2015 44th International Conference on Parallel Processing

Characterizing Loop-Level Communication Patterns in Shared Memory Applications

Arya Mazaheri*‡, Ali Jannesari*†, Abdolreza Mirzaei‡, Felix Wolf*†

*Technical University of Darmstadt, Darmstadt, Germany
†German Research School for Simulation Sciences, 52062 Aachen, Germany
‡Isfahan University of Technology, Isfahan, Iran
{mazaheri, jannesari, wolf}@cs.tu-darmstadt.de, mirzaei@cc.iut.ac.ir

***Abstract*—Communication patterns extracted from parallel programs can provide a valuable source of information for parallel pattern detection, application auto-tuning, and runtime workload scheduling on heterogeneous systems. Once identified, such patterns can help find the most promising optimizations. Communication patterns can be detected using different methods, including sandbox simulation, memory profiling, and hardware counter analysis. However, these analyses usually suffer from high runtime and memory overhead, necessitating a tradeoff between accuracy and resource consumption. More importantly, none of the existing methods exploit fine-grained communication patterns on the level of individual code regions. In this paper, we present an efficient tool based on DiscoPoP profiler that characterizes the communication pattern of every hotspot in a shared-memory application. With the aid of static and dynamic code analysis, it produces a nested structure of communication patterns based on program's loops. By employing asymmetric signature memory, the runtime overhead is around 225× while the required amount of memory remains fixed. In comparison with other profilers, the proposed method is efficient enough to be used with real world applications.**

***Keywords*-nested communication pattern; program analysis; thread dependency; profiler; parallel application;**

I. Introduction

In the past few years multi-core processors have emerged rapidly to address the shortcomings of single-core processors. Nowadays, these processors are also being replaced by heterogeneous system architectures (HSA) due to power limits and dark silicon phenomena [1]. In any case, parallel programming is a must to fully utilize the computing resources of these architectures. Although various parallel programming models and parallelization algorithms have been introduced to make software parallelization easier, the programmers still do not have necessary insight about their softwares being developed to decide about optimization [2]. More importantly, future systems require full collaboration of hardwares and softwares to achieve maximum performance regarding both runtime and power consumption. Understanding communication behavior and characteristics of the underlying performance bottlenecks could be a great hint [3], [4].

One of the promising ways to characterize the application's workload is to identify their inherent communication patterns. These patterns can be used to detect the communication behavior of the target parallel program and also detect the relevant computational pattern [5]. Since, each pattern has a unique communication topology between each processor/thread [6] which helps us to discern them quickly and apply relevant optimizations. Although there are various methods already introduced to extract communication patterns, they are mainly designed for distributed-memory applications while detecting only a single pattern and neglecting dynamic behavior of the target program. Because, they mainly produce a communication pattern for the whole program execution. Additionally, they often suffer from heavy memory usage and high runtime. For instance, runtime overhead is between $80\times \sim 400\times$ and analysis of bzip with an input that requires 25 MB consumes more than 10GB of memory [7] with state-of-the-art approaches. This points out that these methods could not be used for on-the-fly pattern detection. Therefore, considering the important role of patterns in software optimization and workload distribution in HSA platforms, detecting communication patterns with maximum accuracy and efficiency has remained a challenge.

In this paper, we present a tool for automatically extracting multiple communication patterns inside shared-memory applications based on DiscoPoP profiler[8], [9]. Detecting these patterns in shared-memory programs is far different than distributed-memory applications. There is no explicit way (e.g. MPI calls) to detect communications in shared-memory systems and there are additional levels of irregularities and complexities to data communication between processing cores [4]. We have addressed this issue by extending DiscoPoP to act as an inter-thread data dependency profiler which maintains runtime and memory consumption overhead. Furthermore, our method is able to detect multiple communication patterns in hotspots of the target program and create a multi-layer communication matrix.

We experimentally validate our approach against programs in SPLASH [10] benchmark suite. We follow six important properties proposed by E. Cruz et al. [11] for evaluating the applicability of the proposed profiler. Based on these properties a comparison between various dependency

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.85

profilers is provided.

The following items summarize the main contributions of this paper:

- Creating an efficient dynamic inter-thread dependency profiler specifically for shared-memory programs based on DiscoPoP.
- Producing multi-layer communication matrix for hotspot loops.
- Devising a new data structure called "Asymmetric Signature Memory" for recording memory accesses and controlling memory footprint of the profiler.
- Deriving a communication metric named "thread load" for expressing hotspots' efficiency rate quantitatively.

The remainder of this paper is organized as follows: First, a concise review of related works will be presented in Section II. Then, in the next section, we explain the concept behind communication in shared-memory systems and DiscoPoP profiler. In Section IV the main approach will be proposed. Afterwards, an experimental evaluation of our method follows in Section V. Application of the proposed method will be discussed in Section VI and finally, we conclude the paper and discuss possible extensions in Section VII.

II. Related Works

Methods to deliver parallel application characteristics analysis and communication pattern detection can be divided into three categories: Simulation-based and log analysis, Code instrumentation, and Hardware counters measurement.

Simulation-based analysis projects try to log and record every change in the system while the program is running. They do this by either running the application in a controlled sandbox (e.g. by using Virtutech's Simics simulator [12]) with many software probes attached to record the changes. Or, by using debugging helper libraries available in parallel programming frameworks (e.g. IPM [13] in MPI). Papers [4], [14] follow the former approach to trace memory accesses in order to analyze the pattern of communication. Paper [4] clearly shows that communication patterns could be clearly identified in shared-memory programs. However, despite compressing the analysis results, it produces extra large output files more than 100GB for a moderate program input size [4]. Papers [6], [15], [16], [17], [13] follow the latter approach and utilize IPM [18] to collect MPI communications between processors. These efforts are mainly designed for distributed-memory applications and do not take shared-memory systems' characteristics into account. Additionally, since every one of them utilizes IPM, they have high memory overhead since it uses 128-bit signature size for each MPI call. Therefore, beside having high execution runtime, they do not seem fit for on-the-fly analysis. Finally, there is no way to detect multiple and nested patterns inside parallel program using the above methods.

