Affinity-Based Thread and Data Mapping in Shared Memory Systems

MATTHIAS DIENER and EDUARDO H. M. CRUZ, Informatics Institute,
Federal University of Rio Grande do Sul
MARCO A. Z. ALVES, Department of Informatics, Federal University of Paraná
PHILIPPE O. A. NAVAUX, Informatics Institute, Federal University of Rio Grande do Sul
ISRAEL KOREN, Department of Electrical & Computer Engineering,
University of Massachusetts at Amherst

Shared memory architectures have recently experienced a large increase in thread-level parallelism, leading to complex memory hierarchies with multiple cache memory levels and memory controllers. These new designs created a Non-Uniform Memory Access (NUMA) behavior, where the performance and energy consumption of memory accesses depend on the place where the data is located in the memory hierarchy. Accesses to local caches or memory controllers are generally more efficient than accesses to remote ones. A common way to improve the locality and balance of memory accesses is to determine the mapping of threads to cores and data to memory controllers based on the affinity between threads and data. Such mapping techniques can operate at different hardware and software levels, which impacts their complexity, applicability, and the resulting performance and energy consumption gains. In this article, we introduce a taxonomy to classify different mapping mechanisms and provide a comprehensive overview of existing solutions.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → **Main memory**; **Scheduling**

Additional Key Words and Phrases: Survey, shared memory, thread mapping, data mapping, NUMA, cache memories, communication

ACM Reference Format:
Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, and Israel Koren. 2016. Affinity-based thread and data mapping in shared memory systems. ACM Comput. Surv. 49, 4, Article 64 (December 2016), 38 pages.
DOI: http://dx.doi.org/10.1145/3006385

1. INTRODUCTION

Since reaching the practical limits of Instruction-Level Parallelism (ILP) [Brooks et al. 2000], processor manufacturers have been focusing on increasing the Thread-Level Parallelism (TLP) on modern shared memory architectures to continue improving system performance. Such multi-core, multi-threaded architectures exert a high pressure on the memory subsystem as they have to supply large quantities of data to the many functional units.

This work is supported by CAPES, under grant PVE 117/2013, and MCTI/RNP Brazil under the HPC4E project, grant 689772.
Authors' addresses: M. Diener and E. H. M. Cruz, P. O. A. Navaux, Informatics Institute, Federal University of Rio Grande do Sul, Av. Bento Gonçalves, 9500, 91501-970, Porto Alegre, RS, Brazil; emails: {mdiener, ehmcruz, navaux}@inf.ufrgs.br; M. A. Z. Alves, Department of Informatics, Federal University of Paraná, Rua Cel. Francisco Heráclito dos Santos, 100, 81531-990, Curitiba, PR, Brazil; email: mazalves@inf.ufpr.br; I. Koren, 309E Knowles Engineering Building, Department of Electrical & Computer Engineering, University of Massachusetts, Amherst, MA 01003; email: koren@ecs.umass.edu.

DOI: http://dx.doi.org/10.1145/3006385

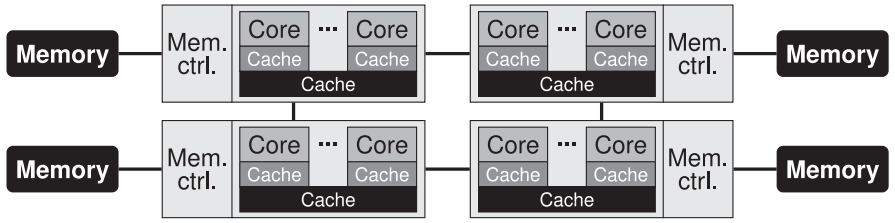


Fig. 1. NUMA architecture with four NUMA nodes. Each node consists of a memory controller with several cores attached to it.

Unfortunately, the progress in memory technologies has not kept pace with the increase in data demand. Therefore, manufacturers have adopted deep and complex memory hierarchies in order to hide memory access latencies from processors. In modern chips, these hierarchies consist of several levels of private and shared cache memories, as well as multiple memory controllers that result in a Non-Uniform Memory Access (NUMA) behavior [Awasthi et al. 2010]. In such memory hierarchies, data movements have a very high impact on the performance and energy consumption of parallel machines [Shalf et al. 2010; Dally 2010].

Figure 1 shows an example memory hierarchy of a modern shared memory system. The system contains four memory controllers, each forming a *NUMA node*, which can access a part of the system memory. Several processing cores are attached to each NUMA node. Furthermore, each core has a private first-level cache and shares a second-level cache with other cores. In this system, a memory access performed by a core can be serviced by a *local* cache or memory controller, or a *remote* one.

In such systems, the latency and energy consumption of memory accesses depend highly on which core performs the request and which cache or memory controller services it. In most cases, local accesses are more efficient than remote ones. It can also become important to prevent overloading some of the caches or memory controllers [Diener et al. 2015; Blagodurov et al. 2011] to reduce contention and increase resource usage fairness.

In addition to the differences in the memory access performance imposed by the hardware, parallel applications also show considerable differences in the way they access memory, leading to the concept of *affinity*. Two types of affinity exist. Threads that access the same data, referred to as *data sharing* [Tam et al. 2007] or *communication* [Barrow-Williams et al. 2009] in the literature, have an affinity between them. Moreover, there is an affinity between threads and the memory pages that they access.

In this context, there are two approaches to affinity-based mapping, namely, *thread mapping* and *data mapping*, and both exploit the differences in memory access behavior and performance. The goal of thread mapping is to assign threads to cores in such a way that memory accesses to data shared between threads are optimized, thus improving the usage of caches and interconnections. Data mapping aims to optimize the usage of memory controllers by improving the assignment of memory pages to controllers. Both types of mapping can focus on improving the *locality* or *balance* of memory accesses [Blagodurov et al. 2011; Dashti et al. 2013]. Thread and data mapping can affect each other, and applying them jointly can result in gains that are higher than when applying each type of mapping separately [Diener et al. 2015b].

The rest of this article is organized as follows. The next section discusses the main concepts behind thread and data mapping. Section 3 presents our taxonomy of mapping mechanisms. An overview and comparison of thread mapping mechanisms is given in Section 4, while Section 5 discusses data mapping mechanisms. In Section 6, we compare the gains of different types of mapping techniques. Section 7 summarizes our

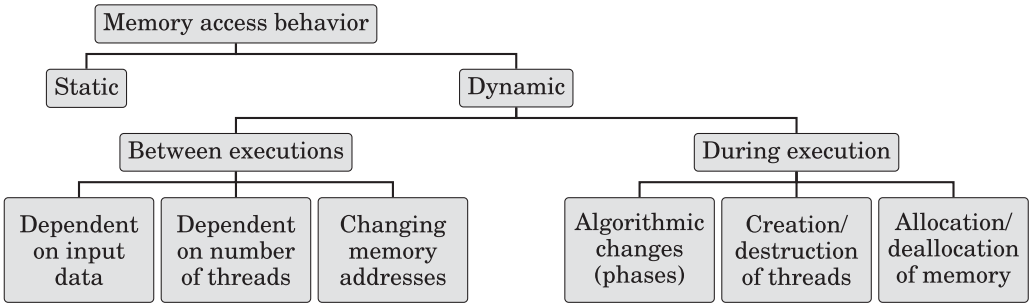


Fig. 2. Classification of static and dynamic memory access behavior of parallel applications.

conclusions and presents future research perspectives. Appendices A and B contain detailed descriptions of the thread and data mapping mechanisms.

2. CONCEPTS OF THREAD AND DATA MAPPING

This section introduces the background for thread and data mapping. We give a brief overview of memory access behavior types of parallel applications, as this determines the mechanisms that can be used. Then, we discuss the main concepts of thread and data mapping and present general policy types and goals for both kinds of mapping. We conclude this section with a summary of the benefits of affinity-based mapping.

2.1. Static and Dynamic Memory Access Behavior of Parallel Applications

Parallel applications may change their memory access behavior for various reasons, and we refer to this as a *dynamic* memory access behavior. Whether an application has a dynamic memory access behavior has an impact on the gains that can be achieved, but it also determines which types of mechanisms are suitable for this application.

At a high abstraction level, we classify the general memory access behavior of an application as *static* or *dynamic*, as shown in Figure 2. We further divide dynamic access behavior into two types. The first type is characterized by dynamic behavior *between separate executions*, where the behavior depends on specified parameters of the application, such as the input data or the number of threads that will be created. Moreover, memory addresses can also change between executions, due to security techniques such as Address Space Layout Randomization (ASLR) [Spengler 2003] or a different order of dynamic memory allocations with functions such as `malloc()` and `new()`.

The second type of dynamic behavior occurs *during the execution* of the application, due to the way a parallel algorithm is implemented (such as using work-stealing [Blumofe and Leiserson 1994] or pipeline programming models), or due to the creation and destruction of threads or allocation/deallocation of memory. If none of these cases occur, we classify the application's behavior as *static*, that is, it remains the same during and between executions.

The aforementioned classification of memory access behavior is important when identifying the types of mapping policies that can be performed. If the memory access behavior is static, no runtime migrations of threads or memory pages need to be performed. Moreover, the behavior can be classified and analyzed through communication or memory access traces, and only the behavior throughout the complete execution needs to be considered.

For applications that show a dynamic behavior only during the execution but do not change their behavior from one execution to the next, traces can be used to analyze their behavior. However, the changing behavior during execution must be considered when

performing mapping, which can require migrations during the execution to achieve optimal gains.

For applications whose behavior changes between executions, trace-based mechanisms require the generation of a new trace for each set of input parameters, as the currently detected behavior would not be valid for future executions of the application. Care must also be taken to limit the impact of changes to memory addresses. Online mechanisms attempt to directly support all types of dynamic behavior.

2.2. Thread Mapping Concepts

Thread mapping[1] is defined as the assignment of threads to execution cores according to a policy that can take various objectives into account [Boillat and Kropf 1990].

2.2.1. Suitable Architectures. Thread mapping in parallel shared memory architectures has become important with the introduction of multicore (Chip Multi-Processing, CMP) and multithreaded (Simultaneous Multi-Threading, SMT) processors. Such systems contain deep cache hierarchies that are shared between different sets of cores. In these architectures, deciding where to execute each thread of a parallel application has a significant impact on how efficiently shared caches and interconnections are used. Earlier systems based on multiple single-core processors (Simultaneous Multi-Processing, SMP) usually featured only simple, private caches and bus-based interconnections, where the mapping of threads to processors did not influence the cache or interconnection usage.

2.2.2. Baseline Thread to Core Assignment. In most shared memory systems, the thread-to-core assignment is handled by default by the operating system scheduler. Traditional schedulers, such as the Completely Fair Scheduler (CFS) [Wong et al. 2008] currently used by default in Linux, focus on load balancing and fairness [Li et al. 2009; Das et al. 2013] and have no information about the memory access behavior of the application. Furthermore, the application itself has usually no direct influence on the mapping. For these reasons, most thread mapping techniques explicitly bind threads to cores, thus overriding the OS scheduler. A comprehensive overview of scheduling techniques is provided by Zhuravlev et al. [2012].

2.2.3. Policy Goals. Affinity-based thread mapping may have two goals, improving the *locality* or *balance* of communication. Locality of communication is usually improved by placing threads that communicate extensively close to each other in the hardware hierarchy in order to make use of shared caches and faster interconnections. A balance policy aims to distribute the amount of communication handled by caches and interconnections fairly. Such a policy can be especially beneficial if the parallel application itself has an imbalance in the communication behavior, that is, if some threads perform more communication than others [Diener et al. 2015], or when not all available cores are used [Velkoski et al. 2013]. In such situations, improving the balance of communication can increase the overall performance [Diener et al. 2015].

2.2.4. Policy Types. Affinity-based thread mapping can be performed in two ways. In an *allocation* policy, each thread is assigned to a particular core, via operating system functions or runtime environment options, and remains on that core until the end of execution. Such a policy does not impose a runtime overhead on the application but cannot react if the behavior changes during execution. In a *migration* policy, threads are migrated between cores during runtime according to the detected communication

[1]*Thread mapping* is also referred to as *task mapping* or *process mapping* in the literature. Since we focus on shared memory systems, we will use the name thread mapping in this article, and refer to task mapping only for runtime environments that explicitly use tasks for parallelization, such as MPI.

behavior. These policies cause a runtime overhead during execution, mostly due to an increase in the number of cache misses [Constantinou et al. 2005] and TLB shootdowns [Villavieja et al. 2011] due to migrations, apart from the overhead of calling the migration functions themselves. Still, migration policies are able to handle dynamic application behavior.

2.2.5. Determining the Thread Mapping. Two pieces of information are necessary to determine the thread mapping. First, the way in which threads access shared data must be known. This behavior is usually represented as a *communication matrix*, in which each element contains the amount of communication between the threads whose IDs are given by the column and row indices. Information about the communication behavior can be generated with the techniques presented in Section 4. Second, the mapping mechanism needs information about the hardware hierarchy, such as the processors, cores, and cache levels. This hierarchy can be discovered with tools such as `hwloc` [Broquedis et al. 2010b].

A thread mapping algorithm uses the communication behavior and hardware hierarchy to determine an improved thread mapping, which is then used to assign threads to cores or migrate them between cores. Such a thread mapping is generally a *global* operation, that is, the mapping is determined for all threads of an application at the same time. As long as the number of threads is sufficiently low, most currently available algorithms can determine an improved mapping in a short time [Jeannot et al. 2014]

Many such algorithms have been proposed in the literature. Since calculating an optimal thread mapping is an NP-complete problem [Radojković et al. 2013], most algorithms use approximations to reduce the complexity to polynomial [Jeannot et al. 2014]. Most algorithms are based on graph representations of the communication behavior and the hardware hierarchy and use graph partitioning [Karypis and Kumar 1998] or graph matching [Cruz et al. 2012] techniques to determine the improved thread mapping. Examples of such algorithms include Scotch [Pellegrini 1994, 2010], METIS [Karypis and Kumar 1996, 1998], and the graph mapping algorithm that is part of the Zoltan toolkit [Devine et al. 2006].

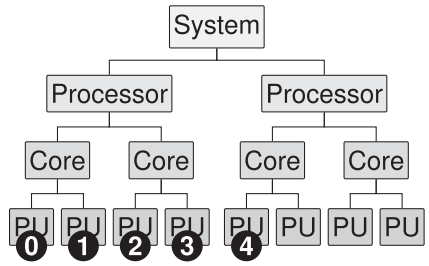
In most shared memory architectures, the hardware hierarchy can be represented as a tree, which can lead to more effective algorithms [Jeannot and Mercier 2010; Träff 2002]. Examples of such algorithms include Treematch [Jeannot and Mercier 2010; Jeannot et al. 2014] and EagerMap [Cruz et al. 2015b]. A thread mapping mechanism uses the output from the mapping algorithm to allocate threads to cores or migrate them. Several studies compare such algorithms in terms of quality, execution time, and stability [Jeannot and Mercier 2010; Jeannot et al. 2014; Glantz et al. 2015; Cruz et al. 2015b].

2.2.6. Thread Mapping Example. Figure 3 presents an example illustrating the operation of various thread mapping policy types and goals. For this example, assume that we execute a parallel application consisting of five threads, 0–4, with the communication behavior shown in Figure 3(a). The matrix entries contain the amount of communication between pairs of threads, expressed as the number of bytes or messages that were exchanged. In the example, two pairs of threads, (0,4) and (1,3), perform equal amounts of communication, while thread 2 does not communicate at all.

