Communication in Shared Memory: Concepts, Definitions, and Efficient Detection

Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux
Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
Email: {mdiener,ehmcruz,mazalves,navaux}@inf.ufrgs.br

***Abstract*—Optimizing the communication behavior of parallel applications has emerged as an important topic in parallel processing. In shared memory architectures, threads communicate implicitly through memory accesses to shared memory areas. The communication behavior can be improved by mapping threads that communicate a lot to processing units that are close to each other in the memory hierarchy, such that they can benefit from shared caches and faster interconnections. An important aspect of such a communication-aware thread mapping is the accurate and efficient detection of communication in shared memory. Previous work used impromptu definitions, without an evaluation of the complexities of different communication types. In this paper, we perform an in-depth, systematic evaluation of communication in shared memory, focusing on its architectural effects. We present an efficient way to detect communication, which is orders of magnitude faster than a cache simulator, while maintaining a high accuracy.**

***Index Terms*—Communication, thread mapping, cache hierarchy, interconnections**

I. Introduction

Due to the large increase of parallelism, communication represents one of the main challenges for the efficiency of parallel applications. Parallel applications need to exchange data to perform their work, which can have a higher impact on the performance and energy consumption than the computation itself [1], [2]. Two basic strategies can be adopted to reduce this impact. First, by reducing the amount of communication (communication avoidance [3]) Second, by performing an assignment of threads to processing units that takes the inter-thread communication into account, communication can be optimized [4]. We refer to this second technique as *communication-aware thread mapping*. Most thread mapping proposals focus on improving *locality*, where threads that communicate a lot are mapped close to each other in the system, to make use of faster interconnections and shared cache memories.

An improved thread mapping impacts the hardware architecture, reducing the number of cache misses, due to more available cache space and less invalidations [5]. Such a mapping also results in less traffic on inter- and intra-chip interconnections, due to fewer cache-to-cache transfers and invalidation messages [6]. These optimizations result in improved performance and energy efficiency of parallel applications. For this reason, it is necessary to describe and evaluate the architectural effects of communication in order to perform an optimized thread mapping.

A critical step of thread mapping is the analysis of the structure of communication, which we call the communication *pattern*, since it determines the mapping that should be applied, as well as the gains that can be achieved. In shared memory architectures and programming models, such as OpenMP and Pthreads, communication is *implicit* and happens through memory accesses to shared data. This presents additional challenges compared to message-passing models such as MPI, because the communication behavior depends on many architectural parameters, such as the cache line size and cache organization. Furthermore, memory accesses are generally hard to study with an acceptable overhead. For these reasons, communication in shared memory is usually described with impromptu definitions, with various granularities [7], sampling strategies [8], or without a focus on the architectural impacts of communication [9]. This can lead to incorrect thread mapping decisions that do not result in optimal gains for many applications.

For a correct and efficient way to detect the communication of parallel applications based on shared memory, this paper makes two main contributions, discussing how communication can be defined and detected efficiently. We first present a systematic description of communication behavior in shared memory, focusing on the various types of communication and their architectural effects. We introduce a method to accurately describe communication, which requires a cache simulator for detection. By relaxing this accurate definition, we construct a new technique that provides a very high accuracy while drastically reducing the overhead of the communication detection. We evaluate the techniques with applications from two parallel benchmark suites.

II. Related Work

Related work that characterizes communication mostly focuses on applications that use explicit message passing frameworks, such as MPI. Examples include [10], [11], [12]. A characterization methodology for explicit communication is presented in [13], [14], where communication is described with *temporal*, *spatial* and *volume* components. We use similar components to describe communication, but apply them in the context of shared memory, where communication is performed implicitly through memory accesses to memory areas that are shared between different threads. Barrow-Williams et al. [9] perform a communication analysis of the PARSEC and Splash2 benchmark suites. They focus on communication on

978-1-4673-8776-7/16 $31.00 © 2016 IEEE
DOI 10.1109/PDP.2016.16

the logical level and therefore only count memory accesses that really represent communication, filtering out memory accesses that occur due to register pressure for example. As we are interested in the architectural effects of communication, we take into account all memory accesses for the characterization.

Most mechanisms that perform communication-aware thread mapping use an informal definition of communication, use hardware or software based memory access sampling with varying granularities, or use other indirect metrics that do not accurately represent communication [7], [8]. Some automatic tools, such as BlackBox [15], perform thread mapping by measuring the IPC of various mappings and selecting the mapping with the highest performance. A related type of proposal that affects communication is based on communication avoidance [3]. Such proposals focus on reducing the impact of communication by reducing the amount of data that needs to be communicated. Even a reduced amount of communication can be optimized with a better thread mapping.

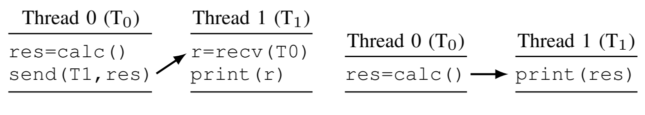
In this work, we introduce a mechanism to describe communication with a higher accuracy as well as a lower detection overhead, leading to better thread mapping solutions.

