Automatic Communication Performance Debugging in PGAS Languages

Jimmy Su[1] and Katherine Yelick[1,2]

[1] Computer Science Division, University of California at Berkeley
[2] Lawrence Berkeley National Laboratory
{jimmysu, yelick}@cs.berkeley.edu

Abstract. Recent studies have shown that programming in a Partition Global Address Space (PGAS) language can be more productive than programming in a message passing model. One reason for this is the ability to access remote memory implicitly through shared memory reads and writes. But this benefit does not come without a cost. It is very difficult to spot communication by looking at the program text, since remote reads and writes look exactly the same as local reads and writes. This makes manual communication performance debugging an arduous task. In this paper, we describe a tool called `ti-trend-prof` that can do automatic performance debugging using only program traces from small processor configurations and small input sizes in Titanium [13], a PGAS language. `ti-trend-prof` presents trends to the programmer to help spot possible communication performance bugs even for processor configurations and input sizes that have not been run. We used `ti-trend-prof` on two of the largest Titanium applications and found bugs that would have taken days in under an hour.

Keywords: PGAS languages, automatic performance debugging

1 Introduction

Titanium is a Partitioned Global Address Space language. It combines the programming convenience of shared memory with the locality and performance control of message passing. In Titanium, a thread running on one processor can directly read or write the memory associated with another. This feature significantly increases programmer productivity, since the programmer does not need to write explicit communication calls as in the message passing model. Unfortunately, this is also a significant source of performance bugs. Many unintended small remote reads and writes go undetected during manual code audits, because they look exactly the same as local reads and writes in the program text. Furthermore, these performance bugs often do not manifest themselves until the program is run with large processor configurations and/or large input sizes. This means the bugs are caught much later in the development cycle, making them more expensive to fix.

In this paper, we describe an automatic communication performance debugging tool for Titanium that can catch this type of bugs using only program runs with small

processor configurations and small input sizes. Trends on the number of communication calls are presented to the programmer for each location in the source code that incurred communication during the program runs. Each trend is modeled by a linear function or a power law function in terms of the number of processors or the input problem size. The models can be used to predict communication performance bottlenecks for processor configurations and problem sizes that have not yet been run. We used the debugging tool on two of the largest Titanium applications and report the bugs that were found using the tool.

2 Motivating Example

To illustrate the difficulty of manual performance debugging in a PGAS language like Titanium, we will use a simple sum reduction example in this section. Processor 0 owns a double array. We would like to compute the sum of every element in the array. To spread the workload among the processors, each processor gets a piece of the array and computes the sum for that part. At the end, the partial sums are added together using a reduction.

Two versions of the code are shown in Figure 1 and Figure 2. The code in Figure 1 has a performance bug in it. The two versions are identical except for two lines of code. The loop that computes the actual sum is identical. In the buggy version, each processor only has a pointer to the array on processor 0. `array.restrict(myPart)` returns a pointer to a subsection of `array` that contains elements from `startIndex` to `endIndex`. Each dereference in the foreach loop results in communication to processor 0 to retrieve the value at that array index. Processor 0 becomes the communication bottleneck as all other processors are retrieving values from it.

```
1  double [1d] array;
2  if (Ti.thisProc() == 0){
3      array = new double[0:999];
4  }
5  array = broadcast array from 0;
6  int workload = 1000 / Ti.numProcs();
7  if (Ti.thisProc() < 1000 % Ti.numProcs()){
8     workload++;
9  }
10 int startIndex = Ti.thisProc() * workload;
11 int endIndex = startIndex + workload - 1;
12 RectDomain<1> myPart = [startIndex:endIndex];
13 double [1d] localArray = array.restrict(myPart);
14 double mySum = 0;
15 double sum;
16
17 foreach (p in localArray.domain()) {
18    mySum += localArray[p];
19 }
```

```
20  sum = Reduce.add(mySum, 0);
```

