Unveiling Thread Communication Bottlenecks Using Hardware-Independent Metrics

Arya Mazaheri
Technische Universität Darmstadt
Darmstadt, Germany
mazaheri@cs.tu-darmstadt.de

Felix Wolf
Technische Universität Darmstadt
Darmstadt, Germany
wolf@cs.tu-darmstadt.de

Ali Jannesari
Iowa State University
Ames, IA, USA
jannesari@iastate.edu

ABSTRACT

A critical factor for developing robust shared-memory applications is the efficient use of the cache and the communication between threads. Inappropriate data structures, algorithm design, and inefficient thread affinity may result in superfluous communication between threads/cores and severe performance problems. For this reason, state-of-the-art profiling tools focus on thread communication and behavior to present different metrics that enable programmers to write cache-friendly programs. The data shared between a pair of threads should be reused with a reasonable distance to preserve data locality. However, existing tools do not take into account the locality of communication events and mainly focus on analyzing the amount of communication instead. In this paper, we introduce a new method to analyze performance and communication bottlenecks that arise from data-access patterns and thread interactions of each code region. We propose new hardware-independent metrics to characterize thread communication and provide suggestions for applying appropriate optimizations on a specific code region. We evaluated our approach on the SPLASH and Rodinia benchmark suites. Experimental results validate the effectiveness of our approach by finding communication locality issues due to inefficient data structures and/or poor algorithm implementations. By applying the suggested optimizations, we improved the performance in Rodinia benchmarks by up to 56%. Furthermore, by varying the input size we demonstrated the ability of our method to assess the cache usage and scalability of a given application in terms of its inherent communication.

KEYWORDS

Shared memory, multi-threading, data locality, profiling, communication.

ACM Reference Format:
Arya Mazaheri, Felix Wolf, and Ali Jannesari. 2018. Unveiling Thread Communication Bottlenecks Using Hardware-Independent Metrics. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3225058.3225142

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08.
https://doi.org/10.1145/3225058.3225142

1 INTRODUCTION

The prevalence of multi-core processors has made parallel programming and therefore performance analysis of parallel applications significant for exploiting utmost efficiency. Despite the emergence of various parallel programming models to make software parallelization easier, programmers yet do not have detailed insights into performance bottlenecks [26]. Thread communication plays an important role in parallel applications, which is often not thoroughly investigated during performance-bottleneck diagnosis. Because memory is much slower than processors, the full characterization of the memory access patterns of important code regions is critical for precise performance diagnosis. This turns out to be more vital in multi-core systems, where data sharing among cores is often performed in last-level caches [27]. Currently, the programmers' approach to performance tuning is manual code restructuring and applying compiler optimizations for each target platform. In general, however, performance optimization needs to improve data-cache locality and reduce communication bottlenecks [19].

Communication between threads is considered a key factor in multi-core performance optimization. Often performance depends on the size and sharing configuration of the underlying cache. Various studies have investigated the sharing behavior in parallel applications, but primarily only for a single cache size [5]. Therefore, they cannot tell how changes in cache size, configuration, or even thread allocation policies might affect performance. Multi-core reuse-distance analysis [29, 33] is a promising approach to assessing the locality of parallel applications and their cache effectiveness, yet it fails to provide a detailed overview of communication among threads. Furthermore, it suffers from high complexity and error-prone results due to thread interleaving. The locality results are also often reported for the whole program. Therefore, the programmer cannot focus on optimizing data sharing patterns and thread communication.

An effective analysis considers communication events, sharing patterns, the number of concurrent communication events, the amount of communication through memory locations in different time spans, and the extent to which communication correlates with particular regions of the source code. We follow this strategy, which yields an architecture-independent perspective on how source code and resource utilization should be optimized, since such analysis would focus on inter-thread communications rather than a specific set of performance results for a given platform.

One way to characterize parallel applications is to identify their inherent communication patterns. Each pattern has a unique communication topology among processors/threads [18], which helps us to discern them quickly and apply relevant optimizations. Although various methods already exist for extracting communication

patterns, they are either only directed toward distributed memory applications [25, 30, 31] or they are not comprehensive enough to be used for memory performance characterization [9, 19]. They mainly produce a single communication pattern for the whole program execution and neglect the dynamic behavior of a parallel application. Additionally, they generate a simple communication topology, failing to provide further insights how to improve data locality.

In this paper, we introduce two new metrics called *communication reuse distance (CRD)* and *communication reuse ratio (CRR)*. We argue that these metrics together can provide a better understanding and analysis of locality and scalability limitations related to data sharing in multi-threaded applications without being tied to a specific architecture. We created an inter-thread data dependence profiler to extract communication events and their re-occurrences during program execution for each code region, which yields communication reuse patterns and distance profiles. The outputs are then used for communication characterization and providing optimization suggestions.

This paper makes the following major contributions:

- A memory-efficient dynamic inter-thread dependence profiler capable of extracting communication events and distinguishing the first from further re-occurrences
- A new concept called communication reuse distance for measuring the locality of data sharing in each parallel region
- Holistic platform-independent metrics for communication characterization, optimization, and scalability analysis
- Suggestions of relevant performance optimizations. For example, based on such suggestions we were able to improve the performance of an application in the Rodinia benchmark suite by 56%

In the remainder of the paper, we first explain the concept behind communication in shared-memory systems and traditional reuse distance analysis. Then, in Section 3 the main approach will be discussed, followed by evaluation results in Section 4. A concise review of related works is presented in Section 5. Finally, we conclude the paper and discuss possible extensions in Section 6.

2 BACKGROUND

Communication in parallel programming models is either explicit or implicit, depending on the way data is shared among threads. Distributed-memory programming models like MPI follow explicit communication through `send()` and `receive()` API calls. In shared-memory applications, exchanging data is implicit and it is mostly accomplished through memory accesses to shared variables [9]. This implies different communication patterns [1] compared with distributed-memory applications, which then bring additional irregularity and complexity to data sharing.

2.1 Communication in shared memory systems

We define the concept of *communication event* as two memory accesses from different threads to the same memory address with a specific pattern. Four different combinations of memory accesses (RaR, RaW, WaR, and WaW) can take place, depending on whether the data is read or written by each thread. However, only the RaW pattern implicates true communication. In the other cases, no thread can be declared as sender/receiver due to performing the same operation. In this case, one thread writes data (sender) which is then read by another thread (receiver). To avoid redundant communication, we only consider the first read after a write as a communication. Other read accesses to the same location will be considered as reuse. Figure 1 illustrates an example of distinguishing between true communication and its reuse for a single memory being accessed by three threads.

