J. Parallel Distrib. Comput. 74 (2014) 2215–2228

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols

Eduardo H.M. Cruz *, Matthias Diener, Marco A.Z. Alves, Philippe O.A. Navaux

Informatics Institute, Federal University of Rio Grande do Sul (UFRGS), Av. Bento Gonçalves, 9500, Campus do Vale, Bloco IV, Lab 201-67, Postal Code 91501-970, Porto Alegre, RS, Brazil

HIGHLIGHTS

- We detect the inter-thread communication in shared memory applications.
- Using the detected communication, we map the threads to improve performance.
- Provide a better usage of hardware resources.
- We reduce execution time, cache misses and traffic on interconnections.
- No need to modify applications or runtime environment.

ARTICLE INFO

Article history:
Received 17 October 2012
Received in revised form
21 November 2013
Accepted 25 November 2013
Available online 6 December 2013

Keywords:
Thread mapping
Cache coherence protocols
Parallel applications
Shared memory
Thread communication
Communication pattern

ABSTRACT

In current computer architectures, the communication performance between threads varies depending on the memory hierarchy. This performance difference must be considered when mapping parallel applications to processor cores. In parallel applications based on the shared memory paradigm, the communication is difficult to detect because it is implicit. Furthermore, dynamic mapping introduces several challenges, since it needs to find a suitable mapping and migrate the threads with a low overhead during the execution of the application. We propose a mechanism to detect the communication pattern of shared memory applications by monitoring cache coherence protocols. We also propose heuristics that, combined with our communication detection mechanism, allow the mapping to be performed dynamically by the operating system. Experiments with the NAS Parallel Benchmarks showed a reduction of up to 13.9% of the execution time, 30.5% of the cache misses and 39.4% of the number of invalidation messages.

1. Introduction

One of the main concerns regarding multi-core architectures is the communication between threads [24]. Communication implies data movement between the cores and impacts the performance and energy efficiency of parallel applications [6]. In most multi-core architectures, some levels of the memory hierarchy are shared by more than one core, which causes a difference in the communication latencies and bandwidths between the cores. Moreover, some architectures have more than one processor, each consisting of several cores. These architectures usually have several levels of memory hierarchy, such that the differences between the communication performance and the overhead due to communication are high. With the upcoming increase of the number of cores, an even higher communication overhead is expected, requiring novel solutions to allow the performance to scale [10].

Thread mapping can help to improve performance by mapping the threads to cores according to a certain policy, such that the usage of the resources is optimized. By mapping the threads considering the amount of communication between them, threads that communicate a lot are mapped to nearby cores on the memory hierarchy. Thereby, the communication performance between these threads is increased. We refer to this type of thread mapping as *communication-aware* thread mapping. In multi-core architectures, some cache lines are replicated in more than one cache, requiring protocols to maintain coherence among all the caches that have replicated cache lines [9]. These coherence protocols invalidate replicated cache lines on every write transaction, which causes a large overhead for communication intensive applications. By mapping the threads considering their communication, there are fewer cache line replications and invalidations, optimizing the usage of cache memories and interconnections.

* Corresponding author.
E-mail addresses: ehmcruz@inf.ufrgs.br, eduardohmdacruz@gmail.com (E.H.M. Cruz), mdiener@inf.ufrgs.br (M. Diener), mazalves@inf.ufrgs.br (M.A.Z. Alves), navaux@inf.ufrgs.br (P.O.A. Navaux).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.11.006

The difficulty to obtain the communication pattern between the threads depends on the parallel programming paradigm. In applications that use the messaging passing paradigm to communicate, detecting the communication pattern can be accomplished by monitoring the origin and destination fields of the messages [23,8,24]. In the shared memory programming model, where our mechanism works, the detection of the communication presents different challenges. The reason is that the communication between the threads is performed implicitly, whenever a thread reads or writes data that is shared between several threads.

Another factor that influences the difficulty of communication-aware thread mapping is whether the mapping is performed statically or dynamically. In static thread mapping, the information on the communication pattern is gathered by profiling the application in a previous execution, using controlled environments such as simulators [3,4]. Static mapping is not suitable if the application has a dynamic behavior, such as programs that use work-stealing algorithms or applications whose behavior depends on input parameters. In order to support applications with static and dynamic communication behaviors, dynamic thread mapping needs to be used, where the detection of the communication pattern and the mapping is performed during the execution of the application.

In this paper, we propose a new lightweight, dynamic mechanism to detect the communication pattern of parallel applications based on shared memory. Our proposed mechanism makes use of cache coherence protocols. It is based on the fundamental idea that a cache line shared by more than one cache indicates that more than one core is accessing the same memory location. These accesses to the same cache line represent communication between the involved threads. Our detection mechanism makes use of invalidation messages of cache coherence protocols to estimate the amount of communication, without changing the protocols themselves.

We also propose a mechanism to dynamically map the threads with a low overhead. The mechanism consists of several steps, including algorithms to detect changes in the communication pattern and to map threads to cores. These detection and mapping mechanisms allow thread mapping to be performed dynamically by the operating system, and do not require simulation or any changes to the source code of the applications.

The remainder of this paper is organized as follows. In the next section, we give an overview of the benefits of optimizing communication in shared-memory architectures and evaluate the theoretical improvements achievable with an oracle mechanism. Section 3 introduces our mechanism to detect inter-thread communication using the cache coherence protocol. Section 4 presents the algorithms that use the detected communication behavior to map threads to cores. In Section 5, we evaluate our proposed mechanism and its overhead. Related work is analyzed in Section 6. Finally, Section 7 summarizes our conclusions and presents ideas for future work.

2. Background: communication-aware thread mapping in shared-memory architectures

In most multi-core architectures, some levels of the memory hierarchy are shared by more than one core, which causes a difference in the communication performance between the cores. Fig. 1 shows an example of such an architecture. In this architecture, the L2 cache and the intrachip interconnection can be exploited by thread mapping. Cores that share the L2 cache (A) communicate faster than cores that use the intrachip interconnection (B), which in turn communicate faster than cores located on different processors (C).

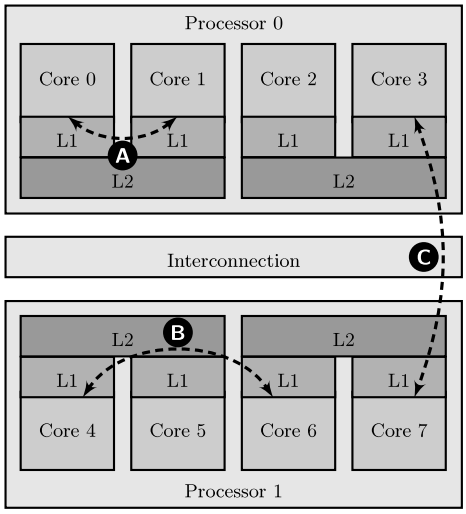


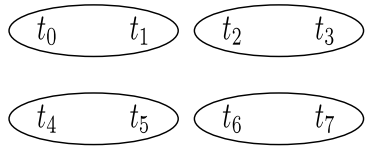
Fig. 1. Processor architecture with one shared (L2) and one private (L1) cache level. There are three different memory access possibilities between two cores: cores that share the L2 cache (A), cores that use the same intrachip interconnection (B), and cores located on different processors (C).

One of the benefits of communication-aware thread mapping is the reduction of cache misses, which we classify into three types. The first type of cache miss is the *invalidation miss*, which happens when a cache line used for communication is constantly invalidated, generating a miss whenever it is accessed. The second type is the *capacity miss*. In shared-memory programs, capacity misses often happen when threads that share a cache evict cache lines accessed by other threads [25]. By mapping threads that communicate a lot to shared caches, fewer cache line evictions are expected, since some cache lines would be accessed by more than one thread. The third type of cache miss is the *replication miss*, which happens due to cache line replication. Replication leads to a virtual reduction of the effective size of the caches [9], as multiple caches store the same cache line. By mapping threads that communicate a lot to cores that share a cache, the space wasted with replicated cache lines can be minimized, leading to a reduction of the cache misses.

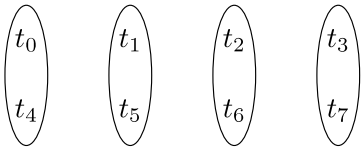
To illustrate how thread mapping affects the performance, consider a producer–consumer situation in shared memory programs, in which one thread writes to an area of memory and another thread reads from the same area. If the cache coherence protocol is based on invalidation, such as *MESI* or *MOESI*, and the consumer and producer do not share a cache, an invalidation message is sent to the cache of the consumer every time the producer writes the data. As a result, after the invalidation, the consumer always receives a cache miss when reading, thereby requiring more traffic on the interconnections, since the cache of the consumer has to retrieve the data from the cache of the producer on every access.

By mapping the threads that communicate on cores that are close to each other in the memory hierarchy, the communication overhead is reduced. In the producer–consumer example, the traffic on the interconnections is reduced if the producer and consumer shared a cache, since both the producer and the consumer access the data in the same cache, eliminating the need for invalidation messages and data transfers. It is important to note that write operations have a greater impact on the performance than read operations, because all writes to shared cache lines invalidate the corresponding lines on the other caches.

Regarding thread communication, we can divide the applications into two main groups: *homogeneous* and *heterogeneous* communication patterns. Homogeneous communication means that each thread presents approximately the same amount of



(a) Phase N.



(b) Phase $N + 1$.

Fig. 2. Phases of the producer–consumer benchmark. Circled threads communicate with each other. In even phases (a), neighboring threads communicate with each other. In odd phases (b), more distant threads communicate.

