Locality vs. Balance: Exploring Data Mapping Policies on NUMA Systems

Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux
Informatics Institute
Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
{*mdiener, ehmcruz, navaux*}*@inf.ufrgs.br*

***Abstract*—In parallel architectures that have a Non-Uniform Memory Access (NUMA) behavior, the mapping of memory pages to NUMA nodes influences the performance of parallel applications. In order to improve traditional data mapping policies, two basic strategies can be employed: optimizing locality or balance of memory accesses. In a locality-based policy, memory pages are mapped to nodes that access the page the most. In a balance-based policy, memory pages are mapped such that the number of memory accesses resolved by each memory controller is similar.**

In this paper, we perform an in-depth exploration of these data mapping policies on the performance of parallel applications. We introduce metrics that describe their memory access behavior and evaluate their suitability for data mapping. We also present new mapping policies that focus on locality, balance or both. These policies were evaluated on three different NUMA architectures with applications from the NAS-OMP and PARSEC benchmark suites. Results show that the performance improvements of each policy depend on the characteristics of the applications and machines. Choosing the wrong policy can actually hurt the performance compared to the default first-touch mapping. Compared to traditional mapping policies and to policies that only focus on either locality or balance, taking into account both locality and balance results in the highest improvements. Furthermore, it avoids the performance reduction caused by the wrong data mapping.

***Keywords*-Data mapping; NUMA; Locality; Load balance;**

I. Introduction

Modern parallel architectures contain multiple memory controllers per machine, which leads to a non-uniform memory access (NUMA) behavior. Each memory controller forms a NUMA node. The available system memory is split between the nodes, such that each node can access its local memory directly, but has to access memory that is located on remote nodes through an interconnection. Memory accesses to a local memory controller have a lower latency and higher bandwidth than accesses to remote nodes. Therefore, to achieve optimal performance on NUMA machines, this property needs to be taken into account when mapping memory pages to NUMA nodes [1].

Most mechanisms to improve data mapping on NUMA architectures rely on optimizing *locality*, that is, they place pages on the NUMA node with the most accesses to the page to minimize the number of memory accesses to remote nodes [2], [3]. Locality is mostly important when the memory access time to access a remote NUMA node is much higher than an access to a local node. However, when the difference between local and remote accesses is low, or the overhead to access a remote node is hidden by cache memories or cache line prefetchers [4], the influence of memory locality becomes lower. In these cases, another important factor to be considered is the *load* of the memory controllers. The goal is to *balance* the memory accesses between the memory controllers such that all controllers handle a similar number of requests [5], [6], [7].

Traditional data mapping policies, such as *first-touch*, *next-touch* and *interleave*, do not perform any analysis of the application [8], resulting in mappings that optimize neither locality nor balance. Newer policies analyze the runtime behavior of the applications when choosing the best mapping for each page [9]. To optimize locality, the policies determine which NUMA node performed the majority of memory accesses to each page. To optimize memory controller load balance, they check the number of memory accesses requested to each memory controller. However, these previous mapping mechanisms are not able to achieve optimal performance improvements and can result in a performance degradation compared to a first-touch policy.

In this paper, we make the following main contributions to the data mapping problem:

1. We introduce metrics to determine the suitability of locality-based and balance-based data mapping policies for parallel applications.

2. We develop two new data mappings policies, *balanced* and *mixed*, that focus on memory balance and combined locality/balance metrics, respectively.

3. We analyze two sets of parallel applications to determine their best mappings and evaluate their performance improvements on three different NUMA architectures. Our mixed policy achieves the highest average improvements while never hurting performance.

The rest of this paper is organized as follows. The next section introduces the metrics used to describe the memory access behavior of parallel applications. Section III describes our data mapping policies. Section IV shows an analysis of parallel applications with the metrics and the data mapping policies. Section V shows the performance results we obtained with the data mapping policies. Section VI discusses related work. Finally, Section VII presents our conclusions.