III. Communication in Shared Memory

Describing the communication behavior presents several challenges that need to be addressed. In this section, we will present definitions of communication in shared memory architectures and discuss their impact on the behavior detection, as well as the thread mapping.

A. Explicit and Implicit Communication

Parallel programming models use different forms of communication. Communication can be *explicit*, where `send()` and `receive()` functions exchange messages between threads, as shown in Figure 1a. In *implicit* communication, communication is performed directly through memory accesses to shared variables, without using explicit functions to communicate, as shown in Figure 1b. Explicit communication supports communication in distributed environments through message transmission over network protocols, such as TCP/IP for nodes interconnected via Ethernet. Implicit communication requires that threads share a physical address space and is therefore limited to shared memory architectures. However, implicit communication has a lower overhead than explicit communication, since it only requires a memory access, while explicit communication has the additional overhead of the socket and packet encapsulation, among others [16].



```
Thread 0 (T0)        Thread 1 (T1)
res=calc()           r=recv(T0)
send(T1,res)         print(r)
```

```
Thread 0 (T0)        Thread 1 (T1)
res=calc()     →     print(res)
```

(a) Explicit communication. (b) Implicit communication.

Fig. 1: Explicit and implicit communication between two threads T_0 and T_1. Arrows indicate communication.

Programming APIs for explicit communication include the Message Passing Interface (MPI) [17] and Charm++ [18], while OpenMP [19] and Pthreads [20] use implicit communication. Since communication via shared memory has a lower overhead [21], many implementations of MPI contain extensions to communicate via shared memory within cluster nodes, such as Nemesis [16] for MPICH2. The extensions allocate a shared memory segment for communication and transform the MPI function calls such that they access these shared segments for communication, bypassing the network layer [16]. For this reason, both explicit and implicit communication can be optimized by improving memory accesses in shared memory architectures [22].

B. True/False Communication and Communication Events

In explicit communication, all communication is *true*, that is, every call to a communication function represents an intention to exchange data between threads. In implicit communication however, not every memory access to shared data by different threads necessarily implies an intention to communicate. We refer to this unintentional communication as *false* communication, which can be further divided into *spatial*, *temporal*, and *logical* false communication. All types of false communication are caused by the way that the hardware architecture, especially the caches and interconnections, work. An overview of the true and false communication types is shown in Figure 2 for two threads T_0 and T_1 that access the same cache line (gray box). The line consists of 4 words.

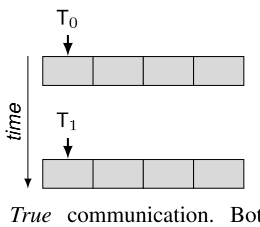
When two threads access the same word in the same cache line while the line is not evicted and the second access is not an unnecessary reload, we call this access *true* communication (Figure 2a).

Spatial false communication happens because the granularity of cache lines and interconnections is larger than the granularity of memory accesses, similar to the classic false sharing problem [23]. As an example, consider that two threads perform memory accesses to the same cache line, but at different offsets within the same line, as shown in Figure 2b. This access is not true communication, as it does not represent an intention to transfer data. However, the architecture treats this access in exactly the same way as it would treat an access to the same offset, in terms of the cache coherence protocol, invalidation and transfer of cache lines. Since we are mostly interested in the architectural effects of communication, we include spatial false communication on the cache line granularity in our definition of communication. In this way, communication-aware mapping can improve accesses to truly shared data, and can reduce the negative impact of false sharing.

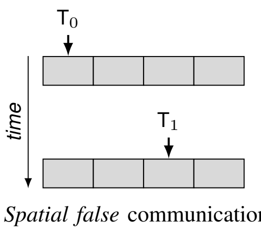
Temporal false communication happens when two threads access the same memory address, but at different times during the execution, such that at the time of the second access, the cache line is not in the caches anymore and needs to be fetched from the main memory. This situation is shown in Figure 2c. This type of false communication is very dependent on the configuration and size of the caches. It can present difficulties for communication detection mechanisms that rely on memory traces and do not have a way to filter communication with a low temporal locality. Since temporal false communication affects the architectural impact of communication, we will reduce its impact by taking into account the temporal locality in our mechanisms.

Logical false communication happens due restrictions of the hardware architecture, especially due to the limited number of registers. For example, if an application requires more registers at the same time than the hardware provides, the compiler needs to spill a register to the memory and re-read the value at a later time. Since this behavior does not constitute an exchange of data, this second access is logical false communication. However, similarly to the spatial false communication, it also affects the architecture. Therefore, we also consider these accesses as communication, in contrast to previous work that focuses on the logical communication behavior [9].

