Performance Evaluation 88–89 (2015) 18–36

Contents lists available at ScienceDirect

Performance Evaluation

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

Characterizing communication and page usage of parallel applications for thread and data mapping

Matthias Diener [a,*], Eduardo H.M. Cruz [a], Laércio L. Pilla [c], Fabrice Dupros [b], Philippe O.A. Navaux [a]

[a] Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
[b] BRGM, Orléans, France
[c] Department of Informatics and Statistics, Federal University of Santa Catarina, Florianópolis, Brazil

ARTICLE INFO

Article history:
Received 23 June 2014
Received in revised form 5 February 2015
Accepted 12 March 2015
Available online 20 March 2015

Keywords:
Shared memory
Thread mapping
Data mapping
Multicore
NUMA

ABSTRACT

The parallelism in shared-memory systems has increased significantly with the advent and evolution of multicore processors. Current systems include several multicore and multithreaded processors with Non-Uniform Memory Access (NUMA) characteristics. These architectures require the adoption of two strategies for the efficient execution of parallel applications: (i) threads sharing data should be placed in such a way in the memory hierarchy that they execute on shared caches; and (ii) a thread should have the data that it accesses placed on the NUMA node where it is executing. We refer to these techniques as thread and data mapping, respectively. Both strategies require knowledge of the application's memory access behavior to identify the communication between threads and processes as well as their usage of memory pages.

In this paper, we introduce a profiling method to establish the suitability of parallel applications for improved mappings that take the memory hierarchy into account, based on a mathematical description of their memory access behaviors. Experiments with a large set of parallel workloads that are based on a variety of parallel APIs (MPI, OpenMP, Pthreads, and MPI+OpenMP) show that most applications can benefit from improved mappings. We provide a mechanism to compute optimized thread and data mappings. Experimental results with this mechanism showed performance improvements of up to 54% (20% on average), as well as reductions of the energy consumption of up to 37% (11% on average), compared to the default mapping by the operating system. Furthermore, our results show that thread and data mapping have to be performed jointly in order to achieve optimal improvements.

1. Introduction

Since reaching the limits of Instruction Level Parallelism (ILP), Thread Level Parallelism (TLP) has become important to continue increasing the performance of shared-memory computer systems. Increases in the TLP are accompanied by more complex memory hierarchies, consisting of several private and shared cache levels, as well as multiple memory controllers that introduce Non-Uniform Memory Access (NUMA) characteristics. As a result, the performance of memory accesses depends on the location of the data [1,2]. Accesses to data that is located on local caches and NUMA nodes have a higher

* Corresponding author.
E-mail address: mdiener@inf.ufrgs.br (M. Diener).

http://dx.doi.org/10.1016/j.peva.2015.03.001
0166-5316/

bandwidth and lower latency than accesses to remote caches or nodes [3]. Improving the *locality* of memory accesses is therefore an important way to achieve optimal performance in modern architectures [4].

For parallel applications, locality can be improved in two ways. First, by executing threads that access shared data close to each other in the memory hierarchy, they can benefit from shared caches and faster intra-chip interconnections [5,6]. We refer to accesses to shared data as *communication* in this paper, and call an optimized mapping of threads to processing units that takes communication into account a communication-aware *thread mapping*. Most parallel programming APIs for shared memory, such as OpenMP and Pthreads, directly use memory accesses to communicate. Even implementations of the Message Passing Interface (MPI), which uses explicit functions to communicate, contain optimizations to communicate via shared memory, such as Nemesis [7] for MPICH2 [8] and KNEM [9] for Open MPI [10]. Second, the memory pages that a thread accesses should be placed on NUMA nodes close to where it is executing to reduce the inter-node traffic, as well as to increase the performance of accesses to the main memory [11]. We call this technique *data mapping*.

The goal of this paper is to characterize the memory access behavior of parallel applications to determine their suitability for mapping and evaluate their performance improvements using mappings that optimize locality. To characterize the communication, we introduce metrics that describe the spatial, temporal and volume properties of memory accesses to shared memory areas. In contrast to previous work that uses a logical definition of communication [12,13], we use a broader definition that focuses on the architectural impact of these accesses. We characterize the memory page usage of the applications by analyzing the distribution of accesses from different NUMA nodes during the execution. The characterizations are then used to perform an optimized thread and data mapping. Related work in this area mostly treats thread and data mapping as separate problems and only handles one of them [14,15]. We make the case that mapping has to be performed in an integrated way to achieve maximum benefits.

The main contributions of this paper are:

- We introduce metrics and a methodology to evaluate the communication and page usage of parallel applications running on shared memory architectures and use it to analyze their potential for thread and data mapping.
- We present a mechanism to employ this information and calculate thread and data mappings that optimize memory access locality.
- We characterize a large set of parallel applications and evaluate their performance and energy consumption improvements using the optimized mappings.

The rest of this paper is organized as follows. The next section briefly discusses the benefits of improved mappings and introduces the workloads that we use in this paper. Section 3 presents related work about characterization of memory access behavior and mapping. Our communication and page usage characterization methodologies are presented in Sections 4 and 5, respectively, together with evaluations of the workloads. Section 6 introduces our optimized mapping mechanism that uses the memory access behavior to perform thread and data mapping that maximizes locality. The experimental results of this mapping mechanism are presented in Section 7. Finally, Section 8 summarizes our conclusions and presents ideas for future work.

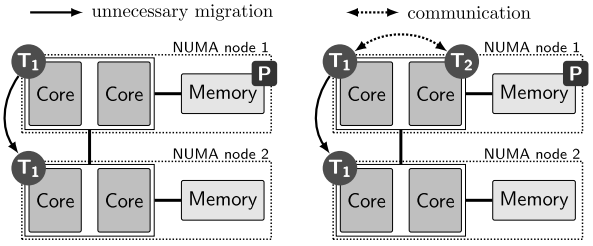
2. Background

2.1. Benefits of improved mappings

Thread and data mapping aim to improve the memory accesses to shared and private data in parallel applications. Thread mapping improves the usage of the interconnections by reducing inter-chip traffic that has a higher latency and lower bandwidth than intra-chip interconnections. It also reduces the number of cache misses of parallel applications. In situations where threads read the same data, executing the threads on the same shared cache reduces data replication, thereby increasing the cache space available for the application [16]. It can also reduce cache-to-cache transfers. In situations where threads write to the same memory addresses, an optimized thread mapping also reduces cache line invalidations. Data mapping improves the memory locality on NUMA machines by reducing the number of accesses to remote memory banks. As thread mapping, it improves the usage of the interconnections. This increases the memory bandwidth available in the system and reduces the average memory access latency.

It is important to note that thread mapping is a prerequisite for data mapping, for the two reasons depicted in Fig. 1. In the figure, an architecture consisting of two NUMA nodes with two cores per node is shown. Consider that two threads T_1 and T_2 are accessing a page P. The first benefit of thread mapping for data mapping is that it prevents unnecessary thread migrations between NUMA nodes, such that threads can benefit from the local data accesses. In Fig. 1(a), if page P was placed on the NUMA node where T_1 is executing and T_1 is migrated to the other node, the data mapping is ineffective.

The second reason is that data mapping alone is not able to improve locality when more than one thread accesses the same page, since the threads may be executing on cores of different NUMA nodes. In this situation, only the threads that are executing on the same node where the data is located can benefit from the increased locality. As shown in Fig. 1(b), when mapping T_1 and T_2 to the same NUMA node, accesses to page P by both threads will be considered as accesses to the local node. In contrast, if T_1 was executing on the other NUMA node, only accesses by T_2 would be considered local. By performing the thread mapping, threads that share a lot of data are executed on the same node, improving the effectiveness of the data mapping.



(a) Prevent unnecessary thread migrations between NUMA nodes.

(b) Increase locality when threads access the same data.

Fig. 1. Why thread mapping is required for data mapping. The figures show a machine with two NUMA nodes, with each node consisting of two cores. Consider that two threads T_1 and T_2 access a page P.

Table 1
Overview of the benchmark suites used in the evaluation, showing parallelization API, benchmark names, input size and average memory usage per application.

Benchmark suite	Parallel API	Benchmark names	Input size	Memory usage
NAS-OMP [17] v3.3.1	OpenMP	BT, CG, DC, EP, FT IS, LU, MG, SP, UA (10)	*D* (DC: *B*)	24.8 GByte
PARSEC [18] v3.0 beta	Pthreads, OpenMP	Blackscholes, Bodytrack, Facesim, Ferret, Freqmine, Raytrace, Swaptions, Fluidanimate, Vips, X264, Canneal, Dedup, Streamcluster (13)	*native*	3.2 GByte
HPCC [19] v1.4.3	MPI	HPCC (single application with 16 workloads)	4000^2 matrix	19.3 GByte
NAS-MZ [20] v3.3.1	MPI, OpenMP	BT-MZ, LU-MZ, SP-MZ (3)	*D*	10.8 GByte

2.2. Benchmark selection and methodology

In this paper, we characterize a diverse set of parallel benchmarks that use several parallelization models and have different memory access behaviors, summarized in Table 1. We selected the following benchmark suites:

NAS-OMP [17] is the OpenMP implementation of the NAS Parallel Benchmarks (NPB), which consists of 10 applications from the HPC domain. All applications except DC are executed with the *D* input size. DC is executed with its largest input size, *B*.