1066-6192/15 $31.00 © 2015 IEEE
DOI 10.1109/PDP.2015.11

II. Memory Accesses of Parallel Applications

In this section, we introduce metrics to describe the memory access behavior of parallel applications on NUMA systems. We present metrics for the suitability of applications for locality-based and balance-based policies. We also discuss dynamic application behavior.

A. Memory Access Locality

Previous research [10] has proposed *exclusivity* as a metric to analyze the suitability of a parallel application for locality-based mapping. The potential of a page for this type of mapping is proportional to the number of memory accesses from a single NUMA node. That is, if a page is mostly accessed from the same node, it has more potential for locality-based mapping than a page that is accessed from several nodes. We call the highest number of memory accesses to a page from a single node compared to the number of accesses from all nodes the *page exclusivity* E_{Page}. The higher the exclusivity of a page, the higher its potential for locality-based mapping. E_{Page} is calculated with Equation 1, where $MemAcc[p][n]$ is the number of memory accesses to page p from node n, and N is the number of NUMA nodes. The max function returns the highest number of memory accesses to a page from a node.

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

The exclusivity is minimal when a page has exactly the same number of accesses from all nodes. In this case, the exclusivity is given by $1/N$. The exclusivity achieves its maximum value of 1 when all accesses to the corresponding page originate from the same NUMA node.

Besides the exclusivity of a page, the number of memory accesses to it has to be taken into account as well. The higher the number of accesses, the higher the potential for data mapping. To calculate the *application exclusivity* E_{App}, we scale the exclusivity of each page with the number of memory accesses to it, and divide this value by the total number of memory accesses. This operation is shown in Equation 2, where P is the total number of pages. Like the page exclusivity, the minimum and maximum of the application exclusivity is $1/N$ and 1, respectively.

$$E_{App} = \frac{\sum_{p=1}^{P} (E_{Page}[p] \cdot \sum_{n=1}^{N} MemAcc[p][n])}{\sum_{p=1}^{P} \sum_{n=1}^{N} MemAcc[p][n]} \tag{2}$$

B. Memory Access Balance

Our second metric, *memory balance*, is used to analyze the suitability of a parallel application for balance-based data mapping. Memory balance is important in applications where the memory accesses are performed in such a way that some memory controllers are overloaded, while others are idle. To measure the *page balance* B_{Pages}, we introduce a new vector, $NodePages$, which consists of N elements. Each element $NodePages[n]$ contains the number of pages mapped to node n. Equation 3 calculates the page balance for all the pages that an application uses. The balance is maximum (with a value of 1) when all nodes store the same number of pages. The balance is minimum (with a value of 0) when one node stores all the pages.

$$B_{Pages} = 1 - \frac{max(NodePages) - min(NodePages)}{\sum_{n=1}^{N} NodePages[n]} \tag{3}$$

Since not all pages have the same number of memory accesses, maximizing load balance considering Equation 3 may not improve the balance of memory controllers. To achieve an improved balance, we need to consider the number of memory accesses to each page. We store the number of memory accesses to each NUMA node in the $NodeAcc$ vector, as shown in Equation 4.

$$NodeAcc[n] = \sum_{p=1}^{P} MemAcc[p][n] \tag{4}$$

We can then calculate the *memory access balance* B_{Acc} of the application with Equation 5.

$$B_{Acc} = 1 - \frac{max(NodeAcc) - min(NodeAcc)}{\sum_{n=1}^{N} NodeAcc[n]} \tag{5}$$

If the memory accesses are equally distributed among the nodes, B_{Acc} is maximized, with a value of 1. If all memory accesses are satisfied from a single node, B_{Acc} is 0.

C. Dynamic Behavior

To analyze the dynamic memory access behavior of the applications, we measure how often pages need to migrated during execution in order to maintain a page always on the node with the most accesses to it. This is done by dividing the execution of each application into time slices of a fixed size and calculating the number of accesses from each node to each page. If the page is not located on the node with the most accesses during the time slice, we store a migration event and update the current node of the page. By counting the number of migration events during the whole execution, it is possible to determine the dynamicity of the memory access behavior of the application.