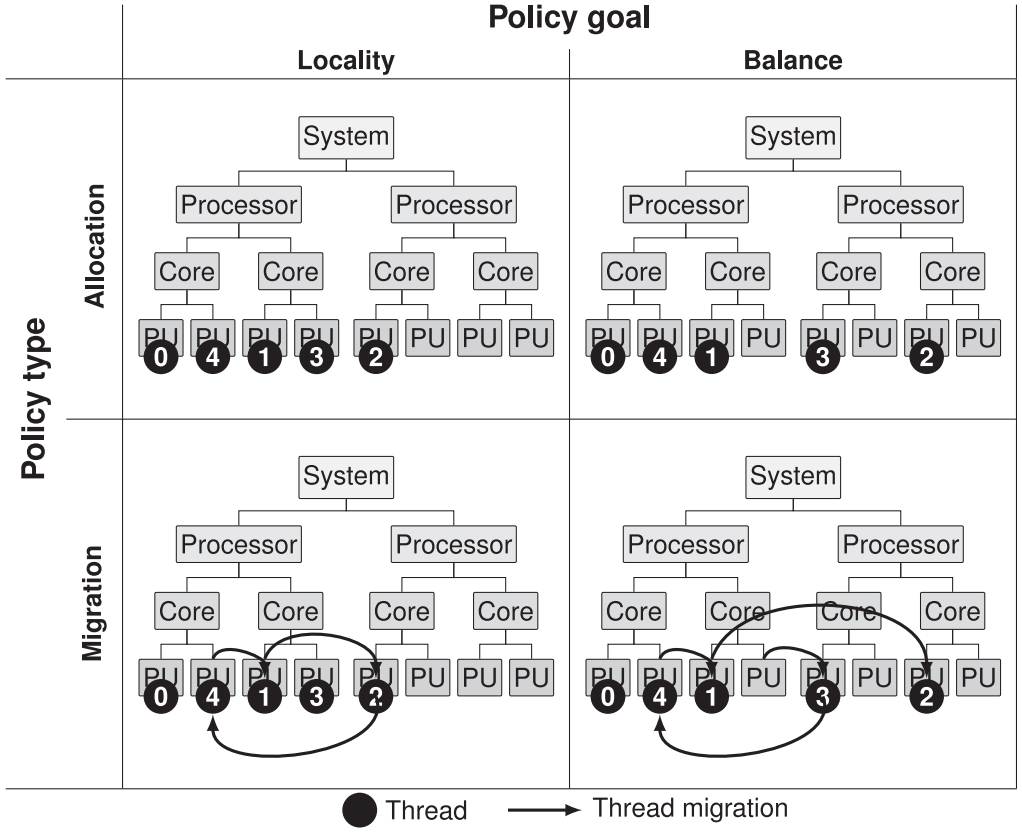
Assume further that we want to execute this application on the hardware architecture shown in Figure 3(b), consisting of two processors with two cores each, where every core can execute two threads at the same time via SMT. The processors have a cache that is shared among the cores, while each core also has a private cache. For the migration policies, assume that the threads are initially assigned to cores using a

4	100	0	0	0	0
3	0	100	0	0	0
2	0	0	0	0	0
1	0	0	0	100	0
0	0	0	0	0	100
threads	0	1	2	3	4

(a) Input: Communication matrix for a parallel application consisting of five threads 0–4. Each entry contains the amount of communication between the corresponding pair of threads.



(b) Input: Hardware hierarchy and round-robin allocation of threads to processing units (PUs). Assume that each processor has a cache level shared among all cores and each core has another private cache level.



(c) Output: Affinity-based thread mapping with different policy types and goals.

Fig. 3. Overview of thread mapping policy types and goals.

round-robin policy, as shown in Figure 3(b), where the first thread is mapped to the first Processing Unit (PU), the second thread to the second PU, and so on.

Figure 3(c) presents how different types of mapping policies could handle such a scenario. An allocation-based policy that focuses on improving locality results in a mapping where most communication is handled by a single processor, achieving a high locality but also resulting in a high imbalance. A balancing policy can distribute threads more fairly, but results in a lower overall locality.

The migration-based policies yield the same final thread mapping, but require several thread migrations, three for a locality policy and four for the balancing policy in this example. In both cases, two threads are migrated across processors, resulting in a higher overhead than intra-processor migrations due to cache misses on more cache levels [Constantinou et al. 2005].

2.2.7. Related Techniques. Several techniques that focus on improving communication are out of the scope of this article and will not be discussed in detail. Many of these techniques focus on mapping in distributed memory environments, such as clusters, in order to improve network interconnection performance. Examples of mapping techniques for cluster environments include proposals by Bhatele [2010], Karlsson et al. [2012], and Soryani et al. [2013]. We will not cover such schemes because our goal is to discuss mapping in shared memory architectures.

A related type of techniques that affects communication is based on *communication avoidance* [Ballard et al. 2014; You et al. 2015]. These techniques focus on decreasing the impact of communication by reducing the amount of data that needs to be communicated among threads in order to overcome the considerable performance and energy consumption impact of communication [Shalf et al. 2010]. They focus on improving parallel algorithms and can be seen as orthogonal to thread mapping techniques, since the impact of a reduced amount of communication can be often further lowered by using a better thread mapping. A comprehensive overview of communication avoidance algorithms is given in Ballard et al. [2014].

2.3. Data Mapping Concepts

Data mapping is defined as the assignment of memory pages to memory controllers (or NUMA nodes) in systems with multiple DRAM memory controllers [Wholey 1991]. For optimal efficiency, data mapping should be combined with thread mapping, in order to reduce thread migrations between NUMA nodes [Corbalan et al. 2003] and to place threads that access the same data on the same node [Brecht 1993].

An improved data mapping policy analyzes how memory pages are accessed by threads and NUMA nodes, and maps a page to the most suitable node. In contrast to thread mapping, the high number of memory pages (up to billions in current systems) often necessitates the use of *local* decisions, that is, data mapping decisions that consider only a single page or a small group of pages at the same time.

2.3.1. Underlying Architectures. NUMA systems have undergone significant changes in the past decades. Early NUMA research prototypes, such as IBM's RP3 [Pfister et al. 1985], Stanford's DASH [Lenoski et al. 1992], and Kendall Square Research's KSR-1 [Frank et al. 1993], failed to gain traction in the computer market but generated considerable research interest. More recent research architectures such as Sun's Wildfire system [Hagersten and Koster 1999] showed that NUMA can improve the scalability of parallel machines compared to traditional Uniform Memory Access (UMA) systems [Noordergraaf and van der Pas 1999].

NUMA machines and data mapping have received renewed attention when chip makers started to integrate memory controllers in their computing systems designs and introduced point-to-point interconnections, such as Intel's QuickPath Interconnect

(QPI) [Ziakas et al. 2010] and AMD's HyperTransport [Conway 2007], between memory controllers and the main memory as well as between processors. Systems with more than one such processor, therefore, have a NUMA behavior. Modern NUMA architectures can include multiple memory controllers on the same chip, leading to a NUMA behavior even on a single processor. Data mapping in shared programming models that target execution in cluster systems, such as Partitioned Global Address Space (PGAS) [Anbar et al. 2016], are out of the scope of this article.

2.3.2. Baseline Memory Page to Node Mapping. Assigning memory pages to NUMA nodes is an important problem in NUMA architectures. It impacts the execution of an application, in the absence of a dedicated migration mechanism, as migrations are expensive due to the requirement to copy potentially large amounts of data between nodes. For this reason, baseline data mapping mechanisms are usually static, in contrast to the standard thread mapping policies.

The most basic policy is based on *home-node allocation*, where memory pages are placed on the NUMA node where a memory allocation is performed (such as via a `malloc()` function), before the first access to the data. That is, whenever a thread allocates memory, this memory is allocated on the node the thread is executing on. Such a policy is only of theoretical interest and not in use nowadays.

A standard policy still in use today is the *interleave* (or *interleaved*) policy, which maps pages according to their address. Usually, the node is determined by the least significant bits of the virtual page address. In this way, contiguous page ranges are mapped to different nodes, leading to similar numbers of pages on each node and a high memory access balance in case pages are accessed uniformly. However, the locality of memory accesses is usually low. Interleave is available on Linux through the `numactl` tool [Kleen 2004].

As a locality improvement to the interleave policy, the *first-touch* policy was developed [Singh et al. 1993; Marchetti et al. 1995]. In this policy, each page is mapped to the NUMA node from which the first memory access to the page is performed, usually during the execution of the page fault handler. The idea behind this policy is that the thread that performs the first access to the page is likely to perform the majority of subsequent accesses to the page, leading to a high memory access locality. However, with this policy the application has a direct influence on the data mapping, and consequently, the application developer should take this policy into account when writing the application. First-touch is the default policy for many current operating systems, including Linux [Lankes et al. 2010], Solaris [Oracle 2010], and Windows [van der Pas 2009], and is generally used as the baseline for performance improvements.

2.3.3. Policy Goals. Affinity-based data mapping, like thread mapping, can focus on two goals, improving the *locality* or *balance* of memory accesses to NUMA nodes. Locality of memory accesses is usually improved by placing memory pages on NUMA nodes that access these pages often, in order to avoid accesses from remote nodes [Corbet 2012b]. Such a policy can, however, result in an imbalance in certain circumstances with some memory controllers performing more accesses than others. Furthermore, an application might allocate data non-uniformly on the nodes, causing an imbalance. A balance policy aims to equalize the number of accesses that each controller handles [Blagodurov et al. 2011] in order not to overload some controllers. For many parallel applications, improving locality also improves balance [Diener et al. 2015].

2.3.4. Policy Types. Affinity-based data mapping can be performed in three ways, *allocation*, *migration*, and *replication*. Allocation and migration are analogous to the respective thread mapping strategies. All three policy types can be applied to memory

pages or to (parts of) data structures. We will refer to memory pages in the explanation below.

In an allocation policy, each page is assigned to a NUMA node and remains on that node until execution is finished. Such a policy does not impose a runtime overhead on the application but cannot react if the behavior changes during execution.

In a migration policy, pages are migrated between NUMA nodes during runtime according to the detected memory access behavior. Such a policy causes a runtime overhead during execution, mostly due to the copying of data between nodes, apart from the overhead for calling the migration functions themselves. However, migration policies are able to respond to changes in the application's behavior during execution.

In a replication policy, pages that are accessed from multiple NUMA nodes are replicated on those nodes. Replication has the advantage of improving both locality and balance of memory accesses simultaneously. However, it suffers from two drawbacks. First, since pages are duplicated, less memory is available to the application. Second, write memory accesses usually require the propagation of the changes to the other copies of the same page in order to maintain coherence, a situation that is similar to cache coherence protocols. For this reason, replication is more beneficial for data that is seldom modified.

2.3.5. Determining the Data Mapping. Deciding on an improved data mapping is usually more straightforward than determining a thread mapping, with simpler data structures and algorithms. Many data mapping algorithms can calculate a mapping for a single page with a constant complexity, with a linear complexity for all pages, while many thread mapping algorithms have a polynomial complexity, as discussed in Section 2.2.5. There is also no need for graph representations of the behavior, as in many thread mapping algorithms. The memory access behavior can be represented in the form of a page usage pattern that describes the number of memory access per page or per data structure from each NUMA node. An example of such a pattern is shown in Figure 4(a). A data mapping algorithm can analyze this pattern to determine the most suitable NUMA node for each page or data structure.

2.3.6. Data Mapping Example. The example shown in Figure 4 illustrates the different policy types and goals of data mapping. Assume that we execute a parallel application with four threads, accessing in total four pages with the page usage pattern shown in Figure 4(a). Out of the four pages, three are accessed exclusively by a single thread. Page 3 is shared uniformly by two threads.

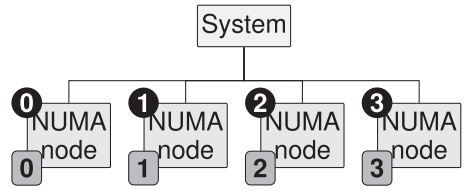
Figure 4(b) shows the hardware architecture that this application is executing on. It consists of four NUMA nodes. Each node is running a single thread mapped with a round-robin thread mapping, as previously shown in Section 2.2.6. This thread mapping is maintained for all data mapping policies. For the data mapping policies that are based on migrations and replications, we further assume that all pages are initially mapped to a NUMA node using a round-robin mapping.

The behavior of the different types of data mapping policies is shown in Figure 4(c). The allocation-based locality policy places two pages on the first node and no pages on the last node, creating an imbalance but resulting in a high memory access locality. The balancing policy maps the same number of pages to all nodes, resulting in a better balance at the expense of a lower locality. The migration-based policies result in the same mapping as the allocation-based policies but require three and two migrations for locality and balance, respectively.

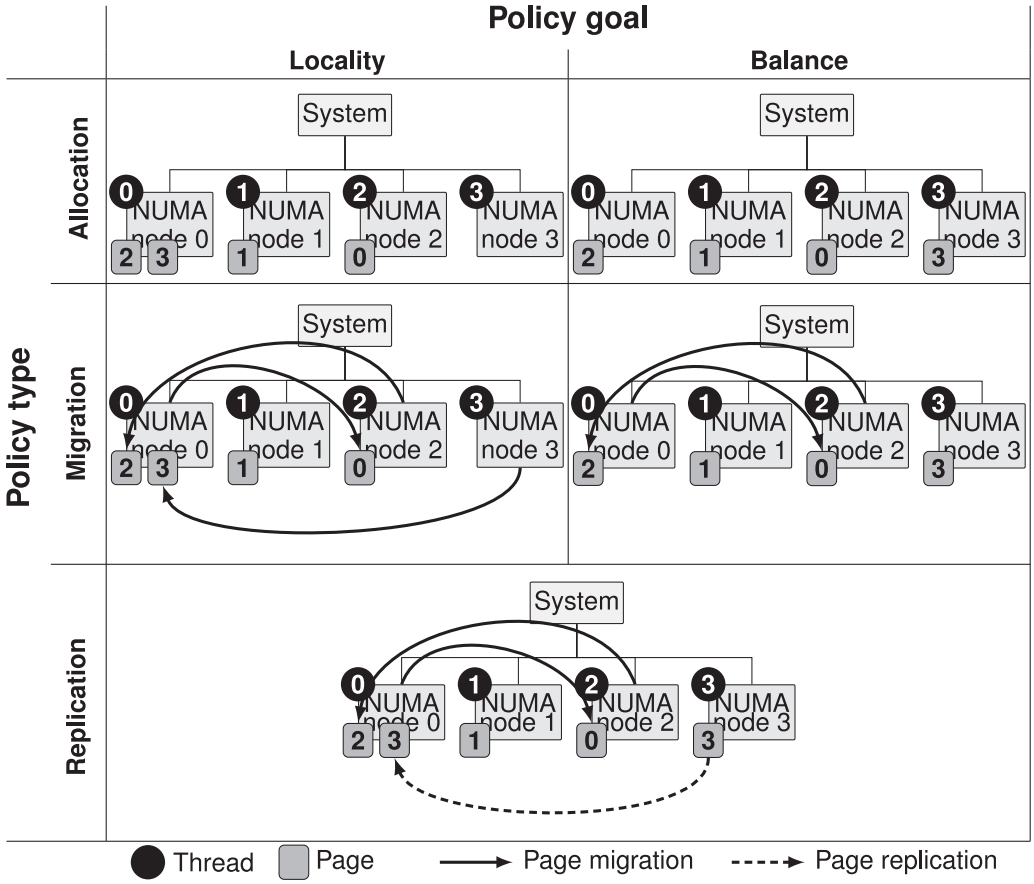
The replication-based policy improves both locality and balance and only one behavior is shown for this type of policy [Brorsson 1989; LaRowe et al. 1992; Bull and Johnson 2002; Dashti et al. 2013]. Since page 3 is accessed by multiple NUMA nodes, it is a candidate for replication. In the example, the policy replicates it from the last node

Page	Memory accesses per thread			
	0	1	2	3
0	0	0	100	0
1	0	100	0	0
2	100	0	0	0
3	100	0	0	50

(a) Input: Page access pattern of a parallel application that consists of four threads and accesses four pages.