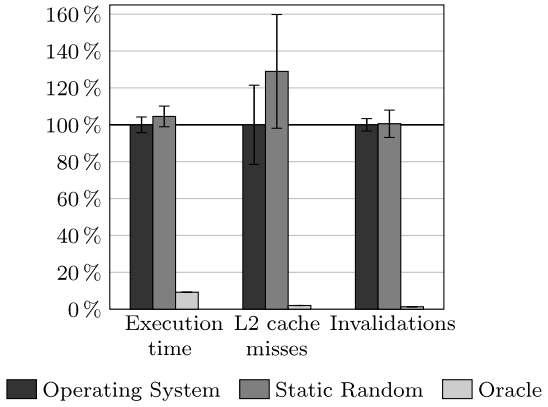


Fig. 3. Performance of the producer–consumer benchmark using three different mappings. All values are normalized to the results of the default OS mapping.

communication to all other threads. On the other hand, in applications that present heterogeneous communication patterns, there are threads that communicate more with a subgroup of threads. By mapping threads according to the amount of communication, we focus on applications with heterogeneous communication patterns. The reason is that in homogeneous applications, the amount of communication between all pairs of threads is about the same, thus any thread mapping would result in the same performance.

Another important aspect of communication is the change of the behavior during the execution, which we call the *dynamic behavior* of an application. Applications with a stable communication pattern do not change their behavior during the execution. Applications with a dynamic behavior present more challenges for thread mapping. The detection mechanism must be able to quickly identify the communication behavior before it changes again. Furthermore, if the behavior changes too frequently, thread mapping cannot improve the performance as the overhead of the migrations becomes higher than the benefits.

There are several ways to dynamically map threads according to the communication. However, some properties are desirable for most applications. A thread mapping mechanism should accurately recognize communication patterns with a low impact on performance. It should be independent from the implementation of the application, such that it does not depend on specific libraries or require modifications to the source code. The mechanism should also consider the false communication problem, which can be spatial or temporal. Spatial false communication is the classical false sharing problem, in which a cache line is present in more than one cache, but the cores are accessing different addresses inside the cache line. Temporal false communication can happen when two threads access the same address, but with large time difference between the accesses, which should not be considered as communication.

2.1. Evaluating thread mapping with a producer–consumer benchmark

To verify the potential of thread mapping, we used a producer–consumer benchmark. It consists of pairs of threads that communicate through a shared vector. The benchmark performs two different phases such that the pairs of producer–consumer threads change in each step. Fig. 2 depicts the two different phases. Applications with this behavior require a dynamic mapping. Static mapping techniques [13,3,4] cannot maximize the performance, since they do not migrate the threads after the application starts running.

We performed experiments with this producer–consumer benchmark on a machine consisting of two quad-core Intel Xeon E5405 processors with private L1 caches and L2 caches that are shared between two cores. The memory hierarchy of this machine is depicted in Fig. 1. We executed the benchmark with three different mappings, the original Linux operating system scheduler, a static random mapping and an oracle mapping. The oracle mapping dynamically migrates the threads at the beginning of each phase, such that threads that communicate are mapped to cores that share the same L2 cache. As the system contains 8 cores, we executed the benchmark with 4 pairs of producer–consumer threads. Each experiment was executed 50 times. We show the average values, normalized to the operating system, and the confidence interval for a confidence level of 95% in the Student's t-distribution.

Fig. 3 shows the results for execution time, L2 cache misses and number of cache line invalidations. The number of cache misses and cache line invalidations was measured using the Papi framework [17]. We can observe that the operating system scheduler and the random mapping perform much worse than the oracle mapping, which reduced the execution time by 90.8%. Furthermore, the number of L2 misses and invalidation messages was reduced to less than 1%, since the data is produced and consumed in the same shared cache. This experiment demonstrates the potential gains of communication-aware thread mapping for parallel applications.

3. Exploiting cache coherence protocols to detect the communication between threads

Cache coherence protocols are responsible for keeping data integrity in shared-memory architectures where more than one cache memory is present, as is common in multi-core and multiprocessor environments. Our mechanism is based on the idea of using the information from these protocols to detect the communication between threads in hardware. In this section, we explain the general concept of using cache coherence protocols to detect the communication, and how to implement it in current architectures.

3.1. Concept of the mechanism

Most coherence protocols keep information about whether a cache line is private or shared between two or more caches. An access to a shared cache line indicates communication between threads. When a read transaction is performed on a shared cache line, it is not always possible to determine which caches share the corresponding line. The reason is that, in most protocol implementations, caches in a high level on the memory hierarchy, such as the L1 cache, keep limited sharing information. These caches usually have states that indicate that the cache line is shared, but there is no information regarding which other

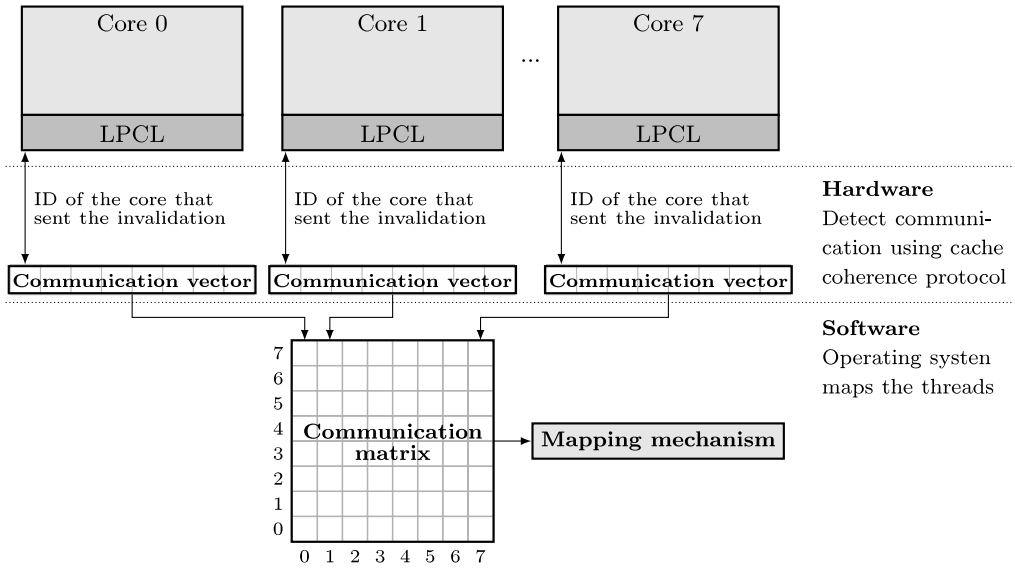


Fig. 4. Proposed mechanism to dynamically map the threads. The proposed hardware modification on the lowest private cache level (LPCL) detects the communication, while the software (operating system) uses the detected communication to map the threads.

caches are sharing the same line. Furthermore, in directory based coherence protocols, a read transaction on a shared cache line does not generate an access to the directory.

On the other hand, a write transaction on a shared line requires that the copies of the same cache line on other caches are invalidated. In this case, cache coherence protocols have to send an invalidation message to all the caches that have the corresponding line. For this reason, we make use of the invalidation messages to detect the communication. Each invalidation message is considered a *communication event* between the core that requested the write transaction and the other cores that have the same cache line in their caches.

As an example, the MOESI protocol provides two states to indicate if a cache line is shared: *shared* and *owner*. Therefore, in MOESI, any write transaction to cache lines that are in the *shared* or *owner* state generate invalidation messages, which we consider a communication event. This can be adapted to other protocols that have similar states, such as MESI and MESIF.

3.2. General hardware implementation

An example implementation of the mechanism for an architecture with 8 cores is presented in Fig. 4. We detect the communication at the lowest private cache level (LPCL) of each core, to be able to identify which core accessed the data being invalidated. To store the amount of communication, our mechanism requires a vector at each cache of the LPCL. The number of elements of this vector is equal to the total number of cores in the system. We call this vector the *communication vector*. It stores the amount of communication of its local core to the other cores in the system. The operating system can see all communication vectors as merged, forming a square matrix, which we call the *communication matrix*.

Our mechanism works as follows. In the original coherence protocol, when a core requests a write transaction on a shared cache line, an invalidation message is sent to the caches that share the same line. In our mechanism, when receiving the invalidation message, the cache in the LPCL increments the communication vector at the position indexed by the ID of the core performing the write transaction. For instance, consider an 8-core architecture in which the LPCL is the L1 cache, and that a given cache line is shared between the L1 caches of cores 2, 5 and 6. If core 5 requests a write transaction on that line, the L1 caches of cores 2 and 6 would receive an invalidation message, and our mechanism would increment the communication vectors of these caches at position 5.

The implementation of our mechanism is attached to the cache memory subsystem. Incrementing the communication vectors can be performed in parallel to the cache access, and does not affect the cache behavior or performance. We need to add a communication vector and an adder unit to each LPCL, as well as instructions that allow the operating system to read and clear the communication vectors. Each communication vector requires C counters to store the amount of communication, where C is the total number of cores of the machine. In our experiments, we observed that saturated counters using 32 bits are enough to correctly detect the communication.

With this mechanism, it is possible to identify which cores are communicating. However, to perform the mapping, we need to know which threads are communicating. This issue is easy to solve, since the operating system knows which thread is executing on each core and can keep a copy of the communication vector of each thread in main memory. Whenever a context switch happens, before loading the new task, the operating system first saves the communication vector into the memory and then sets the elements of the communication vector to zero. All future communication detection is then relative to the new task. In this way, our mechanism is able to detect the communication between the threads of the application.

If there is no private cache level in the memory hierarchy, we can implement our detection mechanism in a shared cache level, following the description in Section 3.3.2.

3.3. Hardware implementation in specific architectures

Our mechanism can be easily adapted to any cache coherent architecture that has a shared state. In this section, we explain how it could be implemented in some specific architectures.

3.3.1. Architectures with multi-threaded cores

If the cores of the architecture are multi-threaded, the data in the LPCL can be accessed by any of its virtual cores. To adapt our mechanism to multi-threaded cores, we need to provide a way to identify which virtual core accessed each cache line. For that, the only required modification is the addition of bits to the LPCL to identify which virtual core accessed each cache line. The hardware

overhead for this is one access bit per virtual core on every cache line of the LPCL. Also, instead of only one communication vector, it would be necessary to add one communication vector for each virtual core in the LPCL. With these modifications, our mechanism is able to handle architectures with multi-threaded cores.