Code instrumentation based methods work by inserting instrumentation function before every target instruction in the program to extract runtime information. Projects which have used this approach mainly employ instrumentation tools like LLVM, Intel Pin, DynamoRIO and Valgrind. DiscoPoP falls into this category, since it uses LLVM instrumentation to analyze thread dependencies. In [19], workloads of PARSEC benchmark suite have been analyzed with Intel Pin. Another project called MACPO [20] uses LLVM to instrument program source code and analyze its data structures. CYPRESS [21] on the other hand tries to find MPI communication traces while compressing to reduce overhead. Helgrind [22] and Helgrind+ [23], [24] are Valgrind based tools to detect synchronization errors. They utilize shadow memory approach with 32 and 64 bits shadow values, respectively. SD3 [7] is another profiler for data dependency analysis of sequential programs which reduces space overhead of tracing memory accesses by compressing strided accesses using a finite state machine. We have compared DiscoPoP to this profiler, since the basic concepts are the same and it finds dependencies in loops.

Hardware performance counter measurement approaches [25], [26], [27], [28] monitor the performance counters available in the underlying hardware to collect required information. They have much less overhead in comparison with the previous methods. As a result they are more suitable for making on-the-fly communication pattern detection approach with the cost of impreciseness. This technique can only estimate thread communications indirectly and therefore leads to inaccurate result [11]. A replacement approach to overcome with this issue is using translation look-aside buffers [11]. Although this approach could only be used for thread migration in shared-memory systems, it could be a great start point for future researches. This approach also needs some tweaks in operating system kernel which makes it hard to use on every system.

III. Background

A. Communication in Shared Memory Architecture

There are various researches [6], [17], [13], [16], [29], [30], [15] which are focused on finding communication patterns in distributed memory applications using MPI and IPM [18] libraries. IPM provides a straightforward approach for collecting explicit MPI call logs to extract communication patterns. However, the amount of research on shared memory systems are far less and need extra effort to fill the gaps. Shared memory systems have fundamental differences in comparison with distributed memory system which demand special attention to attain best performing parallel applications. New parallelization models for shared memory architectures imply different communication patterns [4] as we compared with distributed memory applications. Therefore, a new study is required to accurately determine the communication behavior on this architecture. Besides,

shared memory applications bring additional irregularity and complexity to data sharing and are entirely dependent on efficient communication performance between processors [4]. This irregularity impose additional difficulty on finding the communication pattern, which is implicit and occurs through memory accesses [14]. It happens automatically when one worker writes a value and another one reads it.

Communication time between the tasks may be different depending on which core they are executing and the way memory hierarchy and interconnection are used. The problem is even more important in multi-core machines with NUMA characteristics, since the remote access imposes high overhead, making them more sensitive to thread and data mapping [14]. Detecting automatically a communication phase allows for decreasing frequency and voltage of the processor which leads to reducing power consumption by 30% [26]. Therefore, finding the communication patterns inside parallel programs could be a great source of help for optimizing both the program's performance and power consumption. For instance, exploiting communication patterns can improve performance by mapping threads that communicate a lot to nearby cores on the memory hierarchy. This way, there is less replication of data in different caches. The caches can be used more efficiently, and the number of cache misses is reduced [11]. This can also immensely affect the runtime performance of heterogeneous processors. Since, these architectures hugely rely on managed workload distribution among different processing cores.

B. DiscoPoP Dependency Profiler

DiscoPoP is a dependency profiler mainly designed for detecting dependencies inside sequential and multi-threaded programs. It detects write-after-read (WAR), read-after-write (RAW) and read-after-read (RAR) dependencies among program's instructions with the aid of instrumentation technique using LLVM [31]. The main issue regarding software profiling is runtime overhead and memory consumption which prevents them from being used widely. However, DiscoPoP has succeeded to overcome with this challenge by employing software signatures for recording memory accesses history. Therefore, profiling programs with less than 500MB of memory and 86× average slowdown has been made possible. A great feature about DiscoPoP is that its components can be easily extended to add required functionalities. In this paper, we have used this feature to implement our communication pattern detection based on this profiler.

IV. Inter-Thread Dependency Profiler

A. Architecture

Figure 1 demonstrates the proposed architecture for creating the inter-thread dependency profiler based on DiscoPoP. Highlighted sections show the modules that we extended or added to DiscoPoP. The input for the analysis is the target

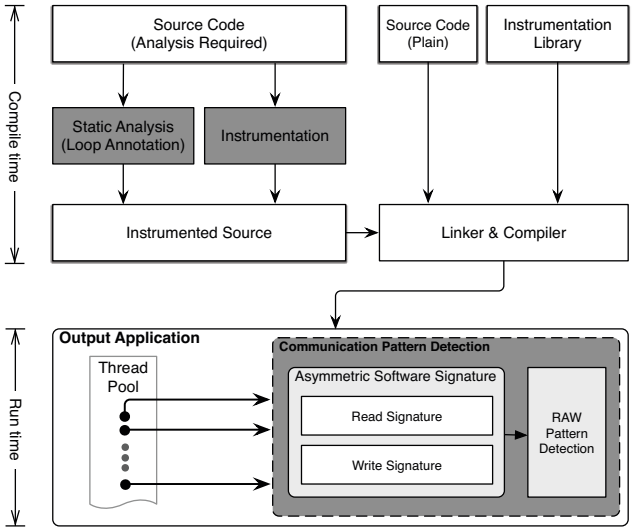


Figure 1. Architecture of the proposed method. Highlighted sections shows the extended modules in DiscoPoP.

parallel program. The source code could be decomposed by user into two pieces: code that has to be analyzed and code that should not be analyzed. This can lead to a significant speedup of the analysis, due to the elimination of unnecessary analysis. The part which should be analyzed is statically instrumented. In the next step the instrumented program is linked with the instrumentation libraries which contains the instrumentation functions and the part of the program under test that should not be analyzed. The result is an executable program along with a communication pattern detection module which then can be executed natively on the processor. In the following sections, we will introduce each component one by one, following the processing flow.