(b) Input: NUMA architecture with four nodes and an initial round-robin mapping of threads and data to nodes.



(c) Output: Affinity-based data mapping with different policy types and goals.

Fig. 4. Overview of data mapping policy goals and types.

to the first one, as shown in the figure. All other pages are accessed only by a single node and are, therefore, only migrated to the corresponding node without replication.

The resulting data mapping has a higher locality than that achieved by the other two policy types, as memory accesses from thread 3 are also performed to a local node. Furthermore, the balance is almost as good as for the other two policy types that focus on improving balance. However, the overall memory usage is higher by 25%, and it is necessary to maintain the coherence of page 3 on write operations.

2.4. Benefits of Improved Mappings

Thread and data mapping aim to improve the memory accesses to shared and private data in parallel applications. This section discusses how such mappings can improve performance and energy efficiency.

2.4.1. Performance Improvements. Thread mapping improves the efficiency of the interconnections, 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 read-only situations, executing threads on the same shared cache reduces data replication in caches, thereby increasing the available cache space [Chishti et al. 2005]. In read-write or write-write situations, an improved thread mapping also reduces cache line invalidations, reducing the traffic on the interconnections as well as preventing a cache miss on the next access to the cache line [Zhou et al. 2009].

Data mapping improves the memory locality on NUMA machines by reducing the number of accesses to remote memory banks. Like thread mapping, it improves the efficiency of the interconnections by reducing the traffic between NUMA nodes. It can also prevent an imbalance in the use of memory controllers. This increases the memory bandwidth available in the system and reduces the average memory access latency.

Thread and data mappings can be performed in an integrated way for increased performance gains [Diener et al. 2015b]. Thread mapping prevents unnecessary thread migrations between NUMA nodes so that the data mapping can be more effective. If several threads that run on different NUMA nodes access the same page, data mapping alone is not effective. By performing thread mapping, threads that access the same data are executed on the same NUMA node, thereby increasing the benefits of data mapping.

2.4.2. Energy Consumption Improvements. Improved thread and data mappings can also reduce the energy consumption of parallel applications. By reducing the application execution time, static energy consumption will be lowered proportionally in most circumstances, since the static energy consumption goes up linearly with the total execution time. Reducing the number of cache misses and traffic on the interconnections also decreases the dynamic energy consumption. Overall, thread and data mappings can result in a reduction in energy consumption that is linearly proportional to the reduction in execution time [Diener et al. 2015].

3. A TAXONOMY OF AFFINITY-BASED MAPPING MECHANISMS

Affinity-based mapping mechanisms generally consist of two parts: the *analysis* of memory access behavior, and the mapping *policy*. Important characteristics of the *analysis* are: when is it performed (whether information is available before execution starts), which metrics are used to describe the behavior, at which level information is gathered (hardware, operating system, runtime environment, or application, among others), and whether the hardware or software need to be modified.

Based on the analyzed behavior, a mapping mechanism needs to follow a *policy* to determine where threads and data should be placed and when they should be migrated. The policy can be characterized in terms of its goals (such as improving locality or

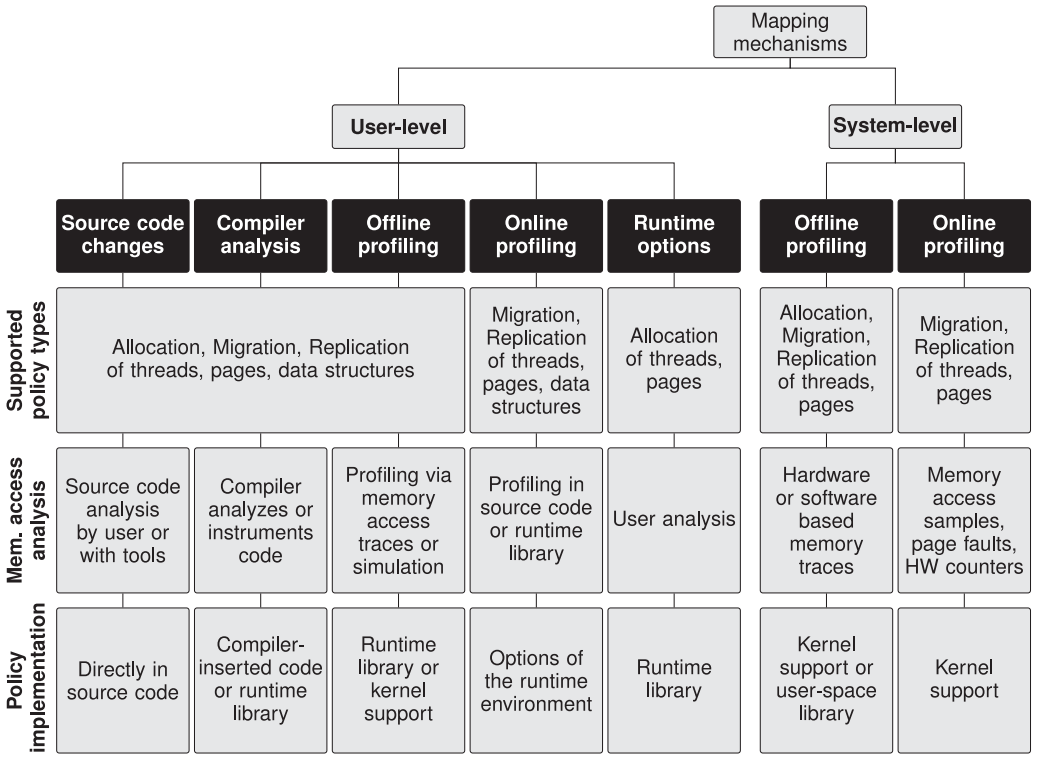


Fig. 5. A taxonomy of affinity-based thread and data mapping mechanisms.

balance), when it is applied (before or during execution), and whether the application or runtime environment need to be modified.

Considering the way the analysis and policy parts are performed (in the context of the execution of a parallel application) and their characteristics, we have developed a taxonomy of affinity-based mapping mechanisms, consisting of two groups (user-level and system-level mechanisms) divided into seven types, as shown in Figure 5.

3.1. User-Level Mechanisms

User-level mapping mechanisms target single parallel applications. They are usually implemented in user-level software and require no special privileges or hardware access for their execution. All policy types and policy goals mentioned in Section 2 are supported.

3.1.1. Source Code Changes. Mapping mechanisms based on source code changes rely on source code or behavior analysis performed by the developer manually or with the help of tools to profile the program (requiring a prior execution). The mapping is then done by changing the source code of the parallel application. To perform thread mapping, application developers can use operating system functions to execute threads on a specified set of processing units. Many runtime environments for OpenMP and MPI offer functions to specify a thread mapping.

Data mapping can be performed by introducing an explicit memory access at the beginning of execution such that each memory page is accessed first by the thread that will access it extensively later, which can be beneficial in *first-touch* data mapping policies. Another possibility is to use libraries that support memory allocation on specific

NUMA nodes. To handle the case where the memory access behavior changes during execution, most operating systems include functions to migrate memory pages during execution.

Such mapping mechanisms can yield significant improvements if the developer has detailed knowledge of the application's behavior. They are, however, intrusive, as they potentially require complex changes to the source code of the application and adaptation to different hardware architectures.

3.1.2. Compiler Analysis. Mechanisms based on compiler analysis perform the mapping in the compiler or its runtime support libraries (such as OpenMP). Such a mapping can be static, for example, by using a preprocessor to analyze the application source code and insert functions to perform the mapping. To support dynamic memory access behaviors, the analysis needs to be performed dynamically as well, for example, by inserting code into the application that performs the analysis online. Compiler-based techniques are generally limited to specific compiler versions and runtime libraries.

3.1.3. Offline Profiling. Mapping mechanisms based on offline profiling consist of two steps: First, the application is profiled to determine its memory access behavior, for example through memory or communication traces. The profile is then analyzed to determine an improved mapping. In the second step, the determined mappings are applied during the execution of the application. The mapping is usually implemented by modifying the source code of the application or by using options of the runtime environment.

The profiling phase is potentially time-consuming and is not applicable if the application changes its behavior between executions, as discussed in Section 2.1. Furthermore, the data generated during profiling might be very large, necessitating a time-consuming analysis [Zhai et al. 2011]. However, it incurs only a minimal runtime overhead, as the behavior analysis and mapping decisions are performed before the application starts.

3.1.4. Online Profiling. Online profiling mechanisms perform the mapping during the execution of the parallel application, using information gathered only during execution. At the user level, relatively little information about memory accesses is available. Therefore, many mechanisms use indirect information about the behavior, such as cache misses or the number of executed instructions per cycle in order to estimate the behavior.

An important advantage of these techniques is that no expensive analysis before execution has to be performed. The main challenge for this type of mechanism is the (potentially variable) tradeoff between accuracy and runtime overhead, as the information gathering and migration may have a large impact on the application, reducing or even nullifying the benefits of mapping. Collecting information about every memory access usually increases considerably the overhead compared to normal execution [Diener et al. 2015b]. Furthermore, since no prior information about the application behavior is available, future behavior must be predicted using past behavior.

The implementation of these mechanisms is usually based on a modified runtime environment, which gathers the information and performs the mapping.

3.1.5. Runtime Options. Some thread and data mappings can be applied directly through options of the runtime environment, including parallel libraries and the kernel, without modifying the parallel application. For thread mapping, many common OpenMP and MPI runtime environments offer options based on simple policies, such as a compact mapping. Most data mapping runtime options are provided by the operating system, for example, by specifying an interleave policy for the application.

Table I. Common Characteristics of Thread and Data Mapping Mechanisms

	User-level					System-level	
Property	Source code changes	Compiler analysis	Offline profiling	Online profiling	Runtime options	Offline profiling	Online profiling
Effort for user	High	Low	Low	Low	Low	Low	Low
Prior information	Yes	Yes	Yes	No	Yes	Yes	No
Req. previous execution	No	No	Yes	No	No	Yes	No
Dynamic behavior	Yes	Yes	Partially[1]	Yes	No	Partially[1]	Yes
Changes to application	Yes	Recomp.	No	No	No	No	No
Multiple applications	No	No	No	No	No	No	Yes
Runtime overhead	Low	Low	Low	Med.	Low	Low	Med.

[1]Support for dynamic behavior *during* execution only.

Runtime environment options do not provide a way to analyze the memory access behavior. Other mechanisms, such as source code analysis or profiling, can be used to determine the behavior and the best mapping. These mapping mechanisms provide the easiest way to improve the memory affinity of a parallel application with little or no overhead. However, the policies are relatively coarse-grained, especially for data mapping, which can limit their gains.

3.2. System-Level Mechanisms

The most important property of system-level mapping mechanisms is that they can take into account the behavior of multiple applications executing at the same time, in contrast to user-level solutions. They are generally implemented in the operating system or in hardware and therefore require special privileges or superuser access to operate. Some mechanisms are only applicable to certain hardware architectures, or even require hardware changes to function properly.

Since they operate at the system level, such mechanisms have no detailed information about the application's data structures and can, therefore, formulate data mapping policies only on top of pages, not data structures. All other policy types are principally supported by system-level mechanisms.

3.2.1. Offline Profiling. Similar to its user-level counterpart, offline profiling uses a separate profiling step before deciding on a mapping. The mapping itself is commonly implemented by modifying the source code of the application or by using options of the runtime environment. System-level mechanisms use special hardware counters or hardware-based memory tracers to monitor the memory access behavior, information that is usually not available at the user level.

3.2.2. Online Profiling. Similar to offline profiling, online profiling uses system-level statistics to guide mapping decisions during application execution. Such statistics include page faults, TLB misses, and cache coherence messages. Based on the measured statistics, mapping is done by invoking operating system functions.

3.3. Characteristics of Taxonomy Types

Each of the seven types discussed in the previous sections have characteristics that are shared by most mechanisms of a particular type. These common characteristics are discussed in this section and summarized in Table I.

3.3.1. Effort for Developer and User. For all mapping types, only source code changes to the applications present a high overhead for the developer or user. All other mechanism types may require extensive modifications to operating systems or runtime

environments, but these modifications are only required once and are then available to many applications.

3.3.2. Availability of Prior Information. Availability of prior information means that information about the memory access behavior is available before the behavior actually occurs. For example, the access behavior within a parallel loop might be known before that loop is executed.

Having prior information has the advantage that future memory access behavior does not have to be predicted using past behavior. Such prior information is also required for mechanisms that perform allocation-based policies. All mechanism types except online profiling can have at least some prior information.

3.3.3. Need for a Previous Execution. A related question is whether a mechanism requires a complete previous execution to gather information about the memory access behavior. As discussed in Section 2.1, for applications whose behavior changes between executions, previous executions (such as those from offline profiling mechanisms) may result in applying incorrect mappings, apart from causing a profiling overhead.

3.3.4. Support for Dynamic Behavior. If an application has a dynamic behavior during execution, as discussed in Section 2.1, mechanisms that perform only static mapping, before application start or at initialization, may yield only limited improvements. Mechanisms that can react to changes at runtime are able to handle better applications with a dynamic behavior, at the expense of a higher runtime overhead.

3.3.5. Changes to Application or Runtime Libraries. Mapping mechanisms that require changes to applications or runtime libraries are more difficult to apply and less general than mechanisms that need no such changes.

3.3.6. Support for Multiple Applications. Modern computer systems are often shared between several users that are executing different applications concurrently. In such scenarios, mapping decisions for different applications might interfere with each other. For greater generality, a mapping mechanism should therefore be able to take several applications into account.

3.3.7. Runtime Overhead. Runtime overhead refers to the impact of the mapping on the running application. Since they do not involve copying of data, allocation-based policies have usually lower overheads than those based on migration or replication.

3.4. Summary

We introduced a taxonomy for mapping mechanisms in this section. Based on where the mapping is applied (user level or system level), we have identified seven classes of mechanisms that have different properties. The same taxonomy can be applied to both thread and data mapping mechanisms. These mechanisms will be presented in the next two sections.

4. THREAD MAPPING MECHANISMS

In this section, we provide an overview of the thread mapping mechanisms according to the taxonomy and characteristics presented in Section 3. A detailed description of each mechanism is provided in Appendix A.