PARSEC [18] is a suite of 13 benchmarks that focus on emerging workloads and are implemented with OpenMP and Pthreads. All benchmarks were executed with the *native* input set, which is the largest input set available.

HPCC [19] is a single application that consists of 16 different HPC workloads, such as HPL and STREAM. It is implemented with MPI.

NAS-MZ [20] is the Multi-Zone implementation of the NAS benchmarks. It uses a hybrid parallelization using the MPI and OpenMP models. All applications are executed with the *D* input size.

For the evaluation in the following sections, we generally execute the applications with 64 threads or processes on a 4-node NUMA system. Benchmarks that cannot be executed with 64 threads are executed with a number of threads as close as possible to 64. The NAS-MZ benchmarks were executed with 4 processes and 16 threads per process.

3. Related work

3.1. Characterizing communication and page usage

Related work that characterizes communication mostly focuses on applications that use explicit message passing frameworks, such as MPI. Examples include [12,21,22]. A characterization methodology for communication is presented in [23,24], 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. [13] perform a communication analysis of the PARSEC and Splash2 benchmark suites. They focus on communication on the logical level and therefore only count memory accesses that really represent communication, filtering out memory accesses that are 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. Regarding the page usage for data mapping, Majo et al. [25] identify shared data as a challenge for improved performance on NUMA systems, as it increases the amount of memory accesses to remote NUMA nodes. They modify four applications from PARSEC to reduce the amount of data that is shared between threads to improve performance. In this paper, we adopt a different approach by optimizing the mapping instead of modifying the applications.

3.2. Thread and process mapping

Several related mechanisms that focus on thread and process mapping considering communication have been proposed.

3.2.1. Message passing

For parallel applications that communicate through explicit message passing, such as MPI, most previous research focuses on methods to trace the messages and use the information to perform a static process mapping. MPIPP [26] is a framework for static process mapping, consisting of a message tracer and mapping algorithm. In [27], an extension for the MPICH2 framework [8] is proposed, where the communicating pattern can be provided either by the application programmer or generated via execution traces. In [28], a similar extension for Open MPI [10] is introduced, where the user needs to specify the desired mapping. Both frameworks focus on static mappings only. Static mappings of particular applications are evaluated in [29–31] for the NAS-MPI applications, a CFD-kernel, and an application based on domain decomposition, respectively. None of the proposals support dynamic mapping.

3.2.2. Multithreaded applications

In [32], techniques to statically collect the communication pattern of the threads of parallel applications based on shared memory were evaluated. Their method consists of instrumenting simulators to generate memory access traces, which are analyzed to determine the communication pattern of the applications. The application is then executed with static mappings based in the detected patterns.

Several mechanisms use indirect communication statistics from hardware counters to perform the mapping. Azimi et al. [33] use hardware counters that provide memory addresses of requests resolved by remote cache memories. It detects incomplete communication patterns, since memory requests resolved by local caches are not considered. The usage of the instructions per cycle (IPC) metric to guide thread mapping is evaluated in Autopin [34]. Autopin measures the IPC of several mappings fed to it and executes the application with the thread mapping that presented the highest IPC. Cruz et al. [5] detect the communication pattern by comparing the contents of the Translation Lookaside Buffer (TLB). SPCD [6] uses page faults of parallel applications to detect communication.

3.2.3. Mapping algorithms

Several mapping algorithms to optimize communication have been proposed. Many algorithms are based on graph partitioning, such as METIS [35], the Zoltan toolkit [36] and Scotch [37]. They use a graph representation of the hardware topology. However, some topologies are hierarchical and may be represented as trees [38]. One example of tree-based mapping algorithm is TreeMatch [39], which focuses on thread mapping in shared memory architectures.

3.3. Data mapping

Traditional data mapping strategies, such as *first-touch*, *next-touch* [40,41] and *interleaving* have been used by operating systems to allocate memory on NUMA machines. These strategies present several disadvantages because they do not gather any page usage statistics from the running application. Some techniques propose the modification of source code, by automatic insertion of mapping code by tools or compilers [11], or by direct optimization of application source code [25].

Several proposals perform data mapping taking into account runtime statistics. Awasthi et al. [42] use information from the memory controller, such as row-buffer hit rates and queuing delays, to guide page mapping policies. Carrefour [43] is a mechanism for AMD-based architectures that uses sampled memory accesses to perform mapping decisions during runtime. Due to the overhead, the mechanism is only enabled for a small number of pages (30,000). For this reason, the improvements (especially for the NAS-OMP benchmarks) are relatively low. MemProf [44] uses a similar sampling mechanism to profile memory accesses and to determine which objects are accessed from remote NUMA nodes. The generated information can then be used to manually change the source code of applications to improve their behavior on NUMA architectures.

Marathe et al. [15] present an automatic page placement scheme for NUMA platforms by tracking memory addresses from the performance monitoring unit (PMU) of the Itanium-2 processor. Marathe and Mueller [45] also make use of the PMU of the Itanium-2, but perform the mapping dynamically. Likewise, Tikir and Hollingsworth [46] perform data mapping dynamically, by using hardware monitors from the Sun Fire architecture. In Section 7, we will compare our proposed mechanism to several of the mechanisms presented here.

Several papers discuss aspects of page placement on NUMA architectures. Gaud et al. [47] discuss some of the challenges of increasing the page size, which can reduce the benefits of mapping due to a lower mapping granularity. Their discussions are echoed by our analysis of page access behavior for large pages in Section 5. Blagodurov [48] discuss contention on NUMA architectures and identify thread migrations between NUMA nodes as detrimental to performance. Majo and Gross [49] identify memory accesses to remote NUMA nodes as a challenge for optimal performance and introduce a set of OpenMP directives to perform distribution of data. The best distribution policy has to be chosen manually and may differ between different hardware architectures. To the best of our knowledge, no systematic evaluation of the page access behavior of parallel applications has been performed before.

4. Characterizing communication behavior

The communication behavior of parallel applications can be described with three properties: the *spatial*, *volume* and *temporal* components [22,24]. In this section, we will first discuss the spatial and volume properties, which we call the *heterogeneity* and *amount* of communication. Then, we will analyze the temporal behavior in terms of changes to these two metrics during the execution of the application. To characterize the applications, we trace all memory accesses of the application in a Pin-based simulator. More details on the simulation environment are given in Section 6.1.

4.1. Communication metrics

We refer to memory accesses to the same address by two different threads as *communication events*. In contrast to previous work that focuses on the logical communication behavior [13], we also consider memory accesses that are not strictly necessary as communication events, such as re-reads of data due to register pressure, since we are mostly interested in the architectural implications of communication. For the same reason, we look at communication at the cache line granularity (64 Bytes on most current architectures), as thread mapping focuses mostly on caches.

4.1.1. Communication heterogeneity

The first metric we introduce describes the spatial communication behavior. It is based on the intuition that, to benefit from thread mapping, it is necessary to have groups of threads that share the same data among each other, and not with other threads. Conversely, if all threads share the same amount of data in the same way among them, no improvements from thread mapping are expected. To formalize this intuition, consider a *communication matrix* M, where each element $M[i][j]$ contains the number of communication events between threads i and j. We can determine the potential for thread mapping of the application by analyzing the differences between the amount of communication between the threads.

We describe the differences as the *heterogeneity* of the communication. First, we normalize the communication matrix to its maximum value, as shown in Eq. (1), to make it easier to compare the heterogeneity between different applications. Then, the heterogeneity C_H is calculated with Eq. (2), where T is the number of threads. The equation calculates the average variance of the communication of each thread. The higher the heterogeneity, the higher the potential for thread mapping, since there are larger differences between the amounts of communication. If heterogeneity is low, the amount of communication between the threads is similar, thus any thread mapping applied would result in similar performance for accesses to shared data.

$$M_{norm} = M / \max(M) \cdot 100 \tag{1}$$

$$C_H = \frac{\sum_{i=1}^{T} \sum_{j=1}^{T} \left(\frac{\sum_{k=1}^{T} M_{norm}[i][k]}{T} - M_{norm}[i][j] \right)^2}{T^2}. \tag{2}$$

4.1.2. Communication amount

To describe the volume of communication, we use another metric, which we call the *amount* of communication C_A. We use the average number of communication events per pair of threads, calculated with Eq. (3). When there is little communication, thread mapping has less potential for improvements, even when the heterogeneity is high. Therefore, both the amount and heterogeneity of the communication must be taken into account when analyzing the potential for thread mapping of an application.

$$C_A = \frac{\sum_{i=1}^{T} \sum_{j=1}^{T} M[i][j]}{T^2}. \tag{3}$$

4.1.3. Temporal communication behavior

The previous two metrics describe the state of the communication matrix at a certain point during execution. To analyze the temporal communication behavior, we separate the execution into time slices of a fixed length and calculate the heterogeneity and amount of communication for each slice. We perform a *phase detection* on the temporal behavior of the application using the heterogeneity value of the time slices. A new phase starts when the average heterogeneity of the last n time slices differs from the n time slices before them by at least $diff_{min}$. The phase detection is formalized in Eq. (4), where t is the current time slice, $(x : y)$ represents the time slice range from x to y, and $havg(t)$ represents the average heterogeneity of time slice t.