Fig. 1. Sum reduction example with performance bug in it (Version 1)

```
12  RectDomain<1> myPart = [startIndex:endIndex];
13  double [1d] localArray = new double[myPart];
14  localArray.copy(array.restrict(myPart));
15  double mySum = 0;
16  double sum;
17
18  foreach (p in localArray.domain()) {
19    mySum += localArray[p];
20  }
21
22  sum = Reduce.add(mySum, 0);
```

Fig. 2. Sum reduction example without the performance bug (Version 2)

Figure 2 shows the version without the performance bug in it. Each processor first allocates space for the `localArray`, then it retrieves the part of `array` that it needs into `localArray` using one array copy call. The array copy results in one bulk get communication. The subsequent dereferences inside the loop are all local.

Although this is a very simple example, this kind of communication pattern is quite common, especially in the initialization phase of a parallel program, where processor 0 typically processes the input before distributing the workload to the rest of the processors. It is difficult to catch this type of bugs manually in Titanium, since the two versions of the program look very similar. For small processor configurations, the performance degradation may not be noticeable given that the initialization is run only once.

We would like a tool that can alert the programmer to possible performance bugs automatically earlier in the development cycle, when we are only testing the program with small processor configurations and small input sizes. For this example, the number of communication calls at the array dereference in the buggy version can be expressed as `(1-1/p)*size`, where `p` is the number of processors and `size` is the size of the array. If we fix the array size at 1000 elements, then we can see that the number of communication calls at the array dereference varies with the number of processors as in Figure 3. The graph shows the actual observed communication calls at the array dereference for 2, 4, and 8 processors along with the predicted curves for both versions of the code.

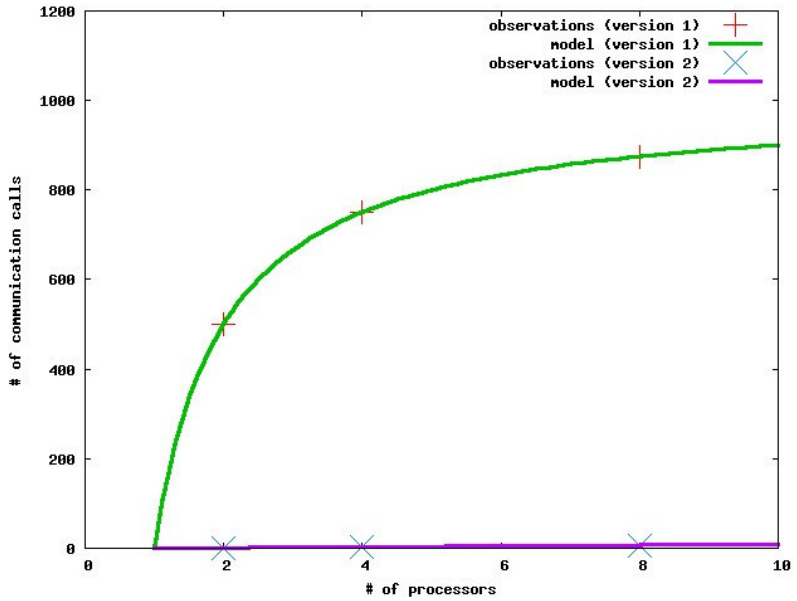

Fig. 3. The number of communication calls at the array dereference is expressed in terms of the number of processors for a fixed array size of 1000 elements for both versions of the program. The X axis is the number of processors, and the Y axis is the number of communication calls. Version 1 is clearly not scalable. For larger array sizes, the gap between version 1 and version 2 would widen.

In the rest of this paper, we will describe a tool called `ti-trend-prof` that can present communication trends automatically given only program traces for small processor configurations and/or small input sizes.

3 Background

Before getting into the details of `ti-trend-prof`, we will give the necessary background information in this section. This includes brief introductions on Titanium, GASNet trace, and `trend-prof`.