Table II contains an overview of the work in affinity-based thread mapping presented in this section. The table shows the supported parallel communication models (based on message passing or accesses to shared memory areas), as well as policy types and goals for each of the mechanisms.

At the user level, there are only a few thread mapping mechanisms that are based on *source code changes*, due to the ubiquity and simplicity of mapping options in runtime

Table II. Summary of the Thread Mapping Mechanisms

	Mapping type	Mechanism	Message passing	Shared memory	Type[1]	Goal[2]
User-level	Source code changes	[Rodrigues et al. 2009]	✓		Alloc	Loc
		[Diener et al. 2010; Diener 2010]		✓	Alloc	Loc
		[Ito et al. 2013]	✓		Alloc	Loc
	Compiler analysis	ForestGOMP [Broquedis et al. 2010a]		✓	Alloc, Mig	Loc
		[Ding et al. 2013]		✓	Alloc	Loc
	Offline profiling	Message tracers, e.g. [Chan et al. 1998]	✓		Alloc	Loc
		MPIPP [Chen et al. 2006]	✓		Alloc	Loc
		[Mercier and Clet-Ortega 2009]	✓		Alloc	Loc
		[Mercier and Jeannot 2011]	✓		Alloc	Loc
		[Brandfass et al. 2013]	✓		Alloc	Loc
		Numalize [Diener et al. 2015b]	✓	✓	Alloc, Mig	Loc
	Online profiling	[Tam et al. 2007; Azimi et al. 2009]		✓	Mig	Loc
		AutoPin [Klug et al. 2008]		✓	Mig	Loc, Bal
		ForestGOMP [Broquedis et al. 2010a]		✓	Alloc, Mig	Loc
		BlackBox [Radojković et al. 2013]		✓	Mig	Loc, Bal
	Runtime options	Compact [Eichenberger et al. 2012]	✓	✓	Alloc	Loc
		Scatter [Eichenberger et al. 2012]	✓	✓	Alloc	Bal
		RoundRobin [Argonne National Laboratory 2014]	✓	✓	Alloc	Bal
		[Hursey et al. 2011]	✓		Alloc	Loc
System-l.	Offline profiling	—				
	Online profiling	[Cruz et al. 2012; Cruz et al. 2015b]		✓	Mig	Loc
		[Cruz et al. 2014a]		✓	Mig	Loc
		SPCD, CDSM [Diener et al. 2015a]	✓	✓	Mig	Loc

[1]Alloc: Allocation, Mig: Migration.

[2]Loc: Locality, Bal: Balance.

systems, as well as the difficulty to detect communication behavior. For the same reasons, few proposals for *compiler analysis* exist. Proposals for *offline profiling* focus mostly on MPI-based applications, where messages can be traced by instrumenting message-passing functions with a relatively low overhead. Mechanisms for user-level *online profiling* rely on indirect execution statistics and are therefore more suitable for shared-memory applications, where offline profiling causes a high overhead. Many parallel environments, both for shared memory and message passing, directly support *runtime options* to map threads.

At the system level, we are not aware of any thread mapping mechanisms that use *offline profiling*. Several solutions for *online profiling* have been proposed, and they are based on statistics gathered from the hardware or operating system.

For thread mapping, the vast majority of mechanisms target single running applications and are implemented at the user level. This is related to the fact that relatively little information is necessary to perform the mapping, as only information about the communication pattern is required, but not the data that is actually communicated. Furthermore, applying a thread mapping is straightforward in many parallel programming environments. Recent years have seen a large number of online profiling mechanisms, both at the user and system levels.

5. DATA MAPPING MECHANISMS

This section presents an overview of the data mapping mechanisms according to our taxonomy presented in Section 3. A detailed description of each mechanism is provided in Appendix B.

Table III. Summary of the Data Mapping Mechanisms

	Mapping type	Mechanism	Thread mapping	Type[1]	Goal[2]
User-level	Source code changes	Libnuma [Kleen 2004]		Alloc	Loc, Bal
		Mai [Ribeiro et al. 2009]		Alloc	Loc
		[Dupros et al. 2010]	✓	Alloc	Loc
		[Cruz et al. 2011]	✓	Alloc	Loc
		[Majo and Gross 2012]		Alloc	Loc, Bal
		[Majo and Gross 2015]		Alloc	Loc
		[Mariano et al. 2016]	✓	Alloc	Loc, Bal
	Compiler analysis	[Nikolopoulos et al. 2000c]		Mig	Loc
		ForestGOMP [Broquedis et al. 2010a]	✓	Alloc, Mig	Loc
		Minas [Ribeiro et al. 2010]		Alloc	Loc
		SPM [Piccoli et al. 2014]		Mig	Loc
	Offline profiling	Numalize [Diener et al. 2015b]	✓	Alloc	Loc, Bal
		TABARNAC [Beniamine et al. 2015]		Alloc	Loc, Bal
	Online profiling	Next-touch, e.g. [Löf and Holmgren 2005]		Mig	Loc
		[Ogasawara 2009]		Mig	Loc
System-level	Runtime options	Numactl [Kleen 2004]	✓	Alloc	Loc, Bal
	Offline profiling	[Bolosky and Scott 1992]		Alloc, Repl	Loc
		[Marathe and Mueller 2006]		Alloc	Loc
		[Marathe et al. 2010]		Alloc	Loc
	Online profiling	[LaRowe et al. 1992]		Alloc, Mig, Repl	Loc, Bal
		[Chandra et al. 1994]	✓	Mig	Loc
		[Verghese et al. 1996b]		Mig, Repl	Loc
		[Tikir and Hollingsworth 2008]		Mig	Loc
		[Awasthi et al. 2010]		Mig	Loc
		NUMA Balancing [Corbet 2012b]	✓	Mig	Loc
		Carrefour [Dashti et al. 2013]		Mig, Repl	Loc, Bal
		LAPT [Cruz et al. 2014b]	✓	Mig	Loc
		kMAF [Diener et al. 2014]	✓	Mig	Loc, Bal
		[Gennaro et al. 2016]	✓	Mig	Loc

[1]Alloc: Allocation, Mig: Migration, Repl: Replication.

[2]Loc: Locality, Bal: Balance.

An overview of the work in data mapping presented in this section is shown in Table III. The table presents, for each proposed mechanism, whether it includes support for thread mapping and shows the policy types and goals supported, according to our discussion in Section 2.3.

At the user level, many data mapping mechanisms are based on *source code changes*, in contrast to thread mapping. The reason for this difference is the higher complexity of data mapping, which is difficult to solve using simple user-level mechanisms. Several techniques using *compiler analysis* have been proposed, although these are often restricted to capturing simple memory access patterns. Only few solutions for *offline profiling* have been proposed, mainly due to the substantial runtime overhead. Similarly, *online profiling* has received little attention in user-space, since such mechanisms can be implemented at the system level with a higher generality. Data mapping based on *runtime options* is uncommon, with few environments providing explicit support for mapping.

At the system level, several *offline profiling* mechanisms have been proposed, which gather information about memory access behavior from hardware counters to guide data mapping decisions on subsequent executions. A large number of proposals use *online profiling* at the system level, gathering hardware or operating system statistics to migrate memory pages at runtime.

Compared to thread mapping, a larger portion of the data mapping mechanisms are implemented at the system level, especially for online profiling. This is related to the comparative difficulty of performing data mapping in user space, since the amount of information required for data mapping is much higher and requires a lower granularity of information. Furthermore, applying the mapping in user space is highly intrusive and prone to conflict with the mapping of other applications that are running at the same time.

Another important observation is that replication-based policies that were dominant in data mapping research for early NUMA architectures have almost completely disappeared. On modern NUMA machines, replication of data requires complex and expensive coherence mechanisms in case of write operations, which make such policies only beneficial for data that is mostly read. Modern policies are commonly based on allocation and migration.

6. THREAD AND DATA MAPPING EXAMPLE

To illustrate the operation and benefits of various types of mapping mechanisms, we present next a case study of a scientific application kernel, *Ondes3D*. Ondes3D simulates the propagation of seismic waves due to earthquakes using a finite-differences numerical method [Aochi et al. 2013]. Its parallel version is implemented with OpenMP [Dupros et al. 2008].

Ondes3D has a static memory access behavior and is, therefore, suitable to a wide range of mapping techniques, including those that require memory access traces. Its communication pattern is determined by domain decomposition, leading to high amounts of communication between neighboring threads. In the baseline version of Ondes3D, all input data is initialized by the master thread, leading to an unfavorable data mapping with a first-touch policy, that causes a high number of remote NUMA accesses as well as a high imbalance.

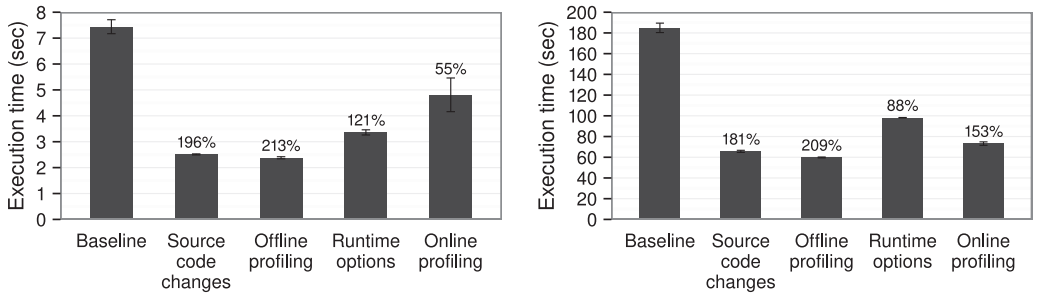
6.1. Mapping Mechanisms

Four different types of mapping mechanisms are compared to the baseline version for Ondes3D. These mechanisms were selected since they represent the most common mapping solutions and can be applied to a large variety of applications and hardware architectures. All four mechanisms perform both thread and data mapping.

6.1.1. Source Code Changes. We modified the source code of Ondes3D to implement a manual thread and data mapping. Due to the nearest-neighbor communication pattern, we implemented a Compact thread mapping via the `sched_setaffinity()` system call of Linux. At the beginning of the parallel phase of the application, after threads have been created, each thread is bound to a core through this system call such that threads with neighboring IDs are placed on cores that are nearby in the memory hierarchy and share a cache.

Data mapping was implemented via forced first-touch from each thread by manually determining which thread will access which part of the data structures, and initializing the parts of the data structures by this thread. In this way, pages will be located on the NUMA node that performs most accesses to them, improving both locality and balance of memory accesses.

6.1.2. Offline Profiling at the User Level. We use the Numalize mechanism [Diener et al. 2015b] to perform an offline profiling of Ondes3D based on a complete memory access trace. Numalize determines optimized thread and data mappings that are applied to the application via a kernel module that allocates threads and data according to the specified location.



(a) Small input set.

(b) Large input set.

Fig. 6. Results of the Ondes3D benchmark with various mapping mechanisms. Percentages on top of the bars show the performance improvements relative to the baseline.

6.1.3. Runtime Options. We executed Ondes3D with a Compact thread mapping via the `GOMP_CPU_AFFINITY` environment variable provided by gcc's OpenMP implementation. Since no runtime environment options exist that can improve locality through data mapping, we decided to improve at least the balance of the data mapping with an Interleave data mapping policy using the `numactl` tool.

6.1.4. Online Profiling at the System Level. As an example of an online profiling mechanisms, we selected the NUMA Balancing technique that is part of recent releases of the Linux kernel. NUMA Balancing migrates threads so that threads that access the same data will be grouped onto the same NUMA node. Furthermore, it migrates memory pages to the node where they are accessed. NUMA Balancing was executed in its default configuration provided by kernel version 3.13.

6.2. Methodology of the Experiments

Ondes3D was executed with two input sets, *small* and *large*, to show the influence of execution time on the gains that can be achieved by each type of mechanism. Both use the same input data but differ in the number of iterations. The application was executed on a 4-node NUMA machine. Each of the four NUMA nodes contains an Intel Xeon X7550 processor with eight cores and 2-way SMT. Each core has private L1 and L2 caches, while the large L3 cache is shared among all eight cores on the processor.

All experiments were performed on Linux kernel version 3.13. Ondes3D was compiled with gcc version 4.6.3 with the `-O2` optimization flag. We show the average execution time and the standard error of 10 executions for each configuration.

6.3. Results

The results of our experiment are shown in Figure 6. It is important to point out that mapping can achieve large performance improvements of more than 200% compared to the baseline. This indicates the significance of data mapping in modern shared memory systems.

Regarding the different mapping techniques, we can see that source code changes and offline profiling provide the largest improvements. However, these are also the mechanisms that have the highest up-front overhead, by requiring source code changes and recompilation or a time-consuming memory trace generation (for this application, the tracing causes a slowdown of 120× compared to normal execution). For both input sets, offline profiling results in slightly higher gains than the source code changes, since the profile also contains information about data structures that were not optimized by the source code changes, such as thread stacks.

The runtime environment options that were specified also caused a significant performance improvement, though not as high as the other mechanisms due to the lack of locality improvements via data mapping. However, this can be an attractive way to improve performance without a deep analysis of application behavior and without overhead. Online profiling, as expected, is more suitable for the larger input set due to the need to learn the application behavior during execution. For the large input set, it achieves a performance improvement that is comparable to the offline profiling.

7. CONCLUSIONS AND RESEARCH PERSPECTIVES

In recent years, the trend to support higher levels of parallelism in shared memory architectures has been evident and is projected to continue in the foreseeable future with the proliferation of many-core chips. In such systems, memory accesses of parallel applications represent a major challenge for application performance and energy consumption. To optimize memory accesses, two types of mapping techniques have been proposed: one that exploits the affinity between threads (thread mapping); and another that exploits affinity between threads and the data that they access (data mapping).

Mapping mechanisms have different characteristics, as well as advantages and disadvantages, depending on where and how they are implemented. These difference affect their applicability, gains, and overhead. In this article, we presented a taxonomy of mapping mechanisms consisting of two groups (user-level and system-level mechanisms) that are subdivided into seven classes. We then classified the large body of research that exists in this area into these groups. This classification showed that online profiling of applications has been the focus of considerable research efforts.

Our analysis also identified two gaps in current mapping research. Filling those gaps could lead to more efficient mechanisms. First of all, most current mechanisms are implemented completely at a single point in the hardware or software. Although some proposals use hardware information to perform mapping decisions in software, multiple software levels have not yet been explored. A *hybrid* mechanism that works at multiple levels can be an interesting solution, for example by combining a memory access analysis via a compiler mechanism with a mapping policy implemented in the kernel. In this way, advantages of both types of mechanisms can be combined.

The second gap that we identified is related to the fact that most current mechanisms are assuming a homogeneous hardware architecture, that is, all cores have the same computational power, processors have a symmetrical cache hierarchy and there is only a simple local/remote hierarchy of memory controllers. However, modern systems are starting to have a *heterogeneous* architecture. For example, processors such as ARM's big.LITTLE [ARM Limited 2013] have some cores that have a higher computational power and energy consumption than other cores on the same chip. Furthermore, architectures with multiple memory controllers on the same chip, often include a hierarchical NUMA behavior, with memory accesses to the local NUMA node, a remote node on the same chip and a remote node on a different chip. These heterogeneity issues have not yet been explored.

APPENDIXES

The appendices contain descriptions of the thread and data mapping mechanisms.