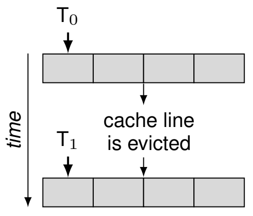
Summarizing this discussion, we will consider spatial and logical false communication in the same way as true communication in this paper, and will filter temporal false communication in our mechanisms. With these considerations, we introduce the concept of a *communication event*, which we define as two memory accesses from different threads to the same cache line while the cache line is not evicted. Some of our mechanisms will relax this definition, by increasing the granularity of the detection to a value that is larger than the cache line size, and by using simpler definitions



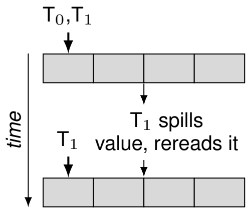
(a) *True* communication. Both threads access the same word in the same cache line while the line is not evicted.



(b) *Spatial false* communication. Both threads access the same cache line, but access different words within the same line.



(c) *Temporal false* communication. Both threads access the same word in the same cache line, but the line is evicted between accesses.



(d) *Logical false* communication. Both threads communicate, but T_1 reloads the same value at a later time due to high register pressure.

Fig. 2: Comparison between true and false communication. Consider that two threads T_0 and T_1 access the same cache line (gray box), which consists of 4 words.

of temporal false communication that are independent of the cache configuration.

C. Read and Write Memory Accesses

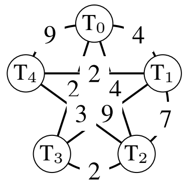
Write operations are generally more expensive than reads, since they imply the invalidation of cache lines in remote caches, requiring more traffic on on-chip interconnections than the cache-to-cache transfers that are caused by read operations. However, read memory accesses are much more numerous than writes. For example, 71.1% of memory transactions in the (sequential) SPEC CPU 2006 benchmark suite [24] are read operations, while they make up 78.1% in the PARSEC suite [25]. Read accesses also have higher chances to stall the pipeline, since they generate more dependencies.

Moreover, the processor needs to wait for a read operation to finish in order to be able to continue operating with the just loaded cache line, which might involve waiting for the main memory. This latency can not always be hidden with Out-of-Order (OoO) execution. On the other hand, write operations are mostly asynchronous. After issuing the write, the processor only needs to wait for an acknowledgment from the L1 data cache to be able to continue with the next instruction. For these reasons, we consider both read and write memory accesses equivalently for the description of communication.

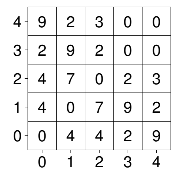
D. Communication Direction and Communication Matrix

In explicit communication, each communication operation has a well-defined sender and receiver (or a group of multiple receivers), in other words, communication is *directed*. In implicit communication however, determining the sender and receiver of communication is much more difficult. Three types of communication events can be defined for implicit communication, depending on whether data is read or written by two threads. These types are *read/read*, *read/write*, and *write/write*. In the *read/read* case, both threads perform read memory accesses to the same cache line, in order to read input data for example. No thread can be identified as the sender/receiver as they perform the same operation. In the *read/write* case, one thread writes data which is read by the other thread. In this case, the writing thread can be considered the sender, and the reading thread the receiver. In the *write/write* case, similar to the read/read case, sender and receiver can also not be identified. Since direction can not be determined in the majority of cases, we treat communication in shared memory as *undirected* in this paper.

With the information about the communication events, it is possible to create an undirected communication graph, where nodes represent threads and edges the number of communication events between each pair of threads. An example of such a graph is shown in Figure 3a for a parallel application consisting of five threads. This type of graph is also referred to as a Task Interaction Graph (TIG) in the literature [26]. In practice, this communication graph is represented as a matrix, which we call *communication matrix* or *communication pattern*. An example communication matrix for the previous graph is shown in Figure 3b. Each cell of

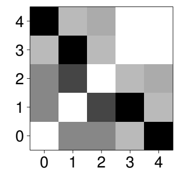


(a) Weighted task interaction graph (TIG). Edges contain the amount of communication between threads. T$_3$ and T$_4$ do not communicate.



4	9	2	3	0	0
3	2	9	2	0	0
2	4	7	0	2	3
1	4	0	7	9	2
0	0	4	4	2	9
	0	1	2	3	4

(b) Communication matrix for the TIG of (a). Cells contain the amount of communication between threads. Axes show thread IDs.



(c) Visualization of the communication matrix shown in (b). Darker cells indicate more communication between the pair of threads.

Fig. 3: Three representations of undirected communication for a parallel application consisting of five threads, T$_0$ – T$_4$.

the matrix contains the number of communication events for the thread pairs, while the axes contain the thread IDs. Since we consider that communication is undirected, the matrix is symmetric. Furthermore, the diagonal of the matrix is kept zero for the discussion in the paper, as memory accesses by the same thread do not constitute communication. Finally, to analyze and discuss the communication patterns, we generally normalize the matrices to their maximum value, to limit the range of values between 0 and 100, for example. To better visualize the communication pattern, we depict the normalized matrix in the form of a heat map, where darker cells indicate more communication. An example of this visualization is shown in Figure 3c.

