Automatic Support for Irregular Computations in a High-Level Language

Jimmy Su and Katherine Yelick
Computer Science Division, University of California at Berkeley
{jimmysu,yelick}@cs.berkeley.edu

Abstract

The problem of writing high performance parallel applications becomes even more challenging when irregular, sparse or adaptive methods are employed. In this paper we introduce compiler and runtime support for programs with indirect array accesses into Titanium, a high-level language that combines an explicit SPMD parallelism model with implicit communication through a global shared address space. By combining the well-known inspector-executor technique with high level multi-dimensional array constructs, compiler analysis and performance modeling, we demonstrate optimizations that are entirely hidden from the programmer. The global address space makes the programs easier to write than in message passing, with remote array accesses used in place of explicit messages with data packing and unpacking. The programs are also faster than message passing programs: Using sparse matrix-vector multiplication programs, we show that the Titanium code is an average of 21% faster across several matrices and machines, with the best case speedup more than a factor of 2x. The performance advantages are due to both the lightweight RDMA (Remote Direct Memory Access) communication model that underlies the Titanium implementation and automatic optimization selection that adapts the communication to the machine and workload, in some cases using different communication models for different processors within a single computation.

1. Introduction

Application scientists increasingly employ irregular data structures such as unstructured grids, particle-mesh structures, adaptive meshes and sparse matrices in an effort to obtain more computationally efficient methods. These methods are not well supported by popular high performance programming models, because they lead to indirect array accesses, pointer-based data structures, and communication that is unpredictable in both timing and volume. Titanium [23] is a Java™-based language designed for high performance computing, with a shared address space to ease programming of pointer-based data structures and a powerful multidimensional array abstraction that has proven useful in both adaptive and non-adaptive block-structured algorithms.

In this paper we propose new compiler and runtime extensions for the Titanium implementation to support programs with indirect array accesses, such as A[B[i]], where the A array may live in a remote processor's memory. These computations arise in sparse iterative solvers, particle-mesh methods, and elsewhere. We add compiler support for an inspector-executor execution model, which optimizes communication performing runtime optimization based on the dynamic pattern of indices in the indirection array, which is B in the previous example. There are several possible transformations that can be done on the communication, and we consider three in this paper: sending the entire remote array, sending exactly those elements that are needed, and sending a bounding box of the required values. While packing is guaranteed to send the minimal number of actual values, it has a higher metadata overhead and is therefore not necessarily optimal. One of the challenges is to select the best communication transformation, because it depends on properties of the application and the machine. We introduce a simple analytical performance model into the compiler, which selects optimizations automatically.

We analyze the benefits of the automated inspector-executor transformation using sparse matrix-vector multiplication on a set of matrices from real applications. Our results show that although the program is significantly simpler when written in Titanium, because it avoids the explicit communication code and the pack and unpack code, the performance is almost always superior to a popular MPI message passing code. The speedup relative to MPI on a suite of over 20 matrices averages 21% on three different machines, with the maximum speedup of more than 2x. The model-based optimization selection is critical to both programmability and performance. Not only does

the model select optimizations that differ by matrix and machine, but also differ between processors within a single matrix-vector multiplication. This runtime complexity is entirely hidden from the programmer, making the application both cleaner and faster.

2. Related Work

The idea of inspector executor optimizations for scientific codes is not new. Walker proposed and implemented the idea of using a pre-computed communication schedule for indirect array accesses to distributed arrays in a Particle-In-Cell application [20]. The idea is widely used in application codes today, where it is programmed manually. Use of the technique with compiler and library support was developed by Berryman and Saltz. The PARTI runtime library [2] and its successor CHAOS [14] provided primitives for application programmers to apply the inspector executor optimization on the source level. The same research group provided the dataflow framework to determine where a communication schedule can be generated, where communication operations are placed, and when schedules can be combined [9]. As an experimental result, they manually carried out the optimizations that would have been suggested by the dataflow framework. The ARF [18] and KALI [11] compilers were able to automatically generate inspector executor pairs for simply nested loops. Slicing analysis was developed to extend the inspector executor paradigm to multiple level of indirection [6]. More recently, the inspector executor technique was used to develop runtime reordering of data and computation that enhance memory locality in applications with sparse data structures [15].

Benkner [3] introduces the concept of halos in HPF for the programmers to specify non-local access patterns to distributed arrays, and control the communication associated with these array accesses. Kelp [1] allows the programmers to use the MotionPlan object to do inspector executor communication. Unlike those two approaches, the optimizations we will describe in this paper are entirely hidden from the programmer.

There have been numerous research works in the area of communication scheduling. Chakrabarti *et al*. [5] implemented an algorithm for optimizing communication schedules across loops in a global manner in the HPF compiler. Dongarra *et al*. [17] gave techniques for performance modeling collective communications.