3.3.2. Implementing the detection mechanism in other cache levels

The simplest way to implement our mechanism is in the LPCL, since it is easier to identify which core accessed each cache line. However, by adding more hardware, we can implement the detection mechanism in any cache level. The additional hardware consists of one access bit per virtual core on every cache line of the cache in which we want to implement our mechanism. Also, it would be necessary to have one communication vector per virtual core. For instance, if we add one access bit per virtual core in all cache lines of the last level cache (LLC), we could implement our mechanism in the LLC instead of the LPCL. Whenever a write or invalidation transaction arrives at the LLC, we can increment the communication vectors corresponding to the access bits in position i, where i is the ID of the virtual core that generated the write transaction.

To illustrate the behavior, consider an architecture with 4 cores sharing each LLC, and that there are 8 cores in total, without multi-threading. Cores 0-3 share the first LLC (LLC-0) and cores 4–7 share the second LLC (LLC-1). Both LLCs would have 4 communication vectors, each with 8 positions. Suppose that cores 0, 1, 3 and 5 access a given cache line. The access bits in LLC-0 corresponding to cores 0, 1 and 3 would be set, as well as the access bit in LLC-1 corresponding to core 5. If core 6 requests a write transaction, when the request arrives at LLC-1, it would increment the communication vector corresponding to core 5 in position 6. The cache coherence protocol would then send an invalidation message to LLC-0. Upon receiving the message, LLC-0 would increment the communication vectors of cores 0, 1 and 3 in position 6.

It is important to note that, in some directory based protocols, the directory is attached to the LLC and already contains the access bits to track which higher level caches have the corresponding line [20]. In these protocols, if there are no multi-threaded cores, we could implement our mechanism in the LLC by adding only the communication vectors.

3.3.3. Including the core ID in the coherence message

Our mechanism needs to know the ID of the core that requested the write transaction. Usually, when a cache sends an invalidation message, all caches that already have the corresponding cache line respond with an acknowledgment message [20]. Therefore, the invalidation message contains information about which cache generated the invalidation, such that the acknowledgment messages can be sent to the correct cache. If the cores are not multi-threaded, the ID of the cache in the LPCL directly identifies the core that requested the write transaction, and no further modification of the coherence protocol is necessary. If the cores are multi-threaded, the ID of the cache in the LPCL could point to any of its virtual cores. One simple way to overcome this issue is to send the ID of the virtual core instead of the ID of the cache in the invalidation message. When a cache receives an invalidation message, it knows which cache corresponds to the virtual core received in the invalidation, since it has knowledge about the memory hierarchy.

If the invalidation message does not contain a core ID, we need to add this information to the invalidation messages. The size of this field is logarithmic to the number of virtual cores in the system. For example, a core ID of 1 Byte can cover systems of up to 256 virtual cores.

3.3.4. Implementation examples in modern processor architectures

The Intel Harpertown Architecture [14] contains 2 cache levels. The L1 cache is private to each core, while the L2 cache is shared by 2 cores. Also, there are 4 cores per processor and hence 2 L2 caches. In this architecture, the simplest way to implement our mechanism is in the L1 cache, following the implementation described in Section 3.2. We could also implement the mechanism in the L2 cache by adding 2 access bits per cache line of the L2 cache, as described in Section 3.3.2, since the L2 cache is shared by 2 cores. Likewise, there would be 2 communication vectors per L2 cache.

A different memory hierarchy is present in the Intel Sandy Bridge [15] architecture, which contains 3 cache levels. The L1 and L2 caches are private to each core, and the L3 cache is shared among all cores. Also, each core has 2-way simultaneous multi-threading (SMT). To implement our mechanism in the LPCL, in this case the L2 cache, we need to add 2 access bits to each cache line of the LPCL to identify which of the 2 virtual cores accessed each cache line, as explained in Section 3.3.1. There would be also 2 communication vectors for each L2 cache.

To implement the detection mechanism in the LLC in Sandy Bridge, we could make use of the directory that is already attached to the cache, following the description in Section 3.3.2. The directory contains bits that identify which private caches contain each cache line, and thereby which core accessed each cache line. If we used one bit per virtual core instead of using one bit per core in the directory, we could identify in the LLC which virtual cores accessed each cache line. Also, there would be one communication vector per virtual core in the LLC.

3.4. Design considerations

As our mechanism is performed entirely by the hardware and the operating system, it does not depend on the parallelization API and does not require any modification to the application or its runtime environment. Moreover, the communication pattern is detected only during the execution of the application. Our mechanism provides a good solution for detecting changes in the behavior of the applications because the number of possible entries in the cache memory is quite low. Data that is not accessed anymore will have its corresponding entry overwritten and will therefore not be counted anymore in the calculation of the communication pattern. This also reduces the impact of the temporal false communication.

Regarding the spatial false communication, our mechanism detects the communication on the cache line granularity. Hence, accesses to different offsets inside the same cache line would still be counted as communication. Cache coherence protocols also have the same problem, considering a cache line as *shared* when different offsets are accessed in different caches. Therefore, our mechanism improves the performance when the application presents spatial false communication by mapping the involved threads to cores that share a cache memory.

4. Using the detected communication to map threads

To dynamically map the threads, we need to provide a way to allow the detection of changes in the communication pattern, as well as to calculate the new mapping. The mapping is performed in software, and can be done on the operating system or library level. We propose a mapping mechanism consisting of several phases, as illustrated in Fig. 5. Our mapping mechanism fetches the values of the communication matrix from the hardware to the memory, and provides the matrix to the algorithms. These algorithms are: the aging algorithm, the communication filter, the mapping algorithm and the symmetric mapping filter.

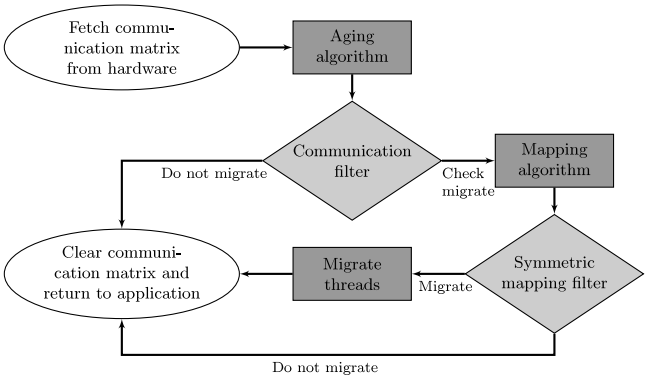


Fig. 5. Software mapping mechanism.

To detect changes in the communication pattern, we apply an aging algorithm on the communication matrix. To reduce the overhead of the mapping algorithm, we also developed a communication filter that determines if the communication behavior has changed sufficiently to warrant a recalculation of the mapping. Since the mapping problem is NP-Hard [5], we developed a heuristic thread mapping algorithm based on the Edmonds graph matching problem [22], which has a polynomial time complexity. Furthermore, different thread mappings may actually be considered equivalent due to symmetries of the memory hierarchy. To avoid unnecessary migrations, we developed a symmetric mapping filter.

After applying the algorithms, our mechanism migrates the threads if necessary. Finally, the elements of the hardware communication matrix are set to zero. By fetching the hardware communication matrix in the beginning and clearing it only after performing the algorithms, the noise introduced by the mapping mechanism into the communication detection mechanism is minimized. In the rest of this section, we will present and discuss these algorithms.

4.1. Aging algorithm

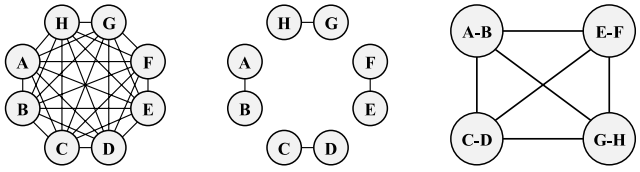
To detect changes in the communication pattern, we adopted an aging strategy. Every time the communication matrix is fetched from the hardware, we apply Eq. (1) to each element of the communication matrix. In this equation, CM_{new} is the communication matrix after the aging, $CM_{previous}$ is the communication matrix from the previous time the aging algorithm was executed, $CM_{hardware}$ is the communication matrix fetched from the hardware, and AF is the aging factor used to give a higher or lower priority to the values of the previous communication matrix.

$$CM_{new}[x][y] = (1 + AF) \cdot CM_{previous}[x][y] + CM_{hardware}[x][y] \quad (1)$$

We vary the value of AF from -0.1 to $+0.1$. Every time we migrate the threads, we set AF to $+0.1$, such that the previous values will have a higher influence. We do this to avoid migrations in consecutive calls to the mapping mechanism, which could harm the performance. On the other hand, every time we do not migrate, we decrease AF by 0.02, which raises the priority of the new values. In this way, the probability of a migration increases over time. We set the minimum value of AF to -0.1 to avoid too many migrations. The time complexity of this algorithm is $O(N^2)$, where N is the number of threads.

4.2. Communication filter

The goal of this filter is to decide if the communication matrix has changed sufficiently to warrant a migration, while presenting a low overhead. First, the filter determines if the communication pattern is homogeneous or heterogeneous. Afterwards, if the pattern is heterogeneous, we detect if the pattern has changed compared to the previous pattern. To decide if a pattern is homogeneous or heterogeneous, we calculate the variance and the average of the values of the communication matrix and divide these two values. It is expected that heterogeneous communication patterns present higher values than homogeneous communication patterns for this metric. If the value is greater than a threshold, called *H-threshold*, the pattern is considered heterogeneous. Otherwise, the pattern is considered homogeneous, not requiring a migration of the threads.



(a) Input. (b) Output. (c) Generate new communication matrix with Eq. (6).