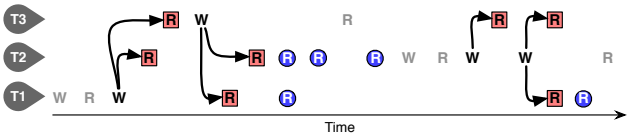


Figure 1: Communicating memory accesses to a single memory location. Red box: true communication. Blue circle: communication reuse.

Identified communication events can be represented by a directed acyclic graph (DAG), where each node represents a thread and edges denote the number of communication events between each pair of threads. Such a graph is often visualized as an adjacency matrix, which is called *communication matrix* or *communication pattern* [9, 22]. Each cell of the matrix contains the number of communication events for a given pair of threads. The diagonal of the matrix is always zero, as memory accesses by the same thread do not imply any communication. Figure 3 shows a sample visualization of a communication pattern, where darker cells indicate more communication.

2.2 Reuse distance analysis

Reuse distance has long been a hardware-independent metric for evaluating the data reuse in programs [12]. The reuse distance of a given reference to element x is the number of distinct data elements accessed between two consecutive uses of x. Data granularity could be anything from pages, cache lines, memory words, or instructions. Reuse distance is typically used for predicting the cache hit ratio of a fully associative LRU cache with N one-word blocks, in which data accesses with reuse distance of N or less would hit. The distribution of reuse distance which is normally represented as a histogram is called the locality signature and shows the overall program data locality (Figure 5). A single run of such an analysis is able to model data locality for all cache sizes and can later be used as a reference for performance optimizations [2, 3, 12].

In multi-threaded applications, such analysis is not hardware-independent anymore because threads interact with each other and memory accesses might interleave [33]. Various researchers proposed an alternative method for computing concurrent reuse distance and private reuse distance for shared and private caches, respectively [17, 29, 34]. Such methods are based on statistical modeling methods and consider specific parallel regions, like loops, due to the high code divergence of task parallelism. Furthermore, these methods consider all memory accesses and do not provide specific insights into communication events and synchronization. In this paper, we address these issues and propose a locality-analysis method to study communication in shared-memory applications.

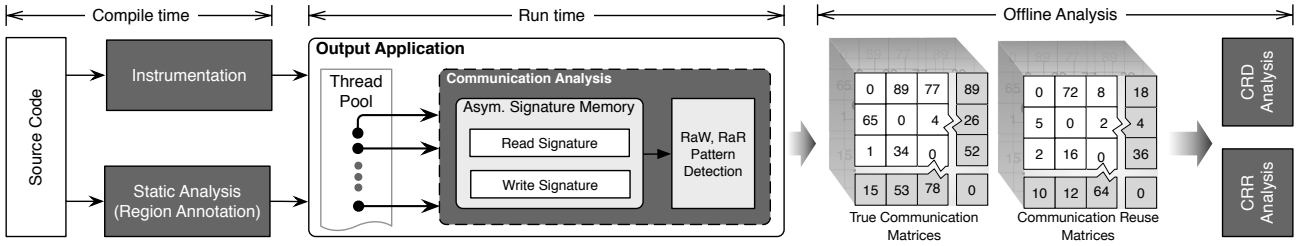


Figure 2: The workflow of the proposed thread communication analysis.

3 APPROACH

Our proposed method for thread communication analysis consists of three different phases: (1) compile-time, (2) runtime, and (3) offline post-analysis. Figure 2 depicts a high-level overview along with the execution flow. In the following, we will explain the main components.

3.1 Instrumentation and profiling

To obtain the required data for extracting communication patterns and communication reuse distance, we extended the DiscoPoP dependence profiler [21]. DiscoPoP detects data dependences among program instructions using LLVM code instrumentation. We tweaked the instrumentation module in DiscoPoP to support pthread and OpenMP parallelization models and instrumented each memory access with its access type, memory address, executing thread ID, region information, and variable size. We also annotated each code region so that later on we could obtain fine-grained information related to each region. The granularity of profiling is loop and function regions in pthread applications and OpenMP regions in OpenMP applications.

The main drawback of software profiling which prevents it from being widely used is runtime and memory overhead. Furthermore, pairwise dependence analysis takes a lot of time and resources to detect inter-thread dependences. DiscoPoP has succeeded to overcome this challenge by utilizing software signatures for recording memory accesses history [22]. Signature memories are used for determining conflicts between two sets: the set of read accesses and the set of write accesses. In addition to being memory efficient, these memories use hash functions with $O(1)$ access time. This enables our profiler to consume a low amount of memory with a reasonable slowdown for a full program analysis. The overhead could be further reduced by sampling the memory accesses. But then, we might lose track of communication events. Therefore, we opted for the full program analysis.

3.2 Communication pattern detection

We are interested in detecting both true communication events (RaW) and their reuse (RaR). By true communication we denote the first read by a thread of a value that has been written by another thread. On the other hand, communication reuse relates to those re-reads after the RaW event. We distinguish these two events from each other because of the different effect that they have on cache usage and memory performance. One critical assumption is that the target application is data race free and synchronization points are used correctly. Otherwise, the gathered information would be misleading.

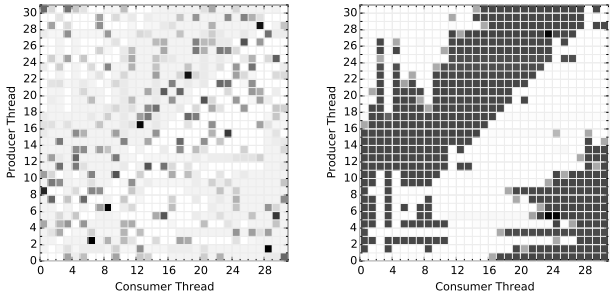


Figure 3: Comparison of (a) true communication and (b) communication reuse of function INTERF() in water_nsquared.