B. Static Analysis

In order to produce nested structure of communication matrices in potential hotspots of the target program, we have devised a simple static analysis module. It analyzes the program and annotates each loop with a unique identifier (UID) using LLVM metadata nodes as depicted in Listing 1. If the instrumented memory access is inside a loop, the UID of the parent loop is fed into the pattern detection for further analysis.

```
1  LLVMContext& C = I->getContext();
2  MDNode* N = MDNode::get(C, ConstantInt::get(Int32,
       loopUIDS++));
3  L->getHeader()->getFirstNonPHI()->setMetadata("loop.md
       .peID", N);
4  currLoopID = L->getHeader()->getFirstNonPHI()->
       getMetadata("loop.md.peID")->getOperand(0);
```

Listing 1. Loop annotation with LLVM

C. Instrumentation

In order to determine the communication pattern, the instrumentation approach has been utilized. There are various

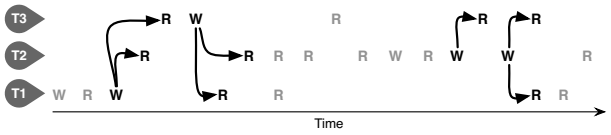


Figure 2. Communicating accesses and memory access ordering for sharing patterns on a single memory location.

ways and frameworks available for instrumenting programs and capturing inter-thread data dependencies. The following items are the reasons behind choosing compiler-based instrumentation over simulation, binary instrumentation and hardware counter analysis:

- Compile-time instrumentation provides greater and easier control over the instrumentation process with the aid of Intermediate representation (IR). For instance, only specific functions, operations or data structures could be selected for instrumentation. In contrast, other methods suffer from this functionality and unable to detect loop structures, boundaries and data structure-related operations [20].
- Compile-time instrumentation enables us to provide on-the-fly analysis feature. Although it degrades the runtime performance by at least $80\times \sim 100\times$, it still outperforms hardware analysis approach with regard to accuracy. Clearly, simulation method could not provide on-the-fly analysis.
- Binary output generated from compile-time instrumentation could run natively on the target architecture utilizing the hardware specific optimizations. While, simulators may not be comprehensive enough to have all features of high-end processors. [20].

Code instrumentation is necessary to access most of the information, required by later performed dynamic analysis in most accurate manner. The input of the pattern detection is memory access operations. We have changed the instrumentation module in DiscoPoP to instrument each memory access with its access type, memory address, function name, variable size, current Loop ID and parent Loop ID.

D. Communication Pattern Detection

From threads sharing adjacency matrix which is called communication matrix in the rest of this paper, we calculate certain reduced quantities that describe the communication pattern at a coarse level. Communication matrix is a $n \times n$ adjacency matrix while n is the number of threads available in the program. It defines the volume of data dependencies among the threads while the program is running.

1) Communicating Memory Accesses: Due to implicit communication behavior of shared memory applications, a standard approach should be devised for identifying implicit communications between software's threads. This approach should avoid redundant memory accesses which often maybe viewed as communicative accesses. In order to determine which words in memory are shared among threads, we inspired from the concept proposed in [4]. A word in memory is defined as shared among threads if it is written to or read from by more than one threads while the program is executing. However, not every reads and writes to such a shared region are actually used to communicate data. Figure 2 illustrates an example of accessing to one memory location while discerning communicating and non-communicating memory access. Communicating accesses are shown in black, while non-communicating accesses are shown in gray.

2) Asymmetric Signature Algorithm: In order to detect communication among threads, an efficient algorithm and data structure should be utilized. Otherwise, in case of using pairwise dependence checking, the overhead of analysis would be very unbearable. Clearly, inter-thread data dependencies can be viewed as data conflict since a dependence exists only when the same memory location is accessed several times in a particular order by more than one thread. Based on this fact, we have employed a modified signature algorithm called asymmetric software signature which determines conflicts between two sets: set of read accesses and set of write accesses. A software signature is a data structure that we borrowed from transactional memory systems. It provides approximate representation of an unbounded set of elements with a bounded amount of state [32]. This data structure is based on hash functions and can access its elements in $\mathcal{O}(1)$. Additionally, the memory overhead is constrained to a fixed value while its size can be adjusted by the user, which makes it suitable for different situations. In other words, the false positive rate could be controlled by changing signature size.