The work presented in this paper extends the inspector executor line of research by looking at the problem of selecting the best communication method. Our work is done in the context of a high level language with a global address space. Our compiler is able to automatically generate code that can accurately choose the best communication method during runtime based on an integrated performance model.

3. Titanium

Titanium is a dialect of Java, but does not use the Java Virtual Machine model. Instead, the end target is assembly code. During compilation, Titanium code is translated into C code, and then a C compiler compiles the generated C code. C is used as an intermediate step for portability. In addition to generating C code to run on each processor, the compiler generates calls to a runtime layer based on the GASNet [4] communication layer. GASNet uses lightweight communication, exploiting hardware support for direct remote reads and writes when it exists. Titanium runs on a wide range of platforms, including uniprocessors, shared memory machines, distributed-memory clusters of uniprocessors or SMPs (CLUMPS), and a number of specific supercomputer architectures (Cray X1, Cray T3E, SGI Altix, IBM SP, Origin 2000, and NEC SX6).

3.1. Memory Consistency Model

Titanium inherits many features of Java, one of which is the Java memory consistency model [8]. Titanium's interpretation of the Java memory consistency model is defined in the language specification [10]. Here are some informal properties of the Titanium model.

- Locally sequentially consistent: All reads and writes issued by a given processor must appear to that processor to occur in exactly the order specified. Thus, dependencies within a processor stream must be observed.
- Globally consistent at synchronization events: At a global synchronization event, such as a barrier, all processors must agree on the values of all the variables. At a non-global synchronization event, such as entry into a critical section, the processor must see all previous updates made using that synchronization event.

The first property implies that a processor must be able to read its own writes. If a processor writes to array elements that have been prefetched from a remote location into a local buffer, subsequent reads by that processor must return the new value. The second property makes data prefetched prior to a synchronization point unusable after that synchronization point. The prefetched data may have been changed by other processors, and reads after the

synchronization point must reflect those changes. This property prevents code motion past synchronization points.

3.2. Foreach Loop

The SPMD model in Titanium means that an instance of the program is run on some number of processors, which is specified at program startup time. Each copy of the program runs independently, without implied global synchronization.

Our transformation targets the built-in foreach loops in Titanium. A foreach loop has the form

foreach (p in D) S

where the iteration space *D* is a rectangular set of Points, where a Point is a n-tuple for an n-dimensional array. *S* any sequence of statements. The loop's semantics specifies that the body, *S*, be executed |*D*| times with *p* bound to each element of *D* in each iteration, but no particular execution order is required. The foreach loop is a local loop executed by a single processor, whereas the parallelism is at a higher level through the SPMD model. Titanium compiler does extensive analysis and optimizations for foreach loops [13].

4. Source Code Transformation

In this section, we give motivations for the optimizations later in the paper using source code transformation on a simple example. The simple example is the indirect sum benchmark in Figure 1.

Processor 0	Processor 1

for (i=0; i<n; i++){
  sum += A[B[i]];
}
	for (i=0; i<n; i++){
	  sum += C[D[i]];
	}

Figure 1: indirect sum benchmark on two processors

The example illustrates the case for two processors. But it can easily be extended for more processors. A is an array that resides on processor 1, and B is an index array owned by processor 0 for accessing array A. When processor 0 reads A[B[i]], it requires communication, because A resides on processor 1. Analogously, C is an array on processor 0, and D is an index array owned by processor 1 for accessing array C.

In Figure 2, we introduce buffers into the indirect sum benchmark. The buffers are local to each processor. The values of each indirect array access are stored in the buffer, and subsequently used in the second loop.

Processor 0	Processor 1

for (i=0; i<n; i++){
  buffer0[i] = A[B[i]];
}
for (i=0; i<n; i++){
  sum += buffer0[i];
}
	for (i=0; i<n; i++){
	  buffer1[i] = C[D[i]];
	}
	for (i=0; i<n; i++){
	  sum += buffer1[i];
	}

Figure 2: indirect sum benchmark on two processors with local buffers

In Figure 1 and Figure 2, the communication uses a pull strategy. Each processor gets the data it needs from the remote processor that owns the data. The next source code transformation in Figure 3 uses a push strategy instead. The code makes the assumption that processor 0 knows what data processor 1 needs from itself, and the location of the buffer on processor 1 to store this data. The same assumption applies to processor 1. In the push strategy, data is put to the processor that needs it.

Processor 0	Processor 1

for (i=0; i<n; i++){
  buffer1[i] = C[D[i]];
}
for (i=0; i<n; i++){
  sum += buffer0[i];
}
	for (i=0; i<n; i++){
	  buffer0[i] = A[B[i]];
	}
	for (i=0; i<n; i++){
	  sum += buffer1[i];
	}

Figure 3: indirect sum benchmark on two processors using push strategy

Two questions arise from these source code transformations. The first question is when is it legal to apply these transformations. The second question is how do these transformations give us the desired speedup. These questions will be addressed later in the paper.