Algorithm 1 Communication extraction 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 dependence to comm. matrix;
      else if lastWrite.tid ≠ a.tid then
        add RaR dependence to reuse matrix;
      end if
    else {a not in write signature}
      insert a to read signature;
    end if
  else {a is write access}
    clear out correspondent entry in read signature;
    insert a to write signature;
  end if
end for
```

The pseudocode for detecting dependences between threads with signature memories is shown in Algorithm 1. This algorithm processes memory accesses in logical order to detect thread dependences and it should be performed by different threads concurrently in order to enable the parallel analysis. The dependences are identified during program execution using lock-free primitives without spawning any new thread.

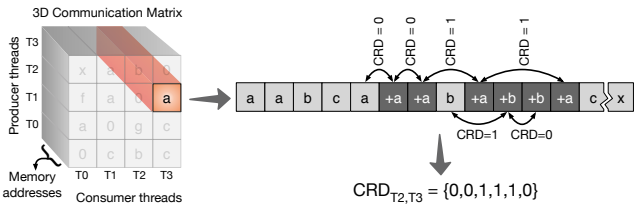


Figure 4: Computing CRD for a sample 3D communication matrix. Dark cells denote reused communication.

In contrast to previous work on communication matrix detection, our profiler generates three-dimensional communication matrices. One for true communication and one for communication reuse. The third dimension contains a sequence of memory addresses shared between corresponding threads, which can later be used for computing communication distances. Obviously, generating two-dimensional communication matrices for the purpose of visualization is also possible. Figure 3 shows the true communication and communication reuse matrices extracted from a function in the application water_nsquared side by side for comparison. The discrepancy between communication reuse and true communication matrices is easily observable. In this specific case, we can spot the threads which are reusing communication more than others. Such observation could be the main source of information for data-locality optimizations like thread/data mapping.

3.3 CRD: Communication reuse distance

Data locality analysis is an established method for evaluating the efficiency of memory accesses within an application. The traditional reuse distance inspired us to propose *communication reuse distance (CRD)* to analyze thread communication. CRD is a measure of data locality for communication events between each pair of threads, with shorter distances having a higher chance to represent cache hits, and longer distances less so.

We created a tool to measure the CRD for each code region (loops and functions in pthread and parallel regions in OpenMP). The input consists of three-dimensional communication matrices, including the sequence of true communication along with their reuse. Communication reuse events are previously annotated with a plus sign to distinguish them from true communication. Given a communication trace for a pair of threads, we define the logical access time of a communication as its index position in the trace, counted from the first communication event. Thus, $CRD_{i,j}$ is the number of distinct data elements accessed between two consecutive usages of the same element among threads i and j.

Figure 4 shows an example of CRD analysis for a pair of threads. In this example, we have two memory locations a and b shared between threads $T2$ and $T3$. Computing the distance between the reused communication on each memory location forms the CRD values. Concatenating all $CRD_{i,j}$ together produces the final CRD. Listing 2 shows our algorithm for measuring the CRD.

To provide further insights into the CRD and potential cache misses related to communication events, we use a histogram. Additionally, we propose two cutoff distances: (1) maximum cutoff and (2) minimum cutoff. The maximum cutoff distance is defined as the total size of cache available in the target system, whereas the minimum cutoff distance is defined as the cache size minus the total size of non-sharing variables in the application. Depending on the value of CRD, the following conditions explain the behavior of the cache:

- CRD[$x >$ Max Cutoff] $\Longrightarrow$ Definite capacity miss
- CRD[Min Cutoff $< x <$ Max Cutoff] $\Longrightarrow$ Probable capacity miss
- CRD[$x <$ Min Cutoff] $\Longrightarrow$ No capacity miss

Figure 5 shows a sample histogram, where most of the communication reuse distances are below the minimum cutoff. Thus, they would not suffer from cache capacity misses. A fraction of reuse events fall between the minimum and maximum cutoffs and therefore the chance of data locality issue exists. Based on this diagram, one can easily evaluate the efficiency of communication reuse and apply further optimizations and data structure modifications to prepare the application for the target platform.

Algorithm 2 Computing communication reuse distance