Our solution consists of two signature memories: Two-level signature memory is designed for "Read Signature" because we need to store the list of all threads which have accessed to the correspondent memory location. It uses a fixed-length array of size n, where n is the signature size, in combination with an efficient MurmurHash [33] function that maps memory addresses to array indexes. We opted for this hash function because it has much lower time complexity while having less collisions in comparison with other hash functions. The first-level array stores the pointers to the second-level arrays which are actually bloom filters. The bloom filter [34] is a simple and space efficient probabilistic data structure for recording and representing a set of data in order to support rapid membership queries. In our case, it has been used to save the list of threads which accessed the same memory address. Figure 3(a) demonstrates the proposed two-level signature approach. Memory location address hashes to an element in the first array. If the element is empty, a pointer to the second array will be allocated and points to the new bloom filter. The size of signature elements and bloom filters are adjustable by the programmer,

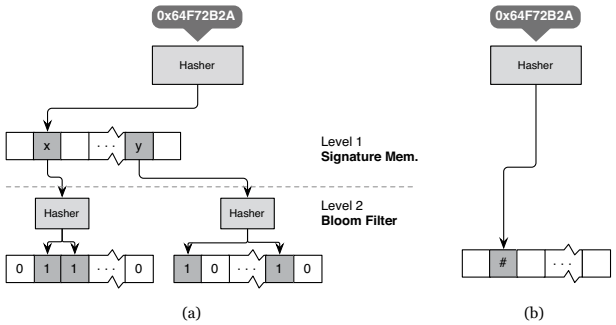


Figure 3. Proposed signature memory architecture. (a) Read signature uses two-level approach, (b) Write signature uses regular signature memory.

so that it can be optimized for a particular program under test. However, the bloom filter has been designed in a way that its size does not need to be adjusted manually. The bloom filter uses a bit vector of size m, where m depends on the number of threads available in the target program. Also a linear combination of hash functions has been devised to automatically adjust the number of hash functions according to the false positive rate required by the user. Since the number of elements going to be inserted to the bloom filter is limited to the number of threads which may access the same memory location, it is guaranteed that the false positive rate does not go beyond the threshold limit.

On the other hand, one-level signature memory tries to only store source thread numbers and is used for representing "Write Signature". In every situation, the values stored in the elements of this signature represent the last thread number which accessed the relevant memory location. Figure 3(b) shows the one-level signature.

The most challenging problem with signature memories is the collision issue. Defining a small signature size could lead to numerous collisions ($h(v_1) = x$ and $h(v_2) = x$ with $v_1 \neq v_2$). This will then produce inter-thread dependencies that do not actually exist ($a.k.a.$ false positive). It is obvious that the accuracy of the algorithm decreases when the size of the signature decreases. Hence, the size of the signature is a trade-off between memory consumption and accuracy.

3) RAW Pattern Detection: Although DiscoPoP already detect various kinds of dependencies, we only need RAW dependency for extracting communication pattern. Moreover, due to using specific signature memory, we have modified RAW dependency algorithm inside DiscoPoP to comply with our asymmetric signature memory. The pseudocode for detecting dependences among threads with signature memories proposed in this paper is demonstrated in Algorithm 1. This algorithm should process memory accesses in temporal order to detect thread dependencies and it should be performed by different threads concurrently in order to enable the parallel analysis. In order to provide parallelism, we use the same threads in the program. Whenever a thread tries to access a memory, it also adds the information

Algorithm 1 RAW thread-dependence using asymmetric signature memory.

```
for all memory access a in the program do
  if Type(a) is read access then
    if a in write signature then
      if a not in read signature & lastWrite.tid ≠ a.tid
      then
        add RAW dependency to comm. matrix;
      end if
    else {a not in write signature}
      insert a to read signature;
    end if
  else {a is write access}
    clear correspondent bloom filter in read signature;
    insert a to write signature;
  end if
end for
```

of its access being performed to the signature memories. Therefore, the dependencies will be identified as the program is running without any need to any extra threads. This simple approach will increase the performance hugely and also reduce the difficulty of implementation. However, special care is required because the signature memory is completely shared with all of the target program's threads. Hence, there is a high risk of contention between threads. We have used C++11 lock-free primitives for implementing signature memory arrays to ensure preventing data race among threads.

E. Extracting Quantitative Metric

We found out that extra valuable metrics could be exploited from communication patterns to quantitatively express performance and bottlenecks of each code region. This feature could be directly fed into an auto-tuner program in order to automatically tune the correspondent parameters and increase the overall runtime performance. One of the sources of bottlenecks in a parallel program could be uneven distribution of workload among threads. However, if this is inevitable, by placing threads on a same socket by thread affinity method, one can diminish this effects [11]. We can transform communication matrices into a simple vector to quantitatively express the overhead of communication on each thread. Equation 1 is a simple example for this mean. The numerator denotes total bytes of communication for $thread_i$ which can be computed by summing all values on that thread's row in communication matrix. $threads\_count$ is total number of threads available in the program.

$$threadLoad_i = \frac{sum(dataCommunicationInBytes_i)}{threads\_count} \quad (1)$$

Table I
COMPARING DISCOPOP AND OTHER WELL-KNOWN PROFILERS BASED ON THE SIX PROPERTIES PROPOSED BY CRUZ ET AL. [11]

Criteria	DiscoPoP	TLB [11]	IPM [18]	SD3 [7]
Real-time detection	Yes	Yes	No	Full support
Memory overhead	Fixed small memory, adjustable by user	N/A	Variable, large output (gigabytes)	Variable memory based on the input size
Runtime overhead (Average)	225×	w/o considerable overhead	N/A	$29x$ - $289x$ (Depend on the threads count)
Communication Pattern Accuracy	Precise*	Approximate	Precise	N/A
Dynamic behavior	Yes	Partial	No	No
Resiliency to FP communication	Yes	Yes	N/A	Yes
Implementation independence	Depend on LLVM	HW architecture dependent	Just MPI applications	Depend on LLVM

∗ In case of having enough signature slots available.