$$\frac{|havg(t-n:t) - havg(t-2n:t-n)|}{havg(t-n:t)} \geq diff_{min}. \tag{4}$$

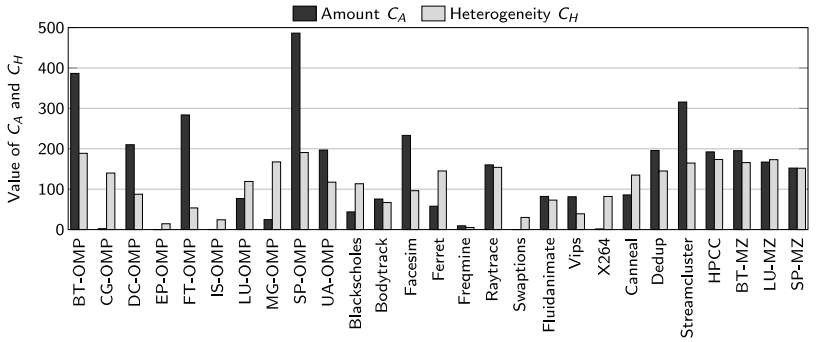


Fig. 2. Global communication behavior.

In this paper, we use a time slice length of 10 ms and $n = 50$, which leads to a minimum phase length of 500 ms. We use a value of 0.2 for $diff_{min}$. It is important to mention that these values are used to analyze and compare the dynamic behavior of the benchmarks. Section 6 will discuss under which circumstances a thread migration is performed.

4.2. Communication behavior of the benchmarks

All benchmarks were executed with the configuration and methodology described in Section 2.2. We begin with a discussion of the global communication behavior and then analyze the dynamic behavior during execution.

4.2.1. Global behavior

Fig. 2 shows the global communication behavior of the parallel applications, in terms of the heterogeneity and amount of communication during their full execution. Fig. 3 shows the global communication matrices for a selection of the benchmarks. Each cell contains the number of communication events between pairs of threads, normalized by the maximum for each benchmark. Darker cells indicate more communication.

Several benchmarks, such as BT-OMP, MG-OMP and SP-OMP, show large amounts of communication between neighboring threads, such as threads 0 and 1. This is a common behavior of applications that are based on domain decomposition, where communication is performed on shared data on the border of each domain. These applications also show significant communication between thread 0 and all other threads, indicating an initialization and reduction behavior. LU-OMP has a high heterogeneity as well, but communication happens mostly between distant threads, with smaller amounts of communication between neighboring threads. FT-OMP and Vips are applications with a low heterogeneity, demonstrated by their homogeneous communication matrices.

Fluidanimate shows communication between pairs of threads that are not neighbors, but farther apart, such as pairs (0,5) and (0,29). Streamcluster shows a pipeline model, where groups of threads (pipeline stages) communicate among themselves. HPCC consists of 16 stages, each with different communication characteristics. Stage 7 is shown in the figure, which shows a complex communication behavior with several groups of threads communicating with each other. In the NAS Multi-Zone benchmarks, most communication is performed between the processes and between threads that are not direct neighbors, but farther apart, such as threads (3,5). For all three Multi-Zone benchmarks, the communication pattern changes when modifying the number of processes and threads.

4.2.2. Dynamic behavior

Fig. 4 shows the number of communication phases per second of each benchmark. We expect more improvements from benchmarks that have a stable communication pattern with less phase changes. Not every phase change requires a migration though. Section 6.2 discusses under which conditions a migration is performed. As an illustration of dynamic behavior, consider Fig. 5, which shows the dynamic behavior of the SP-OMP benchmark. For a better visualization, we modified the application to execute 2 time steps only, which corresponds to about 31,000 time slices. Each black dot represents the amount of communication during a time slice, while the gray dots indicate the heterogeneity for a slice.

We can differentiate several communication phases. During the initialization of the application, a large amount of communication happens between all pairs of threads. Each time step passes its initial 80% of time with a medium amount of communication and a high heterogeneity, corresponding to a communication pattern between neighboring threads. At the end of each time step, bursts of communication occur between the master thread and all other threads, corresponding to a reduction pattern. This regular communication behavior is then repeated for all the time steps of the application. During the finalization, bursts of communication with a low heterogeneity occur.

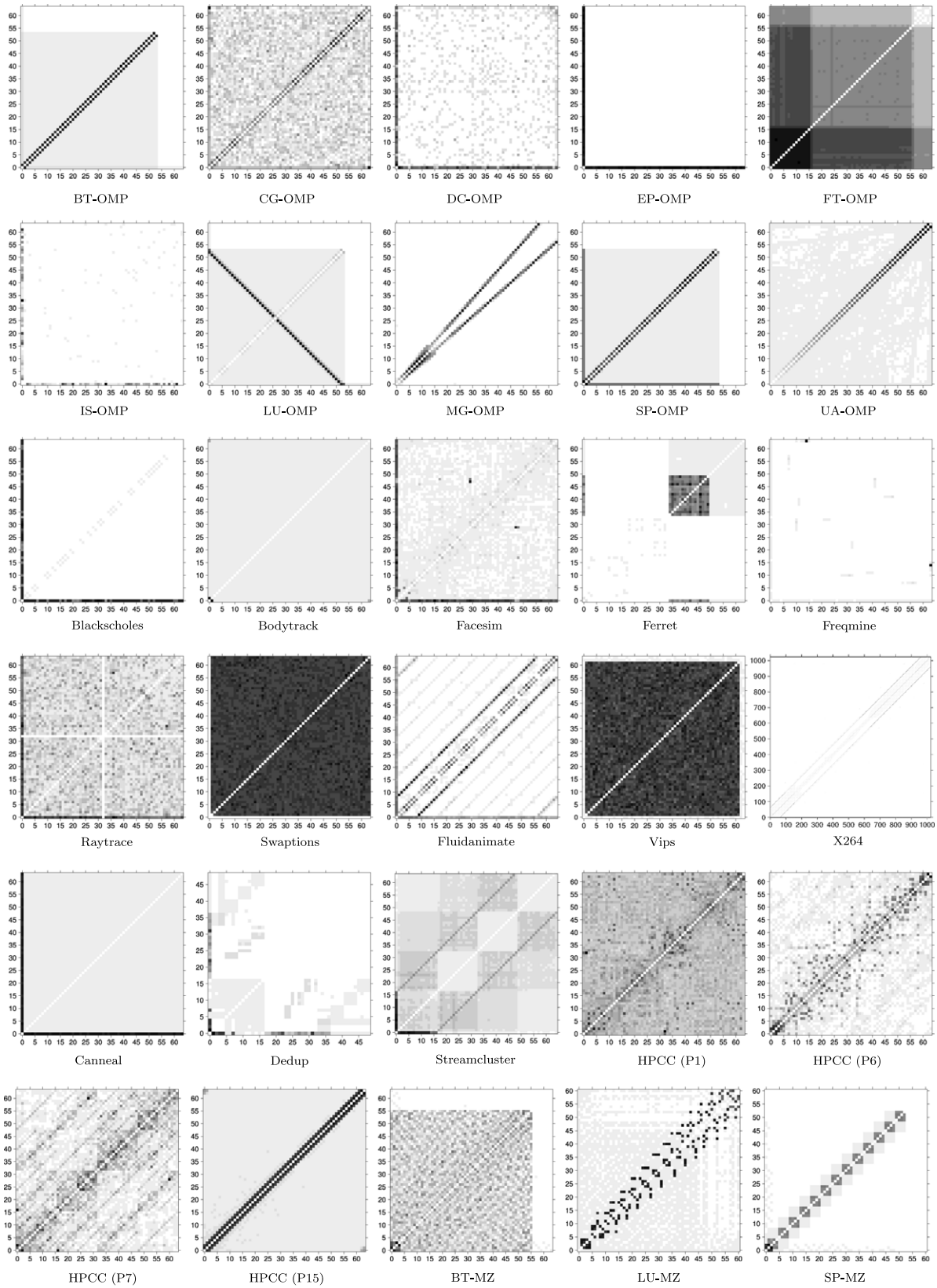


Fig. 3. Global communication matrices of the benchmarks. Axes represent thread IDs. Cells indicate the amount of communication between pairs of threads. Darker cells indicate more communication.

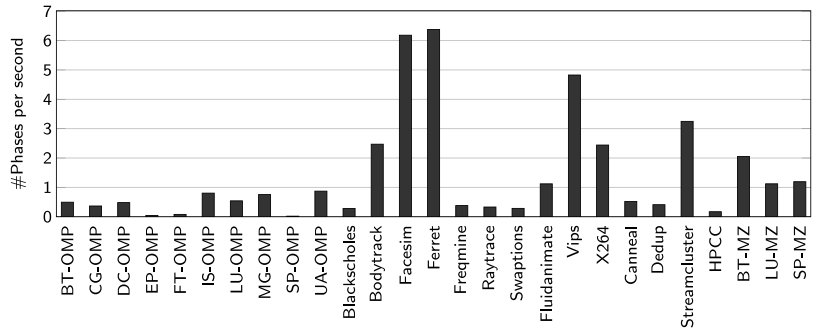


Fig. 4. Number of communication phases per second.