3.1 Titanium

Titanium is a dialect of Java, but does not use the Java Virtual Machine model. Instead, the end target is assembly code. For portability, Titanium is first translated into C and then compiled into an executable. In addition to generating C code to run on each processor, the compiler generates calls to a runtime layer based on GASNet [2], a lightweight communication layer that exploits hardware support for direct

remote reads and writes when possible. 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).

Titanium is a single program, multiple data (SPMD) language, so all threads execute the same code image. A thread running on one processor can directly read or write the memory associated with another. This feature significantly increases programmer productivity, since the programmer does not need to write explicit communication calls as in the message passing model.

3.2 GASNet Trace

Titanium's GASNet backends include features that can be used to trace communication using the GASNet trace tool. When a Titanium program is compiled with GASNet trace enabled, a communication log is kept for each run of the program. In this communication log, each communication event along with the source code line number is recorded.

3.3 `trend-prof`

`trend-prof` [7] is a tool developed by Goldsmith, Aiken, and Wilkerson for measuring empirical computational complexity of programs. It constructs models of empirical computational complexity that predict how many times each basic block in a program runs as a linear or a power law function of user-specified features of the program's workloads. An example feature can be the size of the input. It was previously used on sequential programs for performance debugging.

4 Bug Types

In parallel programming, there are many causes for communication performance bugs. This includes excessive amount of communication calls, excessive volume of communication, and load imbalance. Our work so far in `ti-trend-prof` has been focused on finding the first type of bugs automatically. Our framework can be extended to address the other two types of bugs. In Titanium, there are two main causes for excessive amount of communication calls:

1. Remote pointer dereference
2. Distribution of global meta-data

The first case can come up in two situations. One is when a processor has a shallow copy of an object that contains remote references in its fields. Even though the object is in local memory, accessing its field that contains remote reference would result in a round trip of small messages to a remote processor. If the field is accessed frequently during program execution, it can significantly degrade performance. The second situation comes up during workload distribution among

processors. In parallel program, it is often the case that one processor does I/O during initialization, and then the workload is distributed among all processors. The motivating example in Section 2 fits this description.

The second case comes from distribution of global meta-data. In parallel programs, it is often desirable to have global meta-data available to each processor so that it can find remote objects by following pointers. Each processor owns a list of objects. A naïve way of programming the distribution of meta-data is by broadcasting each pointer individually. This performance bug would not be noticeable when the number of objects is small. Only a large problem size would expose this problem, which is likely to be much later in the development cycle.

In the experimental section, we will show that these types of performance bugs exist in two of the largest Titanium applications written by experienced programmers, and `ti-trend-prof` allowed us to find the bugs automatically within an hour instead of days through manual debugging.

4 ti-trend-prof

In this work, a new tool called `ti-trend-prof` is developed to combine the use of GASNet trace and `trend-prof` to do communication performance debugging for parallel programs. `ti-trend-prof` takes GASNet trace outputs for small processor configurations and/or small input sizes, and feeds them to a modified version of `trend-prof` that can parse GASNet trace outputs. The output is a table of trends per Titanium source code location that incurred communication for the input traces.

The number of processors and the input problem size can be used as features. The linear function `a + bx` and the standard power law function with offset `a + bx^c` are used to model the trend at each source code location. The function which minimizes error is picked to be the model. For example, if we fixed the problem size and varied the number of processors, then the trend would tell us how does the number of communication calls change at this location as we vary the number of processors. Similarly, if we fixed the number of processors and varied the problem size, then the trend would tell us how does the number of communication calls change as we vary the problem size. These trends can be used to predict communication performance bottlenecks for processor configurations and input sizes that we have not run yet. This is particularly useful in the beginning of the development cycle, where we do most of the testing on small processor configurations and small inputs. In the table, the trends are first ranked by the exponent, then by the coefficient. Larger values are placed earlier in the table. The goal is to display trends that are least scalable first to the programmer.