V. EXPERIMENTAL RESULTS

We conducted a range of experiments to measure the applicability and accuracy of the proposed method. First of all we verified the correctness of communication patterns produced by DiscoPoP on all SPLASH benchmark applications. Afterwards, we analyzed the profiler itself. The testbed is a server with 2x 8-core Intel Xeon E5-2680 processors with 64 GB memory, running 64-bit CentOS 6.5. All test programs are based on SPLASH [10] parallel benchmark suite. They were instrumented by LLVM 3.5 and compiled with option -g -O2 using Clang 3.4. We have used various input sizes to consolidate the applicability of our idea. It should be noted that all instrumented programs are set to use $FPRate$ (False positive rate of bloom filter) of 0.001 to obtain accurate results.

A. Profiler Evaluation

E. Cruz, et al. [11] proposed six properties which need to be addressed for every profiler working on finding communication patterns to be suitable for a real-world environment. In table I our enhanced DiscoPoP profiler has been compared with other profilers based on these six properties. It should be noted that, even though DiscoPoP involves false dependencies, its rate can be easily controlled by the user. Having constrained memory consumption and reasonable runtime overhead while getting full insight about dynamic communication phase transition of the target program is a big advantage compared with other methods. In the following, these properties will be discussed in details.

1) Communication Pattern Detection During Execution: Many previous approaches [4], [14], [16], [17], [29], [30] rely on finding the communication pattern in a phase before the actual execution of the workload, for example by using program simulation and sandboxing. This is very time-consuming and potentially takes a lot of storage space to store intermediate data, such as memory traces. However, DiscoPoP is able to detect nested communication patterns while the program is already running without requiring any post-mortem analysis phase.

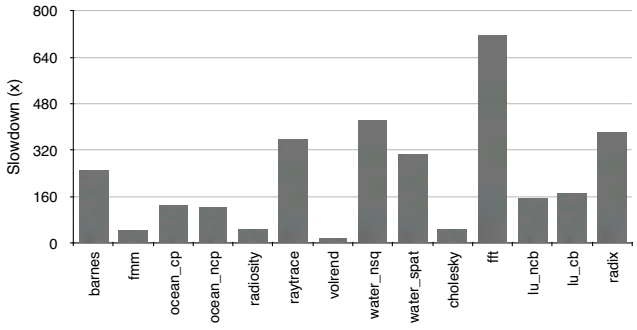
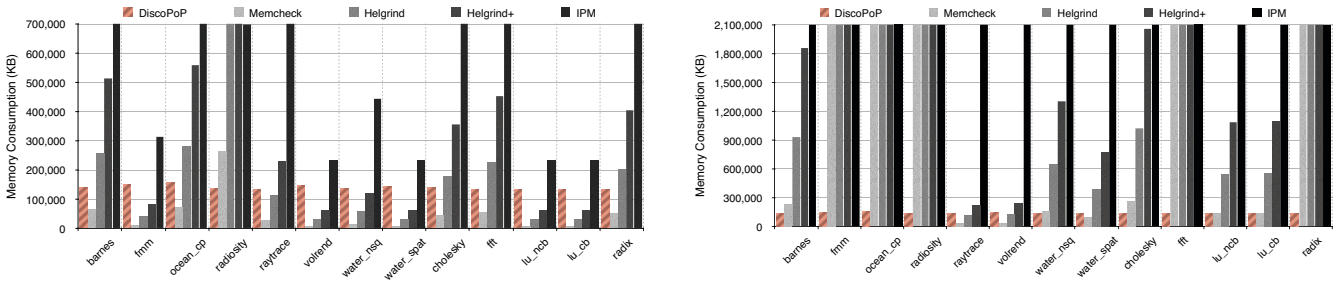


Figure 4. Slowdown of the DiscoPoP on Splash benchmark applications (simdev input size)

2) Runtime Performance Impact: Mainly, performance impact of any profiler is assessed with its runtime and memory overhead. Any real-world profiler should have a low overhead both runtime and memory-wise in order to not interfere with normal execution procedure of the target application. Previous approaches [4], [17], [14], [29] which fall into simulation and code instrumentation approaches have critical problems with performance. They mainly suffer from high memory consumption which in some cases the memory footprint of the profiler could not fit into the memory and lead to occasional system crashes. They also have runtime performance issue where the slowdown is pretty high more than 2500× which absolutely prevent them from working on-the-fly. In the following, runtime and memory overhead of our method will be analyzed and compared with other well-known profilers.

Profiler Runtime: Figure 4 demonstrates the slowdown of SPLASH applications after instrumentation while executing with 32 threads. It is vividly clear that the slowdown rate of target programs are not equal. The range of slowdown spans from 700× to 15× and it largely depends on the inherent communication behavior of the application. If the



(a) simdev input size

(b) simlarge input size

Figure 5. Comparison of memory consumption of the DiscoPoP with other profilers.

target application need a lot of communication, it will suffer from more slowdown after instrumentation. This approach has 225× runtime slowdown which has been computed by computing the average of the slowdown factors. To the best of our knowledge, there is no similar profiler which instrument the application and try to find dependencies among threads exactly like our approach. Therefore, it is not possible to directly compare the runtime of the DiscoPoP method with others.

Memory Consumption: Due to using a fixed length signature memory, the memory consumption is bounded to a specific value. Since, all previous records of memory accesses are stored in signature memory, total memory consumption by the profiler is largely dependent on the signature memory being used and its parameters. Small subset of memory is devoted to store communication matrices of the whole program and its hotspot loops which is negligible in comparison with the size of signature memory.

As described before in section IV, the signature memory used in our extended DiscoPoP is composed of two different signature memories for read and write operations. Equation 2 shows the total memory consumption (Bytes) by the profiler. Where, n is the maximum number of signature elements and t is the number of threads available in the target application. $FPRate$ is the acceptable false positive rate for the bloom filters used in the "read" signature memory.

$$SigMem(n,t) = n.(4 + \frac{-t \times ln(FPRate)}{8 \times ln^2(2)}) \qquad (2)$$

In this section, we have set the signature memory sizes to 10,000,000 because it is not very large and also yields acceptable accuracy in inter-thread dependency results. Additionally, t is set to 32 because all programs are running with 32 threads. Hence, by computing the equation 2, we can see that around 580MB could be sufficient to perform the analysis for any program with moderate input sizes.