Fig. 6. The graph matching problem applied to thread mapping. Each vertex corresponds to a thread or a group of threads. Edges represent the amount of communication between them. Consider that edges A–B, C–D, E–F and G–H of the input graph have the highest weights.

To detect if the heterogeneous pattern changed compared to the previous one, we use a filter that is based on the idea that each thread communicates more with a certain subgroup of threads. We call the threads that belong to the same subgroup *partner threads*. Based on this, for the current communication matrix, the algorithm keeps a vector indexed by thread IDs, where each element contains the ID of its partner thread. We call this vector the *partner vector*. Every time a new communication matrix is evaluated, the algorithm generates the partner vector for it. Afterwards, the algorithm compares the previous and new partner vectors and counts how many threads changed their partner. If the amount of different partners exceed a threshold, called *ID-threshold*, the mapping algorithm is called. Otherwise, the filter algorithm considers that the communication matrix did not change enough to represent a new communication pattern.

Eqs. (2)–(5) formalize the communication filter. $PartnerVector_{previous}$ is the partner vector of the last call to the mapping mechanism. $PartnerVector_{new}$ is the partner vector of the current communication matrix, after applying the aging algorithm. *Ndiff* is a variable that stores how many threads changed their partner. *H-factor* estimates the degree of heterogeneity of the communication matrix. The thread mapping algorithm is called if *CheckMigrate* is true.

$$u(x, y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases} \quad (2)$$

$$Ndiff = \sum_{i=1}^{N} u(PartnerVector_{previous}[i], PartnerVector_{new}[i]) \quad (3)$$

$$H\text{-}factor = \frac{variance(CommMatrix)}{average(CommMatrix)} \quad (4)$$

$$CheckMigrate = \begin{cases} true & \text{if } (Ndiff \geq ID\text{-}threshold) \text{ and } (H\text{-}factor > H\text{-}threshold) \\ false & \text{otherwise.} \end{cases} \quad (5)$$

The ID-threshold directly influences the accuracy and overhead. A low ID-threshold increases the probability of calling the mapping

algorithm, thereby increasing both accuracy and overhead. It increases the overhead because the mapping algorithm is more expensive than the communication filter. The accuracy is increased because the communication filter is only a prediction if the mapping algorithm would return a different mapping. On the other hand, a high ID-threshold decreases both accuracy and overhead. Since each application has its own characteristics, the ideal ID-threshold varies among different applications. If we use a static ID-threshold, we would need to use a more generic value, which would not harm the accuracy for most applications. Therefore, a static ID-threshold would have a low value. This would unnecessarily increase the overhead for applications that have a higher ideal ID-threshold due to too many calls to the thread mapping algorithm. By using an adaptive ID-threshold, we can dynamically adjust the value to find a threshold that has a better trade-off between accuracy and overhead for the running application.

In order to automatically find the ID-threshold for the running application and hence the best trade off between accuracy and overhead, the algorithm dynamically adapts the ID-threshold. The ID-threshold varies between 10% and 50% of threads that change their partners. The initial value is 10%, allowing more migrations in the beginning. When the mapping algorithm is called, and it leads to a migration, we consider that the communication filter correctly predicted that the communication pattern changed. Hence, we decrease the ID-threshold, to make it easier to migrate the threads. However, if calling the mapping algorithm does not lead to a migration, we consider that the communication filter incorrectly predicted the result of the mapping algorithm. Therefore, we increase the ID-threshold, making it more difficult to migrate the threads. In this way, the ID-threshold is automatically adapted to the characteristics of the running application.

For the H-threshold, we empirically determined an ideal value of 250. As this value is normalized to the average of the amount of communication, we expect that this value remains the same for different applications and hardware architectures. Further details about this threshold are given in Section 5.1.

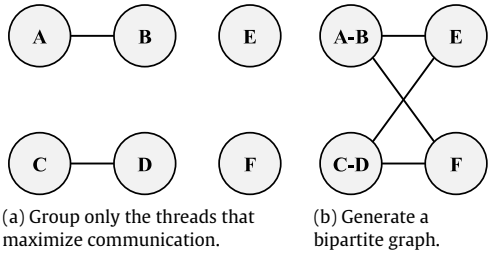
The time complexity of the communication filter is $O(N^2)$, where N is the number of threads, since we need to access all elements of the communication matrix to fill the partner vector.

4.3. Mapping algorithm

Our algorithm to map the threads on the cores is based on maximum weight perfect matching problem for complete weighted graphs. This problem is defined as follows. Given a complete weighted graph $G = (V, E)$, we have to find a subset M of E in which every vertex of V is incident with exactly one edge of M, and the sum of the weights of the edges of M is maximized. This problem can be solved by Edmonds' matching algorithm in polynomial time [22], $O(N^3)$, and a parallel algorithm can solve the problem with a time complexity of $O(\frac{N^3}{P} + N^2 \cdot \lg N)$, where N is the number of vertices and P is the number of processing elements.

To model thread mapping as a matching problem, the vertices represent the threads and the edges the amount of communication between them. A complete graph is obtained directly from the communication matrix, as in Fig. 6(a). The graph is processed by the matching algorithm, which outputs the pairs of threads such that the amount of communication is maximized, illustrated in Fig. 6(b).

If there are only 2 cores sharing a cache, mapping threads to them with the matching algorithm is straightforward. However, there are many architectures in which more than 2 cores share the same cache, or there are more levels of memory hierarchy to be exploited. In these cases, another communication matrix needs to be generated, in which each vertex represents previously grouped threads, and the edges represent the communication between the corresponding groups, depicted in Fig. 6(c). This matrix is generated by Eq. (6).



(a) Group only the threads that maximize communication. (b) Generate a bipartite graph.

Fig. 7. Mapping algorithm when the number of cores sharing a cache is 3. Letters A–F represent a thread. Each vertex represents a thread or a group of threads. Edges represent the amount of communication between them.

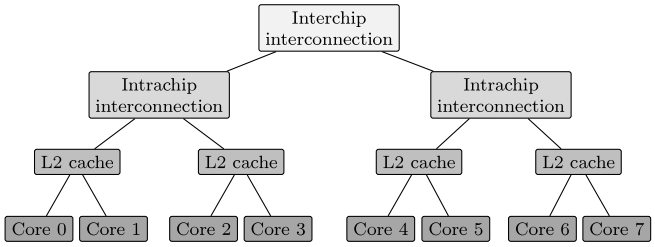


Fig. 8. Representation of the memory hierarchy as a tree.

$$CM_{next}[(x, y)][(z, k)] = CM[x][z] + CM[x][k] + CM[y][z] + CM[y][k] \quad (6)$$

CM_{next} represents the communication matrix that will be used in the next iteration of the mapping algorithm, (x, y) and (z, k) are the matches found in the previous step, and $CM[i][j]$ is the amount of communication between threads i and j. The matching algorithm is re-executed using this new communication matrix as input. This algorithm does not guarantee that the result will contain the pairs of pairs with the most amount of communication, as the communication matrix does not provide communication information about groups with more than 2 threads. However, it is a reasonable approximation and keeps the time and space complexity polynomial.

This procedure is repeated $\log_2 K$ times in each level of the memory hierarchy, where K is the amount of sharers of the corresponding level. Therefore, the complexity of our mapping algorithm is $O(N^3 \cdot \log_2 K)$ for each level, where N is the number of threads. For instance, if there are 4 cores sharing the L1 cache, this procedure is repeated 2 times to calculate which threads will share the L1 cache. The same procedure is applied for each level on the memory hierarchy. For instance, consider the architecture depicted in Fig. 1. In this architecture, we need to apply the matching 2 times. The first matching generates the pairs of threads that will share the L2 cache. The second matching generates the pairs of pair of threads that will communicate through the intrachip interconnection.

The number of cores sharing a cache may not be a power of two. In this scenario, we also apply the matching more than once per cache level. Considering an architecture with 6 cores and 2 caches, where each cache is shared by 3 cores, the matching algorithm would produce 3 disconnected graphs that represent 3 pairs of threads. We generate a new communication graph that groups only the 2 pairs that maximize the communication, as in Fig. 7(a). Afterwards, we insert edges connecting the vertices in such a way to form a bipartite graph that put the grouped threads in the same set, as in Fig. 7(b). Finally, we apply the matching to the bipartite graph. This procedure can be adapted to map any number of threads.

4.4. Symmetric mapping filter

To avoid unnecessary migrations, we developed an algorithm that detects symmetric mappings. To detect symmetries, our algorithm requires a tree representation of the memory hierarchy. The architecture shown in Fig. 1 can be represented by the tree depicted in Fig. 8. In this hierarchy, an example of symmetric thread mappings consists of two threads mapped to cores 0 and 1, or mapped to cores 2 and 3. Although the cores are different, the resources in these 2 different mappings are shared in the same way. The symmetric mapping filter uses this tree to calculate the distance between all the cores using the Floyd–Warshall algorithm. This procedure is performed only at the first time the symmetric mapping filter is called, because the memory hierarchy does not change during execution.

With the distance matrix calculated, the algorithm analyzes the distance between the cores in the new and previous mappings. By accumulating the differences of the distances, we estimate how much the new mapping diverges from the previous mapping. If the sum of the differences is 0, the mappings are symmetric, therefore there is no need to migrate of the threads. The behavior is formalized in Eqs. (7) and (8).

$$Diff = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} |Dist(Core_{new}[i], Core_{new}[j]) - Dist(Core_{previous}[i], Core_{previous}[j])| \tag{7}$$

$$Migrate = \begin{cases} true & \text{if } (Diff > 0) \\ false & \text{otherwise} \end{cases} \tag{8}$$

$Core_{new}[k]$ and $Core_{previous}[k]$ represent the core that thread k is mapped to, for the new and previous mappings, respectively. *Dist* is the distance between the cores that was previously calculated with the Floyd–Warshall algorithm. The threads are migrated if *Migrate* is true. The time complexity of this algorithm is $O(N^2)$, where N is the number of threads.