A. DESCRIPTION OF THREAD MAPPING MECHANISMS

A.1. User-Level Mechanisms

A.1.1. Source Code Changes.

Overview. Information about the memory access behavior is collected typically by a manual analysis of communication behavior (e.g., by determining how data structures

are shared among threads) or by tracing and analyzing communication messages or memory accesses. The communication behavior is used to determine an improved mapping of threads for the target architecture. The improved mapping is applied by calling mapping functions from the application or support libraries to bind threads to specific cores.

Since the most time-consuming parts of the mapping are performed before the actual execution of the application, the runtime overhead is very low, and such mechanisms can usually achieve improvements close to the optimum in case the behavior is determined correctly. However, apart from the requirements to change source codes, these mechanisms are highly dependent on hardware, operating systems, and parallel libraries, which limits their applicability.

To gather information about the hardware, tools such as hwloc [Broquedis et al. 2010b] or the older libtopology/PLPA [The Open MPI project 2009] can be used. Binding threads to specific cores can be achieved via the `hwloc_set_cpubind()` function of hwloc, which offers a platform-independent interface. Examples of system calls for thread mapping on the operating system level include `sched_setaffinity()` (Linux, BSD), `SetThreadAffinityMask()` (Windows) and `processor_bind()` (Solaris).

Examples of binding functions for specific runtime environments include `ippSetAffinity()` (Intel OpenMP) and `pthread_attr_setaffinity_np()` (Pthreads on Linux, BSD). The OpenMP standard also offers the `proc_bind` clause in version 4 of the standard [OpenMP Architecture Review Board 2013], which sets a thread mapping policy for a particular parallel region. The proposed `omp_set_proc_bind()` function interface [Eichenberger et al. 2012] was not included in the OpenMP standard.

Mapping Mechanisms. Rodrigues et al. [2009] evaluate affinity-based mapping of MPI tasks in the BRAMS weather simulation application [Freitas et al. 2009]. Communication is detected by inserting wrapper functions in the application's MPI function calls to trace messages, while the mapping is determined with the Scotch library [Pellegrini 1994]. The authors modify the MPICH runtime library [Gropp 2002] to perform the mapping via the `sched_setaffinity()` system call.

Ito et al. [2013] evaluate task mapping for two parallel applications that are based on domain decomposition of structured and unstructured grids. The communication pattern of each application is specified by the developer. The pattern is defined by the domain decomposition method, where communication happens between neighboring domains. At the beginning of each execution, the communication speed in the architectures is measured through point-to-point exchange of data between all pairs of MPI processes. Scotch is then used to determine the improved mapping. The mapping steps (except the communication pattern) are implemented directly inside the parallel applications.

Diener et al. [2010] and Diener [2010] present a solution for multithreaded applications. The communication pattern of the application is detected by instrumenting the Simics microarchitecture simulator [Magnusson et al. 2002], which traces all memory accesses of all threads and writes them to a file. The traces are checked for memory accesses to the same address by different threads in order to build a communication pattern. Based on the pattern, an improved mapping is determined with a heuristic greedy algorithm. The applications are modified to apply the resulting mapping with the `sched_setaffinity()` system call. Experiments with several parallel applications based on Pthreads were performed on 2-way SMT Intel Nehalem machines and 8-way SMT Sun Niagara2 systems.

A.1.2. Compiler Analysis. The ForestGOMP framework [Broquedis et al. 2010a, 2010c] is an extension to the libGOMP library provided by the GCC compiler. Every time an

OpenMP parallel region is entered, ForestGOMP groups the threads that execute that region into a *bubble*. Nested parallel regions create a tree of bubbles. Such bubbles usually share data and need to synchronize, causing communication. ForestGOMP also creates a tree model of the shared memory architecture based on the cores, cache memories, and processors in the system.

To perform the mapping from thread groups to elements of the hardware tree, ForestGOMP extends the BubbleSched framework [Thibault et al. 2007] and also allows the developer to create new scheduling policies. Their default policy, *Cache*, maintains threads of the same bubble close to each other in the hierarchy in order to improve communication and cache usage. ForestGOMP also allows the programmer to annotate OpenMP pragmas with scheduling hints.

Ding et al. [2013] implement a compiler extension for the Intel compiler that analyzes data reuse patterns in parallel loops. Loop iterations are then mapped to the cache hierarchy such that the locality of data accesses is optimized. Data that is reused in a shorter time is mapped such that is placed in higher-level caches, which are closer to the cores. Threads are mapped to cores with a custom algorithm that takes into account the data reuse and cache hierarchy. The mechanism requires loops to be dependency-free to allow reordering of iterations.

A.1.3. Offline Profiling.

Overview. In mapping mechanisms based on offline profiling, the analysis and policy steps are highly distinct. For the analysis, profiling methods such as message and memory access tracing are employed to determine the communication pattern. An improved mapping is determined with a mapping algorithm. This mapping is then applied to the application by providing runtime options or calling affinity functions from the application, as described in Sections A.1.1 and A.1.5. Most currently available mechanisms target either MPI-based or Pthreads/OpenMP-based applications.

MPI-Based Applications. Communication in MPI-based applications is usually detected by tracing MPI messages between tasks, for example, by creating function wrappers for MPI functions that cause communication. Information stored about the messages includes the sender and receiver tasks, as well as the number of bytes. Examples of such mechanisms are the MPI Parallel Environment (MPE) [Chan et al. 1998], eztrace [Trahay et al. 2011] and the Intel Trace Analyzer and Collector (ITAC) [Intel 2013].

MPIPP [Chen et al. 2006] is a framework to find improved task mappings for MPI-based applications, consisting of a message tracer and a mapping algorithm. MPIPP initially maps each task to a random core. At each iteration, MPIPP selects pairs of tasks to exchange between cores such that the communication cost is reduced. The quality of the determined mapping depends highly on the initial random mapping.

Mercier and Clet-Ortega [2009] evaluate task mapping for the NAS-MPI benchmark suite [Bailey et al. 1991]. Communication patterns are generated via traces, and the mapping is determined with the Scotch algorithm [Pellegrini 1994]. Processes are mapped with the `numactl` command [Kleen 2004].

Mercier and Jeannot [2011] use a different approach to improve affinity. Instead of modifying the location of MPI processes, they reorder ranks in such a way that communication is improved. This has the advantage of supporting migrations in case the communication behavior changes during execution. Communication patterns need to be provided by the developer or can be generated with a trace. The order of the ranks is determined with the Treematch algorithm [Jeannot et al. 2014].

Brandfass et al. [2013] use a similar rank reordering mechanism for MPI-based Computational Fluid Dynamics (CFD) codes. Tasks are assigned by minimizing the overall

communication cost using an approximation algorithm for the Quadratic Assignment Problem (QAP). The mapping is performed by modifying the machine file that describes the location of MPI processes at application start-up.

Applications Based on OpenMP and Pthreads. Offline profiling of applications that use shared memory parallelization APIs, such as OpenMP and Pthreads, usually has a higher overhead than MPI-based applications, as communication is performed implicitly through memory accesses [Diener et al. 2016], and not explicitly through function calls.

Numalize [Diener et al. 2015b] is a mechanism to trace memory accesses of parallel applications, based on the Pin dynamic binary instrumentation tool [Luk et al. 2005]. After analyzing the trace, Numalize outputs the communication matrix and improved thread mappings for the current architecture, which can be applied by providing runtime options.

A.1.4. Online Profiling. Autopin [Klug et al. 2008; Ott et al. 2008] executes a parallel application with various mappings passed to it, monitoring the Instructions Per Cycle (IPC) metric of each mapping. When profiling has finished, the application continues execution with the mapping that resulted in the highest IPC. This profiling and migration process is continued throughout the whole execution. In this way, communication is improved implicitly. However, the gains of this mechanism are highly dependent on the set of initial mappings provided to Autopin.

The BlackBox scheduler [Radojković et al. 2013, 2012] is based on similar concepts. It tests different thread mappings and executes the application with the mapping that resulted in the highest performance. If the number of threads is low, all possible thread mappings are evaluated, but if the number of threads is too high to evaluate all mapping possibilities, BlackBox executes the application with 1,000 random mappings to select the fastest one. BlackBox was evaluated on an 8-way SMT UltraSPARC T2 system with a parallel network packet filter benchmark.

The mechanisms of Tam et al. [2007] and Azimi et al. [2009] use hardware performance counters of the IBM Power5 architecture to analyze memory accesses of parallel applications. The mechanism reads a counter that contains the latest memory address that resulted in a remote cache access and stores a list of these addresses for each core. Periodically, the mechanism analyzes those lists. If several lists contain the same address, the address is used for communication. Threads are then migrated such that the number of remote accesses is reduced. This procedure is repeated throughout the execution. Due to the limitation to remote cache accesses, the accuracy of the mechanism is limited.

The previously mentioned ForestGOMP mechanism [Broquedis et al. 2010a, 2010c] also allows refining the thread mapping based on runtime information provided by hardware counters. However, the authors do not specify which counters can be used and do not provide further information on the implementation.

A.1.5. Runtime Options. When using runtime options to perform thread mapping, it is necessary to analyze the communication through other means, such as source code analysis or offline profiling presented previously. In contrast to source code changes, specifying the mapping as a runtime option has no runtime overhead and requires no recompilation. Runtime options are typically limited to specifying static mappings, without migrations during executions.

Most runtime environments provide a default set of policies from which the user can choose. In the *Compact* mapping, neighboring threads are mapped close to each other in the hardware hierarchy [Eichenberger et al. 2012]. Such a mapping can be beneficial when neighboring threads have a high level of communication among them. The *Scatter* policy distributes threads as evenly as possible within the machine [Eichenberger

et al. 2012], leading to a mapping where neighboring threads are mapped far from each other. The *RoundRobin* mapping assigns threads to cores that have the same ID, that is, thread 0 is mapped to core 0, thread 1 to core 1, and so on Argonne National Laboratory [2014]. Most environments also support a *Custom* mapping, where the exact placement of threads to cores can be specified by the user.

Some environments provide command line options to specify the mapping. For example, MPICH has the `-binding` parameter for the mpirun command [Argonne National Laboratory 2014]. Other MPI libraries provide a similar mechanism via files that specify the mapping, such as Open MPI [Gabriel et al. 2004] with its *rankfile* mechanism [The Open MPI Project 2013].

Many OpenMP frameworks offer selection of a thread mapping policy via environment variables. Two well-known mechanisms are the `GOMP_CPU_AFFINITY` of GCC OpenMP [Eichenberger et al. 2012] and `KMP_AFFINITY` in Intel's OpenMP implementation [Intel 2012]. Version 4 of the OpenMP offers a portable variable, `OMP_PLACES` [OpenMP Architecture Review Board 2013], which also supports specifying different policies for different levels of nested OpenMP statements.

Several tools allow a thread mapping to be performed directly from the command line of the parallel application. These include `hwloc-bind` [Broquedis et al. 2010b], `taskset`, and `numactl` [Kleen 2004].

Hursey et al. [2011] present the Locality-Aware Mapping Algorithm (LAMA) for Open MPI. LAMA is a mechanism to distribute processes among machines, sockets, and cores, according to the general policy selected by the user. It creates a rankfile, which can be used for subsequent executions.

A.2. System-Level Mechanisms

A.2.1. Offline Profiling. Although no explicit system-level offline profiling mechanism currently exists, to the best of our knowledge, the Numalize tool [Diener et al. 2015b] implements communication detection based on the physical system memory, not on the virtual memory of the application. Therefore, it is possible to extend it to support a system-wide profiling and mapping mechanism.

A.2.2. Online Profiling. Online communication detection has two main challenges, accuracy and overhead. Regarding the accuracy, since communication is implicit, care must be taken to obtain accurate temporal information about the communication [Diener et al. 2016]. Limiting the overhead means that efficient sampling strategies have to be used in order to reduce the impact on the running application.

Cruz et al. [2012, 2015a] propose detecting the communication pattern by comparing the contents of the Translation Lookaside Buffer (TLB). The most recently used pages of a core have a corresponding entry in its TLB. The mechanism compares the contents of all TLBs in the system and identifies as communication when the same entry is found in the TLBs of different cores. Temporal information is maintained, since the lifetime of TLB entries is limited. Most current hardware architectures require hardware changes to support this technique, since the TLB contents are usually managed by the processor and can not be accessed by software.

Cruz et al. [2012, 2014a] also proposed monitoring cache line invalidation messages to determine which cores access the same cache line. Each invalidation message is considered as a communication event, while the aggregated amount of communication of all cores is used to estimate the behavior. Temporal information is kept updated by adding an aging mechanism to the detected behavior so that old information is gradually removed. This mechanism requires extensive hardware changes because it adds a vector to each core to store the number of invalidation messages received by all other cores.

SPCD [Diener et al. 2013] and CDSM [Diener et al. 2015a] use page faults to detect communication. Page faults by different processes at the same addresses are considered as communication. To reduce the impact of old information, an aging mechanism is used. In order to increase the accuracy of the mechanism, artificial page faults are introduced during the execution of the parallel applications by iterating over the page table and clearing the page present bit of selected pages, up to a limit of 1% of the total number of pages. With these extra faults, the application behavior can be observed throughout the whole execution, and threads can be migrated according to changes in the behavior.

B. DESCRIPTION OF DATA MAPPING MECHANISMS

B.1. User-Level Mechanisms

B.1.1. Source Code Changes. Many libraries support NUMA-aware memory allocation. These include libnuma [Kleen 2004; Drepper 2007], the Memory Affinity interface (MAi) [Ribeiro et al. 2009; Ribeiro 2011], and Minas [Ribeiro et al. 2010; Ribeiro 2011]. With these libraries, data structures can be allocated according to the specification of the developer, for example, to a particular NUMA node, or with an interleave policy. Such techniques can achieve large performance improvements but place the burden of the mapping on the developer and might require rewriting the code for each different architecture.

An example usage of such libraries is provided by Dupros et al. [2010], who performed an in-depth analysis of the Ondes3D application [Dupros et al. 2008], which simulates the propagation of seismic waves. They evaluate performance with various data mapping strategies implemented with the help of the MAi library. Results show that allocating memory pages on the NUMA node that performs the most accesses to data structures resulted in the highest performance.

Cruz et al. [2011] use memory access tracing with the Simics microarchitecture simulator [Magnusson et al. 2002] to guide thread and data mapping decisions. The traces are analyzed to find the placement of threads and data that results in the highest locality of memory accesses. The mapping is implemented via the Minas framework for the applications from the OpenMP implementation of the NAS Parallel Benchmark (NPB) suite [Jin et al. 1999]. Due to the high simulation overhead, only small input sizes were evaluated.

Majo and Gross [2012] perform an extensive analysis of the memory access behavior of three parallel applications (streamcluster, ferret, and dedup) from the PARSEC benchmark suite [Bienia et al. 2008]. They identify memory accesses to remote NUMA nodes as a challenge for optimal performance and restructure each of the applications to improve memory access locality, cache usage, and hardware prefetcher effectiveness.

Majo and Gross [2015] present a library for Intel Threading Building Blocks (TBB) that allows a programmer to specify data distribution and computation schedules for parallel applications. To improve portability, the programmer specifications can be parametrized with the number of processors available at runtime, which is provided by the runtime environment. The authors modify five applications from the NAS and PARSEC benchmark suites and achieve performance improvements of up to 44% on three machines compared to standard TBB.