Memory consumption of DiscoPoP is compared with Memcheck [35], Helgrind [22], Helgrind+ [23] and IPM [18] which the first three projects all use shadow memory for their analysis and the last one store its output in a log file. Figure 5a and 5b show the memory usage for each SPLASH program while using DiscoPoP and other profiling tools. Figure 5a is for applications using small input size and Figure 5b is for large input size. It is clear that, shadow memory approach consume more memory as the program size grows. However, DiscoPoP memory consumption remains the same disregard to the program's memory allocations.

Simulation based approaches produce more than 100GB [4] log files which is extremely hard to work with. Also, shadow memory based methods clearly could fail if the machine does not have enough memory space. DiscoPoP [8] is the only method which has used signature memory to limit the memory consumption, but due to using queue for analyzing memory accesses orderly, the queue size may increase dramatically if there is burst in accessing memory in the program. However, the DiscoPoP does not have this problem at all and its memory footprint remains the same in every situation.

3) Accuracy of Communication Pattern Detection: Collecting communication patterns should also be as accurate as possible. For example, methods which use hardware counters can only observe the applications behavior indirectly and provide a less accurate view of the communication between the threads [11]. Simulation based and instrumentation approaches produce more accurate results but impose much more overhead on the target program's runtime.

In order to provide an equilibrium in this trade-off, DiscoPoP has equipped with a special signature memory. However, this data structure impose false positive effect in case of choosing a small signature memory. We evaluated the false positive rate (FPR) under four different signature sizes by implementing a perfect signature memory without any collision to be the baseline for FPR comparison. When using 1.0E+6 slots, the average FPR 85.8% which is very high. However, by increasing the slot numbers to 4.0E+6, the FPR significantly reduced to 22.0%. If more accurate results are required, signature size could be set even higher to 1.0E+7 or 1.0E+8 which result in false positive rates of 8.4% and 2.1%, respectively.

4) Dynamic Behavior Detection: Since applications may transition into different phases of computation at runtime,

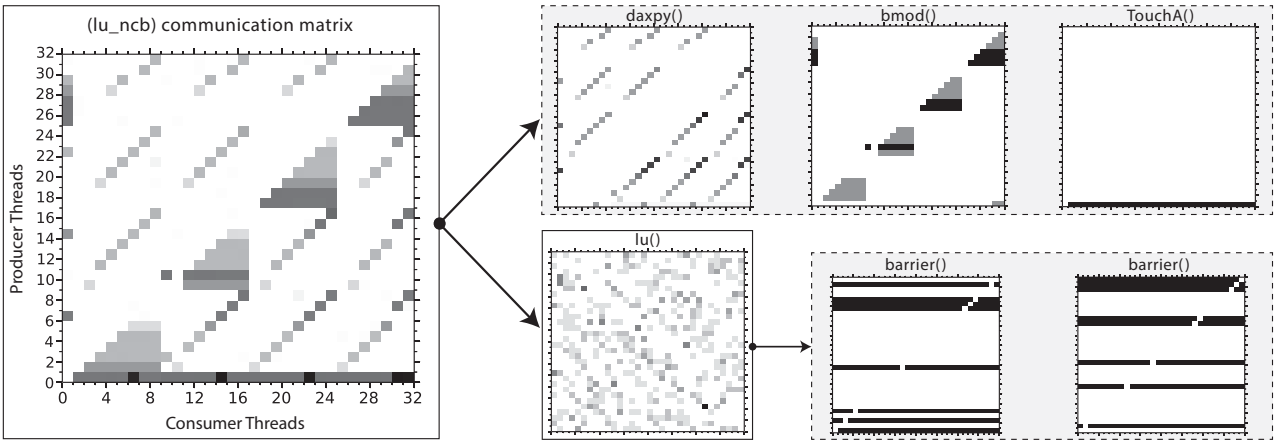


Figure 6. A subset of nested communication patterns found in SPLASH lu_ncb program. Gray boxes denote group of child nodes in nested patterns.

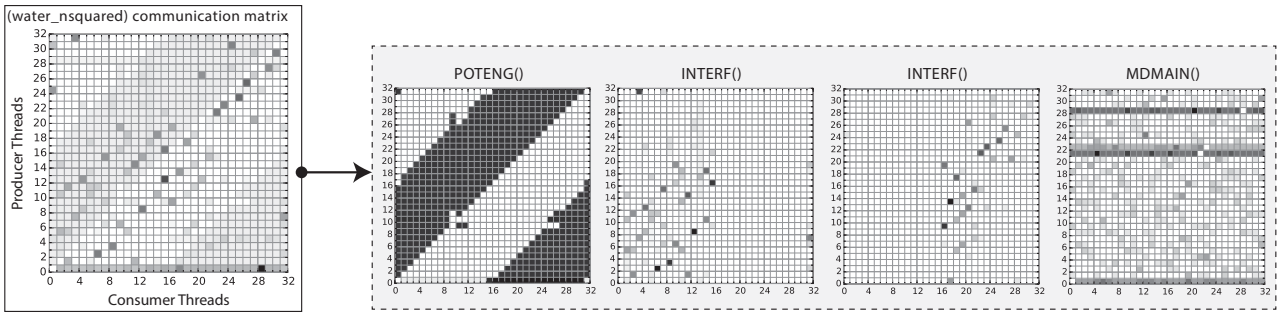
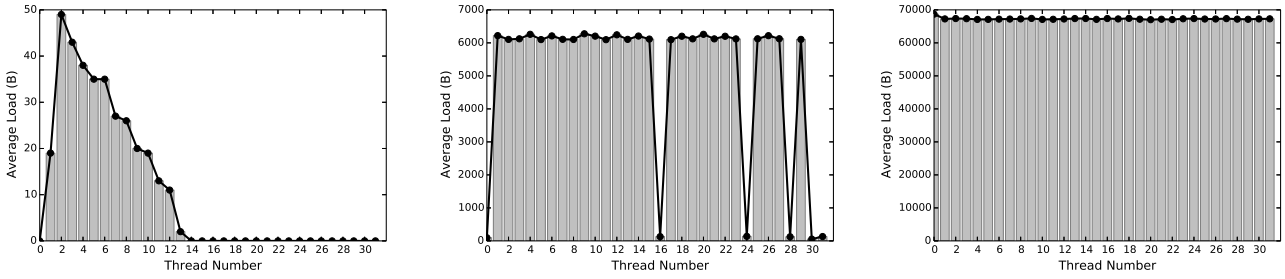


