dynamically to adapt distributed parallel SpMV multiplication kernels to the underlying environments. No preprocessing steps are necessary. Our adaptive *iteration-to-process mapping* system achieves load balance by re-assigning rows to processes, according to the measured execution-time workload of each process. In this way, our tuning system achieves better load balance than previous static allocation systems. Our adaptive system also selects, at runtime, from among several communication methods. We evaluated the proposed tuning system on 26 real sparse matrices from a wide variety of scientific and engineering applications. The experimental results show that our tuning system reduces execution time up to 68.8% (30.9% on average) over a base block-distributed parallel algorithm on a 32-node platform.

This paper is organized as follows: Section 2 provides an overview of basic sequential and distributed parallel SpMV multiplication algorithms, and discusses previous approaches on parallel SpMV optimizations. Section 3 presents our runtime tuning system, which consists of an adaptive iteration-to-process mapping mechanism and a runtime communication selection system. Implementation methodology and experimental results on 26 sparse matrices are shown in Section 4 and Section 5, respectively. Section 6 presents conclusions.

ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.

```
DO k = 1, ITER, 1
  DO i=1, N, 1
    y(i) = 0
    DO j=0, rowptr(i+1)-rowptr(i)-1, 1
      y(i) = y(i) + val(rowptr(i)+j) * x(ind(rowptr(i)+j))
    ENDDO
    x(i) = normalize(y(i))
  ENDDO
ENDDO
```

Figure 1: Sequential algorithm for sparse matrix-vector multiplication, where N is the number of rows in matrix A

2. PARALLEL SPMV MULTIPLICATION ON DISTRIBUTED MEMORY SYSTEMS

There are several sparse storage formats, which favor different parallelization strategies [17]. In this work, compressed row storage (CRS), which is the most widely used sparse-data format in scientific computing applications, is used to store a sparse matrix for SpMV computations. In CRS, rows are stored in consecutive order. A dense array, *val*, is used to store non-zero matrix elements in a row-wise fashion, and two other dense arrays are used to keep the positional information of each non-zero element: *ind*, which contains the column indices of each stored element, and *rowptr*, which contains pointers to the first non-zero element of each row in the array *val*. A sequential algorithm for sparse matrix-vector multiplication ($y=Ax$) is shown in Fig. 1.

For a base distributed parallel implementation, we followed the guidelines in [17]: one-dimensional data distribution and broadcast messages for all data communication. One-dimensional data distribution reduces the complexity of the load-balancing problem on heterogeneous clusters [9]. There are several algorithms for 2-D data partitioning [13, 16, 20]. However, due to their complexity, they are more suitable for single or shared memory systems than for distributed memory systems [12]. The use of broadcast messages for data communication is a natural choice, and it may perform optimally for common cluster interconnections such as Ethernet [17]. As we will see later in this paper, however, different communication methods may be preferred depending on irregular parallel SpMV communication patterns.

In a parallel SpMV implementation for CRS format, the sparse matrix is block-distributed in a row-wise fashion, and the output vector y can also be block-distributed. On the other hand, the input vector x should be replicated to all processes because each process may need the entire input vector x for SpMV multiplication. At each outermost-loop iteration, each process computes its local matrix-vector multiplication part and broadcasts its newly updated input vector to all processes. The parallel algorithm and corresponding data distribution appear in Fig. 2.

In Fig. 2 (a), *s_index* and *e_index* determine the row blocks that each process will compute. For example, the process *P0* in Fig. 2 (b) calculates the matrix-vector product of the dotted region and generates new values *y(1:2)*. Before starting the next calculation, each process updates remotely computed values of x through MPI_Allgatherv() all-to-all communication.

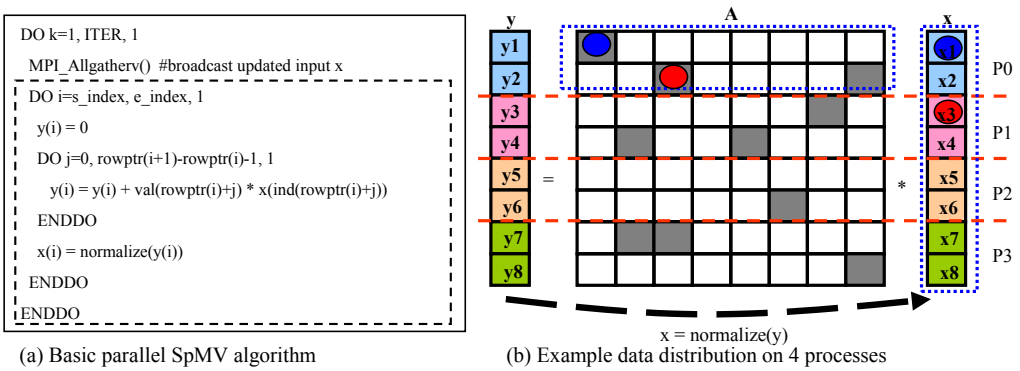
In distributed parallel SpMV algorithms, irregular memory access patterns cause significant load imbalance in terms of both computation and communication. Resolving this issue at compile time is difficult because the data access patterns of the involved computation and communication may be known only at runtime. There exist extensive graph-partitioning-based decomposition algorithms [3, 5, 4, 14] pursuing either minimum communication cost or computational load balancing. But due to their complexity or incurred large overhead, none of them are applicable as runtime techniques. Instead, these techniques are applied to sparse matrices as preprocessing steps. However, this means that users must preprocess every input sparse matrix before running SpMV kernels. Other research has proposed decomposition heuristics [12, 22, 13, 2]; although less complex, they still incur non-negligible overhead. They are engaged as either preprocessing or static runtime allocation methods. Moreover, none of the aforementioned techniques consider dynamic runtime factors such as computing power difference among the assigned nodes, the interconnection characteristics of deployed clusters, and interference from co-existing jobs.

