Array Prefetching for Irregular Array Accesses in Titanium

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

Abstract

Compiling irregular applications, such as sparse matrix vector multiply and particle/mesh methods in a SPMD parallel language is a challenging problem. These applications contain irregular array accesses, for which the array access pattern is not known until runtime. Numerous research projects have approached this problem under the inspector executor paradigm in the last 15 years. The value added by the work described in this paper is in using performance modeling to choose the best data communication method in the inspector executor model. We explore our ideas in a compiler for Titanium, a dialect of Java designed for high performance computing. For a sparse matrix vector multiply benchmark, experimental results show that the optimized Titanium code has comparable performance to C code with MPI using the Aztec library.

1. Introduction

Irregular array accesses arise in many scientific applications such as sparse matrix vector multiply and particle/mesh methods. Figure 1 illustrates a simple example. A is the data array, and B is the indirection array. The array access pattern to A is not known until runtime. When this code is run on a distributed memory machine, array A can be remote, in which case reading the elements of A would require communication. Walker proposed and implemented the idea of using a precomputed communication schedule for indirect array accesses to distributed arrays in a PIC application [18]. His targeted architecture was a hypercube in a crystalline environment, where data need to be forwarded by intermediate processors in order to be transmitted from the source to destination processor. The idea was further developed by Berryman and Saltz [5] in a technique called *inspector executor*. During program execution, the inspector examines the data references made by a processor, and calculates which off-processor data needs to be fetched and where this data will be stored once it is received. The executor loop then uses the gathered data to perform the actual computation.

```
for i = 1 to n do
  sum += A[B[i]]
end do
```

Figure 1: simple example of indirect array access

After the inspector phase, the access pattern is known. In some applications, the communication pattern is reused multiple times, so the runtime may spend some time determining an efficient way to perform the communication. Our experiments show that the best method for doing this communication is both application and machine specific. For some problem sizes, it may even be better to skip the inspector phase entirely, and simply retrieve the entire data array.

We explore our ideas in a compiler for Titanium [19]. Titanium is a dialect of Java designed for high-performance parallel computing using a SPMD execution model. It has extensions for scientific computing, including a multidimensional array construct that we use in this paper. Our compiler can automatically apply the inspector executor optimization to indirect array access code. The generated code is able to accurately choose the best fetch method during runtime to perform the data communication using performance modeling. When the access pattern is repeated over multiple iterations, further optimizations such as schedule reuse, buffer reuse, overlap communication with communication, and overlap communication with computation are applied to the code.

To demonstrate the effectiveness of the optimizations, we develop a sparse matrix vector multiply benchmark in Titanium, and compare the performance of the optimized Titanium code to C code with MPI using the Aztec library [14]. Titanium code is written at a higher level of abstraction and is more concise, yet their performance is comparable.

2. Related Work

The inspector executor technique was developed by Berryman and Saltz. The PARTI runtime library [1] and its successor CHAOS [12] 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 communication schedule can be generated, where communication operations are placed, and when schedules can be combined [7]. As an experimental result, they carried out the optimizations that would have been suggested by the dataflow framework manually. The ARF [16] and KALI [9] 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 [4]. 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 [13].

There have been numerous research works in the area of communication scheduling. Chakrabarti *et al.* [3] implemented an algorithm for optimizing communication schedules across loops in a global manner in the HPF compiler. Dongarra *et al.* [15] 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 without explicit communication. The choice is both application and machine specific. Our compiler is able to automatically generate code that can accurately choose the best communication method during runtime based on performance modeling.

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 the C compiler compiles the generated C code. C is used as an intermediate step for portability. Titanium runs on a wide range of platforms, including uniprocessors, shared memory multiprocessors, distributed-memory clusters of uniprocessors or SMPs (CLUMPS), and a number of specific supercomputer architectures (Cray T3E, IBM SP, Origin 2000).

3.1. Memory Consistency Model

Titanium inherits many features of Java, one of which is the Java memory consistency model [6]. Titanium's interpretation of the Java memory consistency model is defined in the language specification [8]. Here are some informal properties of the Titanium model.

- Locally sequentially consistent: For a single processor, all reads and writes to a given memory location must appear to occur in exactly the order specified.
- 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, subsequent reads by that processor on those array elements must return the new value. The second property makes data prefetched prior to a synchronization point unusable after that synchronization point. The prefetched array elements may have changed, and reads after the synchronization point must reflect those changes.