Figure 7. A subset of nested communication patterns found in SPLASH water_nsquared program. Gray boxes denote group of child nodes in nested patterns.

this may change their behavior and therefore the communication pattern during the execution at different points. A useful mechanism should be able to detect changes dynamically and thereby notify the optimizer from these changes in the target program. Almost every previous approaches analyze the application over the whole execution time and provide a static pattern for overall program execution. This leads to wrong results when the application contains more than one computational task. DiscoPoP on the other hand fully supports this feature. It can identify potential patterns in every hotspot of the program and produce a final report about all patterns found in each stage of the application as depicted in Figures 6 and 7 for "lu" and "water_nsquared" programs in SPLASH benchmark. It is clear that the final communication matrix can be obtained by summing all its child matrices together. It should be noted that, based on experimentations, communication patterns are not identifiable enough while using less than 8 threads.

5) Resiliency to False Positive Communication Problem: False positive communication in shared-memory systems denotes the fact that threads appear to communicate through shared data. However, in reality they are not communicating. The root of the problem is that communication is implicit and happens through shared memory. As an example, false communication can happen when two threads access the same address, but at different times during the execution. Our main idea for detecting RAW dependencies among threads is that only first time access by a thread is counted as a communication between relevant threads. Therefore, it avoids other accesses from counting as communication and therefore do not allow false communication in the analysis to be recorded.

6) Application Implementation Independence: In order to cover a wider range of applications for profiling, the architecture should be independent from the application's implementation. This has two consequences. First, using the mechanism should not depend on a particular parallelization API, such as OpenMP and Pthreads. Second, it should not require the programmer to modify the source code or manually link to additional libraries. Among the previous works, only hardware counter methods have this feature. Projects which have used IPM [18] to analyze the memory trace of the target program, can only work with MPI applications and also need to manually re-link their program with IPM libraries. Other methods which have used Virtutech's Simics simulator [12], can only run their method in a simulated sandbox. Therefore, the application cannot run on a regular system. Additionally, binary code instrumentation methods



(a) (b) (c)

Figure 8. Workload distribution among threads of three hotspots in (a) radix, (b) raytrace and (c) radiosity applications.

discussed earlier which use Intel Pin or Valgrind, have limited support on multi-threaded libraries. They also need to be applied manually on the target applications. DiscoPoP on the other hand has tried to overcome with this issue. Although it needs the program's source-code to inject its instrumentation functions, but this procedure can be done very easy just by adding a compiler option "-pe" to the compiler. Also, since it uses LLVM for instrumentation, it can instrument every common language which LLVM compiler supports. Beside that, currently LLVM supports a wide range of parallelization libraries and its supported frameworks is also growing rapidly.

B. Quantitative Metric Result

We have analyzed radix, raytrace and radiosity applications from SPLASH benchmark and selected three interesting hotspot loops communication matrices to show the applicability of the proposed metric expressed in Equation 1. Figure 8 demonstrates these three communication load detected in loops of selected interesting applications. Each diagram contains the load on each thread. Hence, load balancing among threads in each hotspot could be easily analyzed. For instance, Figure 8a depicts that half of threads are accessing the memory in the correspondent loop and may lead to performance inefficiency. However, threads' load shown in Figure 8c reflects a loop that uses all threads available to do its job. Therefore, the load is evenly distributed among threads.

VI. Application

DiscoPoP can be used for extracting inherent communication patterns of parallel programs for various reasons. One of the most obvious reasons for gathering these patterns is to optimize the runtime performance of parallel programs. Based on these patterns, one can apply most suitable thread mapping to place most communicating thread on the same core for increasing data locality. They can also be used to tune the most relevant performance parameters according to the communication behavior of each code region.

Another usecase of communication patterns is for detecting parallel patterns inside parallel programs. It is known that computational patterns have unique communication pattern which can be utilized for detecting them inside applications. Hence, we made a comprehensive study and found out that based on the communication matrices that we can obtain with DiscoPoP, three classes of parallel patterns could be identified: (1) Computational patterns (Motifs), (2) Architectural patterns and (3) Synchronization patterns. Linear algebra, spectral methods, n-body, structured grids, master/worker, pipeline and synchronization barriers were among the patterns we could identify along with additional performance hints with DiscoPoP. We succeeded to detect these pattern with more than 97% accuracy with the aid of algorithmic methods and supervised learning. We also found out that the negative effect of false positives could be compensated by using machine learning classification methods.