3. ADAPTIVE RUNTIME TUNING SYSTEM

This section presents an adaptive runtime tuning system for distributed parallel SpMV multiplication kernels. The proposed tuning system consists of two steps: *normalized row-execution-time-based iteration-to-process mapping* and *runtime selection of communication algorithms*. The first step achieves computational load balance by mapping rows to processes dynamically, based on measured execution time; the second step finds the best possible communication method for message patterns generated by the first step. In contrast to previous approaches, which statically assign workload to each process according to non-zero element distributions, the proposed tuning system re-distributes the workload dynamically. In this way, our tuning system attains improved load balance. The proposed system assumes compressed row storage (CRS) format, but it can be applied to other storage formats, too. Detailed descriptions of the tuning system are presented in the following two subsections.

3.1 Normalized row-execution-time-based iteration-to-process mapping algorithm

In distributed parallel SpMV multiplication, general decomposition algorithms [12, 22, 13, 2] solve load-balancing problems by partitioning non-zero elements evenly among participating processes. In a one-dimensional data distribution case, the decomposition algorithms map rows to processes such that the numbers of non-zero elements assigned to each process are as even as possible. However, these mapping mechanisms do not consider other factors, such as loop overheads. For example, suppose that two processes are assigned the same number of non-zero elements to compute. If this assignment results in 10 out of 1000 rows being mapped to one process and the remaining 990 rows to the other process, the two processes will incur different loop overheads, even though the amount of SpMV calculations is identical. Another problem of the existing decomposition algorithms is that they do not consider dynamic runtime-system performance. In batch-oriented large clusters, the computing nodes may consist of heterogeneous machines with different clock speeds or physical memory sizes. Moreover, if each



(a) Basic parallel SpMV algorithm

(b) Example data distribution on 4 processes

Figure 2: Parallel algorithm and data distribution for sparse matrix-vector multiplication

node has multiple cores, depending on job configurations, some jobs may run on the same node, sharing its physical memory. The existing algorithms do not cover these dynamic factors affecting runtime performance.

To address these issues, we propose a dynamic runtime mapping mechanism called normalized row-execution-time-based iteration-to-process mapping. In the proposed algorithm, row-to-process mapping is performed at runtime, based on the execution time of each row. The optimization goal of our mapping algorithm is to find the best row-to-process mapping, such that measured SpMV computation times are as even as possible among processes. The basic approach to achieve this goal is as follows: initially rows are block-distributed evenly among processes and each process measures the execution time of each assigned row. The row-execution times are exchanged among processes in an all-to-all manner. The rows are block-distributed again, but inter-process boundaries are adjusted, such that each process has similar row-block-execution time, which is the sum of execution times of rows mapped to the same process. Due to the runtime performance difference, the execution times of the same row may differ on different processes. Therefore, the execution time of each row is measured again. This measure-and-map procedure is repeated until the difference of row-block-execution times are within some threshold (5% in our experiments).

Measuring and exchanging each row execution time may incur large measuring and communication overhead. To minimize the incurred overheads, we approximate each row execution time. At every outermost-loop iteration, which corresponds to one computation of SpMV multiplication, every process measures net computation time consumed to calculate row blocks assigned to the process. *Normalized row execution time (NRET)* is the measured time divided by the number of assigned rows.

$$NRET = \frac{exe.\ time\ for\ assigned\ rows\ computation}{the\ number\ of\ assigned\ rows}$$

In this approximation, each row assigned to one process has identical execution time regardless of the number of non-zeros contained in the row. To increase accuracy, the row execution time is recalculated whenever the row-to-process mapping is changed, and the newly estimated value is used for the next re-mapping. These repeated feedback steps have a grouping effect, so that the algorithm converges. In theory, convergence is not guaranteed, which can occur if there exist a few highly dense rows in small matrices.

To guarantee termination, our algorithm forces tuning to stop after a number of steps (20 in our experiments) and remain off for a *shelter period*. After that period (100 outermost-loop iterations in our experiments) the runtime environment is checked for changes; if unchanged, the algorithm repeats the shelter period. This process continues through the end of the program.

The overall procedure and a simple example are presented in Fig. 3. During the mapping phase, all calculations are redundantly executed by all processes to minimize the involved communication. As shown in the example, some row blocks should be migrated to a new process when the mapping is changed. However, the communication cost to migrate large row blocks may be high. To reduce the migration cost, our tuning system replicates the entire sparse matrix data rather than block-distributing the data among processes. This replication removes data migration complexity and the involved communication cost at the expense of increased memory pressure. Selecting between data replication and migration may be a further tuning target.

3.2 Runtime selection of communication algorithms

On distributed memory systems, data communication is performed by explicit message passing. In this work, we use MPI [1] for data communication. MPI provides rich communication methods, offering variants and parameters that may be tuned for specific communication patterns. Choosing the right methods is important for high performance applications. An extensive study [15] on real MPI applications, performed on a Cray T3E, reveals that the most important performance bottleneck of MPI communication lies in the synchronization delay, which is the difference between the total time spent in the MPI communication call and the underlying data-transfer time. This synchronization delay depends on both computational and communicational load balances. Many existing approaches [7, 19, 8] only focus on individual collective-communication-call tuning, ignoring the synchronization delay caused by computational load imbalance. By contrast, our iteration-to-process mapping system addresses this delay. Given that the computational load-balancing problem is solved and there exist many solutions for individual-communication-call tuning, our communication tuning system focuses on more high-level