3.2. Foreach Loop

Our transformation targets the built-in foreach loops in Titanium. A foreach loop has the form of *foreach (p in D) S*, where the iteration space *D* is a rectangular set of positive integers, and *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. Titanium compiler does extensive analysis and optimizations for foreach loops [11]. The relevant analysis for our purposes is dominator analysis for foreach loops. The classical dominator analysis is used to determine the following:

- Whether a foreach loop is a full domain loop or a partial domain loop. A full domain loop executes on every iteration and cannot be cut short except by a fatal error.

- Which array accesses appear on every iteration of the loop.

4. Inspector Executor in Titanium

4.1. Compile Time Transformations

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

- The array access appears in full domain foreach loops.
- The array access appears on every iteration of the loop.
- B and B[i] do not change inside of the loop.
- A does not change inside of the loop. The reason that A[i] can change is that the conflicts can be resolved by merging against the prefetched data in runtime when A[i] is written to.
- 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 prefetch 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 fetch method is chosen to retrieve the remote data into a local buffer. More details on the choice of the fetch method are presented in the next section. In the executor loop, values for each A[B[i]] are read out of the local buffer.

4.2. Runtime Selection of Fetch Methods

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:

- *Gather method*: use a gather operation to retrieve all the needed elements. The gather operation is a one-sided operation that copies selected elements from a remote array into a local buffer.
- *Bound method*: use a bulk read operation to retrieve a bounding box that contains the needed elements.
- *Bulk method*: use a bulk read operation to retrieve the entire remote array.

The three methods require different amount of set up work:

- The *gather method* needs to run the inspector phase to translate all the indirect array accesses into remote addresses.
- 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 need to run the inspector phase.

We define the best fetch method as the method that takes the least amount of time to complete. Our experiment shows that there is no best method for the general case. The choice 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 latencies for completing a bulk read operation and a gather operation. Without help from the compiler, the application programmer would need to make this decision at the application level, and have three different branches that handle the three fetch methods. This makes the application code much less readable and non-portable. We would like to have the compiler generate code that can choose the best fetch method at runtime based on performance modeling numbers collected on the particular machine.

The total time of a fetch method consists of two parts: the local computation that runs the inspector phase and the communication that actually retrieves the elements from the remote processor. We develop a performance model to account for these two costs. For modeling the local computation, we empirically measure the cost for computing for a single array access in the inspector phase. For modeling communication, we measure the latencies of the gather and bulk read operation with different processor configurations. These empirical measurements only need to be done once when the compiler is built on a particular machine. When an application runs, it simply looks in a table containing this data to make a decision on which fetch method to use.

The decision process for choosing a fetch method happens in two stages. The *bulk method* differs from the *gather* and *bound methods* in that it does not need to run the inspector. Therefore, before running the inspector phase, we need to decide if the *bulk method* is the best choice. The costs of each method are computed as follows:

N:	number of indirect array accesses
T1:	time spent on a single array access in the inspector
T2:	communication time for gathering N elements from a remote array

T3: communication time for bulk reading the bounding box
T4: communication time for bulk reading the entire remote array

Gather method: total time = N*T1 + T2
Bound method: total time = N*T1 + T3
Bulk method: total time = T4

Before running the inspector, the exact size of the bounding box is not known, so we approximate it by the number of array accesses. If the *bulk method* is chosen at this stage, we will skip the inspector phase and use the *bulk method*, otherwise we go on to execute the inspector.

After the inspector phase completes, we need to make a choice between the two remaining candidates: *gather* and *bound*. At this time, we know exactly how big the bounding box is, so we do not need any approximations in computing the cost. We choose the method with the lowest cost.

5. Schedule Reuse

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 fetch method are stored in the schedule. Schedule reuse has been used in prior work, but our schedules contain additional information about the fetch method to be employed.

Schedule reuse enables several optimizations. It amortizes the cost of the inspector over multiple iterations. It also allows the performance model to be more accurate. Since the cost of the inspector is amortized, we always run the inspector during the first iteration, so we no longer need to use an approximation to the size of the bounding box. In fact, we have reduced the two-stage decision process into a single stage process. Because the communication cost of the *bound method* is always less or equal to the communication cost of the *bulk method*. Local buffers used for storing the retrieved data can also be reused.

More importantly, schedule reuse gives us opportunities to overlap communication with other communication or with computation. For example, an application may have repeated patterns of indirect array accesses to an array distributed over multiple processors. In that case, we can overlap the communication for fetching elements from different processors. We can also overlap the computation that only involves local elements or computation with elements that have been fetched with data communication. We have found that limiting the number of outstanding fetch calls helps performance, which makes the choice of communication schedule more difficult.