5. Compile Time Transformations

5.1. Identify Inspector Executor Candidates

The first step is to identify inspector executor candidates for indirect array accesses A[B[i]]. Below is the list of the conditions that the compiler checks for:

- B and B[i] do not change inside of the loop.
- A and A[i] do not change inside of the loop.
- There are no synchronization points inside of the foreach loop, since the memory model requires memory to be globally consistent at a synchronization point.

After identifying the candidates, the compiler performs the inspector executor transformation. In the inspector phase, the array address for each A[B[i]] is computed. The computed values are stored in an index array. After the inspector phase, a communication method is chosen to retrieve the remote data into a local buffer. More details on the choice of the communication method are presented in Section 7. The set of array addresses together with the communication method are stored in a communication schedule. In the executor loop, values for each A[B[i]] are read out of the local buffer.

In some applications, the same pattern of indirect array accesses happens over multiple iterations. One example is an iterative solver. In this case, we would like to store the communication schedule computed during the inspector phase of the first iteration, and reuse the communication schedule on other iterations. A communication schedule may contain information for one or more sets of indirect array accesses to remote arrays. For each set of array accesses, the computed array addresses and the choice of communication method are stored in the schedule. Schedule reuse has been used in prior work, but our schedules contain additional information about the communication method to be employed.

The three properties are sufficient for ensuring the soundness of the inspector executor transformation. First we show that the values read for B[i] are the same for both versions of the program. If B and B[i] do not change during the execution of the loop, then the values read for B[i] in the inspector are the same as the ones in the original loop. B can only change locally, because it is a pointer on the local processor. The changes to B[i] can either come locally or remotely. We know that there are no local changes, because we check that B and B[i] are loop invariant using defuse information. Remote changes to B[i] are possible, since any of the remote processors can be executing code that modifies B[i] while the local processor is inside the loop. If we take a snap shot of memory where the B[i]'s reside before the local processor enters the loop, we can use the values from the snap shot for B[i]'s inside the loop regardless if there are changes to B[i] remotely. The reason is that there is no synchronization event inside the loop, so any remote changes to B[i] during the execution of the loop do not need to be reflected under the Titanium memory consistency model. Any writes to B[i] remotely while the local processor is executing this loop would constitute a race condition.

Now we know the index sets for both versions are the same. We would like to show that the values read from A using this index set are the same for both versions of the program. The argument is similar to the previous step. We know that there are no local changes to A or A[i] using defuse information. Changes to A[i] caused by remote processors during the loop do not need to be reflected, because there is no synchronization event inside the loop.

The requirement that A[i] is not modified by the processor executing the loop can be relaxed. The processor executing the loop has access to the buffer that contains the prefetched values of A[B[i]]. The runtime can conceivably intercept all writes to A[i] from this processor in the loop, and reflect the changes to the values in the buffer. Our experiments show that this relaxation is not worthwhile.

5.2. Pull to Push Transformation

Now we know the requirements for applying the inspector executor transformation. In this section, we turn our attention to the issue of using the push strategy instead of the pull strategy for communication. Extra coordination between processors is needed to use the push strategy. In the case where the schedule can be reused, we would like to communicate the index set and the choice of communication method during the first iteration, and have the processor that owns the data to send the needed data in the subsequent iterations independently. The push strategy uses about half as many messages as the pull strategy. The pull strategy also suffers when the remote processor is not attentive to the network. The communication is not entirely one-sided, because remote packing is required, so if the remote processor is in a computation intensive loop, other processors may be delayed waiting for it.

The extra coordination means that the communication calls need to be placed in the right place so that the data will be coming when it is expected. It is the job of the compiler to find the right place in the code to insert these communication calls, since we are applying the optimizations automatically without hints from the application programmer.

The communication calls need to be placed in such a way that when a processor is about to enter the loop that contains the inspector executor array access, the expected data is on its way from the remote processor. The processor can simply poll on a flag for the arrival of the data.

In Titanium, a barrier statement is executed by all the processors at the same time for the same number of times. A barrier that is executed by a subset of the processors would cause deadlock. The Titanium compiler provides a static analysis called single analysis that eliminates this type of bugs. It conservatively rejects all programs that might run into deadlocks due to misplaced barriers at compile time.

We use single analysis to help us in finding the right place to insert the communication calls. The

property that we are looking for is that whenever a processor is about to enter the loop containing the inspector executor array access, the processor that owns the needed data would execute the communication calls to send the data over. In the top of the loop that contains the inspector executor array access, we insert a barrier node in the control flow graph. Then we run single analysis on this modified control flow graph. Single analysis tells us if it is safe to place the barrier in the top of the loop. If it is not safe, then it is not safe to place the communication calls there, because the processor that needs the data and the processor that owns the data may come to the top of the loop at different times or for different number of times. If single analysis tells us that it is safe to place a barrier in the top of the loop, then we can place the communication calls there, because we are certain that all processors would come to the top of the loop around the same time for the same number of times. After running single analysis, we remove the barrier node from the control flow graph.