```
CRD = [];
for all threadPair in commMatrix do
  uniqueEvents = unique(threadPair);
  for all element x in uniqueEvents do
    indices = find occurrence indices of x and +x in threadPair;
    distances = compute pairwise distance of indices
    append distances to CRD
  end for
end for
```

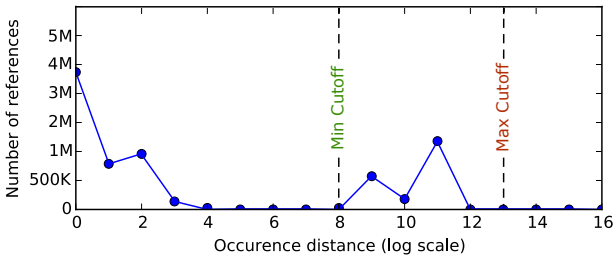


Figure 5: A sample CRD histogram. Max and min cutoffs are shown depending on the cache size.

3.4 CRR: Communication reuse ratio matrix

We introduce the concept of *communication reuse ratio (CRR)* as a matrix similar to a normal communication matrix, where each cell contains the ratio of reuse to true communication instead. To obtain such a matrix, we first extract both true communication and reuse matrices and then we use element-wise division to generate the CRR matrix. The CRR matrix of a program region provides a good overall understanding of the amount of reuse compared to total communication. Hence, those threads which are not reusing communication or are not mainly involved in repeated communication events could be easily identified. Our empirical experiments showed that optimized applications have more homogeneous and balanced CRR matrices.

	Code opt.	Run-time opt.
Data-affinity opt.	Array blocking, Loop reordering, Loop fusion	Data affinity scheduling
Thread-affinity opt.	-	Thread mapping (various policies)

Table 1: The list of optimization types along with some examples categorized into coding and runtime optimizations.

Inspired by Diener et al. [8], we propose two quantitative metrics called homogeneity and balance to characterize and compare CRR matrices. Such comparison guides a programmer to reach an optimal implementation.

3.4.1 Homogeneity. Accessing the outcome of a communication event multiple times is more desired than a long sequence of true communication. Thus, we would like to see homogeneous communication reuse over all threads rather than some threads having no communication reuse at all. We define *communication reuse homogeneity* as in Equation (1), where T denotes the total number of threads and var denotes the variance function. For each row of the CRR matrix, we compute the variance and finally compute the average of variances over all threads. A lower value yields more homogeneous communication reuse.

$$Homogeneity = \frac{\sum_{i=1}^{T} var(CRR[i])}{T} \quad (1)$$

3.4.2 Balance. It is important to determine whether some threads have more communication reuse ratio than others. Such information can be used for a more optimized thread placement. To evaluate this property, we introduce a metric called *communication reuse balance*. We first calculate the total reuse ratio for each row in the CRR matrix (*CommReuseRow*), where each element i of *CommReuseRow* contains the sum of all communication reuse ratios for thread i. Similar to computing load balance, we compute the communication reuse balance as in Equation (2). We seek to have a lower value to have a more balanced CRR matrix.

$$Balance = \left(\frac{max(CommReuseRow[1...T])}{\sum_{i=1}^{T} \frac{CommReuseRow[i]}{T}} - 1\right) \times 100\% \quad (2)$$

3.5 Communication bottleneck analysis

Both communication pattern and CRD analysis are useful methods for identifying bottlenecks and optimizing programs. However, they aim at different aspects of communication issues. CRD analysis informs a programmer as to which region's *communication* is likely to trigger cache misses, while communication and CRR matrices report which *threads* are likely to cause communication and reuse. The combination of CRD and CRR analyses along with homogeneity and balance metrics can provide an architecture-independent instrument suitable for a targeted optimization. For instance, if CRR identifies several threads as communicating or reusing the communication among each other, the CRD analysis can detect whether this communication would be problematic or not. Furthermore, when CRD is high, data structure and data access optimizations are commonly suggested and if CRR metrics are high, it is recommended to apply thread-affinity optimizations. Table 1 contains the optimization categories which our method is able to suggest. For each category, a list of potential optimization methods is also provided.

4 EXPERIMENTAL RESULTS

We conducted a range of experiments to measure the effectiveness of the proposed method. The testbed is a server with two 8-core Intel Xeon E5-2650 processors at 2 GHz and 32 GB memory, running Ubuntu 16.04. We tested the benchmarks with 16 threads and various input sizes, with an average of five independent executions to ensure the correctness of the results.

Previous approaches [1, 23, 30, 31] often rely on program simulation and sandboxing to extract the communication patterns. This is very time consuming and potentially takes a lot of space to store intermediate data such as memory traces. However, our profiler is able to detect communication patterns during execution with an average slowdown of 110× while occupying less than 1 GB memory for allocating software signatures. The overhead largely depends on the inherent communication behavior of the application. The more communication and memory access it performs, the more slowdown will be caused. However, we believe that the level of detail it provides justifies the overhead.