6. Experimental Results

6.1. Experimental Setup

Experiments were performed on three supercomputers: Alvarez, Seaborg, and Lemieux. Table 1 contains a summary of the three machines, and some of their key attributes.

Name	System	Network	CPU
Alvarez	IBM Netfinity cluster	Myrinet 2000	866 MHz Pentium III
Seaborg	IBM RS/6000 SP	SP Switch 2	375 MHz Power 3+
Lemieux	Compaq Alphaserver ES45	Quadrics Elan3	1 GHz Alpha

Chart 1: machine summary

6.2. Performance Results

We developed the following simple benchmark to test our performance model. A is the remote array, and B is the indirection array.

```
foreach (p in B.domain()){
  sum += A[B[p]];
}
```

Figure 2: simple indirect sum benchmark

We varied three parameters during the experiment: size of the array A, minimum and maximum index needed from array A that defines the span, and the number of array accesses. For each problem size, the experiment ran all three methods separately and recorded the timing for each method. We ran this experiment on two nodes with one processor on each node for the three machines. There are a total of 5120 problem sizes in the experiment. Below is the chart that shows the number of times each fetch method is the best choice for the three different machines according to the measured data.

	Alvarez	Seaborg	Lemieux
Gather	438 (9%)	249 (5%)	0 (0%)
Bound	2415 (47%)	3174 (62%)	3414 (67%)
Bulk	2267 (44%)	1697 (33%)	1706 (33%)

Chart 2: the number of times (and percentages) each fetch method is the best choice for a given problem size, span, and number of accesses.

This data shows that there is no best method for the general case. The data also shows an average of 150% slowdown when the worst method is chosen instead of the best method. This suggests that the choice of fetch method is important.

To explain the different behavior across machines, we examine the network hardware in detail. Bulk read operations on Lemieux are implemented using RDMA (Remote Direct Memory Access) gets, which are supported natively in the network hardware on the Quadrics network. During the bulk get operation, the remote processor is not involved, as the network hardware entirely handles servicing the memory request at the remote end. In contrast, there is currently no hardware support for the gather operation on Quadrics. The software implementation of the gather operation includes a network roundtrip and packing done by the remote processor. This explains the data obtained on Lemieux, which does not have the gather method as the best method for any of the test cases. At the time of the experiment, there was no hardware support for bulk get on Myrinet, but hardware support has been added since then. There is no hardware support for bulk get on the SP.

6.3. Using the Performance Model

Next, we use our performance model to see how well it can select the fetch method. For the data collected on Alvarez, the performance model would have chosen the best method 82% of time, the second best method 17% of the time, and the worst method less than 1% of the time. When it chooses the second best method, it pays a performance penalty of 13% slowdown on average. When it chooses the worst method, it pays a performance penalty of 6% slowdown on average. The fact that the penalty for choosing the worst method is smaller than the penalty for choosing the second best method may seem counterintuitive. Upon closer examination of the data, all three methods have very close numbers in cases where the performance model chooses the worst method.

6.4. Sparse Matrix Vector Multiply

In this section, we present data from a sparse matrix vector multiply benchmark in Titanium. We use the compiler to apply the transformations that we have talked about in this paper automatically to this code. The Titanium code uses the GASNet [2] gm backend for communication. As a point of comparison, we also have a C program that does sparse matrix vector multiply by calling a routine in Aztec. Aztec is a library that provides algorithms for the solution of large sparse linear systems. It is written in C. It uses MPI to perform data communication. In terms of the source code size, the C program using Aztec is 55% more than the Titanium code. Both the Titanium and Aztec algorithm partition the matrix by row. Figure 3 illustrates the layout of the matrix in the case with eight processors. Communication is only required for the source vector. A processor needs to fetch a source element for every nonzero outside of its diagonal block.

We run our experiment on Alvarez with various processor configurations. Each run consists of 10000 iterations of sparse matrix vector multiply to offset the granularity of the timer. We present the descriptions of the matrices and the graphs comparing the performance of the two algorithms in the next two sections.

6.4.1. Matrices

We obtained the bcsstk16 matrix from Matrix Market [10]. It is a 4884x4884 matrix with 147631 nonzeros. The nonzeros are concentrated on the diagonal. Due to the location of the nonzeros, each processor has to do data communication with at most two of its neighbors.

The garon2 matrix is taken from the UF Sparse Matrix Collection [17]. It is a 2D finite element method matrix. The size is 13535x13535. There are 390607 numbers of nonzeros. There is more data communication for this matrix than the previous one. Every processor needs some data from every other processor.