Mariano et al. [2016] analyze a large irregular application, HashSieve [Mariano et al. 2015], that is based on parallel memory accesses to large hash tables. Such a memory access behavior is a challenge for NUMA architectures, as caches are mostly ineffective (due to the high memory usage and unpredictable access behavior) and offer few possibilities for locality improvements. The authors were able to improve this

application by implementing a software prefetch mechanism and using an interleave strategy for the data structures.

B.1.2. Compiler Analysis. Bircsak et al. [2000] describe a *next-touch* strategy in the OpenMP runtime environment provided by the Compaq Fortran compiler. In next-touch policies, first proposed by Noordergraaf et al. [1999], a page or data structure is marked in such a way that it will be migrated to the node that performs the next access to it. Bircsak et al. introduce special OpenMP pragmas to annotate the source code to determine which variables should be migrated with this policy.

Nikolopoulos et al. [2000a, 2000b, 2000c] propose UPMlib, an OpenMP library that gathers information about thread migrations and memory statistics of parallel applications to aggressively migrate pages between NUMA nodes when threads are migrated. UPMlib intercepts thread migrations and uses hardware counters on SGI Irix machines to detect pages with large numbers of remote accesses, which are candidates for migration.

The ForestGOMP framework [Broquedis et al. 2010a, 2010c] presented in Section 4 also contains a NUMA-aware memory scheduler to improve memory affinity for OpenMP applications. The thread teams created by parallel OpenMP sections have a memory set attached to them, which the runtime library schedules to the same NUMA node, thereby increasing memory access locality.

The previously mentioned Minas framework [Ribeiro et al. 2010, 2011] also includes an optional source code preprocessor (MApp – Memory Affinity Preprocessor) to identify memory access patterns to large arrays at compile time. For each array, the preprocessor selects the most suitable data mapping policy and modifies the source code to achieve the mapping. Apart from a recompilation, no user intervention is necessary. However, the mechanism is limited to source code that can be analyzed at compile time and does not support migrations.

The Selective Page Migration (SPM) mechanism [Piccoli et al. 2014] performs memory access analysis via compiler-inserted code dynamically during execution. The authors modify the clang compiler [Lattner 2011] to instrument parallel loops. The instrumentation code predicts during runtime the memory access behavior of a loop before it is executed, taking into account the actual execution parameters. The prediction guides the migration of memory pages to the nodes where they will be accessed most during the loop.

B.1.3. Offline Profiling. Several tools provide information about the memory access pattern to pages of parallel applications and suggest policies that can improve the data mapping. Numalize [Diener et al. 2015b] is based on the Pin DBI tool [Luk et al. 2005] and traces all memory access by the application, aggregating the results on the page granularity. Based on the measured behavior, Numalize suggests specific mapping policies and stores them in files. These mappings can be applied with a Linux kernel module that reads these files and applies the specified mapping.

TABARNAC [Beniamine et al. 2015] is an extension to Numalize that can provide a graphical visualization of the page usage, helping the developer to understand and improve an application's memory access behavior. The authors use it to improve the *IS* benchmark from the OpenMP implementation of NPB.

Several other tools have been proposed to analyze the memory access behavior on NUMA machines. These include the HPCToolkit [Adhianto et al. 2010; Liu and Mellor-Crummey 2014], MemAxes [Giménez et al. 2014, 2015], Memphis [McCurdy and Vetter 2010], and MemProf [Lachaize et al. 2012]. Although these tools can identify common memory access issues, they usually do not provide improved mapping policies.

B.1.4. Online Profiling. Löf et al. [2005] present and evaluate a next-touch mechanism based on online profiling for applications running on the Solaris operating system. The profiling is performed by periodically making a system call from the application (`madvise()` with the `MADV_ACCESS_LWP` argument) that specifies address ranges for the next-touch policy. This system call tells Solaris to migrate pages in that range to the NUMA node from which the next memory access is performed. The advantage of such a mechanism for the developer is that he does not have to determine an improved data mapping, he just needs to specify at which points during execution the access behavior might change.

Goglin and Furmento [2009] implement two next-touch mechanisms for Linux based on the ideas presented by Löf et al. [2005]. Their first mechanism was implemented completely in user-space without any special kernel support. An application uses the `mprotect()` system call to mark an address range as inaccessible. On the next memory access to an address within that range, the kernel signals a segmentation fault, which can be caught and handled by the application. In the segmentation fault handler of the application, the page that was accessed is moved to the node where the access came from, and the page is marked as accessible again. Terboven et al. [2008] implement a very similar mechanism for applications based on OpenMP. They additionally discuss the importance of thread affinity.

Goglin and Furmento's [2009] second mechanism contains kernel support for migration, although the next-touch policy is still guided from user space. Similar to before, an application informs the kernel via a system call about an address range for the next-touch and the kernel marks these pages as inaccessible. On the next memory access, the kernel migrates the page and marks it as accessible, without an intervention from the application. The authors compare both implementations and conclude that the mechanism with kernel support is more efficient, mostly due to fewer transitions between kernel and user space.

Lankes et al. [2010] independently developed a very similar kernel-based mechanism for Linux and report comparable results regarding the difference between pure user space and kernel-assisted migrations.

Ogasawara [2009] proposes a data mapping method for Java applications via a NUMA-aware Garbage Collector (GC). During each garbage collection run, the JVM determines which thread accesses each object the most, called the Dominant Thread (DoT) of each object. After determining this, the GC migrates each object to the NUMA node where the thread is running.

B.1.5. Runtime Options. The numactl application [Kleen 2004] is a Linux tool to specify process bindings (as discussed in Section A.1.5) and can also be used to select different data mapping policies for applications, such as an interleave policy or allocation on a specified NUMA node.

B.2. System-Level Mechanisms

B.2.1. Offline Profiling. Bolosky et al. [Bolosky et al. 1991; Bolosky and Scott 1992] create a model for data mapping based on replication and migration of pages for early NUMA systems. The model is based on the cost of memory accesses and the cost of migrations. Their model is evaluated by single-stepping and decoding each instruction of a set of parallel applications in order to trace all memory accesses to data. With the help of the trace and their model, the authors implemented and evaluated an optimal data mapping policy that can greatly reduce the cost of memory accesses.

Chandra et al. [1994] studied page migration policies in the Stanford DASH computer [Lenoski et al. 1992], consisting of 16 MIPS R3000 processors. Information about page usage was gathered offline by measuring cache misses and TLB misses on the

page granularity with information provided by the MIPS R3000 processor. Based on the collected information, the authors simulate several migration policies with different migration limits. Thread mapping is handled by reducing the likelihood of migrations between processors and NUMA nodes in the IRIX scheduler used in the experiments. They find that basing migration decisions on cache miss information results in the highest overall gains, although TLB misses proved to be almost as effective. The authors also configured their mechanism to run online. However, no performance improvements were achieved in this scenario due to the high overhead of the virtual memory subsystem of the IRIX operating system.

Marathe and Mueller [2006] make use of hardware counters from the Performance Monitoring Unit (PMU) of Intel Itanium 2 processors to generate samples of memory addresses accessed by each thread. The profiling mechanism imposes a high overhead, as it requires traps to the operating system on every high-latency memory load operation or TLB miss. For this reason, the profiling is enabled only during the beginning of each application, therefore losing the opportunity to handle changes during the execution of the application. They also propose that the developer of the application should instrument the source code of the application to provide the operating system with information regarding the areas of the application that should have their memory accesses monitored.

Marathe et al. [2010] use the same hardware-based profiling mechanism to generate memory access traces for data mapping. In the software level, they capture the memory addresses sampled by the PMU and associate them to the thread that performed the memory access, thus generating a memory trace. The memory trace is then analyzed to select an improved data mapping based on improving the locality of accesses. In subsequent executions of the application, its pages are mapped according to the improved mapping.

B.2.2. Online Profiling. LaRowe et al. [1991, 1992] present an analytical model of page placement for early NUMA architectures without hardware cache coherence. They propose page migration and replication policies, where the same page is stored on multiple NUMA nodes. Implemented in the DUnX research kernel, policy decisions are performed during page faults, with a page scanner daemon periodically triggering additional evaluations via extra page faults. The coherence between replicated pages is maintained with the same software mechanism that performs cache coherence. No balancing of pages or thread mapping is performed. Similarly, Bolosky and Scott [1992] also mention page-fault–based online policies.

Verghese et al. [1996a, 1996b] propose similar dynamic page migration and replication mechanisms for SGI's IRIX operating system but use cache misses as a metric to guide mapping decisions. They require information about all cache misses and migrate pages to a node with lots of cache misses from a single thread or replicate a pages if it receives a lot of cache misses from multiple threads. The authors also evaluate the number of TLB misses as a metric to guide the mapping but conclude that it is not accurate enough for data mapping. For modern architectures, this detailed information about cache misses cannot be gathered with an acceptable overhead. Cache misses themselves are a more indirect measure of the memory access behavior, especially regarding the node where a page should be mapped to. On modern systems with large caches, cache misses might indicate that the page is not used frequently from a node, which can imply that a page should *not* be migrated to a node with lots of cache misses to that page. Furthermore, this proposal has similar drawbacks as LaRowe's, such as the lack of balance-based data mapping and a thread mapping policy, as well as the overhead of maintaining coherence of replicated pages on modern systems.

Awashti et al. [2010] propose two page migration mechanisms that use queuing delays and row-buffer hit rates from memory controllers. The first is called Adaptive

First-Touch and consists of gathering statistics of the memory controller to map the data of the application in future executions. The second mechanism uses the same information but allows online page migration during the execution of the application. They select the destination NUMA node considering the difference of the access latency between the source and destination NUMA nodes, as well as the row-buffer hit rate and queuing delays of the memory controller. The mechanism does not have information about which data each thread accesses. The page to be migrated is randomly chosen and may lead to an increase in the number of remote accesses.

More recent proposals in operating systems also use page faults for data mapping. Modern versions of the Linux kernel (since version 3.8) include the NUMA Balancing technique [Corbet 2012b] for the x86_64 architecture. NUMA Balancing support was extended to the PowerPC architecture in kernel version 3.14. A previous proposal with similar goals was AutoNUMA [Corbet 2012a]. NUMA Balancing uses the page faults of parallel applications to detect memory accesses and performs a sampled next-touch strategy. Whenever a page fault happens and the page is not located on the NUMA node that caused the fault, the page is migrated to that node. However, this mechanism keeps no history of accesses, which can lead to a high number of migrations, and it performs no thread mapping to improve the gains of the data mapping.

Current research uses a history of memory accesses to limit unnecessary migrations and perform thread mapping. The Carrefour mechanism [Dashti et al. 2013; Gaud et al. 2015] has similar goals as NUMA Balancing, but uses Instruction-Based Sampling (IBS) [Drongowski 2007], available in recent AMD architectures, to detect the memory access behavior. It maintains a history of memory accesses to limit unnecessary migrations. Additionally, it allows replication of pages that are mostly read. However, the authors need to use the sampled accesses to predict if a page will be written to, as these writes have a large overhead due to the coherence, and the OS keeps only very coarse-grained information about the write permissions to pages [Basu et al. 2013]. To limit the runtime overhead, Carrefour limits its characterization and data mapping to 30,000 pages [Dashti et al. 2013] (corresponding to about 120MByte of main memory with 4KByte pages), which limits its applicability to small applications. The authors suggest that Carrefour could be ported to Intel-based architectures via the Precise Event-Based Sampling (PEBS) framework [Levinthal 2009].

The kernel Memory Affinity Framework (kMAF) [Diener et al. 2014; Diener 2015] uses page faults to detect the memory access behavior. Whenever a page fault happens, kMAF verifies the ID of the thread that generated the fault, as well as the memory address of the fault. To increase its accuracy, kMAF introduces additional page faults. These additional page faults impose an overhead to the system, since each fault causes an interrupt to the operating system. Like other sampling-based mechanisms, kMAF has to limit to the number of samples of memory accesses to control the overhead, limiting the accuracy of the detected memory access behavior.

Tikir et al. [2004, 2008] use UltraSPARC III hardware counters to provide information for the data mapping. They bind the threads to the cores using a round-robin policy, without considering any data sharing behavior. The hardware counters are provided by the Sun Fire Link, which counts and samples the transactions on the address bus of the Sun Fireplane interconnect. They insert instrumentation code into the application with the Dyninst library [Buck and Hollingsworth 2000] to gather profiling information, to migrate the memory pages, to bind application threads to processors, and to detect the application termination. The authors also propose using other information to guide mapping, including statistics from the Translation Lookaside Buffers (TLBs). Their proposal focuses on architectures with software-managed TLBs, which is only a minority of current systems.

The Locality-Aware Page Table (LAPT) [Cruz et al. 2014b] is an extended page table that stores the memory access behavior for each page in the page table entry. This memory access behavior comprises the last threads that accessed the pages. LAPT also keeps a communication matrix containing the affinity between the threads. This information is updated by the Memory Management Unit (MMU) on every TLB access. The operating system evaluates the page table periodically, updating the affinity between the page and the NUMA nodes for data mapping. Additionally, it analyzes the communication matrix to determine the thread mapping. Threads and data are periodically migrated according to the determined memory access behavior.

Similarly to the Carrefour, kMAF, and NUMA Balancing mechanisms, Gennaro et al. [2016] propose a mapping technique based on the page faults of multithreaded applications. In contrast to the previous mechanisms, Gennaro et al. identify multiple accesses from different threads to the same pages as a source of inaccuracy for such detection techniques, as a page fault caused by one thread will be resolved, at which point it masks eventual accesses to the page by other threads. The authors solve this issue by creating additional thread-specific page tables that can each resolve their own page faults. In this way, threads that access the same data do not mask each others' page faults.

REFERENCES

L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701. DOI:http://dx.doi.org/10.1002/cpe.1553

Ahmad Anbar, Olivier Serres, Engin Kayraklioglu, Abdel-Hameed A. Badawy, and Tarek El-Ghazawi. 2016. Exploiting hierarchical locality in deep parallel architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 2 (2016), 1–25. DOI:http://dx.doi.org/10.1145/2897783

Hideo Aochi, Thomas Ulrich, Ariane Ducellier, Fabrice Dupros, and David Michea. 2013. Finite difference simulations of seismic wave propagation for understanding earthquake physics and predicting ground motions: Advances and challenges. *Journal of Physics: Conference Series* 454, 1 (Aug 2013), 012010. DOI:http://dx.doi.org/10.1088/1742-6596/454/1/012010

Argonne National Laboratory. 2014. Using the Hydra Process Manager. Retrieved 2015-06-08 from https://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager.