4.1 Communication analysis validation

To demonstrate the validity of the proposed metrics, we looked for multi-threaded applications which have two implementations with different levels of optimization. We found *lu*, *ocean*, and *water* from the SPLASH benchmark suite having this property. We analyzed these three applications (in total six programs) using our method to show the effect of communication locality issues and code optimizations on communication reuse metrics. For each application, we present the results in a tabular format (Figures 6-8), which includes relevant diagrams for the most communication-intensive functions. Each row relates to a function and contains a CRD histogram and two CRR matrices, where the first matrix is for the non-optimized version and the second for the optimized version. Each CRR diagram is annotated with two numbers at the top which represent homogeneity and balance.

4.1.1 lu. `lu_ncb` and `lu_cb` are two different versions, one with a non-contiguous and one with a contiguous block implementation, respectively. `lu_ncb` uses a one-dimensional array in which the matrix to be factored is stored. On the other hand, `lu_cb` uses a two-dimensional array in which all data points in a block (touched by the same processor) are allocated contiguously and locally.

Figure 6 reports the analysis results for communication-intensive functions. The CRD histogram for function `bmod` shows that the contiguous version has a lower distance, which can lead to less cache capacity misses. For validation, we analyzed the cache miss rate using Cachegrind. Detailed cache miss ratios reported for each function in Figure 6 show that the CRD histograms comply with the miss rates. Balance and homogeneity metrics shown on top of the CRR matrices denote higher communication imbalance in the non-contiguous version. However, the optimization increased the heterogeneity. We believe that the balance of communication compensated the heterogeneity. In function `daxpy`, the number of reuse in the contiguous version is much higher than the non-contiguous version. However, the distance of communication is similar. A noticeable change is the homogeneity of communication which improved in the contiguous version. In the last function, we do not see any big improvement, as the CRD histogram and CRR

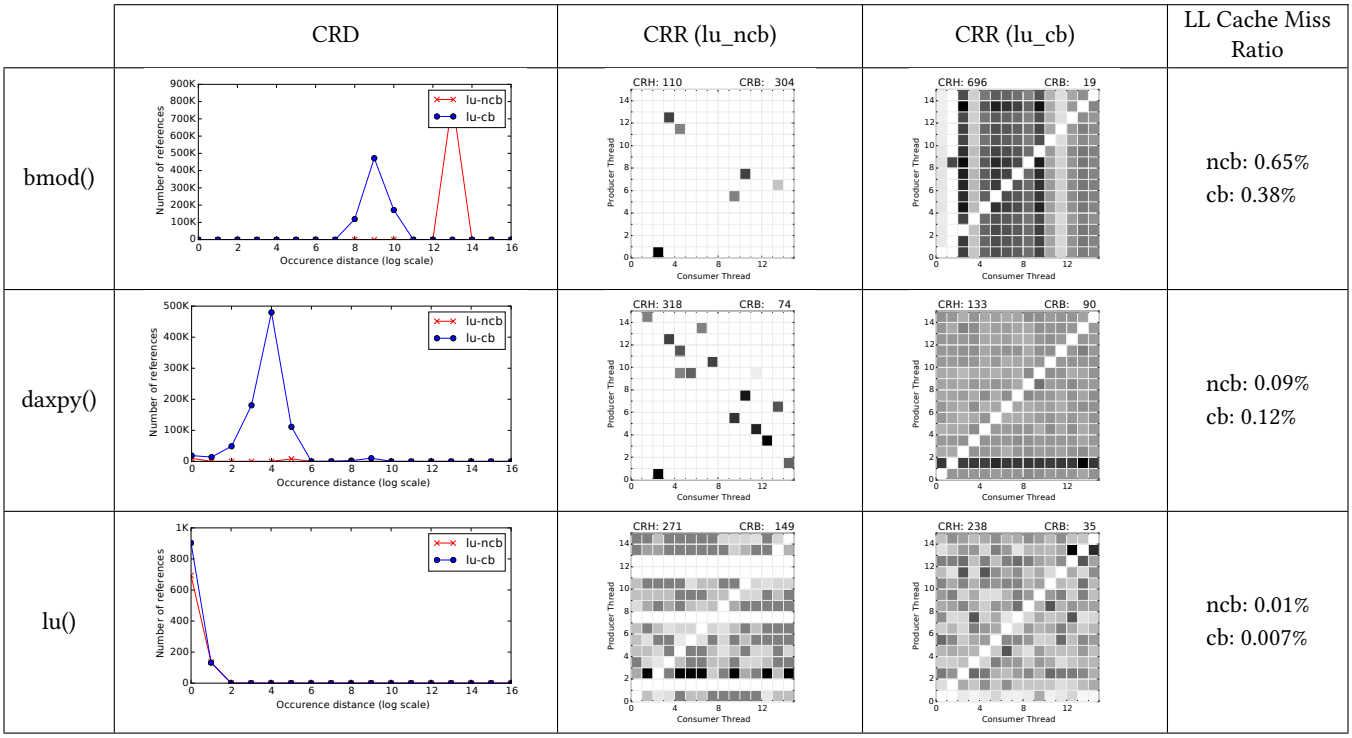


Figure 6: Communication analysis results for lu_ncb and lu_cb.

matrices follow the same pattern. Overall, our fine-grained analysis was able to report slightly better results for the optimized version. Such observation was not made by Biena et al. [4], who reported that the cache behavior of both versions is similar.

4.1.2 ocean. The two versions of this benchmark solve the same problem but use a different memory layout. The non-contiguous implementation uses two-dimensional arrays, which avoids allocating contiguous partitions. The contiguous version is implemented with three-dimensional arrays. The first dimension specifies the processor which owns the partition so that partitions can be allocated contiguously [4].

Figure 7 shows the most communication-intensive functions of both ocean_ncp and ocean_cp. The CRD histograms of functions `laplacalc` and `copy_red` clearly show much higher communication distance for the contiguous implementation, which is in line with the reported cache misses. The CRR matrices of both versions of function `laplacalc` show a similar pattern, while the non-contiguous version seems to be more homogeneous. In this case, the CRD profile is more helpful in finding the source of the locality issue. Some functions, like `jacobcalc`, do not seem to be affected a lot by the optimization, despite having a high cache miss rate and communication insensitivity. We conclude that just by focusing on the hotspot regions, we cannot effectively apply optimizations. We investigated the reason for high CRD distance and poor CRR metrics for the contiguous version and we found out that it can be attributed to the lower number of shared writes at the cost of a much higher miss rate on multi-core devices [4]. Our analysis successfully identified the region where the optimization failed to perform better than the straightforward implementation.

4.1.3 water. water_nsquared and water_spatial are a non-optimized and an optimized implementation, respectively. In water_nsquared, the forces and potentials are computed using an $O(n^2)$ algorithm [32]. water_spatial uses a more efficient algorithm with a computational complexity of $O(n)$. It imposes a uniform three-dimensional grid of cells on the problem domain. Therefore, processors which own a cell will only access neighboring cells.

Figure 8 clearly shows the superiority of the spatial version over non-spatial implementation. All CRD histograms for the representative functions show lower communication reuse distance for the spatial version. Our cache analysis results also show that the cache miss ratio in the spatial version is much lower than in the nsquared alternative. The CRR matrices for the spatial implementation are also more regular and homogeneous. The CRR matrices of function `PREDIC` are a good example for demonstrating that relying solely on CRR is not enough for analyzing the communication behavior. There is not much difference between these two CRRs. However, the CRD diagram shows a shorter distance for the spatial version.