The third matrix is a random matrix. The size is 4000x4000. Each row has 40 randomly distributed nonzeros.

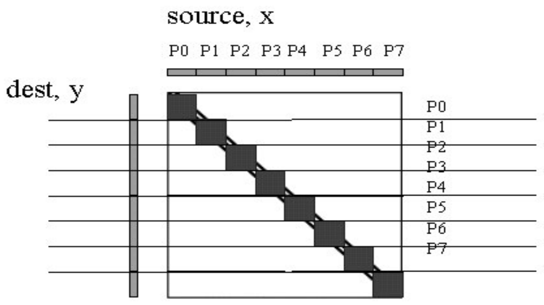

Figure 3: layouts of matrix and vectors

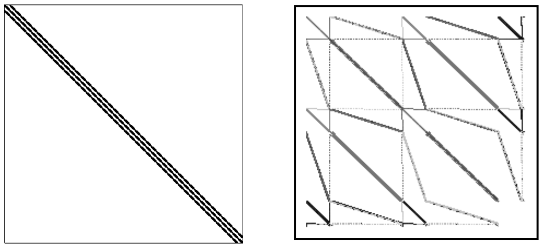

Figure 4: the picture for bcsstk16 is on the left, and the picture for garon2 is on the right.

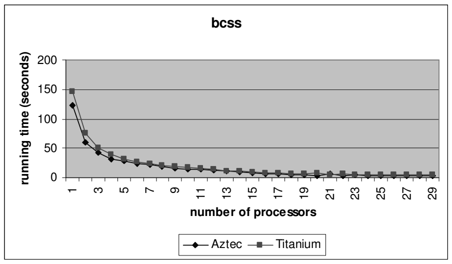

Figure 5: performance graph for bcsstk16

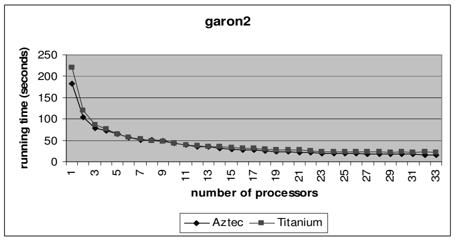

Figure 6: performance graph for garon2

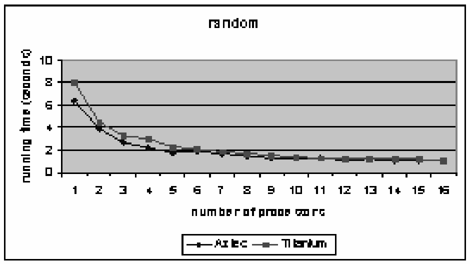

Figure 7: performance graph for random matrix

6.4.2. Performance Analysis

Figure 5, 6, and 7 display the performance graphs comparing Titanium and Aztec on different processor configurations for all three matrices. The performance of Titanium is 15% to 20% slower than Aztec in the single processor case, but the gap shrinks as the number of processors increases. Titanium's serial performance is slightly worst than Aztec, but its communication performance allows it to catch up to Aztec in larger processor configurations. In some large processor configurations, Titanium's performance actually beats Aztec slightly. The reason for the difference in serial performance is the overhead due to the generality of Titanium arrays. Recent optimizations have eliminated most of the overhead, but for a problem with indirect array accesses, the slowdown remains at 20%. There is little or no overhead for the regular case.

7. Conclusion

In this paper, we have described the automatic transformation of irregular array access code in Titanium to take advantage of inspector executor style optimizations and schedule reuse optimizations. We introduced a new 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 get performance comparable to a popular hand-tuned library. In particular, for a sparse matrix vector multiply benchmark, we showed that the optimized Titanium code has comparable performance to C code with MPI using the Aztec library. This serves as a first step toward providing support for irregular applications in Titanium.

10. Acknowledgements

This work was supported in part by the Department of Energy under DE-FC03-01ER25509, by the National Science Foundation under ACI-9619020 and ACI-0090127, and an NDSEG fellowship.

We wish to thank the Pittsburgh Supercomputing Center for the use of their Alpha-based Quadrics system (Lemieux), and the National Energy Research Scientific Computing Center (NERSC) for the use of their IBM SP (Seaborg) and Myrinet 2000 cluster (Alvarez). Thanks also go to the members of the Titanium research group, who provided valuable suggestions and feedbacks on this work. We would like to thank the four anonymous reviewers for their helpful comments on the original submission.

9. References

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

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

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

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

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

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

[12] 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.

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

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

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

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

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

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

[19] 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.