Given that n = # of rows in sparse matrix A, m = # of involved processes

1. Each process, p, calculates average computation time (CT(p)) of the assigned SpMV comp. blocks

2. Each process collects CT(k) from all the other processes (k=1..m) through MPI communication (MPI_Allgather)

3. Each process calculates normalized row-execution time (NRET) and target computation time (tCT)

For k = 1..m

For each row, i, assigned to process k //s_index(k) and e_index(k) are row-block boundaries of process k

$$NRET(i) = \frac{CT(k)}{e\_index(k) - s\_index(k) + 1}$$

tCT = Average of {CT(k), k=1..m}

4. Each process re-maps rows to processes such that all processes have the same expected computation time (eCT)

For k = 1…m //Assume i was initialized to 1 at the beginning

eCT(k) = 0; s_index(k) = i; //Set a new start boundary for process k

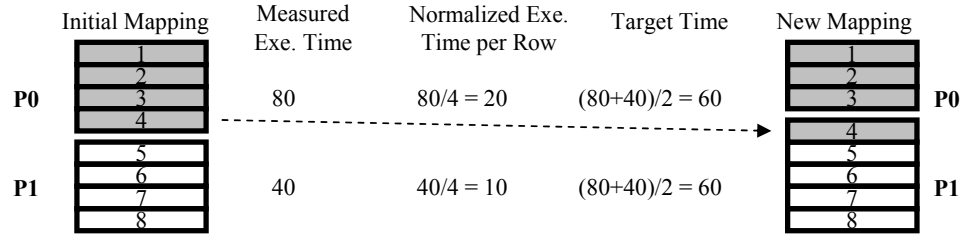
While ((eCT(k) < tCT) && (i<=n)) { eCT(k) += NRET(i); i++; }

e_index(k) = i-1; //Set a new end boundary for process k

e_index(m) = n; //Assign the remaining rows to the last process m if any exists

5. Repeat 1 ~ 4 until the difference of CT(k) among processes is within 5%

(a) Normalized row-execution time based iteration-to-process mapping algorithm



(b) Example case where 8 rows are mapped to 2 processes

Figure 3: Normalized row-execution-time-based iteration-to-process mapping algorithm and an example

algorithmic approaches. In distributed parallel SpMV multiplication, each process updates remotely computed data at every outermost-loop iteration. This data exchange involves many-to-many communication. There are several options for this communication, as shown below.

Block broadcasting method (CM1): each process broadcasts locally written output blocks.

Block point-to-point exchange method (CM2): processes exchange bounding blocks containing needed elements through point-to-point communication.

Packed point-to-point exchange method (CM3): processes exchange exactly needed elements through point-to-point communication. This method involves explicit/implicit packing and unpacking.

The three methods perform differently depending on communication patterns. *CM1* may work well when large numbers of non-zero elements are distributed over an entire matrix. The rationale behind this method is that existing collective communications may be customized to all-to-all communication patterns with large data size. In contrast to *CM1*, *CM2* and *CM3* reduce the communication volume by calculating needed data for each process. *CM3* minimizes communication volume at the cost of additional computation and memory overheads for data packing and unpacking. Compared to *CM3*, *CM2* may include unnecessary elements in the bounding blocks. This does not affect correctness because these unnecessary elements are also the ones written by the sending process. In the proposed tuning system, the best communication method is selected at runtime, based on measured execution time. Since the performance of these methods is heavily dependent on a data distribution, the fastest method is re-selected whenever the adaptive mapping system changes the distribution. Measuring a global time is not simple in parallel executions because nodes have local timers, which are not synchronized. In the proposed tuning system, however, this problem is minimized by the balanced workload. Our selection mechanism simply uses the average measured communication time.

The idea of selecting the best variant among several high-level communication algorithms came from [18], where the best method is chosen at runtime, based on a performance model. One drawback of the previous approach is that the performance model is a simple latency-based point-to-point model, which does not capture complex communication behaviors. By contrast, our simple selection mechanism reflects dynamic runtime performance. Moreover, our system does not depend on prerequisites, such as measuring latencies between nodes to calculate model parameters [18].

4. METHODOLOGY

4.1 Implementation

For our experiments, we hand-coded an MPI-version of *spmul*, a common sequential SpMV multiplication kernel. We combined this base version of the distributed SpMV multiplication kernel with the described tuning system. As communication methods, we used MPI_Allgatherv() for *CM1* and non-blocking receive (MPI_Irecv()) combined with stan-

Table 1: Summary of sparse matrices used in evaluation

Name	Dim (NxN)	Non-zeros	Type	Description
af_shell10	1,508,065	27,090,195	diagonal	sheet metal forming
boneS10	914,898	28,191,660	diagonal	3D trabecular bone
rajat31	4,690,002	20,316,253	diagonal	circuit simulation matrix
Si41Ge41H72	185,639	7,598,452	diagonal	Real-space pseudo-potential method
SiO2	155,331	5,719,417	diagonal	Real-space pseudo-potential method
g7jac200sc	59,310	837,936	diagonal	Jacobian from CEPII
ldoor	952,203	23,737,339	even	INDEED test matrix
appu	14,000	1,853,104	even	fluid dynamics
ASIC_680ks	682,712	2,329,176	even	Xyce circuit simulation matrix
ASIC_680k	682,862	3,871,773	uneven	Xyce circuit simulation matrix
crankseg_2	63,838	7,106,348	uneven	OUTPUT4-Matrix
F1	343,791	13,590,452	uneven	Symmetric indefinite matrix
F2	71,505	2,682,895	uneven	AUDI engine piston rod
hood	220,542	5,494,489	uneven	INDEED Test Matrix (DC-mh)
nd6k	18,000	3,457,658	uneven	ND problem set
nd24k	72,000	14,393,817	uneven	ND problem set
ns3Da	20,414	1,679,599	uneven	3D Navier Stokes
poisson3Db	85,623	2,374,949	uneven	3D Poisson problem
sme3Db	29,067	2,081,063	uneven	3D structural mechanics problem
sme3Dc	42,930	3,148,656	uneven	3D structural mechanics problem
sparsine	50,000	799,494	uneven	structural optimization (CUTEr)
audikw_1	943,695	39,297,771	uneven	symmetric rb matrix
darcy003	389,874	1,167,685	uneven	discretization using mixed FE
inline_1	503,712	18,660,027	uneven	stiffness matrix
kkt_power	2,063,494	8,130,343	uneven	Optimal power flow
msdoor	415,863	10,328,399	uneven	medium size door

dard send (MPI_Send()) for *CM2* and *CM3*. For data packing in *CM3*, we used MPI_Type_indexed(), which provides implicit data packing and unpacking by the underlying MPI implementation.