4.2 Communication scalability analysis

Although the proposed metrics are hardware agnostic and do not change on different systems, the input size affects the metrics. We observed that the amount of change, either the amount of reuse or distance values, can be used for analyzing the scalability of a given application with larger input sizes. This is not only helpful for understanding the requirements of the application but it can also be used for selecting the region which suffers from increased communication overhead.

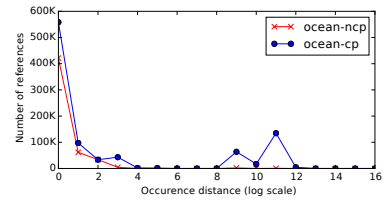
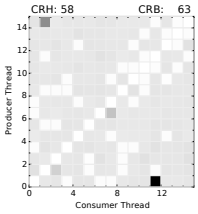
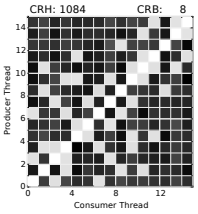
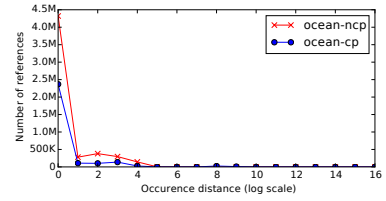
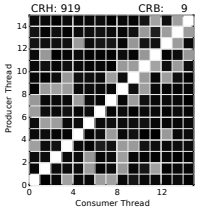
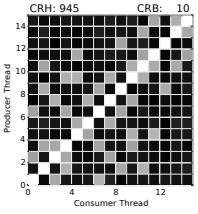
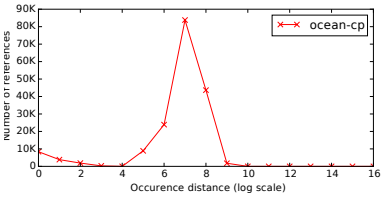
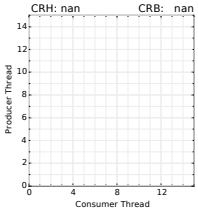
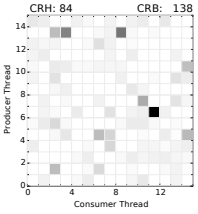
	CRD	CRR (ocean_ncp)	CRR (ocean_cp)	LL Cache Miss Ratio
laplacalc()				ncp: 3.70% cp: 6.25%
jacobcalc()				ncp: 2.19% cp: 2.39%
copy_red()				ncp: N/A cp: 0.09%

Figure 7: Communication analysis results for ocean_ncp and ocean_cp.

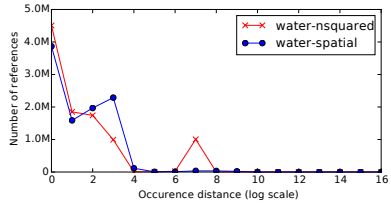
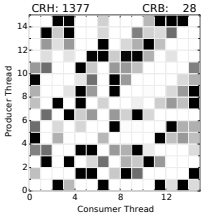
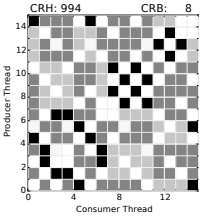
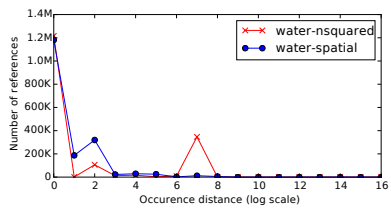
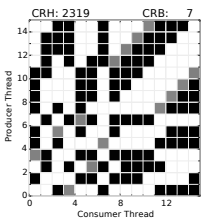
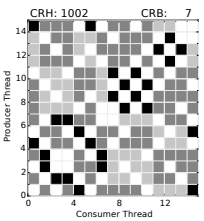
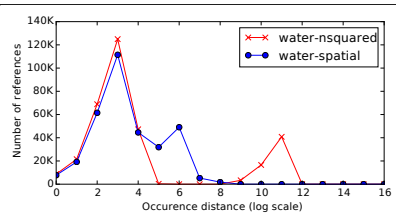
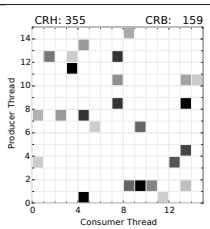
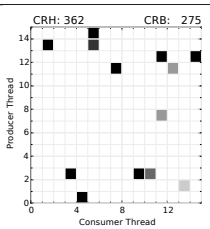
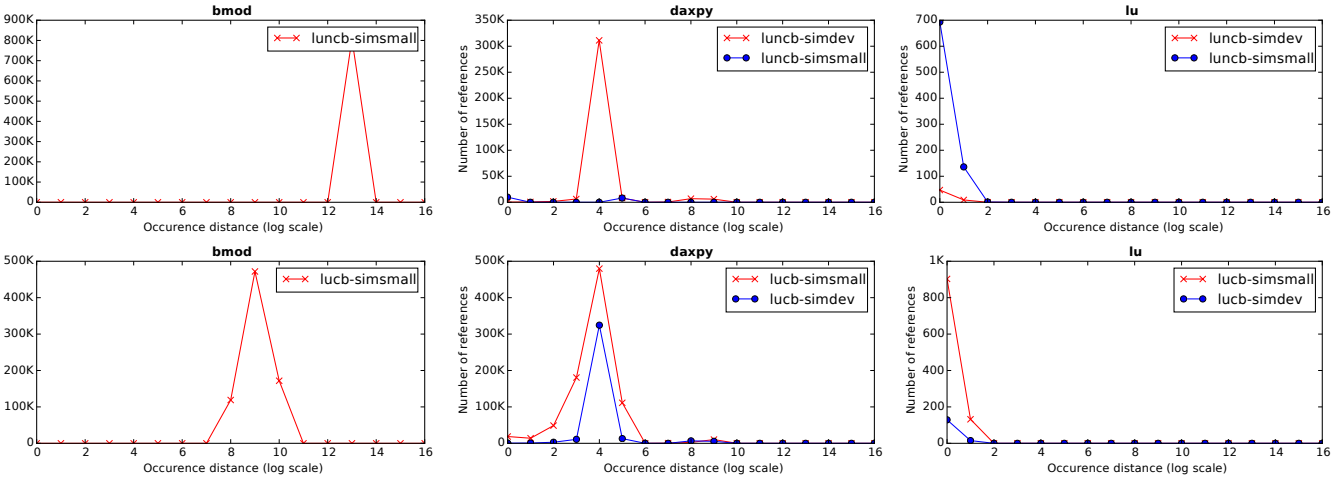
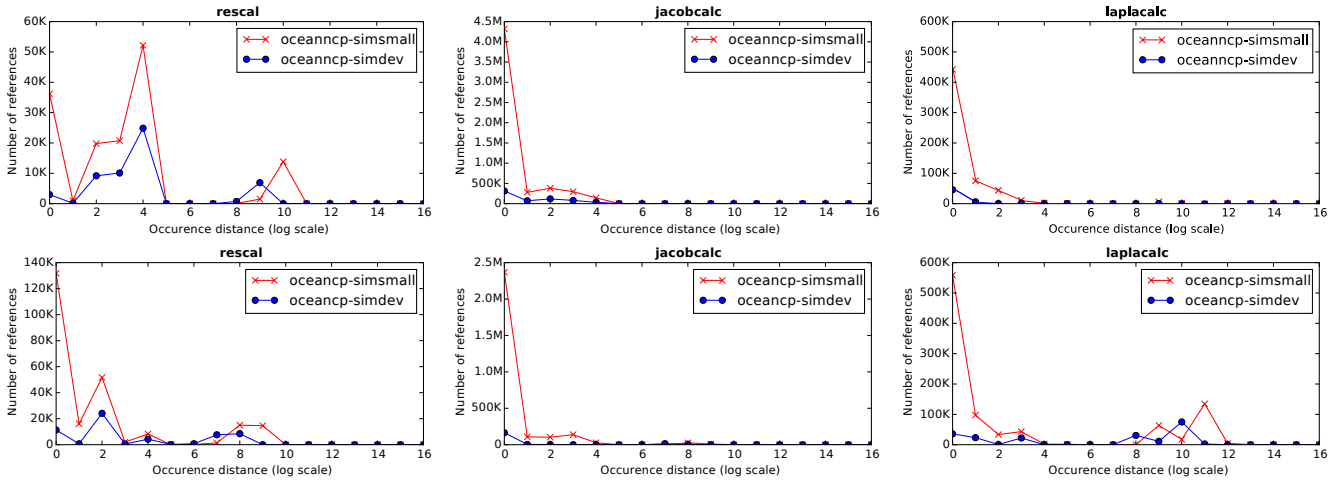
	CRD	CRR (water_nsquared)	CRR (water_spatial)	LL Cache Miss Ratio
INTERF()				nsq: 0.24% spat: 0.02%
POTENG()				nsq: 0.16% spat: 0.007%
PREDIC()				nsq: 1.45% spat: 0.76%

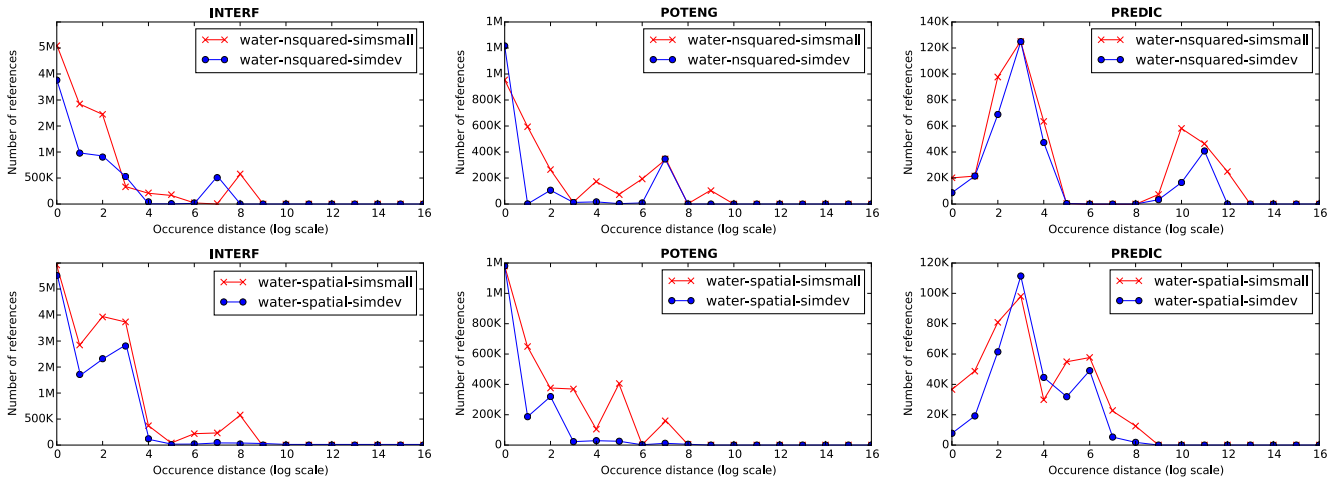
Figure 8: Communication analysis results for water_nsquared and water_spatial.



(a) lu



(b) ocean



(c) water

Figure 9: The effect of two input sizes (`simdev` and `simsmall`) on CRD histograms for the applications lu, ocean, and water.

		Before optimization			after optimization					
Benchmark	Bottleneck region	CRD	H	B	CRD	H	B	Data-affinity opt.	Thread-affinity opt.	Speedup
Back Propagation	backprop.c: L321 - L327	**5**	84	21	2	54	14	Loop reordering, Array blocking	-	15%
Needleman-Wunsch	needle.cpp: L161 - 167, L176-L181	**10**	278	139	5	239	45	Data affinity (32 OMP chunk size), Array blocking	-	29%
SRAD	main.c: L254 - L290, L296-L320	**13**	**1124**	153	4	981	89	Loop reordering	Thread mapping	35%
Particle filter	ex_particle_OPENMP_seq.c: L488 - L495, L480-L482	**8**	321	89	3	384	121	Loop fusion	-	56%
Streamcluster	streamcluster_omp.cpp: L540 - L548	3	**1409**	6	-	-	-	-	Thread mapping	19%

Table 2: The detailed results of communication bottleneck analysis for a subset of the Rodinia [6] benchmarks. Each metric is reported twice. Once for the non-optimized and once for the optimized version. H and B denote homogeneity and balance, respectively. Highlighted bold numbers indicate high metric values, instructing to apply relevant optimization.

Figure 9 demonstrates the effect of two input sizes (`simdev` and `simsmall`) on the CRD histograms of our test cases. The results of both versions of lu show that only function `bmod` is affected by a larger input size. Other diagrams show that a larger input size increases both the number of communication events and their distances. Hence, there is a risk of higher cache misses related to communication after increasing the input size. In the ocean benchmark, we observed a similar behavior. The results show that the optimized version still suffers from high communication overhead. In contrast to lu and ocean, water_spatial shows better scalability, as its CRD profile does not seem to be affected a lot. This clearly shows that the optimized algorithm performs better for larger inputs.

4.3 Communication bottleneck analysis

We used the Rodinia benchmark suite [6] to show the effectiveness of the proposed metrics in finding communication bottlenecks. We succeeded to detect bottlenecks in the five benchmarks of Rodinia listed in Table 2. For each application, the problematic code regions are reported along with the metric results before and after optimization. High CRD value demands to apply data locality optimizations, which are mostly done by code optimizations like array blocking and loop reordering. Conversely, high homogeneity and balance values are a hint to apply runtime thread-affinity optimizations.

Computing the minimum cutoff distance for each application identified high median CRD values in the benchmarks Backpropagation, Needleman-Wunsch, SRAD, and Particle Filter. High CRD indicates potential communication locality issues, which we tried to address with code optimizations and runtime OpenMP scheduling. These optimizations lowered the communication reuse distance and led to a considerable amount of speedup. In Streamcluster and SRAD, we noted high communication reuse heterogeneity among threads, which calls for locality-aware thread mapping optimization. We could not measure the effect of thread mapping on our metrics, because our analysis currently focuses on the data and thread mapping just alters the thread placement. However, a notable speedup was achieved just by optimized thread mapping.