In practice, many of the communication patterns can be modeled by the linear function or the power law function. But there are algorithms that do not fall into this category, such as a tree based algorithms or algorithms that change behavior based on the number of processors used. We don't intend to use the linear or power law trends as the exact prediction in communication calls, but rather as an indicator for possible performance bugs. For example, if the number of communication calls at a

location is exponential in terms of the number of processors, then `ti-trend-prof` would output a power law function with a large exponent. Although this does not match the actual exponential behavior, it would surely be presented early in the output to alert the programmer.

5 Experimental Results

In this section, we show the experimental results on running `ti-trend-prof` on two large Titanium applications: heart simulation [6] and AMR [12]. To obtain the GASNet trace files, the programs were run on a cluster, where each node has a dual core Opteron. We used both cores during the runs. This means that intra-node communication is through shared memory, which does not contribute to communication calls in the GASNet trace counts.

5.1 Heart Simulation

The heart simulation code is one of the largest Titanium applications written today. It has over 10000 lines of code developed over 6 years. As the application matures, the focus has been on scaling the code to larger processor configurations and larger problem sizes. The initialization code has remained largely unchanged over the years. Correctness in the initialization code is crucial. But we have not done much performance tuning on the initialization code, since it is run only once in the beginning of execution.

Recently, we had scaled the heart simulation up to 512 processors on a 512^3 problem. On our initial runs, the simulation never got passed the initialization phase after more than 4 hours on the 512 processors. The culprit is in the following lines of code.

```
  // missing immutable keyword
  class FiberDescriptor{
     public long filepos;
     public double minx, maxx, miny, maxy, minz, maxz;
     ...
  }

  /* globalFibersArray and the elements in it live on
processor 0 */
  FiberDescriptor [1d] globalFibersArray;
  FiberDescriptor [1d] local localFibersArray;
  ...
  localFibersArray.copy(globalFibersArray);
  foreach (p in localFibersArray.domain()){
     FiberDescriptor fd = localFibersArray[p];
     /* Determine if fd belongs to this processor by
examining the fields of fd */
```

```
    . . .
}
```

Fig. 4. Fiber distribution code containing a performance bug due to lack of immutable keyword

The programmer meant to add the "`immutable`" keyword to the declaration for the `FiberDescriptor` class. But the keyword was missing. Immutable classes extend the notion of Java primitive type to classes. For this example, if the `FiberDescriptor` were immutable, then the array copy prior to the foreach loop would copy every element in the `globalFibersArray` to the `localFibersArray` including the fields of each element. Without the "immutable" keyword, each processor only contains an array of pointers in `localFibersArray` to `FiberDescriptor` objects that live on processor 0. When each processor other than processor 0 accesses the fields of a `FiberDescriptor` object, a request and reply message would occur between the processor accessing the field and processor 0. This performance bug is hard to find manually because the source of the bug and the place where the problem is observed are far from each other.

When the processor configuration is small and the number of `FiberDescriptor` objects is small, the effects of this performance bug are hardly observable. Only when we start scaling the application over 100 processors on the 512^3 problem did we notice the problem. The size of the `globalFibersArray` grows proportionately to the problem size of the input. As we increase the number of processors for the same size problem, the number of field accesses to `FiberDescriptor` objects increases linearly. Each processor reads through the entire array to see which fiber belongs to it. Every field access to a `FiberDescriptor` object results in messages to processor 0. At large processor configurations and large problem sizes, the flood of small messages to and from processor 0 becomes the performance bottleneck.

`ti-trend-prof` can catch this bug earlier in the development cycle using only program runs from small processor configurations and small input sizes. It presents the trends in the communication performance both in terms of the number of processors and the input size. Trends are presented for each location in the source code that incurred communication as reflected in the GASNet traces. For a large application such as the heart code, there are many places in the program where communication occurs. In order to present the most interesting results to the user first, trends are sorted first by the exponent followed by the coefficients. Large values get placed earlier in the table. This allows users to see the least scalable locations predicted by the trends first.