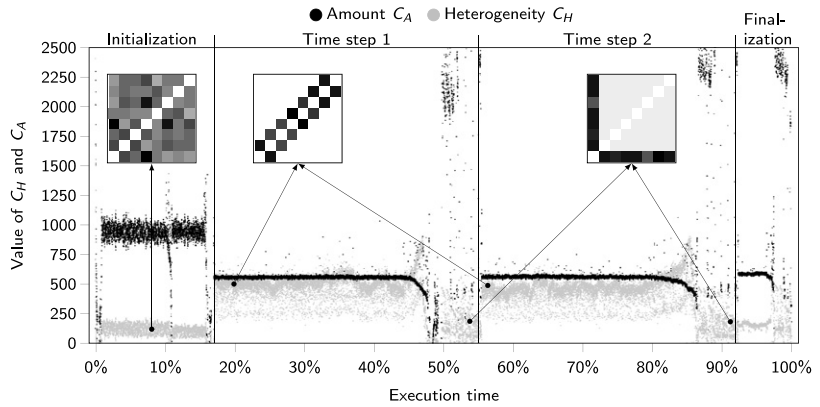


Fig. 5. Dynamic communication behavior of the SP-OMP benchmark for 2 time steps. The arrows point to communication matrices (for the first 8 threads) at specific time slices.

4.3. Summary

Summarizing the application behavior, we expect that applications with a high heterogeneity, high amount of communication and low number of communication phases have a higher suitability for thread mapping. Most of the NAS benchmarks and HPCC fulfill these criteria. Due to their less structured behavior, only a minority of PARSEC benchmarks appear to be suitable for thread mapping. For the reasons mentioned before, even in these applications thread mapping can improve the results of data mapping.

5. Characterizing page usage behavior

This section introduces a methodology that analyzes the page usage of parallel applications to characterize their suitability for data mapping on NUMA machines. We begin with a discussion of the methodology and analyze the benchmarks afterwards. As in the previous section, we use a Pin-based simulator to characterize the benchmarks.

5.1. Metrics for page usage

Similar to the communication behavior, we describe the page usage in terms of the spatial, volume and temporal components. The spatial component is described by how many of the memory accesses to a page originate from a single NUMA node, which we call the *page exclusivity*. By scaling the page exclusivity with the total number of memory accesses (the volume component), we are able to describe the behavior of the whole application, which we call the *application exclusivity*. We describe the temporal behavior of an application through the changes to the exclusivity during execution. We also introduce metrics for the minimum memory usage of an application, below which no gains from data mapping can be expected, and a metric to describe the accuracy of a given data mapping.

5.1.1. Page exclusivity

The potential for data mapping of a page is proportional to the amount of memory accesses from a single NUMA node. That is, if a page presents most of its accesses from the same node, it has more potential for data mapping than a page that is accessed from several nodes. We call the highest number of memory accesses to a page from a single NUMA node compared

to the number of accesses from all nodes the *page exclusivity* E_{Page}. The higher the exclusivity of a page, the higher its potential for data mapping. E_{Page} is calculated with Eq. (5), where $MemAcc[p][n]$ is the number of memory accesses to page p from NUMA node n, and N is the total number of NUMA nodes. The max function returns the highest number of memory accesses to a page from a NUMA node.

$$E_{Page}[p] = \frac{\max(MemAcc[p])}{\sum_{n=1}^{N} MemAcc[p][n]}. \tag{5}$$

The exclusivity is minimal when a page has exactly the same amount of accesses from all nodes. In this case, the exclusivity is given by $1/N$. The exclusivity achieves its maximum when all accesses to the corresponding page originate from the same NUMA node. In this case, the exclusivity is 1. As discussed in Section 2.1, page usage depends on the mapping of threads to NUMA nodes. It is important to note that the thread mapping influences data mapping metrics, as they consider the NUMA node that generated the memory accesses, not the threads themselves. In order to show the maximum gains possible, we assign the threads to the NUMA nodes in such a way that the exclusivity is highest. For example, assigning threads that access the same data to the same NUMA node will result in a higher exclusivity than when assigning them to different nodes.

5.1.2. Application exclusivity

Besides the exclusivity of a page described in the previous metric, the amount of memory accesses to it has to be taken into account as well. The higher the amount of accesses, the higher the potential for data mapping. To calculate the *application exclusivity* E_{App}, we scale the exclusivity of each page with the number of memory accesses to it, and divide this value by the total number of memory accesses. This operation is shown in Eq. (6), where P is the total number of pages. Like the page exclusivity, the minimum and maximum values of the application exclusivity are $1/N$ and 1, respectively. The exclusivity is independent of the mapping of memory pages to NUMA nodes. The data mapping matters for the accuracy of the mapping, discussed in Section 5.1.4.

$$E_{App} = \frac{\sum_{p=1}^{P} (E_{Page}[p] \cdot \sum_{n=1}^{N} MemAcc[p][n])}{\sum_{p=1}^{P} \sum_{n=1}^{N} MemAcc[p][n]}. \tag{6}$$

5.1.3. Total memory usage

Another requirement for data mapping is that the memory usage of the application must be significantly higher than the cache size of the system. We use Eq. (7) as a lower bound, where $PageSize$ is the size of each page, P is the number of pages, C is the total number of caches in the system and $CacheSize[i]$ is the size of Cache i.

$$P \cdot PageSize > \sum_{i=1}^{C} CacheSize[i]. \tag{7}$$

If the memory that an application uses fits into the processor caches, less improvements from a data mapping policy can be expected, as most memory accesses can be filtered by the caches. In practice, this affects two of the applications we discuss in this paper, EP-OMP and Swaptions, whose memory usages are 66 MByte and 7 MByte, respectively.

5.1.4. Accuracy of page mapping

To determine the correctness of a mapping of pages to NUMA nodes, we first use Eq. (8) to define if a single page is located on the correct node. $MaxNode[p]$ is the NUMA node with the highest accesses to page p and $CurNode[p]$ is the NUMA node where page p is currently located. Here, we assign 1 to the accuracy value if the page is located on the NUMA node with the highest number of accesses to the page, and 0 otherwise.

$$Acc_{Page}[p] = \begin{cases} 1, & \text{if } MaxNode[p] = CurNode[p] \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

For the whole application, we scale the accuracy of each page with the number of accesses to it, as shown in Eq. (9), where P is the number of pages.

$$Acc_{App} = \frac{\sum_{p=1}^{P} (Acc_{Page}[p] \cdot \sum_{n=1}^{N} MemAcc[p][n])}{\sum_{p=1}^{P} \sum_{n=1}^{N} MemAcc[p][n]}. \tag{9}$$

Exclusivity and accuracy are connected, although they are not the same. We use the exclusivity to determine if an application is suitable for an improved data mapping and the accuracy to describe the quality of a data mapping. In other words, the exclusivity is a property of a page/application, while the accuracy describes a particular mapping of pages to NUMA nodes.

5.1.5. Dynamic behavior

Similar to the dynamic communication behavior, we divide the execution of each application into time slices of a fixed size and calculate the application exclusivity for each slice. We also analyze the correctness of the page mapping during each slice, to determine how many pages need to be migrated to their correct NUMA node. As before, we use time slices of 10 ms.

5.2. Page usage of the benchmarks

As the baseline for our discussion in this section, we execute the applications with 64 threads on a 4 NUMA node machine with a page size of 4 KByte.

5.2.1. Global behavior

Fig. 6 shows the global application exclusivity for different page sizes with a mapping of threads to NUMA nodes that maximizes the exclusivity. We use page sizes of 4 KByte, which is the default of the x86 architecture, 2 MByte, which is the maximum page size supported by x86, and 64 MByte, which is a possible page size for future systems. In most applications, the exclusivity is high, showing the importance of data mapping. It is important to note that even when increasing the page size, the exclusivity only decreases slightly. On average, the exclusivity for the three page sizes is 84.9%, 77.4% and 69.4%, respectively. To show the influence of the thread mapping on the exclusivity, we also show the difference between the thread mappings that minimize and maximize the exclusivity for each application in Fig. 7. With the minimizing mapping, the average exclusivity for the three page sizes is 78.5%, 70.3% and 65.2%.

Fig. 8 shows the accuracy of the first-touch page mapping for 4 KByte pages with the two metrics defined in Section 5.1.4. The results for the per-page accuracy Acc_{Page} show that the first touch placement has a high accuracy of 78.5%. However, when using the weighed accuracy Acc_{App}, the accuracy falls for most applications. This indicates that, in order to improve the data mapping, it is possible to start with a first-touch mapping and then migrate a relatively low number of pages to their correct NUMA nodes. These pages have an above average number of memory accesses for most of the benchmarks.