VII. Conclusion and Outlook

In this paper, we extended DiscoPoP profiler to create a specialized tool for identifying nested communication patterns inside shared-memory applications. First of all an efficient thread dependency profiler has been proposed based on a specific data structure, called "Asymmetric Signature Memory". Then, we proposed a static analysis approach for annotating loop regions in order to discern potential hotspots of the target program. We also found out that based on the communication patterns, some reduced quantitative metrics could be derived for expressing the performance of the relevant code region. We tested our approach on all SPLASH applications to verify the correctness of produced communication patterns. Furthermore, the results show that communication patterns in each hotspot of the program could be identified distinctively. Based on the evaluation results, it can be concluded that the proposed method can be used for real-world application profiling. It imposes 225× runtime overhead on average which is in acceptable range, because of using code instrumentation approach. Its memory footprint could be less than 500MB and remains fixed regardless of the program input size. In the future we plan to apply sampling technique to reduce the overhead of

instrumentation and use sparse matrices to reduce memory consumption even further.

ACKNOWLEDGMENT

We would like to thank Zhen Li and Tuan Dung Nguyen for their valuable feedbacks and recommendations. We also appreciate the support of the Klaus Tschira Foundation (KTS) for this work.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Int'l Symp. on Computer Architecture*, ser. ISCA'11. IEEE, 2011, pp. 365–376.

[2] A. Raman, A. Zaks, J. W. Lee, and D. I. August, "Parcae: a system for flexible parallel execution," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 133–144, 2012.

[3] J. S. Vetter and A. Yoo, "An empirical performance evaluation of scalable scientific applications," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 16–16.

[4] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of splash-2 and parsec," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 86–97.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer *et al.*, "The landscape of parallel computing research: A view from Berkeley," Uni. California at Berkeley, Tech. Rep., 2006.

[6] S. Kamil, J. Shalf, L. Oliker, and D. Skinner, "Understanding ultra-scale application communication requirements," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE, 2005, pp. 178–187.

[7] M. Kim, H. Kim, and C.-K. Luk, "Sd3: A scalable approach to dynamic data-dependence profiling," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010, pp. 535–546.

[8] Z. Li, A. Jannesari, and F. Wolf, "An efficient data-dependence profiler for sequential and parallel programs," in *29th IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '15, Hyderabad, India, 2015.

[9] ——, "Discovery of potential parallelism in sequential programs," in *Proceeding of 42nd International Conference on Parallel Processing Workshops (ICPPW), Workshop on Parallel Software Tools and Tool In- frastructures (PSTI)*. Lyon, France: IEEE, 2013, pp. 1004–1013.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

[11] E. H. Cruz, M. Diener, and P. O. A. Navaux, "Using the translation lookaside buffer to map threads in parallel applications based on shared memory," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 532–543.

[12] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[13] G. Florez, Z. Liu, S. M. Bridges, A. Skjellum, and R. B. Vaughn, "Lightweight monitoring of mpi programs in real time," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 13, pp. 1547–1578, 2005.

[14] E. H. Molina da Cruz, Z. Alves, A. Carissimi, P. O. A. Navaux, C. P. Ribeiro, and J. Mehaut, "Using memory access traces to map threads and data on hierarchical multi-core platforms," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 551–558.

[15] O. DeMasi, T. Samak, and D. H. Bailey, "Identifying hpc codes via performance logs and machine learning," in *Proceedings of the first workshop on Changing landscapes in HPC security*. ACM, 2013.

[16] S. Peisert, "Fingerprinting communication and computation on hpc machines," *Lawrence Berkeley National Laboratory*, 2010.

[17] C. Ma, Y. M. Teo, V. March, N. Xiong, I. R. Pop, Y. X. He, and S. See, "An approach for matching communication patterns in parallel applications," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[18] K. Fuerlinger, N. J. Wright, and D. Skinner, "Effective performance measurement at petascale using ipm," in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. IEEE, 2010, pp. 373–380.

[19] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 47–56.

[20] A. Rane and J. Browne, "Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics," in *Proceedings of the 21st int'l conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 147–156.

[21] J. Zhai, J. Hu, X. Tang, X. Ma, and W. Chen, "Cypress: combining static and dynamic analysis for top-down communication trace compression," in *High Performance Computing, Networking, Storage and Analysis, SC14*. IEEE, 2014, pp. 143–153.

[22] A. Jannesari and W. F. Tichy, "On-the-fly race detection in multi-threaded programs," in *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*. ACM, 2008, p. 6.

[23] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–13.

[24] A. Jannesari and W. F. Tichy, "Library-independent data race detection," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. PP, no. 99, pp. 1 – 11, 2013.

[25] X. Liu and J. M.Crummey, "Pinpointing data locality problems using data-centric analysis," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 2011, pp. 171–180.

[26] G. Da Costa and J.-M. Pierson, "Characterizing applications from power consumption: a case study for hpc benchmarks," in *Information and Communication on Technology for the Fight against Global Warming*. Springer, 2011, pp. 10–17.

[27] X. Liu and J. M.Crummey, "A data-centric profiler for parallel programs," in *Proceedings of SC13, Int'l Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 28.

[28] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst, "Structuring the execution of openmp applications for multicore architectures," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[29] S. Whalen, S. Engle, S. Peisert, and M. Bishop, "Network-theoretic classification of parallel computation patterns," *International Journal of High Performance Computing Applications*, 2012.

[30] S. Whalen, S. Peisert, and M. Bishop, "Multiclass classification of distributed memory parallel computations," *Pattern Recognition Letters*, vol. 34, no. 3, pp. 322–329, 2013.

[31] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[32] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing signatures for transactional memory," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. IEEE, 2007, pp. 123–133.

[33] A. Appleby, "Murmurhash," 2011.

[34] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[35] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 65–74.