Table 1. Trends output from `ti-trend-prof` for the heart simulation given the GASNet traces for the 128^3 size problem on 4, 8, 16 and 32 processors

Location	Operation	Feature	Max
FFTfast.ti 8727	Get	41p^2 - 416	198400
FFTfast.ti 8035	Put	41p^2 – 416	198400
MyMailBox.ti 384	Put	9p^2 - 789	404120
MetisDistributor.ti 1537	Get	304690p – 1389867	18330567
FluidSlab.ti 3685	Put	200p	12800
FluidSlab.ti 3725	Put	200p	12800

Table 1 shows the trends presented by `ti-trend-prof` given GASNet traces for the heart code on 4, 8, 16, and 32 processors for the 128^3 problem. The trend for the performance bug is in red. The trend shows that the number of get calls on line 1537 in the MetisDistributor file is a linear function with a large coefficient. This clearly alarms the programmer since the number of communication calls should be zero at this location if the "immutable" keyword were not missing. Figure 5 shows the power law model for the buggy line along with observed data.

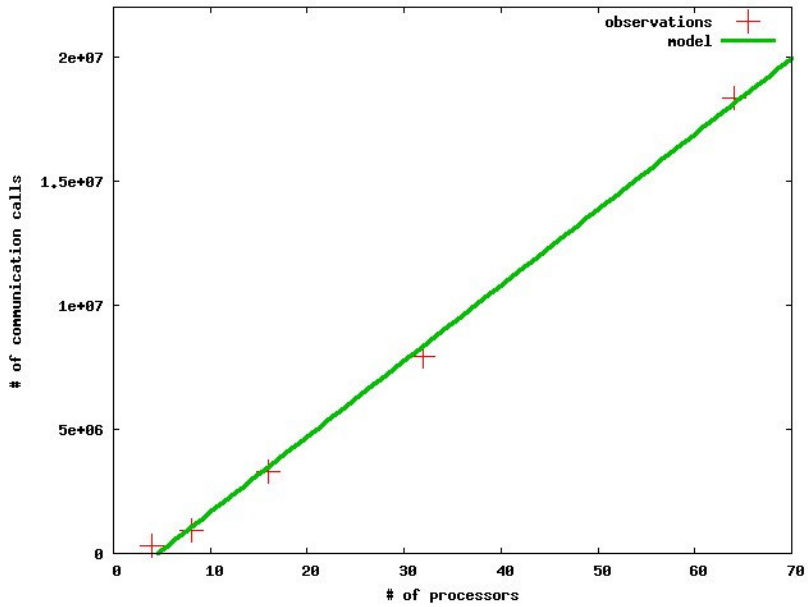

Fig. 5. Graph of the power law function generated by `ti-trend-prof` for the buggy line along with actual observations of communication counts. The X axis is the number of processors, and the Y axis is the count of communication calls.

`ti-trend-prof` can find this same bug in another way. Figure 6 shows the trend when given GASNet traces for the 32^3, 64^3, and 128^3 size problems on 8

processors. The trend for the performance bug location in terms of the input size also clearly indicates that there is a performance bug here. The number of get calls grows super linearly with the problem size. If the "immutable" keyword were there, there should not be any communication calls for this location.