ARM Limited. 2013. *big.LITTLE Technology: The Future of Mobile*. Technical Report. Retrieved 2016-09-01 from https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. 2010. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 319–330. DOI:http://dx.doi.org/10.1145/1854273.1854314

Reza Azimi, David K. Tam, Livio Soares, and Michael Stumm. 2009. Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review* 43, 2 (Apr 2009), 56–65. DOI:http://dx.doi.org/10.1145/1531793.1531803

David H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS parallel benchmarks. *International Journal of Supercomputer Applications* 5, 3 (1991), 66–73. DOI:http://dx.doi.org/10.1177/109434209100500306

G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. 2014. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica* 23, May (2014), 1–155. DOI:http://dx.doi.org/10.1017/S0962492914000038

Nick Barrow-Williams, Christian Fensch, and Simon Moore. 2009. A communication characterisation of splash-2 and parsec. In *IEEE International Symposium on Workload Characterization (IISWC'09)*. 86–97. DOI:http://dx.doi.org/10.1109/IISWC.2009.5306792

Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. In *International Symposium on Computer Architecture (ISCA'13)*. 237–248. DOI:http://dx.doi.org/10.1145/2508148.2485943

David Beniamine, Matthias Diener, Guillaume Huard, and Philippe O. A. Navaux. 2015. TABARNAC: Visualizing and resolving memory access issues on NUMA architectures. In *Workshop on Visual Performance Analysis (VPA'15)*. DOI:http://dx.doi.org/10.1145/2835238.2835239

Abhinav Bhatele. 2010. *Automating Topology Aware Mapping for Supercomputers*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 72–81. DOI:http://dx.doi.org/10.1145/1454115.1454128

John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. 2000. Extending OpenMP for NUMA machines. In *ACM/IEEE Conference on Supercomputing (SC'00)*. 163–181. DOI:http://dx.doi.org/10.1109/SC.2000.10019

Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A case for NUMA-aware contention management on multicore systems. In *USENIX Annual Technical Conference (ATC'11)*. 557–571.

Robert D. Blumofe and Charles E. Leiserson. 1994. Scheduling multithreaded computations by work stealing. In *Symposium on Foundations of Computer Science (FOCS'94)*. 1–29. DOI:http://dx.doi.org/10.1109/SFCS.1994.365680

Jacques E. Boillat and Peter G. Kropf. 1990. A fast distributed mapping algorithm. In *Joint International Conference on Vector and Parallel Processing (CONPAR 90 – VAPP IV)*. 405–416.

William J. Bolosky and Michael L. Scott. 1992. Evaluation of multiprocessor memory systems using off-line optimal behavior. *Journal of Parallel and Distributed Computing (JPDC)* 15, 4 (1992), 382–398. DOI:http://dx.doi.org/10.1016/0743-7315(92)90051-N

William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. 1991. NUMA policies and their relation to memory architecture. *ACM SIGARCH Computer Architecture News* 19, 2 (1991), 212–221. DOI:http://dx.doi.org/10.1145/106975.106994

Barbara Brandfass, Thomas Alrutz, and Thomas Gerhold. 2013. Rank reordering for MPI communication optimization. *Computers & Fluids* 80, July (2013), 372–380. DOI:http://dx.doi.org/10.1016/j.compfluid.2012.01.019

Timothy Brecht. 1993. On the importance of parallel application placement in NUMA multiprocessors. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*. 1–18.

David Brooks, Margaret Martonosi, John-David Wellman, and Pradip Bose. 2000. Power-performance modeling and tradeoff analysis for a high end microprocessor. In *International Workshop on Power-Aware Computer Systems (PACS'00)*. 126–136. DOI:http://dx.doi.org/10.1007/3-540-44572-2_10

François Broquedis, Olivier Aumage, Brice Goglin, Samuel Thibault, Pierre-André Wacrenier, and Raymond Namyst. 2010a. Structuring the execution of OpenMP applications for multicore architectures. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS'10)*. 1–10. DOI:http://dx.doi.org/10.1109/IPDPS.2010.5470442

Franois Broquedis, Jerome Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010b. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'10)*. 180–186. DOI:http://dx.doi.org/10.1109/PDP.2010.67

François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. 2010c. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming* 38, 5–6 (2010), 418–439. DOI:http://dx.doi.org/10.1007/s10766-010-0136-3

Mats Brorsson. 1989. *Performance Impact of Code and Data Placement on the IBM RP3*. Technical Report.

Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for runtime code patching. *International Journal of High Performance Computing Applications (IJHPCA)* 14, 4 (2000), 317–329. DOI:http://dx.doi.org/10.1177/109434200001400404

J. Mark Bull and Chris Johnson. 2002. Data distribution, migration and replication on a cc-NUMA architecture. In *European Workshop on OpenMP (EWOMP'02)*. 1–5.

Anthony Chan, William Gropp, and Ewing Lusk. 1998. *User's Guide for MPE Extensions for MPI Programs*. Technical Report.

Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. 1994. Scheduling and page migration for multiprocessor compute servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*. 12–24. DOI:http://dx.doi.org/10.1145/381792.195485

Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. 2006. MPIPP: An automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'06)*. 353–360. DOI:http://dx.doi.org/10.1145/1183401.1183451

Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 2005. Optimizing replication, communication, and capacity allocation in CMPs. *ACM SIGARCH Computer Architecture News* 33, 2 (May 2005), 357–368. DOI:http://dx.doi.org/10.1145/1080695.1070001

Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. 2005. Performance implications of single thread migration on a chip multi-core. *ACM SIGARCH Computer Architecture News* 33, 4 (2005), 80–91. DOI:http://dx.doi.org/10.1145/1105734.1105745

Pat Conway. 2007. The AMD opteron northbridge architecture. *IEEE Micro* 27, 2 (2007), 10–21.

Julita Corbalan, Xavier Martorell, and Jesus Labarta. 2003. Evaluation of the memory page migration influence in the system performance: The case of the SGI O2000. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'03)*. 121–129. DOI:http://dx.doi.org/10.1145/782814.782833

Jonathan Corbet. 2012a. AutoNUMA: The Other Approach to NUMA Scheduling. Retrieved 2015-06-08 from http://lwn.net/Articles/488709/

Jonathan Corbet. 2012b. Toward Better NUMA Scheduling. Retrieved 2015-06-08 from http://lwn.net/Articles/486858/.

Eduardo H. M. Cruz. 2012. *Dynamic Detection of the Communication Pattern in Shared Memory Environments for Thread Mapping*. Master's thesis.

Eduardo H. M. Cruz, Marco A. Z. Alves, Alexandre Carissimi, Philippe O. A. Navaux, Christiane Pousa Ribeiro, and Jean-François Méhaut. 2011. 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 PhD Forum*. 551–558. DOI:http://dx.doi.org/10.1109/IPDPS.2011.197

Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, and Philippe O. A. Navaux. 2014a. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. *Journal of Parallel and Distributed Computing (JPDC)* 74, 3 (Mar 2014), 2215–2228. DOI:http://dx.doi.org/10.1016/j.jpdc.2013.11.006

Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Laércio L. Pilla, and Philippe O. A. Navaux. 2014b. Optimizing memory locality using a locality-aware page table. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'14)*. 198–205. DOI:http://dx.doi.org/10.1109/SBAC-PAD.2014.22

Eduardo H. M. Cruz, Matthias Diener, and Philippe O. A. Navaux. 2012. Using the translation lookaside buffer to map threads in parallel applications based on shared memory. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS'12)*. 532–543. DOI:http://dx.doi.org/10.1109/IPDPS.2012.56

Eduardo H. M. Cruz, Matthias Diener, and Philippe O. A. Navaux. 2015a. Communication-aware thread mapping using the translation lookaside buffer. *Concurrency Computation: Practice and Experience* 22, 6 (2015), 685–701. DOI:http://dx.doi.org/10.1002/cpe.3487

Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, and Philippe O. A. Navaux. 2015b. An efficient algorithm for communication-based task mapping. In *International Conference on Parallel, Distributed, and Network-Based Processing (PDP'15)*. 207–214. DOI:http://dx.doi.org/10.1109/PDP.2015.25

William J. Dally. 2010. *GPU Computing to Exascale and Beyond*. Technical Report.

Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *International Symposium on High Performance Computer Architecture (HPCA'13)*. 107–118. DOI:http://dx.doi.org/10.1109/HPCA.2013.6522311

Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. 2013. Traffic management: A holistic approach to memory placement on NUMA systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. 381–393. DOI:http://dx.doi.org/10.1145/2451116.2451157

Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit V. Catalyurek. 2006. Parallel hypergraph partitioning for scientific computing. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS'06)*. 124–133. DOI:http://dx.doi.org/10.1109/IPDPS.2006.1639359

Matthias Diener. 2010. *Evaluating Thread Placement Improvements in Multi-core Architectures*. Master's thesis. Berlin Institute of Technology.

Matthias Diener. 2015. *Automatic Task and Data Mapping in Shared Memory Architectures*. Ph.D. Dissertation. Federal University of Rio Grande do Sul.

Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Mohammad S. Alhakeem, and Philippe O. A. Navaux. 2015. Locality and balance for communication-aware thread mapping in multicore systems. In *Euro-Par*. 196–208. DOI:http://dx.doi.org/10.1007/978-3-662-48096-0_16

Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, and Philippe O. A. Navaux. 2016. Communication in shared memory: Concepts, definitions, and efficient detection. In *Euromicro International Conference on Parallel, Distributed, and Network-based Processing (PDP'16)*. 151–158. DOI:http://dx.doi.org/10.1109/PDP.2016.16

Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiss. 2015. Kernel-based thread and data mapping for improved memory affinity. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 26, X (2015), 1–14. DOI:http://dx.doi.org/10.1109/TPDS.2015.2504985

Matthias Diener, Eduardo H. M. Cruz, and Philippe O. A. Navaux. 2013. Communication-based mapping using shared pages. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS'13)*. 700–711. DOI:http://dx.doi.org/10.1109/IPDPS.2013.57

Matthias Diener, Eduardo H. M. Cruz, and Philippe O. A. Navaux. 2015. Locality vs. balance: Exploring data mapping policies on NUMA systems. In *International Conference on Parallel, Distributed, and Network-Based Processing (PDP'15)*. 9–16. DOI:http://dx.doi.org/10.1109/PDP.2015.11

Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. kMAF: Automatic kernel-level management of thread and data affinity. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*. 277–288. DOI:http://dx.doi.org/10.1145/2628071.2628085

Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2015a. Communication-aware process and thread mapping using online communication detection. *Parallel Computing* 43, March (2015), 43–63. DOI:http://dx.doi.org/10.1016/j.parco.2015.01.005

Matthias Diener, Eduardo H. M. Cruz, Laércio L. Pilla, Fabrice Dupros, and Philippe O. A. Navaux. 2015b. Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation* 88-89, June (2015), 18–36. DOI:http://dx.doi.org/10.1016/j.peva.2015.03.001

Matthias Diener, Felipe L. Madruga, Eduardo R. Rodrigues, Marco A. Z. Alves, and Philippe O. A. Navaux. 2010. Evaluating thread placement based on memory access patterns for multi-core processors. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*. 491–496. DOI:http://dx.doi.org/10.1109/HPCC.2010.114

Wei Ding, Yuanrui Zhang, Mahmut Kandemir, Jithendra Srinivas, and Praveen Yedlapalli. 2013. Locality-aware mapping and scheduling for multicores. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. 1–12. DOI:http://dx.doi.org/10.1109/CGO.2013.6495009

Ulrich Drepper. 2007. *What Every Programmer Should Know About Memory*. Technical Report 4. Red Hat, Inc. 114 pages.

Paul J. Drongowski. 2007. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. Technical Report.

Fabrice Dupros, Hideo Aochi, Ariane Ducellier, Dimitri Komatitsch, and Jean Roman. 2008. Exploiting intensive multithreading for the efficient simulation of 3D seismic wave propagation. In *IEEE International Conference on Computational Science and Engineering (CSE'08)*. 253–260. DOI:http://dx.doi.org/10.1109/CSE.2008.51

Fabrice Dupros, Christiane Pousa, Alexandre Carissimi, and Jean-François Méhaut. 2010. Parallel simulations of seismic wave propagation on NUMA architectures. In *Parallel Computing: From Multicores and GPU's to Petascale*. 67–74. DOI:http://dx.doi.org/10.3233/978-1-60750-530-3-67

Alexandre E. Eichenberger, Christian Terboven, Michael Wong, and Dieter An Mey. 2012. The design of OpenMP thread affinity. *Lecture Notes in Computer Science* 7312 LNCS (2012), 15–28. DOI:http://dx.doi.org/10.1007/978-3-642-30961-8_2

Steven Frank, Henry Burkhardt III, and James Rothnie. 1993. The KSR1: Bridging the gap between shared memory and MPPs. In *IEEE Compcon*. 285–294. DOI:http://dx.doi.org/10.1109/CMPCON.1993.289682

S. R. Freitas, K. M. Longo, M. A. F. Silva Dias, R. Chatfield, P. Silva Dias, P. Artaxo, M. O. Andreae, G. Grell, L. F. Rodrigues, A. Fazenda, and J. Panetta. 2009. The coupled aerosol and tracer transport model to the brazilian developments on the regional atmospheric modeling system (CATT-BRAMS) part 1: Model description and evaluation. *Atmospheric Chemistry and Physics* 9, 8 (2009), 2843–2861. DOI:http://dx.doi.org/10.5194/acp-9-2843-2009

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVMMPI'04)*. 97–104. DOI:http://dx.doi.org/10.1007/978-3-540-30218-6_19

Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of memory management on modern NUMA systems. *Commununications of the ACM* 58, 12 (2015), 59–66. DOI:http://dx.doi.org/10.1145/2814328

Ilaria Di Gennaro, Alessandro Pellegrini, and Francesco Quaglia. 2016. OS-based NUMA optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID'16)*. 291–300. DOI:http://dx.doi.org/10.1109/CCGrid.2016.91

Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, and Bernd Hamann. 2014. Dissecting on-node memory access performance: A semantic approach. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. 166–176. DOI:http://dx.doi.org/10.1109/SC.2014.19

Alfredo Giménez, Ilir Jusufi, Abhinav Bhatele, Todd Gamblin, Martin Schulz, Peer-Timo Bremer, and Bernd Hamann. 2015. *MemAxes: Interactive Visual Analysis of Memory Access Data*. Technical Report.

Roland Glantz, Henning Meyerhenke, and Alexander Noe. 2015. Algorithms for mapping parallel processes onto grid and torus architectures. In *International Conference on Parallel, Distributed, and Network-Based Processing (PDP'15)*. 236–243. DOI:10.1109/PDP.2015.21

Brice Goglin and Nathalie Furmento. 2009. Enabling high-performance memory migration for multithreaded applications on linux. In *International Symposium on Parallel & Distributed Processing (IPDPS'09)*. DOI:http://dx.doi.org/10.1109/IPDPS.2009.5161101

William Gropp. 2002. MPICH2: A new start for MPI implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. DOI:http://dx.doi.org/10.1007/3-540-45825-5_5

Erik Hagersten and Michael Koster. 1999. WildFire: A scalable path for SMPs. In *International Symposium on High Performance Computer Architecture (HPCA'99)*. 172. DOI:http://dx.doi.org/10.1109/HPCA.1999.744361

Joshua Hursey, Jeffrey M. Squyres, and Terry Dontje. 2011. Locality-aware parallel process mapping for multi-core HPC systems. In *IEEE International Conference on Cluster Computing (CLUSTER'11)*. 527–531. DOI:http://dx.doi.org/10.1109/CLUSTER.2011.59

Intel. 2012. Using KMP_AFFINITY to create OpenMP thread mapping to OS proc IDs. Retrieved 2015-06-08 from https://software.intel.com/en-us/articles/using-kmp-affinity-to-create-openmp-thread-mapping-to-os-proc-ids.