5.3. Overlap

Our generated code utilizes two types of overlap: communication with communication, and communication with computation. When a processor owns data that is needed by multiple remote processors, non-blocking puts are used to push the needed data to the remote processors. While waiting for the remote data to arrive, we can overlap the computation that only involves local elements or computation with elements that have already arrived.

6. Experimental Platforms

We performed experiments on three parallel machines and developed a performance model for the communication on each of them. The first, RTC, is a cluster of Itanium processors connected by a Myrinet network at Rice University. The second is an IBM Power3 system, Seaborg, which is at NERSC, and the last, Lemiuex, is an HP system at PSC. Table 1 contains a summary of their key features.

Name	System	Network	CPU
RTC	Linux cluster	Myrinet 2000	900 MHz Itanium 2
Seaborg	IBM RS/6000 SP	SP Colony Switch 2	375 MHz Power 3+
Lemieux	Compaq Alphaserver ES45	Quadrics Elan3	1 GHz Alpha

Table 1: machine summary

7. Runtime Selection of Communication

With a set of indirect array accesses to a remote array, there are several options for performing the data communication. The options are listed below:

- *Pack method*: only communicates the needed elements without duplicates. The needed elements are packed into a buffer before sending them to the processor that needs the data.
- *Bound method*: use a bulk put operation to send a bounding box that contains the needed elements.
- *Bulk method*: use a bulk put operation to send the entire array.

The three methods require different amount of set up work. The pack method needs to run the inspector phase to translate all the indirect array accesses into remote addresses and the bound method needs to run the inspector phase to compute the bounding box that contains all the needed elements. The bulk method does not require an inspector phase.

In this paper, we focus on the case where the communication pattern is repeated several times, in which case the cost of the inspector is amortized, so we always run the inspector in the first iteration. In this scenario the bulk method becomes a special case of the bounding box method, so we only discuss the pack and bound methods in the remainder of this paper.

Our experiments show that the choice of communication is both application and machine specific. The application determines the size of the array, number of accesses to that array, and the size of the bounding box. The machine gives different memory and communication costs. Our compiler generates code that can choose the best communication method at runtime based on a performance model.

The total time of a communication method consists of three parts:

1. the time spent on getting the data ready for communication in the remote processor
2. the communication time for sending the data
3. the time for reading the data out of local buffer in the executor loop

Both 1 and 3 are local processor costs, which are dominated by memory access times. We use cache and memory latency numbers provided by the vendor in a performance model. In the pack method, the needed elements are gathered into a buffer by the remote processor. The gathering from the source array is random access, while the storing of the elements into the buffer and the reading of indices are sequential access. In the bound method, no packing of the data is needed, since the entire bounding box is sent. Figure 4 shows the model components. N is the number of

distinct elements being packed, $L1_{line}$ and $L2_{line}$ are cache line sizes, and α_1, α_2 and α_{mem} are latencies for L1, L2, and memory, respectively. Cache line sizes are adjusted to match the word size in each formula.

buffer gather:

$N\alpha_1 + N / (L1_{line})(\alpha_2 - \alpha_1) + N / (L2_{line})*(\alpha_{mem} - \alpha_2)$

index read:

$N\alpha_1 + N / (L1_{line})(\alpha_2 - \alpha_1) + N / (L2_{line})*(\alpha_{mem} - \alpha_2)$

source:

if (source fits in L1) then $N\alpha_1$
else if (source fits in L2) then $N\alpha_2$
else $N\alpha_{mem}$

Figure 4: performance model for packing

To estimate the communication cost, we use a piecewise linear model. For large messages, the cost is a fixed per message latency plus a per Byte bandwidth cost. We found that using this simple linear model for small messages was not accurate enough, and instead use different latency and bandwidth terms in different size ranges. For example, in the GM network, the steps are due to the 4KB MTU size of the packets. Figure 5 shows how well our models fit the actual. The average error comparing our models with the actual on all three machines is less than 1%. These latency and bandwidth numbers are computed empirically for each machine.

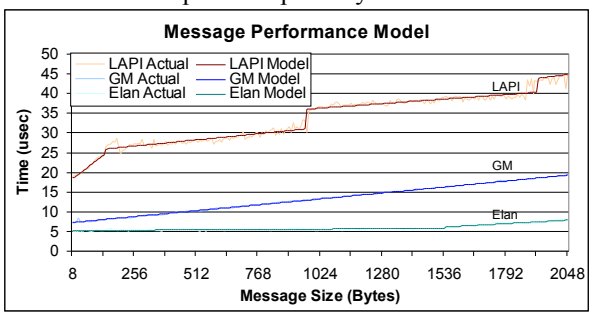

Figure 5: comparing the latency bandwidth models to actual on LAPI, GM, and Elan.