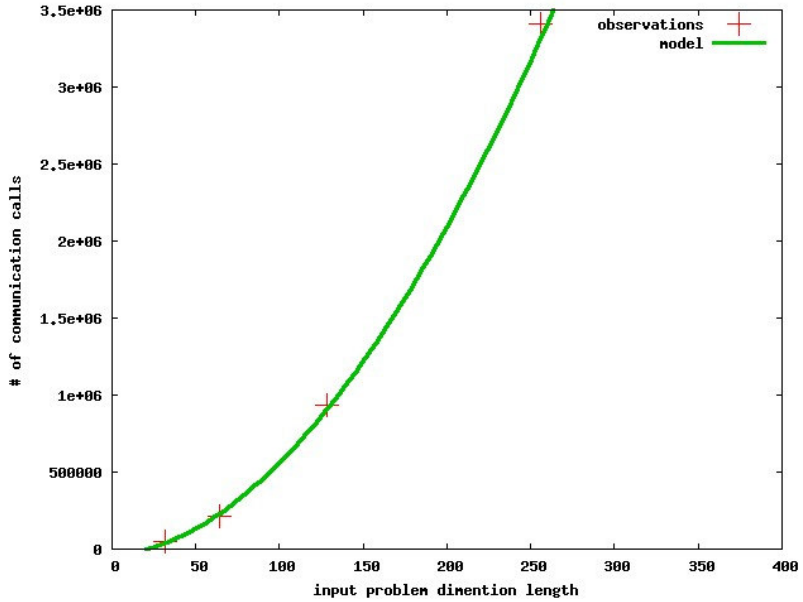

Fig. 6. Graph of the power law function generated by `ti-trend-prof` for the buggy line along with actual observations of communication counts. The X axis is the dimension length of the input problem, (dimension length)^3 gives us the input problem size. The Y axis is the communication count.

We also note that not all trends presented by `trend-prof` are performance bugs. For example, the first trend presented in Table 1 represents the communication pattern during the global transpose in the FFT. The global transpose uses an all to all communication pattern, which makes the number of communication calls grow as the square of the number of processors. The trend presented by `trend-prof` confirms this.

5.2 Adaptive Mesh Refinement

Adaptive Mesh Refinement (AMR) is another large Titanium application. AMR is used for numerical modeling of various physical problems which exhibit multiscale behavior. At each level of refinement, rectangular grids are divided into boxes distributed among processors. Using `ti-trend-prof`, we were able to find two performance bugs in AMR, where one was known prior from manual debugging and the other was not found previously.

5.2.1 Excessive Use of Broadcasts

The first bug appears in the meta-data set up of the boxes at each refinement level. Boxes are distributed among all the processors. But each processor needs to have the meta-data to find neighboring boxes that may live on another processor. Figure 7 shows the code for setting up the meta-data. Instead of using array copies to copy the array of pointers from each processor, it uses one broadcast per box to set up the global box array `TA`. For a fixed size problem, the number of broadcasts due to the code in Figure 7 is the same regardless of the number of processors. But each processor must wait for the broadcast value to arrive if the broadcast originates from a remote processor. As more processors are added for the fixed size problem, more of the values come from remote processors. Subsequently, each processor performs more waits at the barrier as the number of processors increases, and the total number of wait calls sum over all processors increases linearly as shown in Figure 8. If array copies were used, the number of communication calls should only increase by `2p-1` when we add one more processor.

```
/* Meta-data set up*/
for (k=0;k<m_numOfProcs;k++)
    for (j=0;j<(int single)m_layout.numBoxesAt(k);j++)
        TA[k][j]=broadcast TA[k][j] from k;
```

Fig. 7. Fiber distribution code containing a performance bug due to lack of immutable keyword

Figure 8 shows the trend presented by `ti-trend-prof` given the GASNet traces for 2, 4, 6, and 8 processors for the 128^3 problem. It clearly indicates to the programmer that the increase in number of communication calls is larger than expected. Prior to the development of `ti-trend-prof`, it took three programmers to find this bug manually in four days. Similar to the bug in the heart code, the bug was caught late in the development cycle. This performance bug did not become noticeable until we ran the code beyond 100 processors.