III. Data Mapping Policies

To evaluate the impact of data mapping on the performance of the parallel applications, we compare several mapping policies to the default first-touch policy. These policies will be presented in this section.

The following mapping policies were evaluated: *random*, *interleave*, *locality*, *balanced*, *mixed*. The interleave policy is available in many operating systems, such as via the `numactl` tool in Linux [11]. Locality is a policy similar to previous mechanisms that focus on locality improvements in NUMA systems [9], [10]. Balanced and mixed are two new mapping approaches. Balanced distributes pages in such a way that all memory controllers resolve the same number of memory accesses. We also introduce a mixed policy, which presents a trade-off between locality and balance. In the following description of the mapping policies, $node[p]$ represents the NUMA node of a page p, and N represents the total number of nodes.

1) Random: In the *random* mapping, each page gets assigned randomly to a NUMA node. We use the random mapping to validate the importance of data mapping.

2) Interleave: In the traditional *interleave* policy, pages get assigned to NUMA nodes according to their address. Usually, the lowest bits of the page address are used to determine the node. Equation 6 describes this behavior, where $addr(p)$ returns the page address of page p.

$$node[p] = addr(p) \bmod N \tag{6}$$

This policy ensures that memory accesses to consecutive addresses are distributed among the nodes. A similar policy, a round-robin mapping of pages to nodes, showed almost the same results as interleave and will therefore not be discussed in this paper.

3) Locality: In the *locality* policy, each page gets assigned to the NUMA node with the most memory accesses to the page. This policy ensures that the highest possible number of memory accesses are resolved by the local NUMA node. However, it does not take the balance between nodes into account and can lead to overloaded nodes. Equation 7 describes this behavior, where $maxnode(p)$ returns the node with the most accesses to page p.

$$node[p] = maxnode(p) \tag{7}$$

4) Balanced: In the *balanced* mapping, we maximize the balance between the nodes, leading to an optimal load, while still taking into account the locality of each page. Algorithm 1 calculates this mapping. In the algorithm, we first sort the list of pages by the number of accesses. Then, we map each page to the node with the most accesses to the page that is not overloaded.

Balanced and locality are opposite policies: Balanced optimizes memory access balance over locality, while the locality policy maximizes local accesses at the expense of memory balance. For this reason, these policies can determine which of the metrics is more important for the performance of parallel applications.

5) Mixed: In the *mixed* mapping, we combine locality and balance. For each page, if its exclusivity E_{Page} is above a threshold $minExcl$, we allocate the page on the node with the highest number of accesses, as in the locality policy. If the exclusivity is below the threshold, we allocate the page on the node given by the interleave mapping. This behavior is shown in Equation 8.

$$node[p] = \begin{cases} maxnode(p), & \text{if } E_{Page}[p] > minExcl \\ addr(p) \bmod N, & \text{otherwise} \end{cases} \tag{8}$$

Algorithm 1. Calculate the *Balanced* mapping.

```
Input: p.AccNode[n]: number of accesses to page p from
       node n; N: number of NUMA nodes
Output: node[p]: mapping of pages to nodes
1  sort pages by number of accesses;
2  TotalAcc = 0;
3  for each Page p do
4      p.TotalAcc = 0;
5      for each Node n do
           // total memory accesses per page
6          p.TotalAcc += p.AccNode[n];
       // total memory accesses
7      TotalAcc += p.TotalAcc;
8  for each Node n do
9      AccNode[n] = 0;
10 for each Page p do
       // find node with the most accesses to p
          that is not overloaded
11     n = node with the most accesses to p;
12     while AccNode[n]/TotalAcc > 1/N do
13         n = node with the next highest accesses to p;
       // map page p to node n and update values
14     node[p] = n;
15     AccNode[n] += p.TotalAcc;
```

This mapping provides a trade-off between locality and balance. We experimented with various values for $minExcl$, between 80% and 95%. For the applications we evaluated, the optimal results were achieved with 90%, but were not very sensitive to the value.