E. Comparing Communication Patterns

An important aspect of communication is the question of how to compare different communication behaviors. Since a communication matrix can be thought of as a grayscale image, we use a concept from image comparison to compare different matrices. To compare two normalized communication matrices A and B that have the same numbers of threads, we calculate the Mean Squared Error (MSE) [27] with Equation 1, where N is the number of threads of each matrix. If A and B are equal, the MSE is equal to zero. The MSE is maximized when only a single pair of threads communicates in one matrix and all threads except that pair communicate equally in the other matrix. In that case, the MSE is given by $\frac{N^2-N}{N^2} \times \max(M)^2$, where $\max(M)$ is the maximum value of both matrices (the value that the matrices are normalized to).

$$\mathit{MSE\ (A,B)} = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \big(A[i][j] - B[i][j]\big)^2 \qquad (1)$$

With the MSE, it is possible to compare different communication behaviors with each other, as well as to measure the accuracy of different communication detection mechanisms.

IV. A Relaxed Definition of Communication

The definition of communication presented in Section III-B is the most accurate definition to analyze the architectural impact of communication, but it has three disadvantages. First, since it is based directly on the size and configuration of the cache levels, different cache configurations might result in a different communication behavior, making it less useful to describe the application behavior itself. Second, determining the communication behavior with the accurate definition requires either analyzing the application in a full cache simulator, which has a high overhead, or access to the contents of the cache on real hardware, which is not possible on most modern architectures. Third, storing and analyzing communication on the cache line granularity (64 bytes in most current architectures) has a high storage overhead due to the need to save large amounts of data. This overhead can be reduced by increasing the granularity of the analysis to large sizes than the cache line size. For these reasons, we present a relaxed definition of communication and compare it to the accurate definition in this section.

We relax the accurate definition of communication in the following ways. First, we remove the requirement on the cache hierarchy and consider all accesses to memory addresses on a granularity derived by a common cache line size as communication events. To reduce the impact of temporal false communication, we maintain a small queue of the two most recent threads that accessed each cache line. Second, we increase the granularity to a higher value than the cache

Algorithm 1: The relaxed definition of communication.

```
Input: address: memory address that was accessed;
       tid: thread ID of the thread that performed the access;
       g: granularity of detection, number bits to shift
Output: updates communication events
// memory block of the address, contains a
   queue of up to 2 threads:
block = address >> g;
// number of threads that accessed the
   block; can be 0, 1 or 2:
nthreads = block.size();
if nthreads == 0 then
    // no previous access
    block.push_back(tid);
if nthreads == 1 && block.front() != tid then
    // 1 previous access
    communication_event(block.front(), tid);
    block.push_back(tid);
if nthreads == 2 then
    // 2 previous accesses
    t1 = block.front();
    t2 = block.back();
    if t1 != tid && t2 != tid then
        communication_event(t1, tid);
        communication_event(t2, tid);
        block.pop_front();
        block.push_back(tid);
    else if t1 == tid then
        communication_event(t2, tid);
    else if t2 == tid then
        communication_event(t1, tid);
        block.pop_front();
        block.push_back(tid);
```

line size, separating the memory address space into memory blocks. Algorithm 1 shows the function that is executed on each memory access. The memory block is calculated by bit shifting the address with the chosen granularity. The block contains a queue that stores the ID of the previously accessing threads. Then, the number of threads that previously accessed the block are counted. If other threads had accessed the block before, communication events are recorded and the queue is updated.

V. Methodology

This section presents the experimental methodology.

A. Simulation Environments

1) Simulated Machine: We simulate a machine consisting of 64 PUs with a cache hierarchy inspired by the Intel Nehalem microarchitecture [28], which is the base of our real evaluation system. 32 L1 caches (size: 32 KByte) and L2 caches (size: 256 KByte) are shared between pairs of PUs, while the L3 cache (size: 18 MByte) is shared between 16 PUs.

2) Accurate Definition of Communication: The accurate communication is generated with a full cache simulator[1] based on the Pin dynamic binary instrumentation tool [29]. The tool traces all memory accesses of a parallel application and simulates a 64-core architecture with a 3-level cache hierarchy. We evaluate *true*, *spatial false* and *temporal false* communication. We did not find an automated way to measure *logical false* communication, and therefore do not consider it separately in this discussion. Depending on its particular form, the logical false communication is included in one of the other three communication types. We calculate the temporal false communication by simulating an infinite last level cache. In this way, repeated accesses to the same cache line from different threads will always be counted as communication. The difference between amounts of communication with the limited and infinite cache is the temporal false communication.

3) Relaxed Definition of Communication: For the relaxed definition of communication, we developed a custom memory tracer based on Pin, for the application characterization as part of this paper[2]. The tool records all memory accesses of all threads, storing the address and thread ID of each memory access. For each access, an analysis routine for the relaxed definition of communication is executed, as described in Section IV.