4.2 Parallel platforms

We used the Intel compiler 9.0 with option -O2 and MPICH2 1.2.7. We carried out the experiments on the Hamlet linux cluster at Purdue University. Hamlet consists of 308 IA-32 P4 nodes with two different processor speeds (3.06 GHz and 3.2 GHz) and two different physical memory sizes (2 GB and 4GB). We used 32 nodes connected with InfiniBand.

4.3 Evaluated sparse matrices

To evaluate the performance of our tuning system on the parallel SpMV multiplication kernel, we conducted experiments on 26 real sparse matrices in the UF Sparse Matrix Collection [6]. The matrices cover a wide range of application areas, such as finite element method, circuit simulation, and linear programming. A summary of the matrices appears in Table 1. Most of the matrices in the table are symmetric. For some of these cases, only one half of the matrix is stored, and we performed our experiments only on the stored information. While the overall execution time for these cases would be twice the numbers we obtained, the load-balancing results would be the same. The same algorithm would essentially be performed twice – once on the upper half and once on the lower half of the input matrix.

5. EXPERIMENTAL RESULTS

This section presents the performance of the tuned parallel SpMV multiplication kernel on real matrices listed in Table 1. We compared our tuned version with the base parallel SpMV multiplication kernel described in Section 2. In the base implementation, rows were block-distributed evenly among processes. To separate the contributions of adaptive mapping and communication method selection, we also ran variants with only one of these techniques turned on. We implemented a static allocation method and compared it with our iteration-to-process mapping system. The overhead incurred by the tuning system was analyzed and the upper limit of the overhead was measured. Each kernel executed SpMV multiplication 1000 times and each test per input matrix was repeated three times. For time measurements, we used MPI_Wtime(), a built-in MPI function.

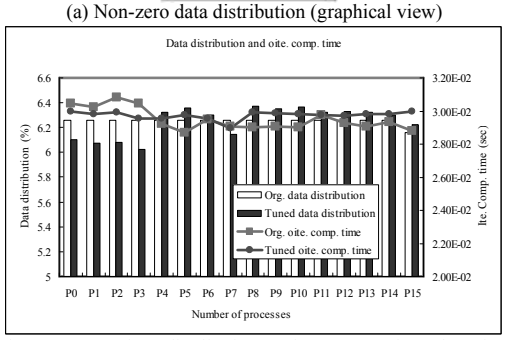
5.1 Parallel performance

Performance impact of matrix structure: In SpMV computations, the data structure of the input matrices has a significant impact on the performance. In Table 1, matrices are classified into three types: diagonal, even, and uneven. In diagonal-type matrices, non-zero elements are allocated along and around the diagonal. These matrices are usually highly sparse and evenly distributed with respect to rows.

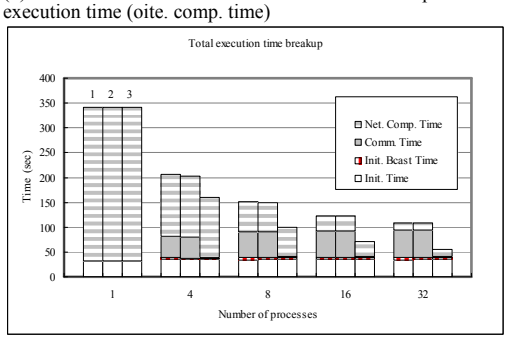
Fig. 4 shows an example of the diagonal-type matrices. Fig. 4 (b) displays non-zero element distribution and one outermost-loop iteration execution time (oite. comp. time), which is the time to execute SpMV computation blocks assigned to each process. The graph contains results of the original version (base parallel version) and the tuned version run on 16 nodes. In this case, the original data distribution is already fairly even. Nevertheless the effect of load balancing is evidenced in the figure.

Fig. 4 (c) shows execution time breakups for original, computation-tuning-only (*CTuned*), and tuned versions. The left bar in each group, marked as *1*, represents the original version, followed by the CTuned and tuned versions. From the graph, we can see that adaptive mapping does not improve significantly, but communication tuning is very effective (averaged total execution time reduction = 36%). In

(a) Non-zero data distribution (graphical view)



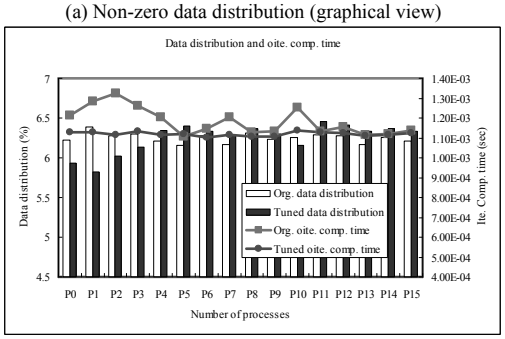
(b) Non-zero data distribution and outermost-loop iteration execution time (oite. comp. time)