5. Experimental evaluation

We executed the applications in a machine consisting of 2 quad-core Intel Xeon E5405 processors that are based on the Harpertown microarchitecture [14], running the Linux kernel version 3.2. The memory hierarchy is illustrated in Fig. 1. Since current machines do not implement our proposed detection mechanism, we generate the information regarding the communication by implementing our detection mechanism in the Simics simulator [19] extended with the GEMS-Ruby memory model [21]. Our implementation follows the description in Section 3.2. The parameters of the simulated machine follow the real machine as close as possible.

To evaluate our proposal, we used the OpenMP implementation of the NAS parallel benchmarks (NPB) [16], version 3.3.1, as well as the producer–consumer (PC) benchmark described in Section 2.1. The NPB applications were executed using the *W* input size due to simulation time constraints. We ran all the benchmarks except *DC*, which takes too much time to simulate. All benchmarks were executed with 8 threads, which is the number of threads the architecture can execute in parallel.

The communication matrices are detected inside the simulator and are then fed into the mapping mechanism on the real machine during the execution of the applications to perform the thread mapping. To keep the execution inside the simulator synchronized with the execution in the real machine, we keep track of the barriers of the applications in the simulator. This is possible because the applications from our workload execute the same amount of barriers and in the same order along different executions. Therefore, when the execution in the simulator and real machine reach the same barrier, their executions are in the same state.

We check for thread migrations in the synchronization points of the applications, such as in barriers. By performing the mapping in the synchronization points, we can make use of mandatory stall times that would put the running thread to sleep. In this way, the overhead imposed by the mapping is reduced. Since it needs to wait for a barrier, the mapping is performed a bit later than the optimal time, which slightly reduces the gains from an optimized mapping. This approach is suitable if the application contains a reasonable amount of synchronization points, as is common in parallel applications.

For applications that have large parallel phases without synchronization, it would be necessary to create a separate thread that awakes periodically to perform the mapping. Although this approach is more generic, it can have a higher interfere on the performance, since the mapping thread may awake in situations where all the threads of the application are running, making one thread of the application sleep. Since most applications from our workload have a high number of barriers, we make use of the stall time of the threads to execute the mapping. If the detection mechanism was available in the hardware, it would be possible to implement the mapping mechanism in the kernel scheduler, either performing the mapping when a thread sleeps or by creating a separate thread for the mapping mechanism.

In the rest of this section, we first analyze the communication behavior of the applications. Afterwards, we show the performance results. Finally, we discuss the overheads associated with our proposal.

5.1. Communication behavior

For a better understanding of the applications, we first analyze the degree of heterogeneity of the benchmarks. Fig. 9 shows how heterogeneous each benchmark is, considering the *H-factor* function described in Section 4.2, as well as the aging algorithm from Section 4.1. We empirically determined the value of the *H-threshold* for the communication filter algorithm, setting it to 250. Therefore, our algorithms consider that the applications that present heterogeneous patterns are BT, IS, LU, MG, SP, UA and PC, while the applications CG, EP and FT are considered homogeneous. In the following paragraphs, we investigate if this classification is correct by directly analyzing the communication matrices.

Fig. 9 shows if the communication pattern is heterogeneous, but it does not show if the communication pattern changes during the execution. To analyze the dynamic behavior, we included Fig. 10, where we check how much consecutive communication matrices differ. To calculate these values, we normalize every cell (i, j) from the previous and current communication matrices to the highest values of their matrices. After normalization, we calculate the dynamicity factor with Eq. (9), where $CM_{current}$ is the communication matrix after applying the aging algorithm, and $CM_{previous}$ is the communication matrix evaluated in the last time the mapping mechanism was called. We use the power of 4 to increase the influence of the higher differences.

$$Dynamicity = \sum_{i=1}^{N} \sum_{j=1}^{N} (CM_{current}[i][j] - CM_{previous}[i][j])^4 \tag{9}$$

In the PC application, two different communication phases are repeated twice. If a static mapping mechanism was used, it would consider the communication as depicted in Fig. 11(j), which makes static mapping unable to map the application. On the other hand, our mechanism clearly detects the two phases, whose communication matrices are illustrated in Fig. 12(b). In the first and third phases, the neighbor threads communicate (Fig. 12(b1)). In the second and fourth phases, threads with the same congruence modulo 4 communicate (Fig. 12(b3)). Due to these patterns, there is

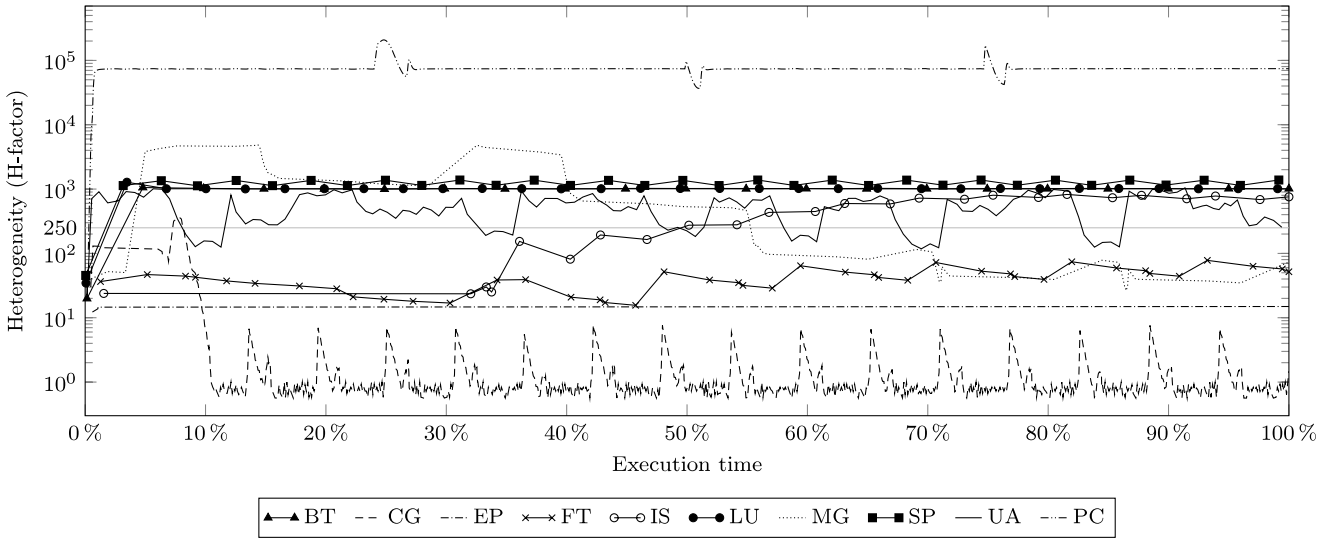


Fig. 9. Degree of heterogeneity (H-factor) of the applications. Applications with a heterogeneity level higher than 250 (H-threshold, solid gray line) are considered as heterogeneous.

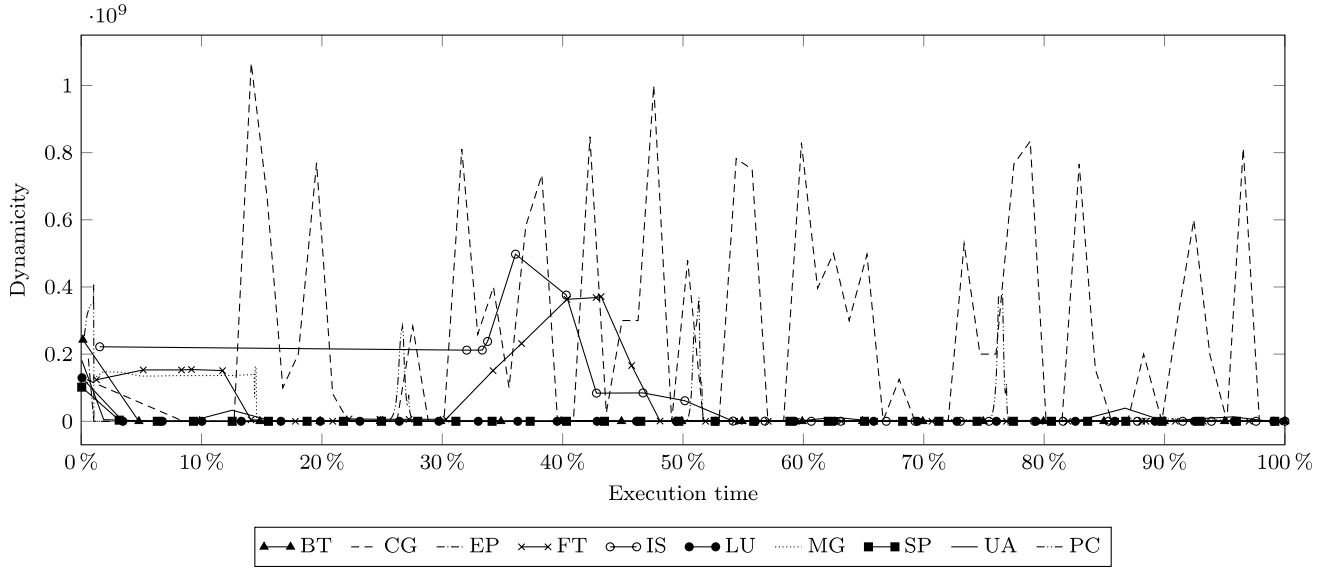


Fig. 10. Dynamic behavior of the applications. High values indicate a change in the communication pattern.

a high level of heterogeneity in Fig. 9 for PC. On every phase change, as shown in Fig. 12(b2), there is a peak in Fig. 10. The 4 phases can be observed in Fig. 9, since the degree of heterogeneity varies at the times corresponding to each phase change. Due to this highly dynamic behavior, we classify PC as having a heterogeneous and non-stable communication.