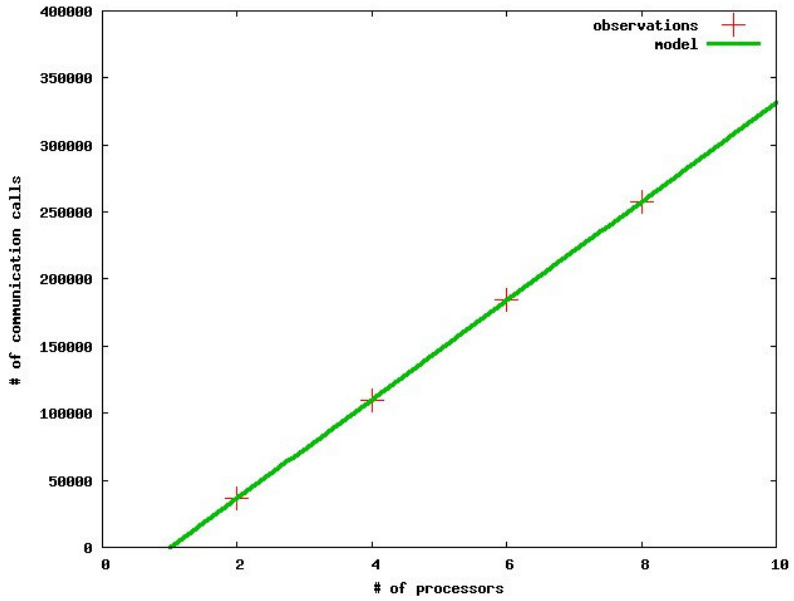

Fig. 8. Graph of the power law function generated by `ti-trend-prof` for the excessive broadcast along with actual observations of communication counts. Each processor must wait at the broadcast if the broadcast originates from a remote processor. As the number of processor increases for a fixed size problem, more of the broadcast values come from remote processors.

5.2.2 Shallow Copy of Meta-data

After the set up of meta-data, each processor only has a pointer to boxes that live remotely. Whenever it needs to perform operations on the underlying array for the box, it needs to call an accessor method for the box, which incurs communication if the box is remote. The number of calls that require communication increases with the number of processors, because more neighboring boxes become remote as processors are added. `ti-trend-prof` reports that the number of communication calls resulting from the accessor method grows almost as the square of the number of processors. If we had a deeper copy of the meta-pointer, which includes the caching of the pointer to the underlying array, we would avoid a majority of the communication calls at the accessor method. The meta-data for the boxes are reused over many iterations. This bug was not found previous through manual performance debugging.

6 Related Work

There has been vast amount of work in the area of performance debugging in both sequential programs and parallel programs. For sequential programs, gprof [8] is a widely used tool for estimating how much time is spent in each function. Gprof samples the program counter during a single run of the program. Then it uses these samples to propagate back to the call graph during post processing. The key difference is that we use multiple runs of the program to come up with trends that can predict performance problems for processor configurations and/or problem sizes that have not been run. Gprof only gives performance information for a single run of the program.

Kluge et al. [9] focus specifically on how the time a MPI program spends communicating scales with the number of processors. They fit these observations to a degree two polynomial, finding a, b, and c to fit `y = a+bx+cx^2`. Any part of the program with a large value for c is said to parallelize badly. Our work differs in that we can use both the number of processors and the input size as features to predict performance. We have used our tool on large real applications. The experiment in [9] only shows data from a Sweep3D benchmark on a single node SMP. Their technique is likely to have much worst errors when used on a cluster of SMPs. They are modeling MPI time, which would be affected by how many processors are used within a node to run MPI. All processors within a node share resource in communication with other nodes. Furthermore, our target programs are written in a PGAS language instead of MPI, which are much harder to find communication locations manually by looking at the program text.

Vetter and Worley [11] develop a technique called performance assertions that allows users to assert performance expectations explicitly in their source code. As the application executes, each performance assertion in the application collects data implicitly to verify the assertion. In contrast, `ti-trend-prof` does not require additional work from the user to add annotations. Furthermore, it may not be obvious to the programmer as to which code segment should have performance assertions. `ti-trend-prof` found performance bugs in code segments where the user didn't think was performance critical. But those performance bugs severely degrade performance only on large processor configurations and large problem sizes, and `ti-trend-prof` helps the user to identify them by presenting the trends.