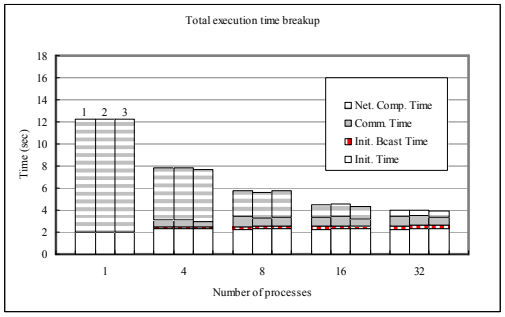
(c) Execution time breakups for original, computation-tuning-only (CTuned), and tuned versions

Figure 4: Non-zero data distribution and parallel performance of sparse matrix *af_shell*

(a) Non-zero data distribution (graphical view)



(b) Non-zero data distribution and outermost-loop iteration execution time (oite. comp. time)



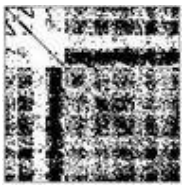
(c) Execution time breakups for original, computation-tuning only (CTuned), and tuned versions

Figure 5: Non-zero data distribution and parallel performance of sparse matrix *appu*

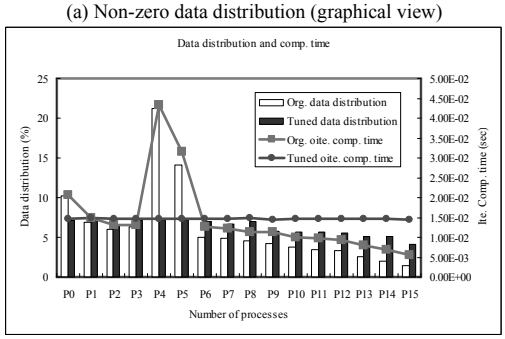
diagonal matrices such as *af_shell10*, each process has to exchange data only with its neighbors. In this case, point-to-point communications such as *CM2* or *CM3* are more suitable than heavy all-to-all communication (*CM1*). In real experiments on 4, 8, 16, and 32 nodes, *CM3* is selected for 8 out of 12 executions and *CM2* was selected for the remaining executions.

Fig. 5 represents the case for even-type matrices, whose non-zero elements are not allocated diagonally but still distributed evenly with respect to rows. As shown in Fig. 5 (a), the matrices of this type usually have non-zeros distributed randomly over the entire matrix. As the *af_shell10* case did, Fig. 5 (b) also shows that computational load balance can be achieved with uneven non-zero distribution, even though the performance benefits are negligible. In this case, both adaptive mapping and communication tuning are of little use. Communication tuning is not useful because the data distribution in this case requires all-to-all communication, where *CM1* may perform better than others, depending on the sparsity and communication volume. In our experiments, *CM3* was selected more often than *CM1*, but the results indicate that the performance of *CM3* is no better than *CM1*, in this case.

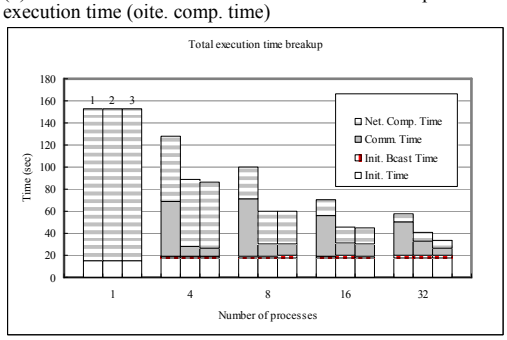
The matrix *F1* in Fig. 6 is highly uneven. Consequently, the data distribution is also uneven and its communication pattern is irregular. Fig. 6 (b) and (c) illustrate the power of our adaptive mapping system. In the *F1* input case, our mapping system achieves load balance with 5.6 tuning calls on average, and the incurred overhead is 0.45% on average. With this small overhead, our tuning system reduces the total execution time by 37%. The main execution time (the time to execute the main SpMV computation body excluding the initial input data distributing section,) is reduced by 49.6% on average. From Fig. 6 (c), we can see that computational load balancing reduces communication time. By balancing the workload among processes, the synchronization delay, mentioned in Section 3.2, can be reduced. As shown in Fig. 6 (a), *F1* has non-zero elements randomly distributed among the entire matrix. Therefore, communication tuning is not effective. On the other hand, in the 32-node case (rightmost group in Fig. 6 (c)), there is a noticeable time reduction between the middle bar (CTuned version) and the right bar (tuned version). This suggests that communication characteristics can be changed within the same input matrix, depending on the number of involved processes. In our experiments on the *F1* input, *CM1* or *CM2* were selected for the 4, 8, and 16 node cases, but the 32-node case preferred *CM3*. *CM3* minimizes the communication volume, but it requires complex data packing and unpacking. If large numbers of non-zero elements are located closely but not



(a) Non-zero data distribution (graphical view)



(b) Non-zero data distribution and outermost-loop iteration execution time (oite. comp. time)



(c) Execution time breakups for original, computation-tuning only (CTuned), and tuned versions

Figure 6: Non-zero data distribution and parallel performance of sparse matrix *F1*

continuously, *CM3* will perform worse than *CM2*, when its packing overhead offsets the benefits from communication volume reduction. This happened on 4, 8, and 16 nodes. On 32 nodes, however, communication volume reduction by *CM3* outweighed the packing overhead. This case benefits from tuning even within the same input data and the same target architecture.

These three cases are representative of the performance behavior of our tuning system.

Overall performance on 26 matrices: Speedups of all 26 matrices on 4 and 32 nodes are presented in Fig. 7. In the figure, total speedups are the speedups in terms of total execution time and main comp. speedups show main execution speedups. The graphs reveal several interesting behaviors. First, in the 4-node cases, both adaptive mapping and communication tuning work well, but on 32 nodes, communication tuning is more important than adaptive mapping. This effect is caused by computational load balancing, which becomes less problematic as the number of involved processes increases. When computation is distributed to a large number of processes, less work is assigned to each process on average. Hence, the synchronization delay caused by load imbalance tends to be smaller. Another reason is that the benefit of using *CM2* or *CM3* becomes less on small numbers of nodes; communication volume tends to be large and it is more likely to be all-to-all communication. In such case, *CM1* may perform better than the others. In our experiments, *CM3* was chosen more often than *CM1* even on 4 nodes, but the performance benefit of *CM3* was much less than on 32 nodes.