BT, SP and UA are applications that present most of their communication between neighboring threads. This behavior is common when the application is based on domain decomposition, where most of the communication happens between neighbors and most of the shared data is located on the borders of each sub-domain. LU presents communication between neighbors, but it also performs communication between distant threads. With this behavior, the degree of heterogeneity of these applications is high, as shown in Fig. 9. In Fig. 10, we can also observe that the communication pattern stabilizes after a short period of initialization. This means that the domain decomposition pattern is present during almost the entire execution. Therefore, we classify these four applications as having a heterogeneous and stable communication pattern.

The communication patterns of IS and MG are considered heterogeneous for about half of the execution time. For IS, the first half of the execution time refers to its initialization. As observed in Fig. 9, IS starts to change from homogeneous to heterogeneous at around 40% of its execution time, which is the same point where there is a peak in Fig. 10. After the stabilization, there is a lot of communication between neighbors, which is evident in Fig. 11(e). In MG, the overall communication looks similar, but the degree of heterogeneity decreases over time. The reason is that the amount of communication between neighbors compared to non-neighbor threads is higher at the beginning of the execution than at the end. Therefore, although communication between neighbors is present during the entire execution, the heterogeneity is much lower at the end of the execution. Due to these reasons, we also classify IS and MG as having heterogeneous and stable patterns, although the degree of heterogeneity is low.

The communication pattern of CG changes several times during the execution, as illustrated in Fig. 10. However, contrary to PC, where both patterns were heterogeneous, one pattern of CG is homogeneous and the other is heterogeneous. This can be observed in

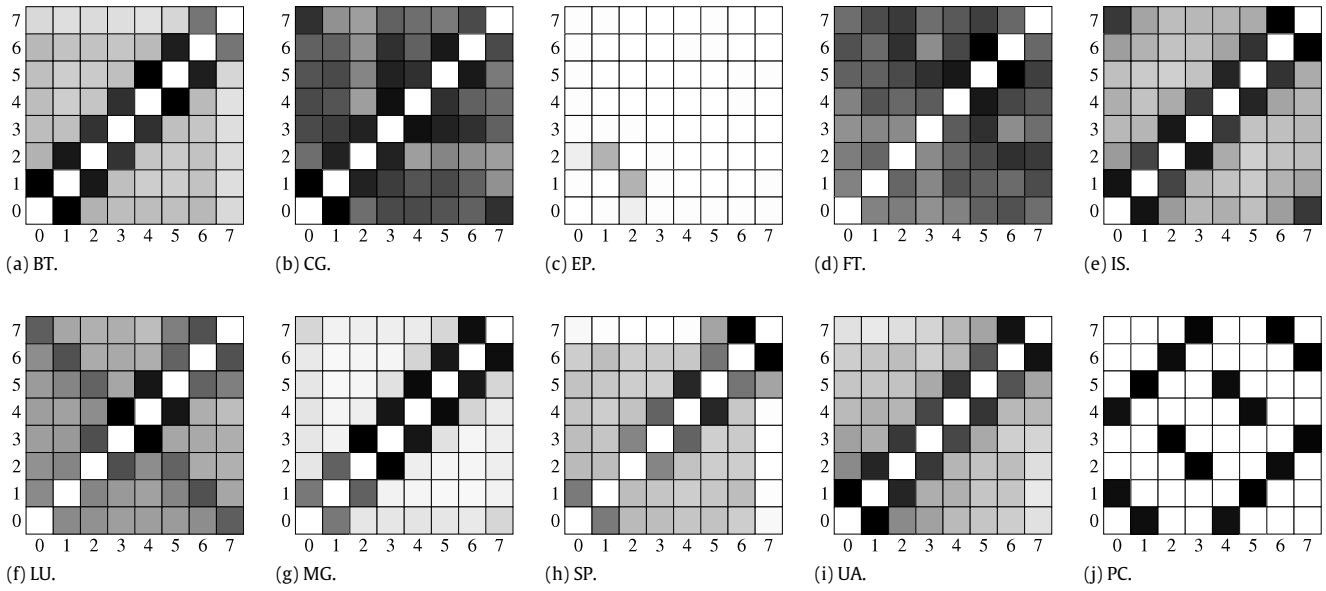


Fig. 11. Overall communication matrices of the NAS and PC benchmarks. Axes represent thread IDs. Darker cells indicate higher amounts of communication between the threads.

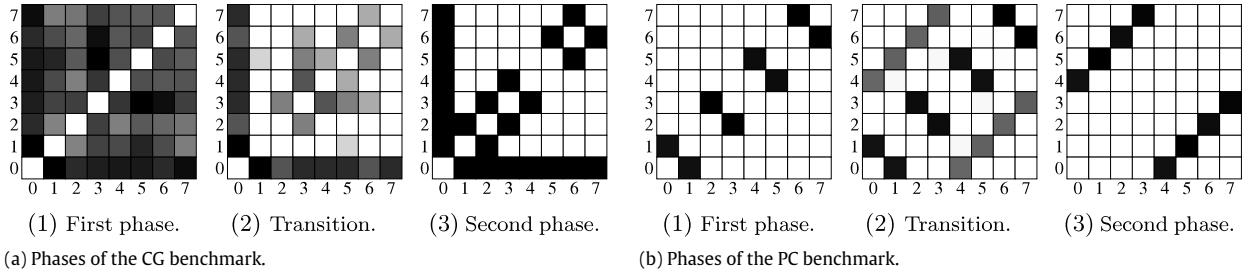


Fig. 12. Different communication phases of the CG and PC benchmarks.

Fig. 12(a). In phase 1, there is little difference in the amount of communication between any pairs of threads. Phase 2 depicts the point in the execution in which the communication pattern is changing. Then, in phase 3, the neighbors communicate, indicating a domain decomposition pattern. Furthermore, the detected pattern is also similar to a reduction pattern, since thread 0 communicates with all other threads. In Fig. 9, although we can observe that the degree of heterogeneity of CG varies, our mechanism classifies CG as always having homogeneous patterns. The reason is that the communication matrices we show are normalized to their own maximum values, while the values of Fig. 9 are not normalized. Hence, the absolute values of the communication matrices of CG are much lower compared to the values of the heterogeneous applications, thereby being classified as homogeneous.

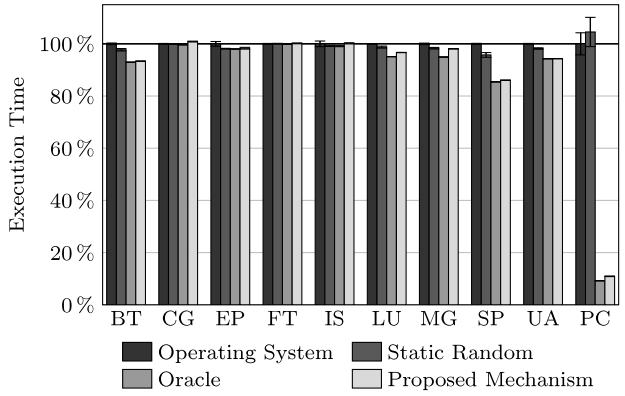
EP is an application with a homogeneous communication pattern and without any kind of dynamic behavior. We can observe that the communication matrix of EP, shown in Fig. 11(c), shows very little communication. Furthermore, most threads do not communicate at all. The FT application is also considered homogeneous and without any dynamic behavior. Despite that, Fig. 10 shows a peak at around 40% of the execution, it is actually due the initialization, since it takes some time for the communication pattern to stabilize. FT presents a short execution time, such that the time spent in the initialization has a big influence on the communication pattern.

Summarizing the communication pattern results, we can observe that 6 NAS benchmarks (BT, SP, UA, LU, IS and MG) have a heterogeneous and stable communication behavior. The PC benchmark has a dynamic communication behavior, with two different heterogeneous phases. For these, we expect gains in the performance when using our mapping mechanism. The CG benchmark periodically changes its communication behavior between heterogeneous and homogeneous and we therefore expect smaller improvements compared to the applications with an exclusively heterogeneous pattern. For the other two NAS benchmarks (EP and FT), we do not expect improvements, as their communication patterns are homogeneous.

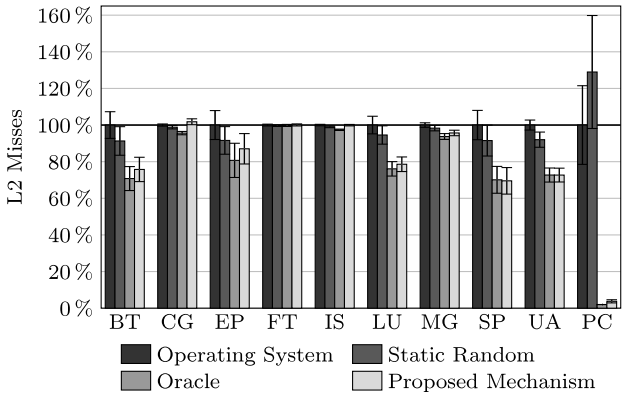
5.2. Performance results

To analyze the performance, we measure the execution time and cache memory events obtained from hardware counters using the Papi framework [17]. Fig. 13(a)–(c) present the execution time, L2 cache misses and number of invalidation messages, respectively. We focused only on the L2 cache misses because the L1 caches are private and do not benefit from mapping. Each benchmark was executed 50 times. We show the average values, as well as the confidence interval for a confidence level of 95% in a Student's t-distribution. We compare the results of our proposal to a static random mapping and to the oracle mapping. For the oracle mapping, we generated traces of all memory accesses for each application and perform an analysis of the communication pattern, similarly to [13]. All values are normalized to the results using the default operating system scheduler. We also present the absolute values of the results from our mechanism in Table 1.