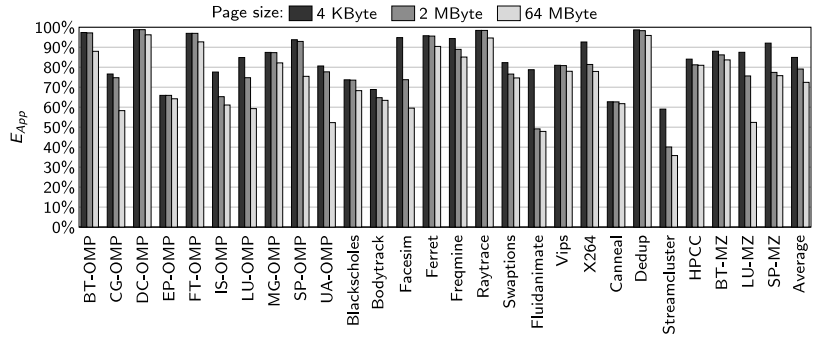


Fig. 6. Application exclusivity E_{App} for different page sizes, for a thread mapping that maximizes the exclusivity.

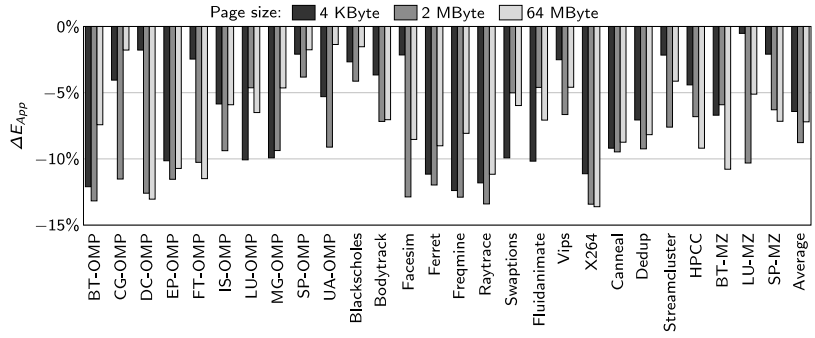


Fig. 7. Difference of the application exclusivity E_{App} between the thread mappings that minimize and maximize communication.

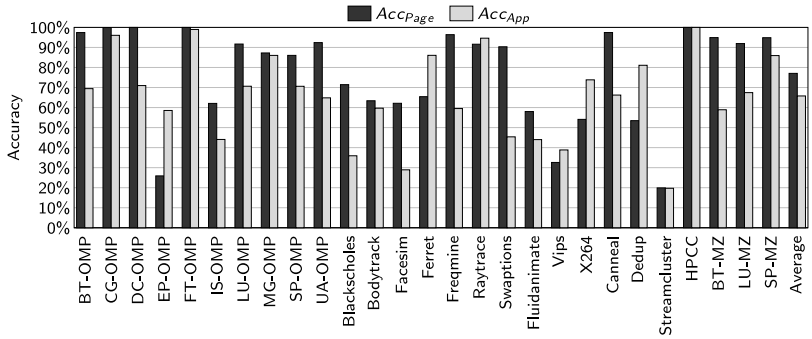


Fig. 8. Accuracy of the first-touch page mapping for 4 KByte pages.

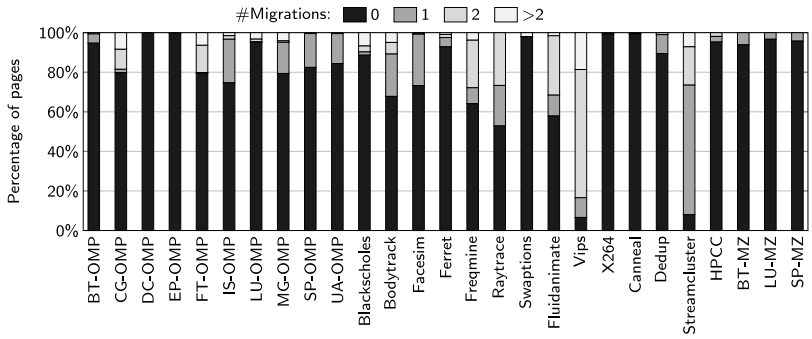


Fig. 9. Number of page migrations.

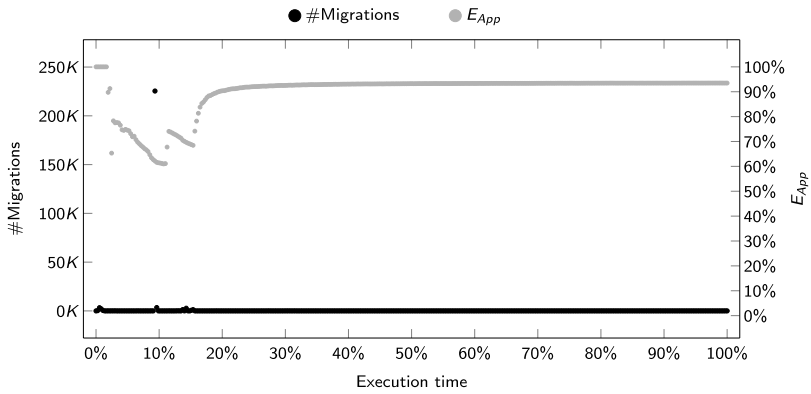


Fig. 10. Dynamic page usage behavior of SP-OMP. Black dots correspond to the number of page migrations at each time slice, while gray dots indicate the changing application exclusivity due to a changing memory access behavior.

5.2.2. Dynamic behavior

To evaluate the dynamic behavior of the workloads, we measure the number of page migrations that have to be performed in order to maintain each page on the NUMA node with the most accesses. Fig. 9 shows the migrations for all the workloads, considering no thread migrations and a first-touch policy. The results show that most of the pages have to be migrated only once or not at all. The dynamic page usage behavior of the SP-OMP benchmark is shown in Fig. 10. For each time slice, we show the number of pages that need to be migrated to their correct NUMA node as well as the application exclusivity E_{App}. The results show that during the initialization of the application, the exclusivity varies between 60% and 80%, indicating shared accesses to memory pages. As soon as the parallel part of SP-OMP starts, the exclusivity begins to stabilize and soon reaches its maximum at 92%. For the number of migrations, we can confirm that a significant number of page migrations are necessary only during the time slice at the start of the parallel computation. For the rest of the execution, the page usage is stable and no pages need to be migrated.

5.3. Summary

To summarize the page usage behavior of the evaluated benchmarks, we can affirm that most of them are suitable for data mapping due to their high exclusivity. Even when increasing the page size from the default 4 KByte to 2 MByte and 64 MByte, the exclusivity reduces only slightly. Compared to a first-touch mapping, only few pages need to be migrated in order to achieve an accurate mapping (less than 22% of pages on average). However, these pages represent a much higher percentage of memory accesses (more than 35% of total memory accesses on average), showing the importance of data mapping. Furthermore, most pages need to be migrated only once, which limits the overhead of migrations.

6. Optimized mapping mechanism

This section describes a thread and data mapping mechanism that optimizes memory access locality, which we refer to as *OPT* in the paper. It uses the characterization methodology described in the previous sections. OPT consists of three parts: A memory tracer to determine the communication and page usage characteristics of a parallel application, a thread mapping mechanism that calculates optimized thread to core assignments based on the communication behavior and a data mapping mechanism to assign memory pages to NUMA nodes based on the page usage.

6.1. Detecting memory access behavior

To determine the memory access behavior, we built a memory tracer using the PIN Dynamic Binary Instrumentation tool [50]. For every time slice (10 ms by default), the tool outputs the communication matrix and the page usage of the whole application. We consider all memory accesses to the same cache line (with a default size of 64 Bytes) within the same time slice as communication events. In this way, we can characterize communication independently of the hardware architecture. Despite the overhead caused by the tracing (about 10x to 20x depending on the application), it is still feasible to characterize very large applications, such as the ones we use in the paper. Since we do not save the full memory access trace, but only the behavior for each time slice, the data generated is very small (less than 80 MByte per application).

For statically allocated memory addresses, the memory addresses contained in the memory trace are valid, since they are defined by the linker during compilation. However, regarding the dynamically allocated memory, the memory addresses may change between executions. To overcome this issue, we keep track of all dynamic memory allocation functions, as in [15]. Since multi-process applications (such as MPI) communicate through shared memory segments, we convert the virtual address to the physical address using the `proc` filesystem for these segments in order to detect the communication. In our experiments, the thread mapping does not need to be modified when changing the input data, as long as the application is executed with the same number of threads. Changing the input data usually changes the data mapping however, in this case the application would need to be profiled again.

6.2. Thread mapping

The communication matrix is evaluated by the thread mapping algorithm. We describe first the mapping algorithm itself, followed by the way we handle migrations during execution.

6.2.1. Calculating the thread mapping

The information provided by the communication detection is used to calculate an optimized mapping from processes and threads to processing units (PUs). Processes and threads are handled in the same way by the mapping algorithm. The thread mapping problem is NP-hard [51], therefore it is necessary to use efficient heuristic algorithms to calculate the mapping. This section describes the mapping algorithm we use. The mapping problem is modeled with two undirected graphs, a communication graph and a hierarchy graph. In the communication graph, vertices represent processes and edges represent the amount of communication between them. In the hierarchy graph, vertices represent levels of the memory hierarchy and PUs, while edges represent the interconnections between them. The communication graph is obtained from the communication matrix during runtime, while the hierarchy graph is generated from the hardware information provided by the kernel.