B. Real Machine

We use a real machine to compare the overhead of communication detection with the proposed methods and to evaluate the performance gains of thread mapping. This machine consists of four 8-core Intel Xeon processors (Nehalem microarchitecture [28]) that support 2-way SMT. The L1 and L2 caches are private to each core, while the L3 caches are shared among all the cores on each processor. The machine can execute up to 64 threads in total. Table I contains an overview of the configuration parameters of the real machine.

[1]The simulator is available at https://github.com/matthiasdiener/CacheSim
[2]The tracer is available at https://github.com/matthiasdiener/numalize

TABLE I: Overview of the real machine used in the evaluation.

Property	Value
Processors	4× Intel Xeon X7550, 2.0 GHz, 8 cores, 2-way SMT
Caches per proc.	8× 32 KB+32 KB L1, 8× 256 KB L2, 18 MB L3
Memory	128 GB DDR3-1066, page size 4 KB
Operating system	Ubuntu 12.04, Linux kernel 3.8 (CFS scheduler), 64 bit

C. Parallel Applications

For the experiments, we selected five applications from two different parallel benchmark suites, which have communication behaviors that are representative for all other applications in these two suites. From the OpenMP implementation of the NAS Parallel Benchmarks [30] (NAS-OMP), we selected the LU and UA benchmarks. Both were executed with the *A* input size. We also selected Blackscholes, Ferret, and Swaptions from the PARSEC benchmark suite [25], which are applications implemented with Pthreads. PARSEC applications were executed with the *simlarge* input size. All benchmarks have a stable communication behavior, with only minimal changes between or during executions.

VI. Experimental Evaluation

This section presents the results of our experiments, discussing the communication behavior, accuracy of the communication definitions, detection overheads, as well as performance gains from thread mapping.

A. Communication Behavior of the Benchmarks

We begin with an evaluation of the communication behavior of the benchmarks, discussing the amount of different types of communication as well as the patterns.

1) Communication Statistics: Figure 4 presents the communication statistics of the benchmarks running with 64 threads, calculated with the cache simulator. We show the number of communication events for true, spatial false, and temporal false communication, as well as the number of memory accesses to cache lines that were only accessed by a single thread during the whole execution, labeled *Private* in the graph. The y-axis is scaled logarithmically.

Several important conclusions can be drawn from these results. First of all, the amount of communication is widely different between benchmarks. For example, although LU and UA have the same amount of private memory accesses, UA has several orders of magnitude more communication events, which can indicate that thread mapping is more beneficial for this benchmark. All applications have significant amounts of spatial false communication, similar but slightly lower than the true communication. Moreover, temporal false communication is the highest form of communication in all benchmarks. This indicates that it is important to filter this communication type in order not to reach wrong conclusions regarding the behavior and the thread mapping.

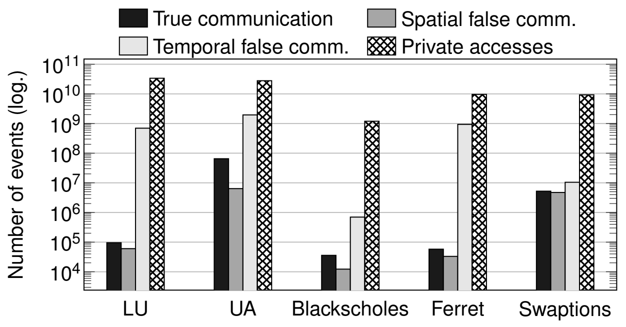
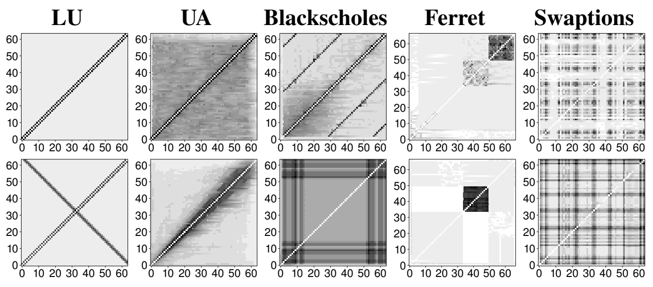


Fig. 4: Communication behavior of the benchmarks, measured with the cache simulator, showing the number of communication events (for true, spatial false, and temporal false communication), as well as private memory accesses (accesses to cache lines that were never accessed by more than one thread).