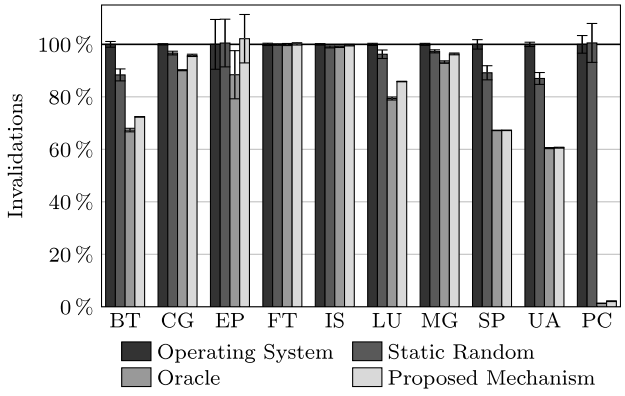
In the PC benchmark, we obtained the highest improvements. Compared to the operating system, our mechanism reduced the execution time by 89.1%, the L2 cache misses by 96.3% and the



(a) Execution time.



(b) L2 cache misses.



(c) Invalidations.

Fig. 13. Performance results, normalized to the OS scheduler.

number of invalidations by 97.9%. The improvements are very high because PC presents a high level of heterogeneity. Additionally, our mechanism was able to provide improvements very close to the oracle mapping, because the communication phases of PC are clearly defined. The operating system and static random mappings show large confidence intervals, which is expected, since different mappings are used on each execution. Also, even the best random mappings present improvements worse than our mechanism, because, even if the random mapping is suitable for one of the communication phases, it will not be suitable for the other phases. We can observe in Table 1 that our algorithms migrated PC 11 times, despite having only 4 phases. This happens because our algorithms are heuristics designed to work with any type of application, therefore they are not completely precise.

BT, LU, SP and UA were classified as having heterogeneous and stable communication patterns. We can observe that, in these applications, our proposal performs better than the operating system and the static random mapping. However, the improvements are lower than in PC. There are two main reasons for that. First, the degree of heterogeneity is lower than in PC. Second, the mapping does not change during execution. Hence, the random and operating system mappings do not perform as bad as in PC.

It is important to note that the number of invalidations and cache misses in BT, LU, SP and UA were greatly reduced. The execution time was also reduced, but by a smaller factor. Invalidations are more sensitive to thread mapping than cache misses and the execution time. The reason is that a good mapping directly influences the number of invalidations, while cache misses and execution time are also influenced by other factors, such as cache lines prefetches and competition for cache lines by the cores that share the cache, among others. Regarding the NAS benchmarks, using our mechanism, SP presented the highest reduction of execution time (13.9%) and L2 cache misses (30.5%), and UA presented the highest reduction of cache line invalidations (39.4%).

IS and MG are also classified as having heterogeneous and stable communication patterns. Despite that, only small improvements were achieved, since the overall level of heterogeneity is lower than the other applications from the same category. In MG, the oracle mapping reduced the execution time by 5.0%, while our mechanism reduced it by 1.8%, which is 0.4% better than the random mapping. We can also observe a small improvement in the L2 cache misses and number of invalidations. Regarding IS, the execution time is very low, and the communication pattern is heterogeneous for only half of the execution, leading for a negligible performance improvement even for the oracle mapping.

As stated in Section 2, if the communication pattern among the threads is homogeneous, negligible performance improvements are expected to be achieved by thread mapping, as in the case of EP and FT. In CG, one of the two communication patterns is slightly heterogeneous, and the oracle was able to reduce the number of L2 cache misses and invalidations by a small amount. Although our mechanism was also able to detect the two communication patterns, the overhead that it introduced in CG was the greatest compared to the other applications, because it has the highest number of synchronization points per second.

EP, besides having a homogeneous communication pattern, does not share data between the threads, which is the reason that the absolute number of invalidations are low compared to the other applications. These low values imply that small, unpredictable events during the execution have a large impact on the results, as reflected by the large confidence intervals in Fig. 13(b) and (c).

All NAS benchmarks except CG have communication patterns that stabilize after a period of initialization. In these applications, communication-aware mapping can achieve higher improvements because the threads do not need to be migrated anymore after the initialization. CG is an application with a highly dynamic behavior and changes its communication pattern during each time step. In this type of application, the potential for improvements is much lower, since migrations would have to be performed with very short intervals, increasing the overhead as well as reducing the time the new mapping can be effective. As our evaluation is limited to the NAS benchmarks, other applications with a highly dynamic behavior need to be analyzed carefully to determine if they can benefit from communication-aware mapping.

Our proposal presented results similar to the oracle mapping, demonstrating its effectiveness. In most cases, it performed significantly better than the random mapping. This shows that the gains compared to the operating system are not due to the unnecessary migrations introduced by the operating system, but due to a more efficient usage of the resources of the machine.

Table 1
Absolute values of the results using our mechanism.

Parameter	BT	CG	EP	FT	IS	LU	MG	SP	UA	PC
Execution time (s)	0.63	0.13	0.43	0.09	0.08	2.12	0.22	2.14	2.00	0.40
Kilo L2 misses per second	1395	1722	36	4231	4864	2621	9497	2649	1351	1643
Kilo invalidations per second	7580	3778	126	17 211	12 662	13 315	35 971	13 572	4747	6525
Synchronization points per second	1198	16 277	9	446	318	716	508	753	760	10 040
Number of migrations	2	1	0	0	1	2	1	1	1	11

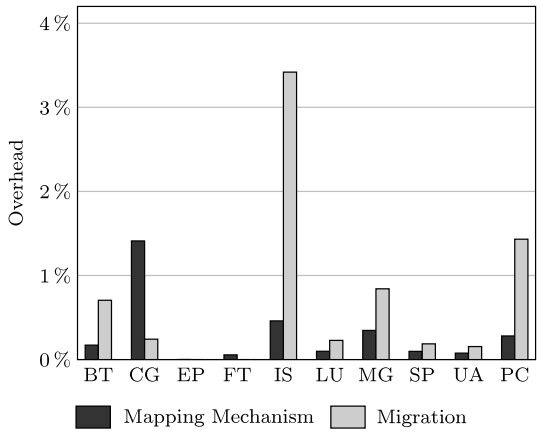


Fig. 14. Overhead of the mapping mechanism (consisting of the aging algorithm, the communication filter, the mapping algorithm and the symmetric mapping filter) and the migration, in % of the total execution time.

5.3. Overhead of the proposed mechanism

The overhead of our proposed mechanism can be divided into 3 categories. The first category is the mechanism to detect the communication patterns. The second type of overhead consists of the algorithms used to calculate the mapping and migrate the threads (the aging algorithm, communication filter, mapping algorithm and symmetric mapping filter). This overhead depends on the amount of synchronization points, as explained in the beginning of Section 5. The third category consists of the side effects of migrating the threads.

The mechanism to detect the communication patterns has no time overhead, since it is implemented directly in the cache memory hardware and does not influence the critical path. It imposes a small hardware overhead, consisting of the registers to store the communication vectors and the adder units to increment them. For the architecture used in the experiments, it is necessary to add eight 32-bit wide registers and 1 adder unit to each L1 cache memory. This represents a negligible chip area.

Fig. 14 shows the performance overhead of the mapping mechanism and the migration overhead as a percentage of the total execution time of each application. Regarding the mapping mechanism, CG presented the highest overhead, because it has the highest rate of synchronization points. EP only has 4 synchronization points during its entire execution, so the overhead from the mechanism is very low. For the other applications, the highest overhead from the mapping mechanism is in IS, where it represents only 0.46% of the execution time.

The most relevant side effect of migrating the threads is the addition of cache misses. Since the cores and their associated caches that are running the threads change, memory accesses from the migrated thread will generate cache misses right after the migration. Furthermore, these additional cache misses increase the load in the interconnections, because the data needs to be transferred from one cache to another. Although only 1 migration is performed in IS, it presented the highest overhead, since its execution time is the shortest. PC presented the second highest overhead because it migrated the most times. For the other applications, the migration overhead is less than 1%. The average total overhead, consisting of the mapping mechanism and the migrations, is 1.12%.

6. Related work

In this section, we contextualize the state of art in communication detection and thread mapping mechanisms and compare them to our proposal. We also evaluate work regarding process mapping, since it is related to thread mapping.

6.1. Static communication detection and thread mapping

A technique to collect the communication pattern of the threads of parallel applications based on shared memory is evaluated in [3]. They instrumented the Simics simulator to trace all the memory accesses, which were then analyzed to determine the communication pattern of the applications. In [4], the authors analyze the communication pattern of applications using the Pin dynamic binary analysis tool [2]. In [13], the potential of mapping the threads of applications taking into account the communication between them was evaluated. The Simics simulator was instrumented to monitor all memory accesses and detect the communication patterns of the applications. With these patterns, they created a static thread mapping and measured the performance improvement. Despite improving performance, these methods are infeasible for real applications, as they require simulation or dynamic binary analysis, which demand a lot of processing time. Furthermore, these methods detect only static communication patterns and cannot optimize the communication when the application presents different communication patterns along the execution.

6.2. Dynamic communication detection and thread mapping

An approach to improve the performance of distributed shared memory (DSM) systems is proposed in [18]. They divide the memory address space into blocks and keep the memory access pattern of each block in a pattern table. The pattern contains information on which threads accessed the corresponding block, as well as which type of access the threads performed, such as a read or write access. By looking at the patterns, the DSM coherence protocol can make speculative transactions in order to minimize the impact of maintaining the coherence. The pattern table used to predict future transactions could be also used to detect the communication. However, the amount of additional hardware required to store and update the pattern table is higher than the amount of hardware necessary to implement our proposal.

In [1], the authors show that hardware performance counters already present in current processors can be used to dynamically map parallel applications. They schedule threads by taking into account an indirect estimate of the communication pattern using hardware counters of the Power5 processor. These hardware counters monitor memory accesses that are resolved by cache memories located in remote chips. To decrease the overhead of the proposed mechanism, the mapping system is only triggered

after the number of core stall cycles and cache misses exceeds a given threshold. Memory accesses resolved by local cache memories or the main memory are not considered when detecting the communication, generating an incomplete communication pattern.