To calculate the mapping, we use the dual recursive bipartitioning algorithm of the Scotch mapping library [37], version 6.0, which is used widely in the community [29,30,52]. It seamlessly supports under/overprovisioning, that is, executing with less or more processes than the architecture can execute in parallel and does not assume one process or thread per core. Scotch has a complexity of $\mathcal{O}(N^3)$, where N is the number of processes to be mapped [52]. In our experiments, Scotch takes less than 1 s to calculate the thread mapping of 4000 threads. Since most applications run with less threads and the mapping is calculated before execution, its overhead is negligible. The algorithm receives the communication and hierarchy graphs as input, and outputs the PU for each process such that the total cost of communication is minimized.

6.2.2. Dynamic behavior

We handle migrations during executions by determining if the communication behavior has changed sufficiently to warrant a migration of threads. We perform a migration only if all of the following conditions are met:

1. The communication phase has changed, as described in Section 4.1.3.
2. The heterogeneity C_H of the new phase is higher than H_{min}.
3. The time between migrations is at least t_{min}.
4. The amount of communication events C_A per second is higher than A_{min}.
5. The calculated mapping is different from the previous mapping.

In this paper, we use $H_{min} = 100$, $t_{min} = 1$ s and $A_{min} = 150$ as a lower bound, below which only marginal improvements from a migration can be expected.

6.3. Data mapping

For each time step, we calculate the correct mapping of memory pages to NUMA nodes, using Eq. (8). We take into account the dynamic thread mapping calculated in the previous subsection. We only migrate the pages at most once every t_{min} to prevent excessive migrations, where $t_{min} = 10$ s.

6.4. Implementation of the mechanism

We implemented the OPT mechanism as a runtime tool which receives as input the detected behavior at each time slice and the command to execute. The tool uses `ptrace` to track the creation of threads and processes and the memory allocations. In this way, the mechanism is independent of the parallel API that the application uses. For the static allocations, we match the address of the allocation against the trace to select the appropriate NUMA node. For the dynamic allocations, the size and the order of the allocations are matched against the trace to force a page allocation on the selected NUMA node, similar to [15]. To migrate the processes and threads during execution, we synchronize the trace using synchronization points, such as barriers and locks, as in [53]. We start the application directly with the first calculated thread mapping.

7. Results of the OPT mechanism

This section presents and discusses the improvements achieved by the OPT mechanism. After a brief overview of the experimental methodology, we present several sets of results: performance and energy consumption improvements using thread and data mapping together, performance improvements using either thread or data mapping alone, and a comparison of OPT with manual data mapping.

7.1. Methodology of the experiments

We perform experiments with the following mechanisms:

Operating system (OS): We use the Linux kernel, version 3.8, with its default scheduler and first touch mapping. It represents the baseline for our experiments. We also evaluated other data mapping policies available in the kernel, the NUMA balance mode[1] and a forced interleave policy (through numactl). As neither policy improved the performance compared to first touch, we only show the results for the first touch case.

We compare the following policies:

Random: We generated random mappings of threads to PUs and memory pages to NUMA nodes. All benchmarks were executed with the same set of random mappings.

Compact: The compact mapping is similar to existing implementations in some OpenMP frameworks[2] and MPI implementations.[3] It performs a round-robin scheduling of threads and processes to cores that takes into account the memory hierarchy of the machine.

OPT: Our OPT mechanism was executed with the configuration shown in the last section.

For the experiments, we use a machine with 4 NUMA nodes that can execute 64 threads in parallel. The details of the system are summarized in Table 2. We use the same benchmarks and input sizes as before, presented in Table 1. For the results, we show the average values of 10 executions (normalized to the OS) as well as a confidence interval for a confidence level of 95% in a Student's t-distribution.

1 http://lwn.net/Articles/486858/.

2 http://software.intel.com/en-us/articles/using-kmp-affinity-to-create-openmp-thread-mapping-to-os-proc-ids.

3 http://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager.

Table 2
Configuration of the evaluation system.

Parameter	Value
Processors	4x Intel Xeon X7550, 2.0 GHz, 8 cores, 2-SMT
Caches/proc	8x 32 KByte L1, 8x 256 KByte L2, 18 MByte L3
Memory	128 GByte DDR3-1333, 4 KByte page size

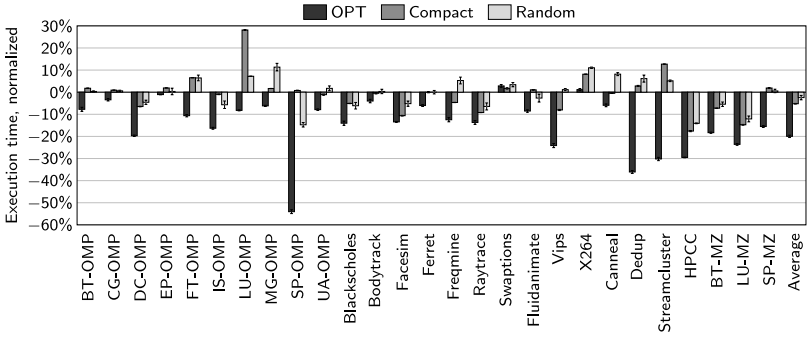


Fig. 11. Execution time, normalized to the OS.

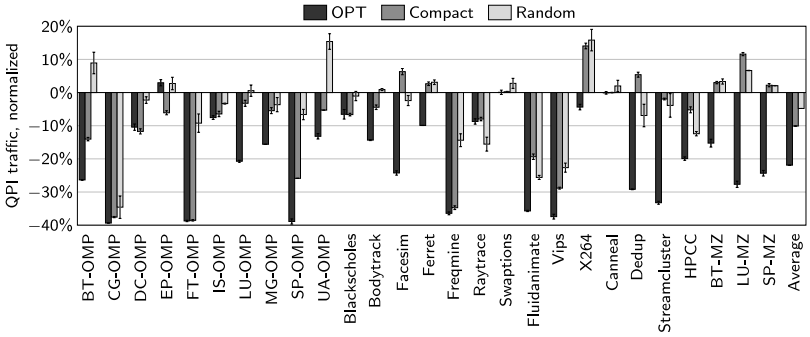


Fig. 12. QPI traffic, normalized to the OS.

7.2. Performance results

Fig. 11 shows the results for the execution time normalized to the baseline, the OS. Fig. 12 shows the normalized results for traffic on the QuickPath Interconnect (QPI), measured with the Intel performance counter monitor (PCM). Applications that have high indicators of heterogeneity, amount of communication and exclusivity levels tend to present higher performance improvements. For instance, SP-OMP, the application that achieved the highest improvements (54.1%), has high indicators for all metrics. On the other hand, EP-OMP presents low indicators, and no performance improvements. However, these indicators do not necessarily correlate with the improvements, since the improvements depend on how much the operating system mapping optimized the communication as well, and also because there are other runtime scheduling effects which are not focus of our work.

Both thread and data mappings influence QPI traffic. A better thread mapping results in a placement of threads that share data on the same NUMA node, thus reducing QPI traffic. A better data mapping reduce QPI traffic by decreasing the amount of remote memory accesses. We can observe that, in most cases, the reduction of QPI traffic is higher than the reduction of execution time. This is expected, since a better mapping directly influences the usage of interconnections, while the execution time is influenced by other factors as well.

On average, execution time was reduced by 19.9%, 5.2% and 1.9% compared to the OS by OPT, Compact and Random, respectively. By analyzing the random and compact mappings, we note that simple mapping strategies are not sufficient to handle a variety of applications, since they have very different characteristics. Also, this shows that our performance improvements are due to a more efficient mapping, and not due to unnecessary migrations introduced by the operating system.

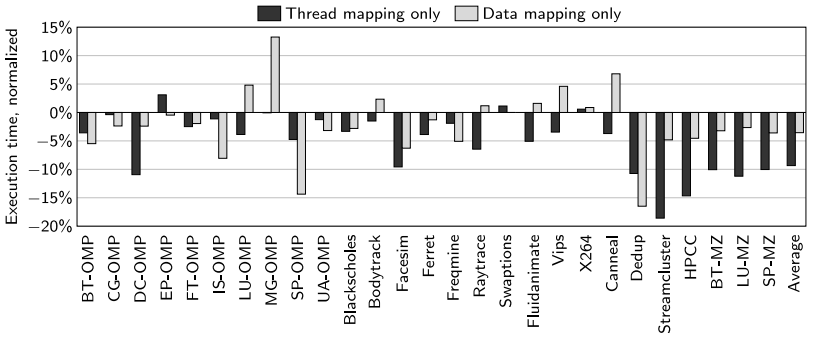


Fig. 13. Application execution time when running only thread or data mapping, normalized to the OS.