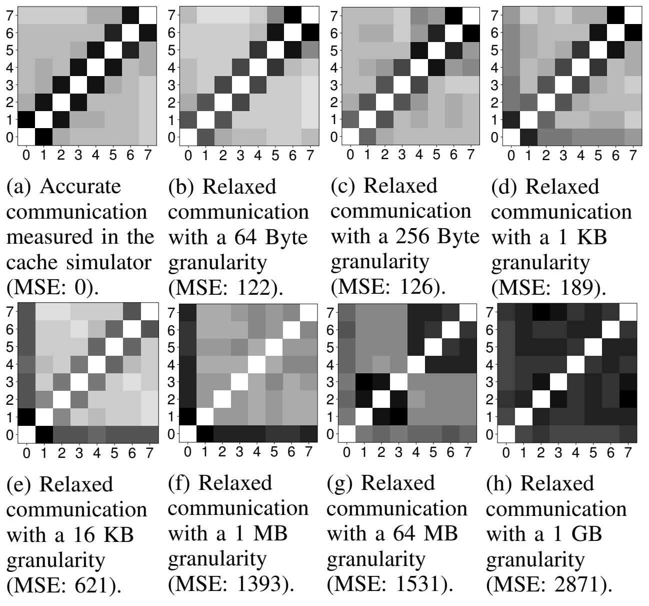
MSE: 543 MSE: 1229 MSE: 968 MSE: 837 MSE: 137

Fig. 5: Communication patterns of the benchmarks, calculated with the accurate definition (top row). The bottom row of pattens includes temporal false communication.

2) Communication Patterns: The detected communication patterns are shown in Figure 5. The top row shows the patterns of the true and spatial false communication (which are relevant for the mapping), while the bottom row includes the temporal false communication as well. The figure also shows the MSE between both patterns for all benchmarks, as introduced in Section III-D. As indicated by the communication statistics, temporal false communication can change the detected communication behavior substantially, resulting in potentially wrong mapping decisions. All applications except Swaptions show significant differences and have high MSEs.

The nearest-neighbor communication pattern of LU, where thread pairs with close thread IDs (0,1), (1,2), ..., communicate a lot, has additional false temporal communication between threads that are far apart, such as pairs (0,63). UA has a less pronounced nearest-neighbor pattern, where also threads that are not direct neighbors communicate. With the temporal false communication, this general pattern remains similar, but the differences between neighboring threads and threads that are farther apart increase, which can result in an overestimation of the benefits of mapping.

Blackscholes' communication structure is completely modified by the temporal false communication. The Ferret benchmark consists of four pipeline stages, where the last two stages perform most of the communication. When considering temporal false communication, only the third phase appears to communicate, potentially resulting in a not optimal thread mapping. Swaptions has similar amounts of communication between most thread pairs. This pattern changes only slightly when including temporal false communication, indicated by the low MSE.



(a) Accurate communication measured in the cache simulator (MSE: 0). (b) Relaxed communication with a 64 Byte granularity (MSE: 122). (c) Relaxed communication with a 256 Byte granularity (MSE: 126). (d) Relaxed communication with a 1 KB granularity (MSE: 189). (e) Relaxed communication with a 16 KB granularity (MSE: 621). (f) Relaxed communication with a 1 MB granularity (MSE: 1393). (g) Relaxed communication with a 64 MB granularity (MSE: 1531). (h) Relaxed communication with a 1 GB granularity (MSE: 2871).

Fig. 6: Comparison of the communication matrices of the UA benchmark, generated with the accurate (a) and relaxed (b) – (h) definitions of communication with different detection granularities. The MSE is calculated with the accurate matrix (a).

3) Summary: Communication behaviors differ considerably between applications. We expect more gains from communication-aware thread mapping with patterns that have a clearer structure. We also conclude that temporal false communication has a high impact on the communication pattern, due to its high amount and different structure than true and spatial false communication. Therefore, reducing temporal false communication is important when performing thread mapping in order to apply the correct mapping.

B. Accuracy of the Relaxed Communication Detection

We compare the accurate and relaxed definitions of communication by measuring the MSE of the generated communication matrices. We also evaluate the impact of various detection granularities (larger than the cache line size) on the accuracy of the detection.

1) UA Benchmark: As an example, we analyze the behavior of the UA benchmark from NAS-OMP, which has a high sensitivity to these characteristics. For a better visualization, we show the results with 8 threads.

In Figure 6, the communication matrices of the different detection mechanisms are shown. The baseline of our evalua-

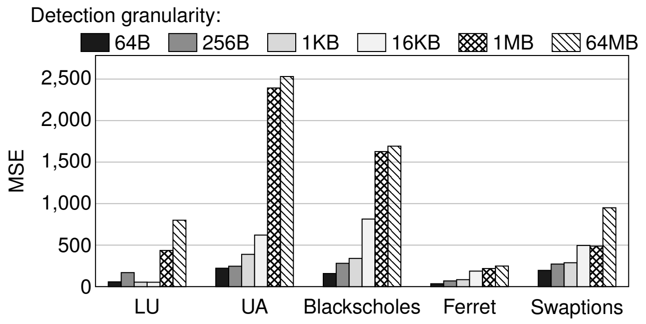


Fig. 7: MSE of the relaxed communication definition (with different granularities) compared to the accurate definition, considering 64 threads. Lower values are better.

tion, the matrix generated with the cache simulator (including true and spatial false communication) is shown in Figure 6a. Figures 6b – 6h show the matrices generated with the relaxed definition and increasing granularity of memory blocks. The figure also contains the values of the MSE, calculated between the baseline and each matrix generated with the relaxed definition. Higher MSEs indicate a higher inaccuracy of the detected communication. In this configuration with 8 threads, the maximum possible MSE is 8750.