In practice, each processor may communicate with multiple other processors, while our model is for a single pair of processors and does not account for network contention. The number of simultaneous communication events depends on the application characteristics. While our point to point model does not capture this more complex communication behavior, we find it is sufficient for selecting a good communication method, as we will show in the performance section.

After the data arrives at the destination processor, the data is read out of the buffer during the executor phase in a random access pattern. In our model, we assume the costs for both methods are the same, although the bound method may suffer more cache misses in practice, because it is a larger data buffer than in the pack case

We pick the communication method that gives the lowest estimated cost by adding the packing and communication cost estimates. A choice is made separately for each processor pair. A schedule may contain several pairs, since each processor may need to communicate with several remote processors. The communication method selection automatically trades off network bandwidth and cache misses. By packing the needed elements into a buffer, the pack method uses a smaller message than the bounding box, but it incurs more memory traffic for the packing process.

8. Optimizing a Sparse Matrix Kernel

Indirect array accesses and the irregular memory and network access patterns that result are common in sparse matrix code. In this section we use a sparse matrix kernel, matrix vector multiplication, to evaluate our compiler and runtime techniques, which are entirely automatic. In this case, the matrix is sparse while both the source and result vectors are dense.

The parallel algorithm partitions the matrix by rows, with each processor getting a contiguous block of complete rows. Each processor also holds the corresponding piece of the result vector, so the only communication that is required is on the source vector. Because the source vector is often computed from an earlier result, it is partitioned in the same manner as the result vector. Figure 6 illustrates the layout of the matrix in the case with eight processors. Communication is only required for the source vector, and only for those elements in which a processor holds a nonzero outside of the processor's diagonal block. The off-diagonal nonzero shown will result in communication from P5 to P1, for example.

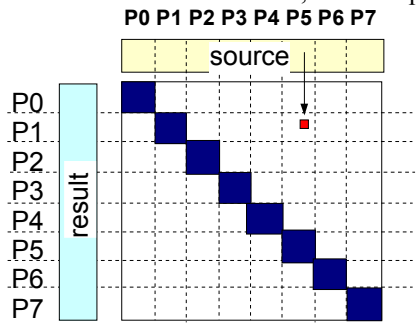

Figure 6: Parallel layouts of matrix and vectors

Due to the different nonzero structures of the matrices, the communication requirements vary widely

across matrices. We therefore use a set of benchmark matrices from real applications to evaluate our optimizations. Figure 7 gives two examples to illustrate the differences. On the left we have the nemeth21 from Matrix Market [12]. It is a 9506x9506 matrix with 1173746 nonzeros. Because the nonzeros occur only near the diagonal, each processor needs to communicate with at most two of its neighbors. The garon2 matrix on the right is taken from the UF Sparse Matrix Collection and is a 2D finite element method matrix [19]. The size is 13535x13535 and there are 390607 nonzeros. There is more data communication for this matrix than the previous one, because nonzeros are spread throughout the matrix, albeit in a regular pattern. Every processor will need some data from every other processor.

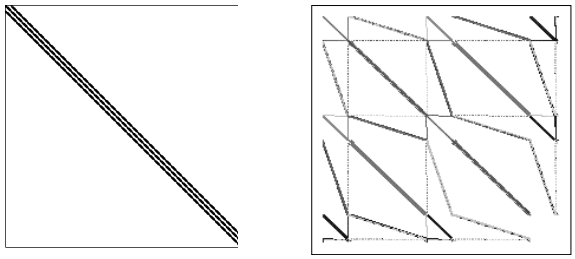

Figure 7: Structure of nemeth21 (left) and garon2 (right)

8.1 Evaluating Each Optimization

We begin by analyzing the performance of several different Titanium implementations using the garon2 matrix, which will highlight the differences in communication costs between the versions. The Titanium source code is the same across the versions, but the compiler and runtime support differ. Figure 8 shows the performance on the RTC Itanium/Myrinet cluster using the GASNet implementation for Myrinet's GM1 communication layer.

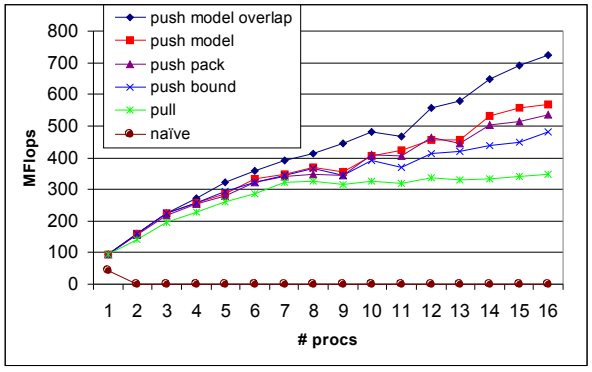

Figure 8: performance on the garon2 matrix

Naïve: With the naïve version, the generated code uses a remote read for each indirect array access when the array is remote. As expected, the performance does not scale.