Our analysis results provided optimization insights, both related to data and threads, which improved the runtime performance by up to 56%. To the best of our knowledge, other methods do not provide such detailed information on program communication for selecting the right type of optimization. Our method can be used in companion with other performance debugging tools to extend the scope of performance analysis.

5 RELATED WORK

To the best of our knowledge, no similar paper is published on evaluating the locality of communication in shared memory systems. Gprof [16] and Threadspotter [13] are two well-known methods for optimization suggestion, yet they are unable to produce comparable results. Profiling multi-threaded applications using Threadspotter will primarily generate results only valid for a single thread (master). Furthermore, these methods do not necessarily focus on communication bottlenecks, which is the main target of this paper. Nevertheless, we discuss previous efforts on communication pattern detection and multi-core reuse distance below.

Various methods have been proposed for analyzing the communication pattern in parallel applications. Most of them target distributed-memory applications [14, 15, 18, 20, 25] though, which makes the communication analysis straightforward but cannot be extended to shared-memory programs. Simulation-based methods try to simulate the system by logging and recording every change during program execution [1, 23]. This is impractical as it implies high runtime overhead and produces extra large output files of more than 100 GB for a moderate input size [1]. On the contrary, our method uses a memory efficient profiler. Other studies [7, 10, 11] utilized Intel Pin to extract communication patterns and used them for thread and data mapping. However, their output is a single communication matrix. Thus, they are not able to detect the dynamic behavior of an application in different code regions.

Reuse distance analysis has been studied extensively for single-core architectures [12, 35]. However, due to nondeterministic runtime scheduling of threads in multi-threaded applications, alternative methods have been introduced to preserve hardware independence [17, 24, 28, 29, 34]. Recent methods [33] are based on statistical models to predict the concurrent reuse distance. However, this method only works for specific regions like loops. Furthermore, none of these methods can be used to analyze communication events. Our method combines communication reuse distance with thread communication analysis, which yields more comprehensive results.

Two studies [19, 27] tried to combine reuse distance analysis with other metrics. Rane et al. [27] proposed a tool comprised of different analyses and metrics, such as reuse distance analysis, cycles per access, hit ratios and access strides to investigate data structures and find out their bottlenecks. However, they only focus on data structures and neglect the effect of threads on communication. Another study [19] introduces architecture-independent metrics, including multi-core reuse distance and communication analysis

to perform performance analysis. However, no additional insights into thread communication are provided.

6 CONCLUSION AND OUTLOOK

Platform-independent communication metrics enable the analysis of parallel programs in terms of their inherent data communication without being tied to a particular hardware. In this paper, we propose a set of metrics to identify data communication bottlenecks. Communication reuse distance (CRD) reflects the effect of communication on the cache, while the communication reuse ratio (CRR) matrix sheds light on the amount of reuse after a true communication between threads. With the help of two quantitative metrics, the homogeneity and balance of CRR matrices can be easily evaluated. We showed that our method is not only able to detect communication bottlenecks in different code regions, but also can help programmers apply a suitable type of optimization. Additionally, we demonstrated that such analysis is helpful for determining the input scalability of a given application with regard to its cache usage. As a next step, we plan to compute the probability of cache misses between the minimum and maximum cutoff distances. Moreover, we aim at reducing the profiling overhead via sampling.

Acknowledgment. This research has been supported by the Klaus Tschira Foundation, the Hessian LOEWE initiative within the Software-Factory 4.0 project, German Academic Exchange Service (DAAD), the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH16008D, and the US Department of Energy under Grant No. DE-SC0015524. We thank our colleagues Mohammad Norouzi and Dr. Alexandru Calotoiu for their invaluable feedback and insights.

REFERENCES

[1] Nick Barrow-Williams, Christian Fensch, and Simon Moore. 2009. A communication characterisation of Splash-2 and Parsec. In *International Symposium on Workload Characterization (IISWC)*. IEEE, 86–97.

[2] Kristof Beyls and Erik D'Hollander. 2001. Reuse distance as a metric for cache behavior. In *IASTED Conference on Parallel and Distributed Computing and systems*, Vol. 14. 350–360.