Coarfa et al. [4] develop the technique for identifying scalability bottlenecks in SPMD programs by identifying parts of the program that deviates from ideal scaling. In strong scaling, linear speedup is expected. And in weak scaling, constant execution time is expected. Call path profiles are collected for two or more executions on different numbers of processors. Parts of the program that do not meet the scaling expectations are identified for the user.

Brewer [3] constructs models to predict performance of a library function implementation as a function of problem parameters. The parameters are supplied by the user. For example, the radix width can be a parameter for an implementation of the radix sort algorithm. Based on those parameters, the tool picks the implementation that the model predicts to be the best. Our tool does not require the user to have the knowledge to supply such parameters.

There are also vast amount of work based on the LogP [5] technique. In particular, Rugina and Schauser [10] simulate the computation and communication of parallel programs to predict their worst-case running time given the LogGP [1] parameters for the targeted machine. Their focus is on how to tune a parallel program by changing communication patterns given a fixed size input.

7 Conclusion

In this paper, we described a tool called `ti-trend-prof` that can help Titanium programmers to do communication performance debugging automatically. Given only program traces from small processor configurations and/or small input sizes, `ti-trend-prof` provides trends for each source code location that incurred communication. Trends are modeled as a standard power law function with offset. Programmers are alerted to trends with large exponents and coefficients, which correspond to possible communication performance bug in the program. The technique is completely automatic without any manual input from the user.

We used `ti-trend-prof` on two large Titanium applications, and we found three real performance bugs in the code. Two of them were known previously from time consuming manual debugging. The third was unknown prior to the use of the tool. These results show the feasibility of using an automatic tool to find communication performance bugs in PGAS languages, given only the program traces from small processor configurations and small input sizes.

8 Acknowledgements

The authors would like to thank Simon Goldsmith and Daniel S. Wilkerson for introducing us to `trend-prof`. Their enthusiasm and persistence greatly encouraged us to adapt `trend-prof` for our needs. Simon Goldsmith also helped us in implementing the parsing of GASNet trace outputs. Thanks go to the members of the Titanium research group, who provided valuable suggestions and feedbacks about this work. We would also like to thank the anonymous reviewers for their helpful comments on the original submission.

This work was supported in part by the Department of Energy under DE-FC02-06ER25753, by the California State MICRO Program, by the National Science Foundation under ACI-9619020 and EIA-9802069, by the Defense Advanced Research Projects Agency under F30602-95-C-0136, by Microsoft, and by Sun Microsystems.

References

1. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. Journal of Parallel and Distributed Computing, 44(1):71–79, 1997.
2. D. Bonachea, GASNet specifications, 2003.
3. E. A. Brewer. High-level optimization via automated statistical modeling. In PPOPP '95: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 80–91, New York, NY, USA, 1995. ACM Press.
4. C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability Analysis of SPMD Codes Using Expectations. PPOPP, 2007
5. D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 1–12, 1993.
6. E. Givelberg and K. Yelick, Distributed Immersed Boundary Simulation in Titanium, 2004
7. S. Goldsmith, A. Aiken, and D. Wilkerson, Measuring Empirical Computational Complexity, Foundations of Software Engineering, 2007
8. S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, pages 120–126, New York, NY, USA, 1982. ACM Press.
9. M. Kluge, A. Knüpfer, and W. E. Nagel. Knowledge based automatic scalability analysis and extrapolation for MPI programs. In Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lecture Notes in Computer Science. Springer-Verlag.
10. R. Rugina and K. Schauser. Predicting the running times of parallel programs by simulation. In Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, 1998.
11. J. Vetter and P. Worley, Asserting performance expectations, SC, 2002
12. T. Wen and P. Colella, Adaptive Mesh Refinement in Titanium, IPDPS, 2005
13. 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.