The results show that the communication detected with the relaxed definition remains very accurate up to a granularity of 1 KB, with low values for the MSE and matrices that are visually similar to the baseline. When increasing the granularity to values above 1 KB, the MSE keeps rising and the matrices lose their similarity to the baseline, with a complete divergence starting at about 1 MB.

2) All Benchmarks: Figure 7 presents the MSE of the relaxed definition for all benchmarks, varying the granularity of detection, considering an execution with 64 threads. Similar to the results of UA, we find that detection accuracy remains good with granularities of up to 1–16 KByte. Ferret is the least sensitive to the block size. These results show that the relaxed definition successfully filters the temporal false communication, indicated by the fact that the MSE is substantially lower than the MSE of the accurate+temporal false communication discussed in Section VI-A2.

C. Overhead of Communication Detection

Table II shows the overhead of both communication detection mechanisms, presented as the number of times application execution is slower compared to execution without detection. As expected, generating the communication with the relaxed definition is much faster than the cache simulator that is needed for the accurate measurement. In some cases, such as LU and Ferret, the relaxed detection results in an execution more than 100 times faster than the cache simulator. These results also show that communication detection can be reasonably performed even for large applications with the relaxed detection, since the detection has to be run only once.

TABLE II: Overhead of the communication detection mechanisms. Increase in execution time compared to the non-instrumented execution. Lower values are better.

Mechanism	LU	UA	Blackscholes	Ferret	Swaptions
Accurate	5944×	2860×	1771×	5304×	6157×
Relaxed	49×	72×	113×	39×	148×

D. Application Performance with Thread Mapping

We evaluate the performance impact of mapping using the communication behavior detected with the mechanisms.

1) Methodology: Performance improvements of thread mapping were measured on the real machine presented in Section V-B. The baseline for the experiments is the default thread mapping by the Linux Completely Fair Scheduler (CFS) of kernel 3.8. We calculate optimized mappings from the detected communication behaviors with the EagerMap mapping algorithm [31]. EagerMap receives the communication matrix and a description of the hardware hierarchy as input, and outputs a thread mapping that optimizes overall communication locality. We evaluate the communication matrix detected by the accurate definition (with and without temporal false communication) and various granularities of the relaxed definition. For each mapping, we measured the average execution time of 10 executions, and present the performance gains compared to the OS mapping.

2) Results: The performance gains are shown in Figure 8. All benchmarks except Swaptions have significant gains from mapping, reaching up to 37% in the case of LU. Swaptions can not benefit from mapping due to its unstructured communication behavior, where all pairs of threads have similar amounts of communication. It is important to mention that taking temporal false communication into account results in substantially lower performance improvements, as already indicated by our accuracy measurements. In several cases, performance is reduced compared to the OS mapping.

Regarding the performance improvements with the relaxed definition communication behavior, we find that with the

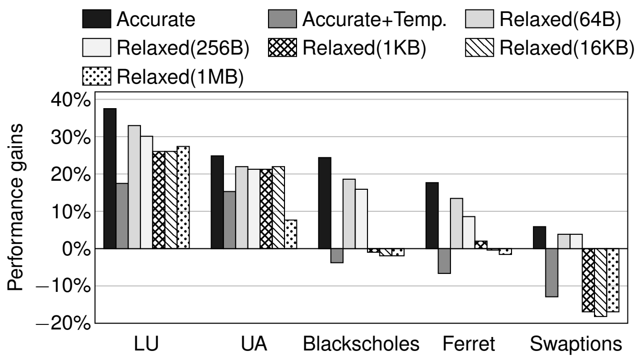


Fig. 8: Performance gains of thread mapping with various detected communication behaviors, compared to the OS. Higher values are better.

64 bytes and 256 bytes granularities, gains are very similar to the improvements with the accurate detection. For most benchmarks, increasing the granularity reduces gains, as expected. For the three PARSEC applications, the high granularities (starting from 1 KB) result in performance losses, while the NAS-OMP remain more stable. These results indicate that communication can be analyzed with the relaxed definition and a granularity of less than 1 KByte with a high accuracy.

VII. Conclusions

Improving the communication behavior of parallel applications is one of the main challenges for optimal performance. In shared-memory architectures, communication can be optimized by mapping threads that communicate a lot to cores that are close to each other in the memory hierarchy, improving the usage of caches and interconnections. For a successful mapping, it is important to determine the communication behavior in an accurate and efficient way. In this paper, we performed an in-depth investigation of the types of communication, as well as their impact on the hardware architecture and the performance improvements. We introduced an optimized, tracing-based mechanism to detect communication that is orders of magnitude faster than a full cache simulator but maintains a very high level of accuracy.