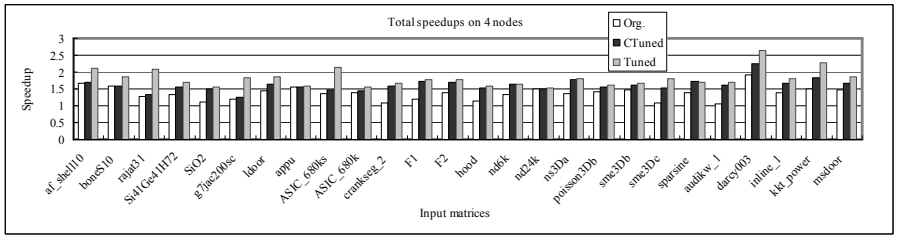
Second, there is a gap between total execution speedup and main execution speedup and the gap increases as the number of nodes increases. In the current parallel implementation, input data are read by process *P0* and the data are broadcast to all the other processes. The broadcast message replicates input data to all involved processes, which adds non-negligible overhead. If we block-distribute the input data, the initial overhead may reduce, but the distribution may cause several input data migrations during the adaptive mapping phase. There is an opportunity to further develop efficient algorithms for data migration. Table 2 summarizes the execution time reductions measured for the 26 matrices.

5.2 Comparison of static allocation and adaptive iteration-to-process mapping

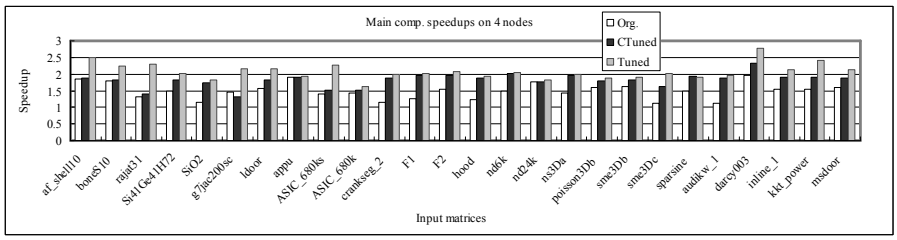
This section compares our adaptive iteration-to-process mapping system with a static allocation method, which maps rows to processes statically, such that non-zero elements are distributed evenly among processes. Fig. 8 contains the speedups of both our mapping algorithm (CTuned) and the static allocation algorithm (SA) on 4 and 16 nodes. The figure reveals that our adaptive mapping system performs equally or better than the static allocation method in most cases. *darcy003* and *kkt_power* are quite interesting cases; the static method gets hardly any speedups on both 4 and 16 nodes, while our adaptive mapping achieves reasonable ones. *darcy003* and *kkt_power* have a large number of consecutive rows that do not contain any non-zero elements. When the static method is applied, these rows are assigned to one process. Fig. 2 (a) shows that the rows containing no non-zero elements still have some computations, such as loop checking and normalization. If a great number of such rows are assigned to one process, as the static allocation does, these small computations build up significant workloads, causing severe load imbalance. By contrast, our adaptive mapping system considers effective workloads experienced by each process. Therefore, our mapping system causes no such load imbalance problem. These results indicate that non-zero-element-distribution-based load-balancing mechanisms may not be optimal, depending on input matrix characteristics.

5.3 Comparison of tuned and fixed communication

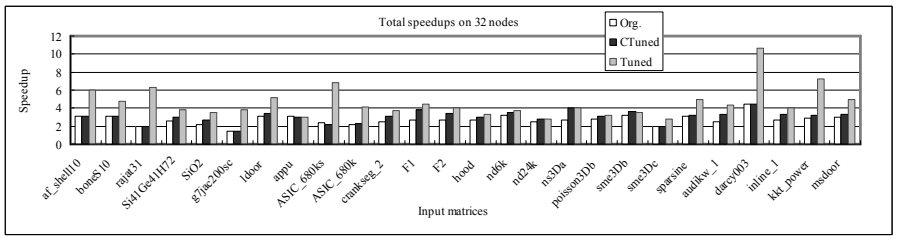
In this section, we study the effect of runtime selection of communication methods. For this study, we compared our tuned version with fixed-communication-method versions (*CM1*, *CM2*, and *CM3*). Adaptive mapping is applied to all versions to rule out the effect of load balancing. The speedups on 4 nodes and 16 nodes are shown in Fig. 9. The results indicate that *CM3* works well in most cases, except for *ns3Da*, *poisson3Db*, *sme3Db*, and *sme3Dc*, where *CM1* works best. This phenomenon occurs because we deal with sparse matrices. Table 1 shows that most of the matrices used in the experiments are very large but highly sparse at the same time. On these matrices, *CM3* may generate much smaller communication volume than *CM1* or *CM2*, such that the packing overhead can be overcome by reduced communi-



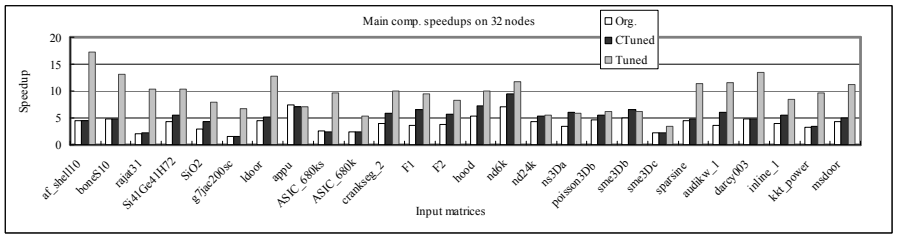
(a) Total execution time speedups on 4 nodes