Pull: The next line shows the benefit of an initial inspector executor optimization on this code. This significantly improves on the naïve version, because data is packed into larger messages. Remote gets are used in this case to pull data from the remote parts of the source vector.

Push: The next three versions of the code uses remote puts instead of gets for communication. On Myrinet GM1, puts are significantly faster than gets because puts are implemented using lower level RDMA support. Pulling requires a round-trip with remote work to pack the data, which requires that all processors be attentive to the network. The service of the packing request can be delayed if the remote processor is in the middle of a computation intensive loop when the request arrives.

Model vs Pure: The three push lines differ in the choice of communication mechanism. One uses the bounding box approach throughout the machine, while another uses packing throughout. The third line uses our performance model, which performs at least as well as the best of the pure methods. For some processor configurations, the model actually chooses a mixed strategy, such as in the 16 processors case. In a mixed strategy, a processor communicates with some neighbors using the bound method, and other neighbors with the pack method. This is the reason that we see the gap between the model performance and the maximum of the pure method performances. This result shows that applying the inspector executor optimization manually at the source code level can be a daunting task. Because there is a choice of communication methods between each pair of processors, the number of different choice configurations grows exponentially as the number of neighbor increases. Furthermore, the search for the best configuration would have to be done for each combination of input and processor configuration. We believe that it is a much cleaner solution to develop a performance model, and have the compiler to apply the inspector executor optimization automatically. The performance model only has to be developed once for a given machine.

Overlap: The last version of the code overlaps communication with computation. After sending the needed data using non-blocking puts, each processor does its local computation with its nonzeros that do not require communication. After the local computation completes, the processor polls on the arrival of the remote data as they are needed. On some matrices, there are sufficiently many nonzeros that do not require

communication that by the time the processor needs the incoming data, the data has already arrived. In those cases, the cost of communication is largely hidden behind the local computation.

8.2. Comparison with an MPI Library

In this section we compare the best Titanium implementation using the performance model and overlapped communication with an MPI implementation that uses the same basic data layout and algorithm for sparse matrix-vector multiplication. The MPI implementation is from a popular sparse solver library called Aztec, which is written in C [16].

We use matrices from Matrix Market and the UF Sparse Matrix Collection. Table 2 lists some of the matrices along with the dimensionality (all the matrices are square) and the number of nonzeros.

#	Name	Dim (NxN)	Non-zeros	Area
1	appu	14000	1853104	Structures
2	av41092	41092	1683902	Unknown
3	barrier2-1	113076	2129496	Physics
4	bbmat	38744	1771722	Structures
5	cage12	130228	2032536	Biology
6	cfd2	123440	3085406	Graphics
7	crystk02	13965	968583	Structures
8	crystk03	24696	1751178	Structures
9	lin	256000	1766400	Eigenval
10	nasasrb	54870	2677324	Structures
11	nemeth21	9506	1173746	Chemistry
12	nemeth22	9506	1358832	Chemistry
13	nemeth23	9506	1506810	Chemistry
14	nemeth24	9506	1506550	Chemistry
15	nemeth25	9506	1511758	Chemistry
16	nemeth26	9506	1511760	Chemistry
17	oilpan	73752	2148558	FEM
18	qa8fk	66127	1660579	FEM
19	qa8fm	66127	1660579	FEM
20	t3dh_a	79171	4352105	Physics
21	t3dh_e	79171	4352105	Physics
22	vanbody	47072	2329056	FEM

Table 2: matrix characteristics

Programmability differences between C+MPI and Titanium are difficult to quantify. The Titanium code contains only array accesses and field dereferences, whereas the MPI code has explicit send and receive routines as well as code to pack required source vector elements into buffers. Lines of source code, while far from perfect, may provide some insight into the differences in programming difficulty. The C program using Aztec is 55% longer than the Titanium code.

We performed experiments on the three machines described earlier: RTC, Seaborg, and Lemieux. The Titanium implementation uses tuned GASNet implementations for each of the three message layers: GM1 on Myrinet (RTC), Elan3 on Quadrics (Lemieux) and LAPI on the IBM SP (Seaborg). Aztec uses a pure packing approach for communication. We use different processor configurations from 1 to 16 processors, and always use only one processor per node. Extending the performance model to handle clusters of SMPs remains as future work.

Figures 9 through 11 show the average and maximum speedup of the Titanium version relative to the Aztec version on 1 to 16 processors. In general, the Titanium version is faster, sometimes by more than a factor of 2x. Across all processor configurations, matrices, and machine, the Titanium code was an average of 1.2x faster than the MPI code. The speedups are highest on the Myrinet machine, where the RDMA support used by GASnet is most significant. The Quadrics network is fast for both MPI and GASNet, and for matrices with less communication, you see little difference between the two languages. There are some cases where the MPI code outperforms the Titanium code, usually on smaller problem sizes. We continue to investigate issues related to barrier performance and serial code differences that account for these slowdowns, and presumable additional tuning for these machines may also be possible in the Aztec code.