Intel. 2013. Intel Trace Analyzer and Collector. Retrieved from http://software.intel.com/en-us/intel-trace-analyzer.

Satoshi Ito, Kazuya Goto, and Kenji Ono. 2013. Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments. *Computers & Fluids* 80 (jul 2013), 88–93. DOI:http://dx.doi.org/10.1016/j.compfluid.2012.04.024

Emmanuel Jeannot and Guillaume Mercier. 2010. Near-optimal placement of MPI processes on hierarchical NUMA architectures. In *Euro-Par Parallel Processing*. 199–210. DOI:http://dx.doi.org/10.1007/978-3-642-15291-7_20

Emmanuel Jeannot, Guillaume Mercier, and François Tessier. 2014. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems* 25, 4 (Apr 2014), 993–1002. DOI:http://dx.doi.org/10.1109/TPDS.2013.104

H. Jin, M. Frumkin, and J. Yan. 1999. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*. Technical Report October. NASA.

C Karlsson, T Davies, and Zizhong Chen. 2012. Optimizing process-to-core mappings for application level multi-dimensional MPI communications. In *IEEE International Conference on Cluster Computing (CLUSTER'12)*. 486–494. DOI:http://dx.doi.org/10.1109/CLUSTER.2012.47

George Karypis and Vipin Kumar. 1996. Parallel multilevel graph partitioning. In *International Parallel Processing Symposium (IPPS'96)*. 314–319. DOI:http://dx.doi.org/10.1109/IPPS.1996.508075

George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (Jan 1998), 359–392. DOI:http://dx.doi.org/10.1137/S1064827595287997

Andi Kleen. 2004. *An NUMA API for Linux*. Technical Report. Retrieved rom http://andikleen.de/numaapi3.pdf.

Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. 2008. autopin – automated optimization of thread-to-core pinning on multicore systems. In *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC)*. 219–235. DOI:http://dx.doi.org/10.1007/978-3-642-19448-1_12

Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A memory profiler for NUMA multicore systems. In *USENIX Annual Technical Conference (ATC'12)*. 53–64.

Stefan Lankes, Boris Bierbaum, and Thomas Bemmerl. 2010. Affinity-on-next-touch: An extension to the linux kernel for NUMA architectures. *Lecture Notes in Computer Science* 6067 LNCS, PART 1 (2010), 576–585. DOI:http://dx.doi.org/10.1007/978-3-642-14390-8_60

Richard P. LaRowe, Mark A. Holliday, and Carla Schlatter Ellis. 1992. An analysis of dynamic page placement on a NUMA multiprocessor. *ACM SIGMETRICS Performance Evaluation Review* 20, 1 (1992), 23–34. DOI:http://dx.doi.org/10.1145/133057.133082

Richard P. LaRowe Jr. 1991. *Page Placement For Non-Uniform Memory Access Time (NUMA) Shared Memory Multiprocessors*. Ph.D. Dissertation. Duke University.

Chris Lattner. 2011. LLVM and clang: Advancing compiler technology. In *Free and Open Source Developers European Meeting (FOSDEM'11)*.

Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. 1992. The DASH prototype: Implementation and performance. In *International Symposium on Computer Architecture (ISCA'92)*. 92–103. DOI:http://dx.doi.org/10.1109/ISCA.1992.753307

David Levinthal. 2009. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*. Technical Report.

Tong Li, Dan P Baumberger, and Scott Hahn. 2009. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. 65–74. DOI:http://dx.doi.org/10.1145/1504176.1504188

Xu Liu and John Mellor-Crummey. 2014. A tool to analyze the performance of multithreaded programs on NUMA architectures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. 259–272. DOI:http://dx.doi.org/10.1145/2555243.2555271

Henrik Löf and Sverker Holmgren. 2005. Affinity-on-next-touch: Increasing the performance of an industrial pde solver on a cc-NUMA System. In *International Conference on Supercomputing (ICS'05)*. 387–392. DOI:http://dx.doi.org/10.1145/1088149.1088201

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. 190–200. DOI:http://dx.doi.org/10.1145/1065010.1065034

Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *IEEE Computer* 35, 2 (2002), 50–58. DOI:http://dx.doi.org/10.1109/2.982916

Zoltan Majo and Thomas R. Gross. 2012. Matching memory access patterns and data placement for NUMA systems. In *International Symposium on Code Generation and Optimization (CGO'12)*. 230–241. DOI:http://dx.doi.org/10.1145/2259016.2259046

Zoltan Majo and Thomas R. Gross. 2015. A library for portable and composable data locality optimizations for NUMA systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. 227–238. DOI:http://dx.doi.org/10.1145/2688500.2688509

Jaydeep Marathe and Frank Mueller. 2006. Hardware profile-guided automatic page placement for ccNUMA systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. 90–99. DOI:http://dx.doi.org/10.1145/1122971.1122987

Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. 2010. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *Journal of Parallel and Distributed Computing (JPDC'10)* 70, 12 (2010), 1204–1219. DOI:http://dx.doi.org/10.1016/j.jpdc.2010.08.015

Michael Marchetti, Leonidas Kontothanassis, Ricardo Bianchini, and Michael L. Scott. 1995. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *International Parallel Processing Symposium (IPPS'95)*. 480–485. DOI:http://dx.doi.org/10.1109/IPPS.1995.395974

Artur Mariano, Matthias Diener, Christian Bischof, and Philippe O. A. Navaux. 2016. Analyzing and improving memory access patterns of large irregular applications on NUMA machines. In *Euromicro International Conference on Parallel, Distributed, and Network-based Processing (PDP'16)*. 382–387. DOI:http://dx.doi.org/10.1109/PDP.2016.37

Artur Mariano, Thijs Laarhoven, and Christian Bischof. 2015. Parallel (probable) lock-free hashsieve: A practical sieving algorithm for the SVP. In *International Conference on Parallel Processing (ICPP)*. 590–599. DOI:http://dx.doi.org/10.1109/ICPP.2015.68

Collin McCurdy and Jeffrey Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS'10)*. 87–96. DOI:http://dx.doi.org/10.1109/ISPASS.2010.5452060

Guillaume Mercier and Jérôme Clet-Ortega. 2009. Towards an efficient process placement policy for MPI applications in multicore environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. 104–115. DOI:http://dx.doi.org/10.1007/978-3-642-03770-2_17

Guillaume Mercier and Emmanuel Jeannot. 2011. 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'11)*. 39–49. DOI:http://dx.doi.org/10.1007/978-3-642-24449-0_7

Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. 2000a. User-level dynamic page migration for multiprogrammed shared-memory

multiprocessors. In *International Conference on Parallel Processing (ICPP'00)*. 95–103. DOI:http://dx.doi.org/10.1109/ICPP.2000.876083

Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. 2000b. UPMLIB: A runtime system for tuning the memory performance of OpenMP programs on scalable shared-memory multiprocessors. In *Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR'00)*. 85–99. DOI:http://dx.doi.org/10.1007/3-540-40889-4_7

Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. 2000c. Is data distribution necessary in OpenMP? In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'00)*. DOI:http://dx.doi.org/10.1109/SC.2000.10025

Lisa Noordergraaf and Ruud van der Pas. 1999. Performance experiences on sun's wildfire prototype. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'99)*. 1–16. DOI:http://dx.doi.org/10.1109/SC.1999.10052

Takeshi Ogasawara. 2009. NUMA-aware memory manager with dominant-thread-based copying GC. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*. 377–390. DOI:http://dx.doi.org/10.1145/1640089.1640117

OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface, Version 4.0. (2013).

Oracle. 2010. Solaris OS Tuning Features. Retrieved 2015-06-16 from http://docs.oracle.com/cd/E18659_01/html/821-1381/aewda.html

Michael Ott, Tobias Klug, Josef Weidendorfer, and Carsten Trinitis. 2008. autopin – automated optimization of thread-to-core pinning on multicore systems. In *Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG'08)*.

François Pellegrini. 1994. Static mapping by dual recursive bipartitioning of process and architecture graphs. In *Scalable High-Performance Computing Conference (SHPCC'94)*. 486–493. DOI:http://dx.doi.org/10.1109/SHPCC.1994.296682

François Pellegrini. 2010. *Scotch and Libscotch 5.1 User's Guide*. Technical Report.

Gregory F. Pfister, William C. Brantley, David A. George, Steve L. Harvey, Wally J. Kleinfelder, Kevin P. McAuliffe, Evelin S. Melton, V. Alan Norton, and Jodi Weiss. 1985. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *International Conference on Parallel Processing (ICPP'85)*. 764–771.

Guilherme Piccoli, Henrique N. Santos, Raphael E. Rodrigues, Christiane Pousa, Edson Borin, and Fernando M. Quintão Pereira. 2014. Compiler support for selective page migration in NUMA architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*. 369–380. DOI:http://dx.doi.org/10.1145/2628071.2628077

Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. 2012. Optimal task assignment in multithreaded processors: A statistical approach. *SIGARCH Computer Architecture News* 40, 1 (2012), 235–248. DOI:http://dx.doi.org/10.1145/2189750.2151002

Petar Radojković, Vladimir Cakarević, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. 2013. Thread assignment of multithreaded network applications in multicore/multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 24, 12 (2013), 2513–2525. DOI:http://dx.doi.org/10.1109/TPDS.2012.311

Christiane Pousa Ribeiro. 2011. *Contributions on Memory Affinity Management for Hierarchical Shared Memory Multi-core Platforms*. Ph.D. Dissertation. Univeristy of Grenoble.

Christiane Pousa Ribeiro, Marcio Castro, Jean-François Méhaut, and Alexandre Carissimi. 2010. Improving memory affinity of geophysics applications on NUMA platforms using Minas. In *International Conference on High Performance Computing for Computational Science (VECPAR'10)*. 279–292. DOI:http://dx.doi.org/10.1007/978-3-642-19328-6_27

Christiane Pousa Ribeiro, Jean-François Méhaut, Alexandre Carissimi, Marcio Castro, and Luiz Gustavo Fernandes. 2009. Memory affinity for hierarchical shared memory multiprocessors. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'09)*. 59–66. DOI:http://dx.doi.org/10.1109/SBAC-PAD.2009.16

Eduardo R. Rodrigues, Felipe L. Madruga, Philippe O. A. Navaux, and Jairo Panetta. 2009. Multicore aware process mapping and its impact on communication overhead of parallel applications. In *IEEE Symposium on Computers and Communications (ISCC'09)*. 811–817. DOI:http://dx.doi.org/10.1109/ISCC.2009.5202271

John Shalf, Sudip Dosanjh, and John Morrison. 2010. Exascale computing technology challenges. In *High Performance Computing for Computational Science (VECPAR'10)*. 1–25. DOI:http://dx.doi.org/10.1007/978-3-642-19328-6_1

Jaswinder Pal Singh, Truman Joe, Anoop Gupta, and John L. Hennessy. 1993. An empirical comparison of the Kendall square research KSR-1 and stanford DASH multiprocessors. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'93)*. 214–225.

Mohsen Soryani, Morteza Analoui, and Ghobad Zarrinchian. 2013. Improving inter-node communications in multi-core clusters using a contention-free process mapping algorithm. *Journal of Supercomputing* 66, 1 (apr 2013), 488–513. DOI:http://dx.doi.org/10.1007/s11227-013-0918-7

Brad Spengler. 2003. *PaX: The Guaranteed End of Arbitrary Code Execution*. Technical Report.

David Tam, Reza Azimi, and Michael Stumm. 2007. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*. 47–58. DOI:http://dx.doi.org/10.1145/1272998.1273004

Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. 2008. Data and thread affinity in OpenMP programs. In *Workshop on Memory Access on Future Processors: A Solved Problem? (MAW'08)*. 377–384. DOI:http://dx.doi.org/10.1145/1366219.1366222

The Open MPI project. 2009. Portable Linux Processor Affinity (PLPA). Retrieved 2015-06-08 from https://www.open-mpi.org/projects/plpa/.

The Open MPI Project. 2013. mpirun(1) man page (version 1.6.4). Retrieved 2016-02-08 from http://www.open-mpi.org/doc/v1.6/man1/mpirun.1.php#sect9.

Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2007. Building portable thread schedulers for hierarchical multiprocessors: The bubblesched framework. In *Euro-Par*. 42–51. DOI:http://dx.doi.org/10.1007/978-3-540-74466-5_6

Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2004. Using hardware counters to automatically improve memory performance. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'04)*. DOI:http://dx.doi.org/10.1109/SC.2004.64

Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing (JPDC)* 68, 9 (Sep 2008), 1186–1200. DOI:http://dx.doi.org/10.1016/j.jpdc.2008.05.006

Jesper Larsson Träff. 2002. Implementing the MPI process topology mechanism. In *ACM/IEEE Conference on Supercomputing (SC'02)*. DOI:http://dx.doi.org/10.1109/SC.2002.10045

François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. 2011. EZTrace: A generic framework for performance analysis. In *International Symposium on Cluster, Cloud and Grid Computing (CCGrid'11)*. 618–619. DOI:http://dx.doi.org/10.1109/CCGrid.2011.83

Ruud van der Pas. 2009. *Getting OpenMP Up To Speed*. Technical Report.

Goran Velkoski, Sasko Ristov, and Marjan Gusev. 2013. Loosely or tightly coupled affinity for matrix-vector multiplication. In *International Convention on Information & Communication Technology Electronics & Microelectronics (MIPRO'13)*. 228–233.

Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996b. Operating system support for improving data locality on CC-NUMA compute servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*. 279–289. DOI:http://dx.doi.org/10.1145/237090.237205

Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996a. *OS Support for Improving Data Locality on CC-NUMA Compute Servers*. Technical Report.

Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. 2011. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 340–349. DOI:http://dx.doi.org/10.1109/PACT.2011.65

Skef Wholey. 1991. Automatic data mapping for distributed-memory parallel computers. In *International Conference on Supercomputing (ICS'91)*. 25–34. DOI:http://dx.doi.org/10.1145/143369.143377

Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. 2008. Towards achieving fairness in the linux scheduler. *ACM SIGOPS Operating Systems Review* 42, 5 (ul J2008), 34–43. DOI:http://dx.doi.org/10.1145/1400097.1400102

Yang You, Jammes Demmel, Kenneth Czechowski, Le Song, and Richard Vuduc. 2015. CA-SVM: Communication-avoiding support vector machines on distributed systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'15)*. 847–859. DOI:http://dx.doi.org/10.1109/IPDPS.2015.117

Jidong Zhai, Tianwei Sheng, and Jiangzhou He. 2011. Efficiently acquiring communication traces for large-scale parallel applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 22, 11 (2011), 1862–1870. DOI:http://dx.doi.org/10.1109/TPDS.2011.49

Xing Zhou, Wenguang Chen, and Weimin Zheng. 2009. Cache sharing management for performance fairness in chip multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 384–393. DOI: http://dx.doi.org/10.1109/PACT.2009.40

Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2012. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 1–32. DOI: http://dx.doi.org/10.1145/2379776.2379780

Dimitrios Ziakas, Allen Baum, Robert A. Maddox, and Robert J. Safranek. 2010. Intel quickpath interconnect - architectural features supporting scalable system architectures. In *Symposium on High Performance Interconnects (HOTI'10)*. 1–6. DOI: http://dx.doi.org/10.1109/HOTI.2010.24

Received June 2016; revised September 2016; accepted October 2016