(b) Main execution time speedups on 4 nodes



(c) Total execution time speedups on 32 nodes



(d) Main execution time speedups on 32 nodes

Figure 7: Speedups of all 26 matrices on 4 and 32 nodes

Table 2: Execution time reduction by the proposed tuning system: In *A(B)* format, *A* represents the average of 26 matrices and *B* is the maximum value

	4 nodes	8 nodes	16 nodes	32 nodes	overall
Main time reduction (CTuned) (%)	17.2 (40.4)	22.1 (55.2)	20.7 (51.7)	16.3 (45)	19.1 (55.2)
Main time reduction (Tuned) (%)	27.8 (44.1)	38.8 (62.1)	46.1 (75.9)	53.2 (79.3)	41.5 (79.3)
Total time reduction (CTuned) (%)	14.9 (34.2)	16.9 (44.7)	14 (37.8)	9.6 (32)	13.8 (44.7)
Total time reduction (Tuned)(%)	23.9 (39.6)	30.6 (55.6)	33.3 (66.7)	36 (68.8)	30.9 (68.8)

cation size. By comparing graphs in Fig. 9 and Table 1, we can find the trend that *CM1* or *CM2* is preferred to *CM3*, as input matrices become denser. Fig. 9 shows that our tuning system can capture the best communication method in all cases.

5.4 Tuning overhead

Fig. 10 presents tuning overhead, which is represented by the percentage of tuning time in the total execution time. Fig. 10 (a) contains measured overheads on 4, 8, 16, and 32 nodes. From the graph, we can see that the overheads increase as the number of nodes increases. The proposed tuning system involves a small number of all-to-all communications for each process to collect timing information of the others. Even though the required communication volume is small, these collective calls become heavier, as the number of nodes increases. Fig. 10 (b) compares incurred overhead and upper limit, which is the case where the tuning system is called continuously until the program finishes. The upper limit tests were conducted on 4 and 16 nodes. On 4 nodes, the upper limit is still small (0.72% on average), but on 16 nodes, it goes up to 12% on average. However, the actu-

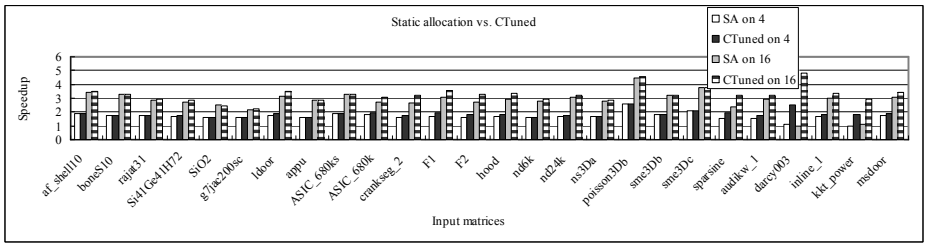
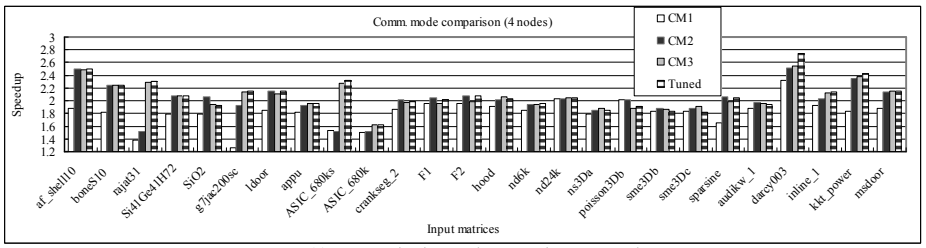
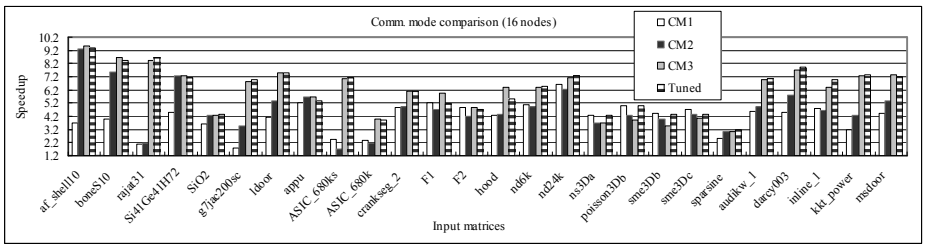


Figure 8: Performance comparison of static allocation method (SA) vs. adaptive iteration-to-process mapping method (CTuned)