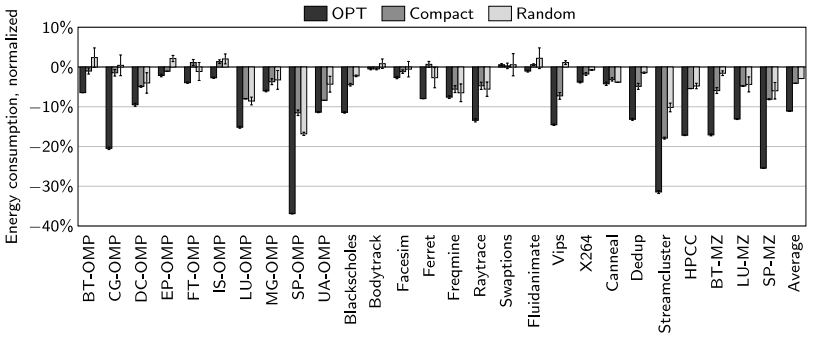


Fig. 14. Energy consumption, normalized to the OS.

7.3. Energy consumption

Improved thread and data mappings can also reduce energy consumption of parallel applications. By reducing application execution time, static energy consumption (leakage) will be reduced proportionally in most cases, since the processing cores will be in a high power-consuming state for a shorter time. A shorter execution time does not directly affect the dynamic energy consumption, since the amount of work performed by the application (in terms of executed number of instructions, for example) will remain about the same. However, reducing the traffic on the interconnections reduces the dynamic energy consumption, leading to a more energy-efficient execution of parallel applications due to improved mappings.

We measure the system energy consumption during the execution of each application with the help of the Baseboard Management Controller (BMC), which exposes the energy consumption of the system through IPMI. The results for the energy consumption are shown in Fig. 14. As expected, applications that presented higher performance gains also showed higher reductions of the energy consumption. SP-OMP improved the most, by 36.9%. On average, OPT reduced energy consumption by 11.1%, while Compact and Random only reduced it by 5.1% and 1.9%, respectively.

7.4. Running thread and data mapping separately

To evaluate the influence of thread and data mapping, we executed OPT only with the thread mapping and data mapping parts. Three configurations were evaluated: thread and data mapping managed by the OS (baseline), OPT thread mapping + OS data mapping, OS thread mapping + OPT data mapping. Fig. 13 shows the execution time for the three configurations, normalized to the baseline. For most benchmarks, the improvements from the thread mapping only are higher than for the data mapping only. On average, data mapping reduced execution time by 3.2%, thread mapping by 8.5%. It is important to note that the improvements from managing thread and data mapping jointly are higher than the sum of improvements when managing the affinities separately. This shows that integrated mechanisms are necessary for optimal results.

For most of the benchmarks, the results correlate with our observations in Sections 4 and 5. For example, FT-OMP was characterized as unsuitable for thread mapping and suitable for data mapping, which correlates to the improvements that were achieved. Not all applications correlate however. Vips was characterized as not suitable for thread mapping, yet shows significant improvements. This can be caused by effects such as contention on functional units in SMT, for example. It is also important to mention that data mapping alone reduces performance compared to the OS in many cases. The reason is that the thread migrations of the OS after the data migrations have a high impact on performance.

Table 3
Execution time of two versions of Ondes3D.

Ondes3D version	OS mapping	OPT mapping
Unmodified code	184.89 s	61.38 s
Manually optimized code	65.55 s	61.35 s

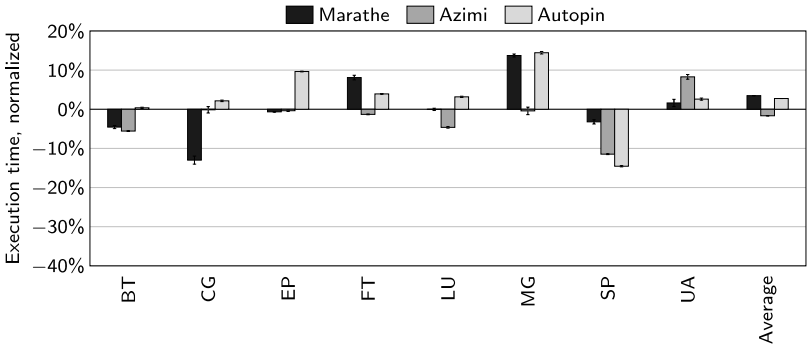


Fig. 15. Comparing application execution time with OPT to related work. Values are normalized to the OS.

7.5. Comparing OPT to manual mapping

Several previous works suggest manual changes to the source code of applications to improve the mapping. In this section, we consider the main numerical kernel extracted from the Ondes3D [54] application in order to compare the OPT mechanism to manual optimization of the code. Ondes3D simulates the propagation of seismic waves using a finite-differences numerical method. On shared-memory architectures, a common way to extract parallelism of this numerical stencil is to exploit the triple nested loops of the three dimensional problem. Classical manual optimizations rely on the default Linux first-touch memory policy. By exploiting the regular memory access pattern of finite-differences applications, we guarantee that the memory accessed by each thread is allocated close to the thread [55].

Table 3 shows the absolute execution time of the kernel, comparing the OS to the OPT mechanism when executing the unmodified and the optimized code. Firstly, we can underline the improvement from the OPT mechanism in comparison with the original version of the code (x3.01). This is coming from the memory-bound characteristics of this kernel with a huge impact of efficient threads and memory mapping in order to maximize the overall performance on NUMA platforms. Secondly, the substantial gain (6.4%) obtained when using the OPT mechanism in comparison with the manually optimized version of the code can also be noticed. This shows that an automatic mechanism can achieve the same or higher improvements than manual code changes.

7.6. Comparison to previous proposals

We compare OPT to three previous techniques that were presented in Section 3: Autopin [34], the Azimi thread mapping [33], and to the Marathe data mapping mechanism [45]. Autopin was executed with 5 mappings: the Compact mapping, as well as 4 random mappings. After a warmup time of 500 ms, every mapping was evaluated for 150 ms. Then, the mapping that resulted in the highest IPC was selected for the rest of the execution. Autopin was directly executed on the real machine. We implemented Azimi and Marathe in Pin, generating mapping information that is fed to a runtime system during the execution of the application in the real machine. For Azimi, the system simulated inside Pin consists of 4 last level caches, each with a size of 16 MByte. For Marathe, we used the same cache configuration as in Azimi, and a long latency load based profile, as described in [45].

Fig. 15 shows the execution time of the related work for the NAS-OMP applications. Values are normalized to the results of the operating system. None of the previous mechanisms is able to consistently improve performance. For three benchmarks, BT-OMP, CG-OMP and SP-OMP, several of the mechanisms are able to achieve improvements that are similar to OPT. However, for the other benchmarks, the mechanisms cannot improve performance and even reduce performance. On average, Marathe and Autopin reduced performance by 3.4% and 3.7%, respectively. Azimi improved performance by 1.6%.

8. Conclusions

Improving the efficiency of memory accesses is an important goal of current and future computer systems. Strategies to increase the locality of accesses need to take into account the communication as well as the page usage of parallel applications. This information can be used to perform an optimized mapping of threads to cores and of memory pages to NUMA nodes.

In this paper, we presented a method to characterize these factors and evaluate which memory access behaviors can be improved by an optimized mapping. This method was employed to analyze a large set of parallel applications based on a variety of parallel APIs. We proposed a mechanism to evaluate the memory access behavior and calculate an optimized thread and data mapping. Using this mechanism, the parallel applications previously analyzed had their performance improved by up to 54% (20% on average) when compared to the mapping done by the OS. We also show that thread and data mapping have to be performed together to achieve optimal results. Additionally, when compared to manual mechanisms that perform the mapping in the source code of applications, our proposal was able to achieve slightly better improvements.

For the future, we plan to characterize more applications, such as the Splash-2, SpecOMP and Rodinia benchmark suites, as well as larger scientific applications, and evaluate their improvements using thread and data mapping.

Acknowledgments

This work was partially funded by CNPq and CAPES.