[3] Kristof Beyls and Erik H D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223–250.

[4] Christian Bienia, Sanjeev Kumar, and Kai Li. 2008. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *International Symposium on Workload Characterization (IISWC)*. IEEE, 47–56.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 72–81.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization (IISWC)*. IEEE, 44–54.

[7] Eduardo HM Cruz, Matthias Diener, Laércio L Pilla, and Philippe OA Navaux. 2015. An efficient algorithm for communication-based task mapping. In *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 207–214.

[8] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, Mohammad S Alhakeem, Philippe OA Navaux, and Hans-Ulrich Heiß. 2015. Locality and balance for communication-aware thread mapping in multicore systems. In *European Conference on Parallel Processing*. Springer, 196–208.

[9] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, and Philippe OA Navaux. 2016. Communication in shared memory: Concepts, definitions, and efficient detection. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE, 151–158.

[10] Matthias Diener, Eduardo HM Cruz, and Philippe OA Navaux. 2015. Locality vs. balance: Exploring data mapping policies on NUMA systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 9–16.

[11] Matthias Diener, Eduardo HM Cruz, Laércio L Pilla, Fabrice Dupros, and Philippe OA Navaux. 2015. Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation* 88 (2015), 18–36.

[12] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 245–257.

[13] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 55–65.

[14] Ahmad Faraj and Xin Yuan. 2002. Communication characteristics in the NAS parallel benchmarks. In *IASTED PDCS*. 724–729.

[15] German Florez, Zhen Liu, Susan M Bridges, Anthony Skjellum, and Rayford B Vaughn. 2005. Lightweight monitoring of mpi programs in real time. *Concurrency and Computation: Practice and Experience* 17, 13 (2005), 1547–1578.

[16] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, Vol. 17. ACM, 120–126.

[17] Yunlian Jiang, Eddy Zhang, Kai Tian, and Xipeng Shen. 2010. Is reuse distance applicable to data locality analysis on chip multiprocessors?. In *Compiler Construction*. Springer, 264–282.

[18] Shoaib Kamil, John Shalf, Leonid Oliker, and David Skinner. 2005. Understanding ultra-scale application communication requirements. In *International Symposium on Workload Characterization (IISWC)*. IEEE, 178–187.

[19] Milind Kulkarni, Vijay Pai, and Derek Schuff. 2011. Towards architecture independent metrics for multicore performance analysis. *ACM SIGMETRICS Performance Evaluation Review* 38, 3 (2011), 10–14.

[20] Ingyu Lee. 2009. Characterizing communication patterns of NAS-MPI benchmark programs. In *IEEE Southeastcon 2009*. IEEE, 158–163.

[21] Zhen Li, Ali Jannesari, and Felix Wolf. 2015. An Efficient Data-Dependence Profiler for Sequential and Parallel Programs. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*.

[22] Arya Mazaheri, Ali Jannesari, Abdolreza Mirzaei, and Felix Wolf. 2015. Characterizing Loop-Level Communication Patterns in Shared Memory. In *44th International Conference on Parallel Processing (ICPP)*. IEEE, 759–768.

[23] Eduardo Henrique Molina da Cruz, Zanata Alves, Alexandre Carissimi, Philippe Olivier Alexandre Navaux, Christiane Pousa Ribeiro, and J Mehaut. 2011. Using memory access traces to map threads and data on hierarchical multi-core platforms. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 551–558.

[24] Qingpeng Niu, James Dinan, Qingda Lu, and P Sadayappan. 2012. PARDA: A fast parallel reuse distance analysis algorithm. In *26th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 1284–1294.

[25] Sean Peisert. 2010. Fingerprinting communication and computation on HPC machines. *Lawrence Berkeley National Laboratory* (2010).

[26] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: a system for flexible parallel execution. *ACM SIGPLAN Notices* 47, 6 (2012), 133–144.

[27] Ashay Rane and James Browne. 2011. Performance optimization of data structures using memory access characterization. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 570–574.

[28] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 53–63.

[29] Derek L Schuff, Benjamin S Parsons, and Vijay S Pai. 2010. Multicore-aware reuse distance analysis. In *International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1–8.

[30] Sean Whalen, Sophie Engle, Sean Peisert, and Matt Bishop. 2012. Network-theoretic classification of parallel computation patterns. *International Journal of High Performance Computing Applications* (2012).

[31] Sean Whalen, Sean Peisert, and Matt Bishop. 2013. Multiclass classification of distributed memory parallel computations. *Pattern Recognition Letters* 34, 3 (2013), 322–329.

[32] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, Vol. 23. ACM, 24–36.

[33] Meng-Ju Wu and Donald Yeung. 2013. Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. *ACM Transactions on Computer Systems (TOCS)* 31, 1 (2013).

[34] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. 2013. Studying multicore processor scaling via reuse distance analysis. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 499–510.

[35] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 6 (2009).