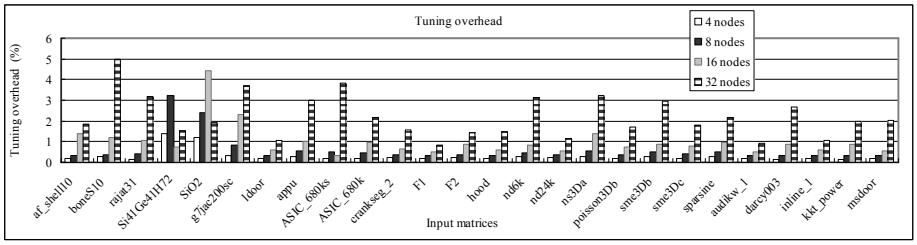


(a) Communication mode comparison on 4 nodes

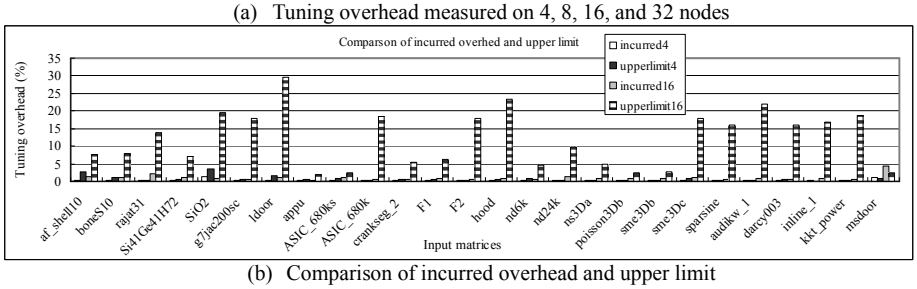


(b) Communication mode comparison on 16 nodes

Figure 9: Performance comparison of fixed communication modes vs. tuned mode



(a) Tuning overhead measured on 4, 8, 16, and 32 nodes



(b) Comparison of incurred overhead and upper limit

Figure 10: Tuning overhead (percentage of tuning time in the total execution time)

ally incurred overhead remains small (0.3%, 0.6%, 1%, and 2.2% on 4, 8, 16, 32 nodes, respectively). This shows that our tuning system is efficient and converges quickly in most cases.

6. SUMMARY AND CONCLUSIONS

We have presented an adaptive runtime tuning system for distributed parallel SpMV multiplication kernels. Our adaptive mapping system solves load-balancing problems by dynamically re-assigning matrix rows to processes, according

to measured real-time workloads. To minimize the incurred communication cost, our runtime selection system finds the best communication method for the data distribution tuned by our mapping mechanism. Experiments on 26 sparse matrices from various scientific and engineering applications led to several key findings. First, load-balancing mechanisms based on non-zero element distribution may not be suitable. Most previous work attempts to achieve load balance by distributing non-zero elements among processes as evenly as possible. However, our results show that load balancing may be lowest with uneven data distribution, when considering the underlying runtime environment. Second, computational load balancing can play an important role in minimizing communication cost, as it can reduce synchronization delay. Third, different data distributions favor different communication methods, even though point-to-point communication methods work well in many cases. This is because the sparsity of matrices is often very high. In these cases, it is likely that either communication volume is small or all-to-all communication is unnecessary. Fourth, initial input data transfer times add non-negligible overhead to the parallel execution time. To reduce this overhead, more studies on various storage formats or data migration algorithms are needed.

Even though this paper focuses on sparse matrix-vector multiplication tuning, our adaptive tuning system can be applied to general irregular applications such as N-body problems. The work presented in this paper generalizes a compiler-driven adaptive tuning system, where a compiler identifies irregular loops showing repetitive computation and communication patterns and generates necessary inspector-executor codes. Future work will include applying our techniques to other irregular applications as well as applying these techniques to a large range of sparse storage formats.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] The message passing interface (MPI) standard [online]. available: http://www-unix.mcs.anl.gov/mpi/.

[2] J. I. Aliaga and V. Hernandez. Symmetric sparse matrix-vector product on distributed memory multiprocessors. *Conf. on Parallel Computing and Transputer Applications*, 1992.

[3] R. H. Bisseling and W. Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005.

[4] E. G. Boman and U. Catalyurek. Constrained fine-grain parallel sparse matrix distribution. *SIAM Workshop on Combinatorial Scientific Computing*, 2007.

[5] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed systems*, 10, July 1999.

[6] T. Davis. University of Florida Sparse Matrix Collection [online]. available: http://www.cise.ufl.edu/research/sparse/matrices/.

[7] G. E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. J. Dongarra, and E. Jeannot. Flexible collective communication tuning architecture applied to Open MPI. *Euro PVM/MPI*, 2006.

[8] A. Faraj and X. Yuan. Automatic generation and tuning of MPI collective communication routines. *Int. Conf. on Supercomputing*, pages 393–402, 2005.

[9] B. Hendrickson and T. G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM J. on Scientific Computing*, 21(6):2048–2072, 2000.

[10] E. J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applicatins*, 18(1):135–158, 2004.

[11] D. Jin and S. G. Ziavras. A super-programming technique for large sparse matrix multiplication on PC clusters. *IEICE Trans. Inf. & Syst.*, E87-D(7), July 2004.

[12] S. G. Nastea and O. Frieder. Load-balancing in sparse matrix-vector multiplication. *Proceedings of 8th IEEE Symposium on Parallel and Distributed Processing*, page 218, 1996.

[13] A. T. Ogielski and W. Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, 14(3):519–530, 1993.

[14] C.-W. Qu and S. Ranka. Parallel incremental graph partitioning. *IEEE Trans. on Parallel and Distributed Systems*, 8(8):884–896, 1997.

[15] R. Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. *Euro PVM/MPI*, pages 35–42, 1999.

[16] E. Rothberg and R. Schreiber. Improved load balancing in parallel sparse Choleski factorization. *Proceedings of Supercomputing (SC)*, 1994.

[17] R. Shahnaz, A. Usman, and I. R. Chughtai. Implementation and evaluation of parallel sparse matrix-vector products on distributed memory parallel computers. *IEEE Int. Conf. on Cluster Computing*, pages 1–6, 2006.

[18] J. Su and K. Yelick. Automatic support for irregular computations in a high-level language. *Proc. of 19th IPDPS*, page 53, 2005.

[19] S. S. Vahdiyar, G. E. Fagg, and J. J. Dongarra. Automatically tuned collective communications. *Proceedings of Supercomputing (SC)*, page 46, 2000.

[20] B. Vastenhouw and R. H. Bisseling. A two-dimmensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[21] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Proceedings of Supercomputing (SC)*, 2007.

[22] L. H. Ziantz, C. C. Ozturan, and B. K. Szymanski. Run-time optimization of sparse matrix-vector multiplication on SIMD machines. *Int. Conf. Parallel Architecture and Languages*, 817:313–322, July 1994.