References

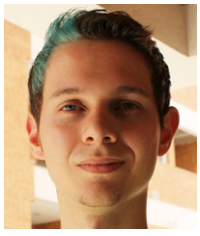
[1] W. Wang, T. Dey, J. Mars, L. Tang, J.W. Davidson, M.L. Soffa, Performance analysis of thread mappings with a holistic view of the hardware resources, in: IEEE International Symposium on Performance Analysis of Systems & Software, ISPASS, 2012.
[2] P.W. Coteus, J.U. Knickerbocker, C.H. Lam, Y.a. Vlasov, Technologies for exascale systems, IBM J. Res. Dev. 55 (5) (2011) 14:1–14:12.
[3] J. Feliu, J. Sahuquillo, S. Petit, J. Duato, Understanding cache hierarchy contention in CMPs to improve job scheduling, in: International Parallel and Distributed Processing Symposium, IPDPS, 2012.
[4] S. Borkar, A.A. Chien, The future of microprocessors, Commun. ACM 54 (5) (2011) 67–77. http://dx.doi.org/10.1145/1941487.
[5] 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.
[6] 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.
[7] D. Buntinas, G. Mercier, W. Gropp, Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2006.
[8] W. Gropp, MPICH2: a new start for MPI implementations, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2002.
[9] B. Goglin, S. Moreaud, KNEM: a generic and scalable kernel-assisted intra-node MPI communication framework, J. Parallel Distrib. Comput. 73 (2) (2013) 176–188.
[10] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, Open MPI: goals, concept, and design of a next generation MPI implementation, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2004.
[11] C.P. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, L.G. Fernandes, Memory affinity for hierarchical shared memory multiprocessors, in: International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD, 2009, pp. 59–66.
[12] A. Faraj, X. Yuan, Communication characteristics in the NAS parallel benchmarks, in: Parallel and Distributed Computing and Systems, PDCS, 2002.
[13] N. Barrow-Williams, C. Fensch, S. Moore, A communication characterisation of splash-2 and parsec, in: IEEE International Symposium on Workload Characterization, IISWC, 2009.
[14] R. Natarajan, M. Chaudhuri, Characterizing multi-threaded applications for designing sharing-aware last-level cache replacement policies, in: 2013 IEEE International Symposium on Workload Characterization, IISWC, 2013.
[15] J. Marathe, V. Thakkar, F. Mueller, Feedback-directed page placement for ccNUMA via hardware-generated memory traces, J. Parallel Distrib. Comput. 70 (12) (2010) 1204–1219.
[16] Z. Chishti, M.D. Powell, T.N. Vijaykumar, Optimizing replication, communication, and capacity allocation in CMPs, ACM SIGARCH Comput. Archit. News 33 (2) (2005) 357–368.
[17] H. Jin, M. Frumkin, J. Yan, The OpenMP implementation of NAS parallel benchmarks and its performance, 1999.
[18] C. Bienia, S. Kumar, J.P. Singh, K. Li, The PARSEC benchmark suite: characterization and architectural implications, in: Parallel Architectures and Compilation Techniques, PACT, 2008, pp. 72–81.
[19] P. Luszczek, J.J. Dongarra, D. Koester, R. Rabenseifer, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, D. Takahashi, J. Jack, R. Rabenseifner, Introduction to the HPC challenge benchmark suite, 2005.
[20] R.F. Van der Wijngaart, H. Jin, NAS Parallel Benchmarks, Multi-Zone Versions, Tech. Rep., 2003.
[21] I. Lee, Characterizing communication patterns of NAS-MPI benchmark programs, in: IEEE Southeastcon, 2009.
[22] J. Kim, D. Lilja, Characterization of communication patterns in message-passing parallel scientific application programs, in: International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications, 1998.
[23] S. Chodnekar, V. Srinivasan, A.S. Vaidya, A. Sivasubramaniam, C.R. Das, Towards a communication characterization methodology for parallel applications, in: International Symposium on High Performance Computer Architecture, HPCA, 1997.
[24] J.P. Singh, E. Rothberg, A. Gupta, Modeling communication in parallel algorithms: a fruitful interaction between theory and systems?, in: ACM Symposium on Parallel Algorithms and Architectures, 1994.
[25] Z. Majo, T.R. Gross, (Mis)understanding the NUMA memory system performance of multithreaded workloads, in: IEEE International Symposium on Workload Characterization, IISWC, 2013, pp. 11–22.
[26] 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.
[27] G. Mercier, E. Jeannot, Improving MPI applications performance on multicore clusters with rank reordering, in: European MPI Users' Group Conference on Recent Advances in the Message Passing Interface, EuroMPI, 2011.
[28] J. Hursey, J. Squyres, T. Dontje, Locality-aware parallel process mapping for multi-core HPC systems, in: IEEE International Conference on Cluster Computing, CLUSTER, 2011.
[29] G. Mercier, J. Clet-Ortega, Towards an efficient process placement policy for mpi applications in multicore environments, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface.
[30] B. Brandfass, T. Alrutz, T. Gerhold, Rank reordering for MPI communication optimization, Comput. & Fluids (2012) 372–380.
[31] S. Ito, K. Goto, K. Ono, Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments, Comput. & Fluids 80 (2013) 88–93. http://dx.doi.org/10.1016/j.compfluid.2012.04.024.
[32] E.H.M. Cruz, M.A.Z. Alves, A. Carissimi, P.O.A. Navaux, C.P. Ribeiro, J.-F. Méhaut, Using memory access traces to map threads and data on hierarchical multi-core platforms, in: IEEE International Symposium on Parallel and Distributed Processing Workshops and Ph.D. Forum, 2011.

[33] 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 (2) (2009) 56–65.
[34] T. Klug, M. Ott, J. Weidendorfer, C. Trinitis, autopin — automated optimization of thread-to-core pinning on multicore systems, High Perform. Embedded Archit. Compilers 3 (4) (2008) 219–235.
[35] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.
[36] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, U.V. Catalyurek, Parallel hypergraph partitioning for scientific computing, in: IEEE International Parallel & Distributed Processing Symposium, IPDPS, 2006.
[37] F. Pellegrini, Static mapping by dual recursive bipartitioning of process and architecture graphs, in: Scalable High-Performance Computing Conference, SHPCC, 1994, pp. 486–493.
[38] J.L. Träff, Implementing the MPI process topology mechanism, in: ACM/IEEE conference on Supercomputing, SC, 2002.
[39] E. Jeannot, G. Mercier, Near-optimal placement of MPI processes on hierarchical NUMA architectures, in: Euro-Par Parallel Processing, 2010.
[40] H. Löf, S. Holmgren, Affinity-on-next-touch: increasing the performance of an industrial PDE solver on a cc-NUMA system, in: International Conference on Supercomputing, SC, 2005, pp. 387–392.
[41] B. Goglin, N. Furmento, Enabling high-performance memory migration for multithreaded applications on linux, in: IEEE International Symposium on Parallel & Distributed Processing, IPDPS, 2009.
[42] M. Awasthi, D.W. Nellans, K. Sudan, R. Balasubramonian, A. Davis, Handling the problems and opportunities posed by multiple on-chip memory controllers, in: Parallel Architectures and Compilation Techniques, PACT, 2010, pp. 319–330.
[43] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, M. Roth, Traffic management: a holistic approach to memory placement on NUMA systems, in: Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2013, pp. 381–393.
[44] R. Lachaize, B. Lepers, V. Quéma, MemProf: a memory profiler for NUMA multicore systems, in: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, 2012.
[45] J. Marathe, F. Mueller, Hardware profile-guided automatic page placement for ccNUMA systems, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, 2006, pp. 90–99.
[46] M.M. Tikir, J.K. Hollingsworth, Hardware monitors for dynamic page migration, J. Parallel Distrib. Comput. 68 (9) (2008) 1186–1200.
[47] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, V. Quema, Large pages may be harmful on NUMA systems, in: Usenix Atc, 2014, pp. 231–242.
[48] S. Blagodurov, S. Zhuravlev, M. Dashti, A. Fedorova, A case for NUMA-aware contention management on multicore systems, in: USENIX Annual Technical Conference, 2010, pp. 557–571.
[49] Z. Majo, T.R. Gross, Matching memory access patterns and data placement for NUMA systems, in: International Symposium on Code Generation and Optimization, CGO, 2012.
[50] C. Luk, R. Cohn, R. Muth, 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.
[51] S. Bokhari, On the mapping problem, IEEE Trans. Comput. C 30 (3) (1981) 207–214.
[52] T. Hoefler, M. Snir, Generic topology mapping strategies for large-scale parallel architectures, in: International Conference on Supercomputing, ICS, 2011.
[53] E.H.M. Cruz, M. Diener, M.A.Z. Alves, P.O.A. Navaux, Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols, J. Parallel Distrib. Comput. 74 (3) (2014) 2215–2228. http://dx.doi.org/10.1016/j.jpdc.2013.11.006.
[54] F. Dupros, H. Aochi, A. Ducellier, D. Komatitsch, J. Roman, Exploiting intensive multithreading for the efficient simulation of 3D seismic wave propagation, in: International Conference on Computational Science and Engineering, CSE, 2008.
[55] F. Dupros, C. Pousa, A. Carissimi, J.-F. Méhaut, Parallel simulations of seismic wave propagation on NUMA architectures, in: Parallel Computing: From Multicores and GPU's to Petascale, ParCo 2009, 1–4 September, Vol. 10, Lyon, France, 2010, pp. 67–74.

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), Porto Alegre, 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.

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



Laércio Lima Pilla holds an associate professor position in the Federal University of Santa Catarina, Brazil. He obtained his Ph.D. degree in Computer Science in 2014 in a joint doctorate between the Federal University of Rio Grande do Sul, Brazil, and the University of Grenoble, France. His research interests include topology-aware scheduling, and software-based fault-tolerance on GPUs.

Fabrice Dupros has over 10 years of experience in parallel computing applied to various topics in Earth Sciences. He worked on numerical algorithms for air quality modelling at the Center for Advanced Computing and Data Systems of the University of Houston. He joined BRGM (France's leading public institution in Earth Science applications) in 2003 working on high performance computing applied to geosciences. He is currently leading the research program on scientific computing and 3D visualization. Fabrice Dupros graduated in Applied Mathematics and Engineering from the University of Lyon and from the University of Houston. He received a Ph.D. in Computer Science from the University of Bordeaux.

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.