Acknowledgments

This work was partially supported by CNPq and Capes. It was developed in the context of the HPC4E project of RNP/EU.

References

[1] W. J. Dally, “GPU Computing to Exascale and Beyond,” Tech. Rep., 2010.

[2] J. Shalf, S. Dosanjh, and J. Morrison, “Exascale Computing Technology Challenges,” in *High Performance Computing for Computational Science (VECPAR)*, 2010, pp. 1–25.

[3] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz, “Communication lower bounds and optimal algorithms for numerical linear algebra,” *Acta Numerica*, vol. 23, no. May, pp. 1–155, 2014.

[4] J. Zhai, T. Sheng, and J. He, “Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 22, no. 11, pp. 1862–1870, 2011.

[5] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, “Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[6] W. Wang, T. Dey, J. Mars, L. Tang, J. W. Davidson, and M. L. Soffa, “Performance Analysis of Thread Mappings with a Holistic View of the Hardware Resources,” in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2012.

[7] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, “Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring,” *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 56–65, Apr. 2009.

[8] M. Diener, E. H. M. Cruz, and P. O. A. Navaux, “Communication-Based Mapping Using Shared Pages,” in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013, pp. 700–711.

[9] N. Barrow-Williams, C. Fensch, and S. Moore, “A Communication Characterisation of Splash-2 and Parsec,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 86–97.

[10] A. Faraj and X. Yuan, “Communication characteristics in the NAS parallel benchmarks,” in *Parallel and Distributed Computing and Systems (PDCS)*, 2002, pp. 724–729.

[11] I. Lee, “Characterizing communication patterns of NAS-MPI benchmark programs,” in *IEEE Southeastcon*, 2009, pp. 158–163.

[12] J. Kim and D. J. Lilja, “Characterization of communication patterns in message-passing parallel scientific application programs,” in *International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC)*, 1998, pp. 202–216.

[13] S. Chodnekar, V. Srinivasan, A. S. Vaidya, A. Sivasubramaniam, and C. R. Das, “Towards a communication characterization methodology for parallel applications,” in *International Symposium on High Performance Computer Architecture (HPCA)*, 1997, pp. 310–319.

[14] J. P. Singh, E. Rothberg, and A. Gupta, “Modeling Communication in Parallel Algorithms: A Fruitful Interaction between Theory and Systems?” in *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1994, pp. 189–199.

[15] P. Radojković, V. Cakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, “Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 12, pp. 2513–2525, 2013.

[16] D. Buntinas, G. Mercier, and W. Gropp, “Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem,” in *International Symposium on Cluster Computing and the Grid (CCGRID)*, 2006, pp. 521–530.

[17] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard,” Tech. Rep., 2012.

[18] L. V. Kale and S. Krishnan, “CHARM++: A Portable Concurrent Object Oriented System Based On C++,” in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1993, pp. 91–108.

[19] L. Dagum and R. Menon, “OpenMP: An Industry-Standard API for Shared-Memory Programming,” *IEEE Computational Science and Engineering (CSE)*, vol. 5, no. 1, pp. 46–55, 1998.

[20] D. Buttlar and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*, 1996.

[21] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, “Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis,” in *International Conference on Parallel Processing (ICPP)*, 2009, pp. 462–469.

[22] M. Diener, E. H. M. Cruz, P. O. A. Navaux, A. Busse, and H.-U. Heiß, “Communication-Aware Process and Thread Mapping Using Online Communication Detection,” *Parallel Computing*, vol. 43, no. March, pp. 43–63, 2015.

[23] W. J. Bolosky and M. L. Scott, “False sharing and its effect on shared memory performance,” *USENIX Systems on USENIX Experiences*, vol. 1801, no. 14520052, pp. 1–15, 1993.

[24] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, X. Tian, M. Girkar, H. Saito, and U. Banerjee, “Comparative Architectural Characterization of SPEC CPU2000 and CPU2006 Benchmarks on the Intel Core 2 Duo Processor,” in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2008, pp. 132–141.

[25] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72–81.

[26] D. L. Long and L. A. Clarke, “Task Interaction Graphs for Concurrency Analysis,” in *International Conference on Software Engineering*, 1989, pp. 44–52.

[27] A. Hore and D. Ziou, “Image Quality Metrics: PSNR vs. SSIM,” in *International Conference on Pattern Recognition*, Aug. 2010, pp. 2366–2369.

[28] Intel, “Intel® Xeon® Processor 7500 Series,” Tech. Rep. March, 2010.

[29] C. Luk, R. Cohn, R. Muth, and H. Patil, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005, pp. 190–200.

[30] H. Jin, M. Frumkin, and J. Yan, “The OpenMP implementation of NAS Parallel Benchmarks and Its Performance,” NASA, Tech. Rep. October, 1999.

[31] E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux, “An Efficient Algorithm for Communication-Based Task Mapping,” in *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2015, pp. 207–214.