Our overall summary is that the performance of the Titanium code is usually better than that of MPI and from the analysis in section 8.1, we can attribute these differences to the combination of the RDMA support in GASNet, avoiding the cost of packing for some matrices, and the use of a performance model to select the best communication mechanism for each pair of processors, and the effectiveness of the overlap using the RDMA communication model.

9. Conclusions

In this paper, we have described an automatic transformation of programs with indirect array accesses in Titanium to take advantage of inspector executor style optimizations. We introduced a performance modeling technique to select communication methods using a combination of data collected at compiler install time and runtime information about the application's access patterns. This allows application programmers to write Titanium code in a straightforward way and obtain performance superior

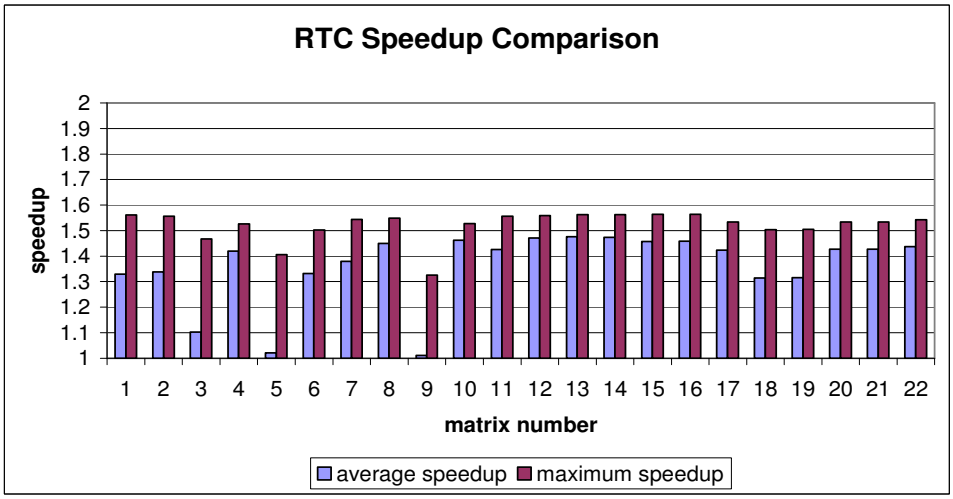

Figure 9: performance comparison between Titanium and Aztec on Linux cluster

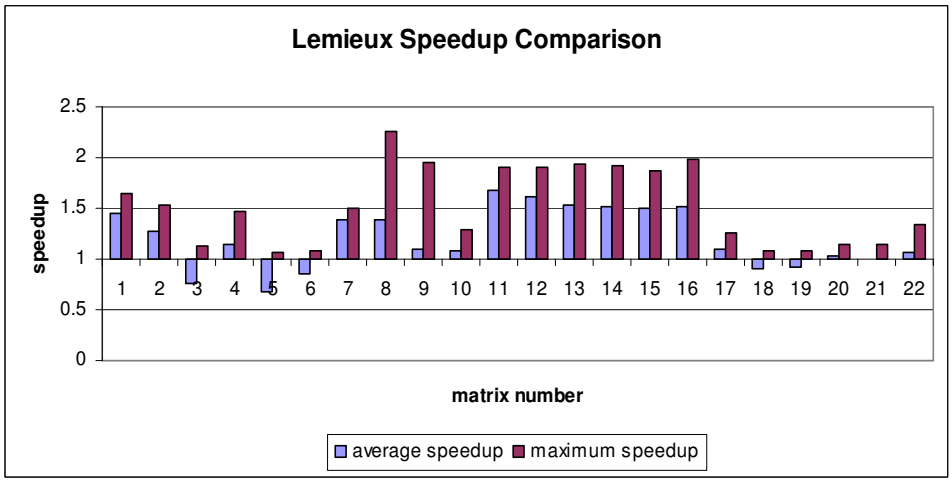

Figure 10: performance comparison between Titanium and Aztec on Compaq Alphaserver