The ForestGOMP mapping library is introduced in [7]. This library integrates into the OpenMP runtime environment and gathers information about the different parallel sections of applications from hardware performance counters. The library generates data and thread mappings for the regions of the application. The data mapping is suitable for Non-Uniform Memory Access (NUMA) machines, as in these machines the latency to the memory banks may be different for each processor. ForestGOMP tries to keep the threads that share data nearby according to the memory hierarchy, as well as to place the memory pages in NUMA nodes close to the core that is accessing the page. The hardware counters they used to guide the thread and data mapping only indirectly estimate the communication patterns. Also, their work is limited to parallel applications that are based on OpenMP.

In [11], the Translation Lookaside Buffer (TLB) is used to dynamically detect the communication pattern. The TLB is responsible to perform the translation of virtual addresses to physical addresses and is present in most architectures that support virtual memory. As there is one TLB for each core, the communication pattern was detected by searching all TLBs for matching entries. The impact of spatial false communication is higher than in our cache coherence based mechanism, since the TLB based mechanism detects communication at the page level granularity. In [12], the authors propose a mechanism that uses the page table of the parallel application to detect the communication pattern by introducing additional page faults during the execution. In these mechanisms, the operating systems is responsible for calculating the communication matrix, while in our coherence based mechanism the hardware generates the communication matrix automatically, which reduces the overhead.

6.3. Process mapping of message-passing based applications

In [23], virtual machines running on clusters are migrated among the different nodes by considering the amount of communication between them. They detect the communication between the virtual machines by monitoring the source and destination fields of the packets sent on the network. Both the execution time and the network traffic were reduced by dynamically migrating the virtual machines to nearby nodes of the cluster. A similar method is proposed in [8], but it focuses on static mapping of parallel applications that are based on the Message Passing Interface (MPI). Compared to message passing environments, detecting communication in shared memory applications presents different challenges, since the communication is implicit and therefore requires new mechanisms to perform the detection.

7. Conclusions and future work

The communication between the threads of parallel applications is an important issue of multi-core architectures. The ongoing increase of the number of cores imposes a high communication overhead, and mechanisms to optimize the communication are necessary to take advantage of the higher level of parallelism. In this paper, we presented a new thread mapping mechanism that optimizes the communication using information provided by cache coherence protocols. Our mechanism is implemented directly in the cache memory subsystem and detects the communication during the execution of the applications. Furthermore, it is independent from the implementation of the parallel applications and presents no time overhead. We also proposed algorithms to dynamically migrate the threads. Our algorithms impose a low overhead and are able to migrate the threads of applications with very different characteristics.

We evaluated our proposal using the NAS parallel benchmarks and with a producer–consumer benchmark. We were able to dynamically identify the communication patterns for all applications and migrate them during the execution. Our detection mechanism and algorithms were also able to recognize the different communication patterns along the execution of the applications. We compared our mechanism to the operating system scheduler, a random mapping and to an oracle mapping. Compared to the operating system, we reduced the execution time, cache misses and number of invalidations by up to 13.9%, 30.5% and 39.4%, respectively.

Improvements were dependent on the communication characteristics of the applications. As expected, applications that communicated more and present heterogeneous communication patterns showed the greatest improvements. Applications that have homogeneous communication patterns did not present improvements. This is the expected result, as there is no difference in the communication among the threads to be exploited. Also, we observed the importance of dynamic mapping over static mapping, since only dynamic mapping mechanisms are capable of handling applications that present several communication patterns.

For the future, we intend to evaluate our mechanism using other benchmarks, with different communication characteristics. We also plan to evaluate how our proposal affects the energy consumption, since it improves the cache and interconnection usage.

Acknowledgments

This work was partially supported by CNPq and CAPES.

References

[1] R. Azimi, D.K. Tam, L. Soares, M. Stumm, Enhancing operating system support for multicore processors by using hardware performance monitoring, ACM SIGOPS Oper. Syst. Rev. 43 (2009) 56–65.
[2] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C. Luk, G. Lyons, H. Patil, A. Tal, Analyzing parallel programs with pin, IEEE Comput. 43 (2010) 34–41.
[3] N. Barrow-Williams, C. Fensch, S. Moore, A communication characterisation of splash-2 and parsec, in: IEEE International Symposium on Workload Characterization, IISWC, 2009, pp. 86–97.
[4] C. Bienia, S. Kumar, K. Li, PARSEC vs. SPLASH-2: a quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors, in: IEEE International Symposium on Workload Characterization, IISWC, 2008, pp. 47–56.
[5] S. Bokhari, On the mapping problem, IEEE Trans. Comput. C-30 (1981) 207–214.
[6] S. Borkar, A.A. Chien, The future of microprocessors, Commun. ACM 54 (2011) 67–77.
[7] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.A. Wacrenier, R. Namyst, Structuring the execution of OpenMP applications for multicore architectures, in: IEEE International Parallel & Distributed Processing Symposium, IPDPS, 2010.
[8] H. Chen, W. Chen, J. Huang, B. Robert, H. Kuhn, MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters, in: International Conference on Supercomputing, 2006, pp. 353–360.
[9] Z. Chishti, M.D. Powell, T.N. Vijaykumar, Optimizing replication, communication, and capacity allocation in CMPs, ACM SIGARCH Comput. Archit. News 33 (2005) 357–368.
[10] P.W. Coteus, J.U. Knickerbocker, C.H. Lam, Y.A. Vlasov, Technologies for exascale systems, IBM J. Res. Dev. 55 (2011) 14:1–14:12.
[11] E.H.M. Cruz, M. Diener, P.O.A. Navaux, Using the translation lookaside buffer to map threads in parallel applications based on shared memory, in: IEEE International Parallel & Distributed Processing Symposium, IPDPS, 2012, pp. 532–543.
[12] M. Diener, E.H.M. Cruz, P.O.A. Navaux, Communication-based mapping using shared pages, in: IEEE International Parallel & Distributed Processing Symposium, IPDPS, 2013, pp. 700–711.
[13] M. Diener, F.L. Madruga, E.R. Rodrigues, M.A.Z. Alves, P.O.A. Navaux, Evaluating thread placement based on memory access patterns for multi-core processors, in: IEEE International Conference on High Performance Computing and Communications, HPCC, 2010, pp. 491–496.
[14] Intel, Quad-Core Intel Xeon Processor 5400 Series Datasheet, Technical Report, 2008.
[15] Intel, 2nd generation Intel Core Processor Family, Technical Report, 2012.

[16] H. Jin, M. Frumkin, J. Yan, The OpenMP Implementation of NAS Parallel Benchmarks and its Performance, Technical Report, 1999.
[17] M. Johnson, H. McCraw, S. Moore, P. Mucci, J. Nelson, D. Terpstra, V. Weaver, T. Mohan, PAPI-V: performance monitoring for virtual machines, in: International Conference on Parallel Processing Workshops, ICPPW, 2012, pp. 194–199.
[18] A.C. Lai, B. Falsafi, Memory sharing predictor: the key to a speculative coherent DSM, in: International Symposium on Computer Architecture, ISCA, 1999.
[19] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: a full system simulation platform, IEEE Comput. 35 (2002) 50–58.
[20] M.M.K. Martin, M.D. Hill, D.J. Sorin, Why on-chip cache coherence is here to stay, Commun. ACM 55 (2012) 78.
[21] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, D. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, ACM SIGARCH Comput. Archit. News 33 (2005) 92–99.
[22] C. Osiakwan, S. Akl, The maximum weight perfect matching problem for complete weighted graphs is in PC, in: IEEE Symposium on Parallel and Distributed Processing, SPDP, 1990, pp. 880–887.
[23] J. Sonnek, J. Greensky, R. Reutiman, A. Chandra, Starling: minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration, in: International Conference on Parallel Processing, ICPP, 2010, pp. 228–237.
[24] J. Zhai, T. Sheng, J. He, Efficiently acquiring communication traces for large-scale parallel applications, IEEE Trans. Parallel Distrib. Syst. 22 (2011) 1862–1870.
[25] X. Zhou, W. Chen, W. Zheng, Cache sharing management for performance fairness in chip multiprocessors, in: International Conference on Parallel Architectures and Compilation Techniques, PACT, 2009, pp. 384–393.

Eduardo Henrique Molina da Cruz graduated in Computer Science at the State University of Maringa (UEM), Brazil, and received his masters degree at the Federal University of Rio Grande do Sul (UFRGS), Brazil, where he is currently a Ph.D. student.

His research focuses on improving the communication between threads on shared-memory architectures and to improve the locality of the memory accesses in architectures with non-uniform memory access (NUMA).

Matthias Diener graduated in Computer Engineering at the Berlin Institute of Technology (TU Berlin), Germany, and is currently a Ph.D. student at the Federal University of Rio Grande do Sul (UFRGS), Brazil.

His research interests are in improving the performance and energy efficiency of parallel applications that run on shared-memory architectures, by taking into account the memory access behavior of the application.

Marco Antonio Zanata Alves graduated in Computer Science at the São Paulo State University (UNESP), Brazil, and received his masters degree from the Federal University of Rio Grande do Sul (UFRGS), Brazil, where he is currently a Ph.D. student.

His research focuses on increasing the energy efficiency of cache memories for high performance computer architectures.

Philippe Olivier Alexandre Navaux is a Professor at the Federal University of Rio Grande do Sul (UFRGS), Brazil, since 1973.

He graduated in Electronic Engineering from UFRGS in 1970. He received his masters degree in Applied Physics from UFRGS in 1973 and his Ph.D. in Computer Science from INPG, France in 1979.

He is the head of the Parallel and Distributed Processing Group (GPPD) at UFRGS and a consultant to various national and international funding agencies such as DoE (US), ANR (FR), CNPq (BR), CAPES (BR) and others.