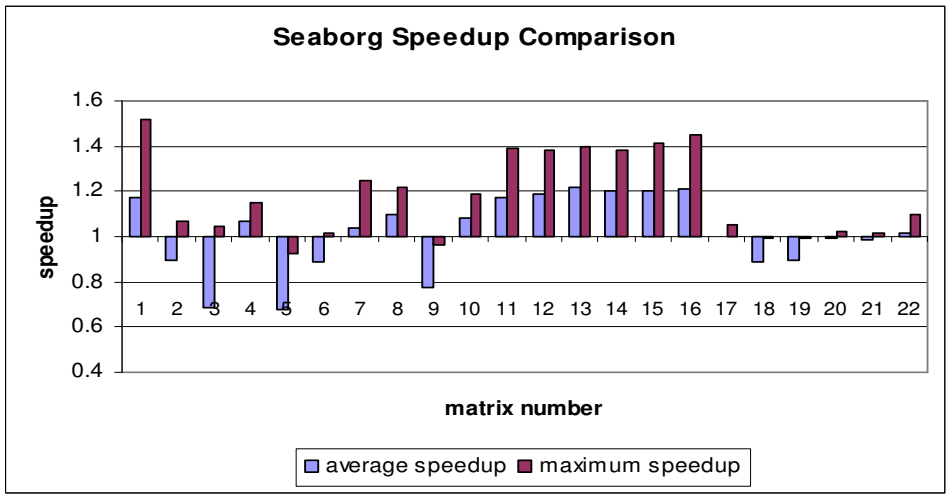

Figure 11: performance comparison between Titanium and Aztec on IBM SP

to a popular hand-tuned library. In particular, for a sparse matrix vector multiply problem, we showed that not only is the program more concise, but the optimized Titanium code is up to 2.25x faster. In future work we plan to extend the optimizations to multidimensional arrays, and analyze its usefulness for particle-mesh methods as well as extending the communication performance model to handle hybrid shared and distributed memory machines.

These results show the feasibility of using a high level language for high performance scientific programming on programs with indirect array accesses on remote arrays. In other papers we have demonstrated the use of Titanium on large applications, including a heart simulation [22] and adaptive mesh refinement [21] problems. The use of a high level parallel language enables compiler optimizations, like the one described in this paper, that cannot be done automatically in a library-based approach like MPI.

10. References

[1] S. Baden, and S. Fink, "The Data Mover: A Machine-Independent Abstraction for Managing Customized Data Motion", *LCPC,* 1999.

[2] H. Berryman, and J. Saltz, "A manual for PARTI runtime primitives", 1990.

[3] S. Benkner, "Optimizing Irregular HPF Applications Using Halos", *Irregular,* 1999.

[4] D. Bonachea, "GASNet specifications", 2003.

[5] S. Chakrabarti, J. Demmel, and K. Yelick, "Modeling the benefits of mixed data and task parallelism", *Symposium on Parallel Algorithms and Architectures*, 1995.

[6] R. Das, J. Saltz, and R. v. Hanxleden, "Slicing analysis and indirect accesses to distributed arrays", *Workshop on Languages and Compilers for Parallel Computing*, 1993.

[7] R. Das, M. Uysal, J. Saltz, and Y. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures", *Journal of Parallel and Distributed Computing*, 1993.

[8] J. Gosling, B. Joy, and G. Steele, "The Java language specification", 2000.

[9] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz, "Compiler Analysis for Irregular Problems in Fortran D", *Workshop on Languages and Compilers for Parallel Computing*, 1992.

[10] P. Hilfinger et al, "Titanium language reference manual", 2001.

[11] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "Supporting shared data structures on distributed memory machines", *Symposium on Principles and Practice of Parallel Programming*, 1990.

[12] Matrix Market, http://math.nist.gov/MatrixMarket.

[13] G. Pike, and P. Hilfinger, "Reordering and Storage Optimizations for Scientific Programs", *Supercomputing*, 2002.

[14] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz, "Run-time and compile-time support for adaptive irregular problems", *Supercomputing*, 1994.

[15] M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings", *Programming Language Design and Implementation*, 2003.

[16] R. Tuminaro, M. Heroux, S. Hutchinson, and J. Shadid, "Official Aztec user's guide: version 2.1", 1999.

[17] S. Vadhiyar, G. Fagg, and J. Dongarra, "Performance Modeling for Self Adapting Collective Communications for MPI", *LACSI Symposium*, 2001.

[18] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani, "Distributed memory compiler design for sparse problems", 1991.

[19] UF Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices.

[20] D. Walker, "The Implementation of a Three-Dimensional PIC Code on a Hypercube Concurrent Processor", *Conference on Hypercubes, Concurrent Computers, and Application*, 1989.

[21] T. Wen, and P. Colella, "Adaptive Mesh Refinement in Titanium", *IPDPS*, 2005.

[22] S. Yau, "Experiences in Using Titanium for Simulation of Immersed Boundary Biological Systems", 2002.

[23] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect", *Workshop on Java for High-Performance Network Computing*, 1998.

Acknowledgement

This work was supported in part by the Department of Energy under DE-FC03-01ER25509, by the National Science Foundation under ACI-9619020, ACI-0090127, and CNS-0325873, by the California State MICRO Program, by Sun Microsystems, and by an NDSEG fellowship. Experiments were performed on facilities provided by Rice University, the National Energy Research Scientific Computing Center, and the Pittsburgh Supercomputing Center. Thanks go to the members of the Titanium research group, who provided valuable suggestions and feedbacks on this work. We would also like to thank the anonymous reviewers for their helpful comments on the original submission.