These 5 mapping policies will be compared to a *first-touch* policy, where each page gets allocated on the node that accessed the page for the first time. This policy is the default for most operating systems, such as Linux.

IV. Analysis of Benchmark Behavior

In this section, we analyze two sets of parallel benchmarks with the locality and balance metrics described in Section II to determine their suitability for the data mapping policies introduced in Section III.

Data mapping can be influenced by the assignment of threads to NUMA nodes for pages that are accessed by several threads. If a page is accessed by multiple threads, the number of remote accesses is reduced by assigning the threads to the same NUMA node. Therefore, we perform a static assignment of threads to NUMA nodes considering the page sharing [12].

A. Methodology of the Analysis

1) Benchmarks: We selected two parallel benchmark suites for the evaluation. *NAS-OMP* [13] is the OpenMP implementation of the NAS Parallel Benchmarks (NPB), which consists of applications from the HPC domain. Benchmarks were executed with the *C* input size. The average memory consumption per application is 6.1 GByte. The *PARSEC* benchmark suite [14] focuses on emerging parallel workloads for shared memory architectures. All benchmarks were executed with the *native* input set. The average memory consumption per application is 4.8 GByte.

2) Environment: For our discussion, we show the results for the applications with 64 threads on a simulated 4 NUMA node machine with a page size of 4 KByte. To determine the memory access behavior, we built a memory tracer using the PIN Dynamic Binary Instrumentation tool [15]. For every time slice, the tool outputs the page usage of the whole application. Despite the overhead caused by the tracing (about 10x of execution time), it is still feasible to characterize very large applications.

B. Evaluation of Benchmark Behavior

1) Exclusivity: The application exclusivity E_{App} is presented in Fig. 1 for two configurations: (i) our baseline, with 64 threads on 4 NUMA nodes (threads are assigned to NUMA nodes with the mechanism described in the beginning of this section), (ii) 64 NUMA nodes (1 thread per node), to show the inherent exclusivity of the applications. Results are further divided into two page sizes, 4 KByte (which is the default page size in x86) and 2 MByte.

Most applications have a high exclusivity, even for 64 NUMA nodes, which demonstrates the importance of a locality-based mapping policy. Several benchmarks, such as EP, Canneal and Streamcluster, have a lower exclusivity however. Even when increasing the page size or the number of NUMA nodes, the exclusivity only decreases slightly for most benchmarks. A slight decrease of exclusivity is expected for larger pages, as there will be more falsely shared data. When increasing the number of NUMA nodes, more shared pages will be located on different nodes, lowering the exclusivity. On average, the application exclusivity for the 4 KByte and 2 MByte page sizes is 84% / 78% (for 4 nodes), and 73% / 65% (for 64 nodes), respectively.

2) Balance: The values of the memory balance metrics are shown in Table I. Fig. 2 visualizes the page and memory

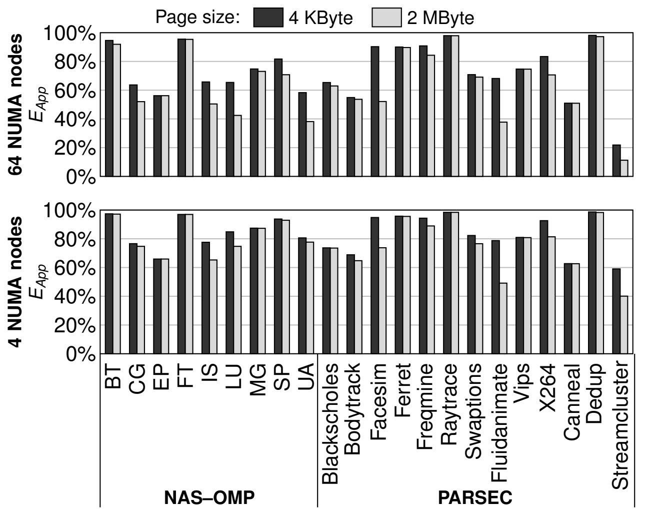

Figure 1. Application exclusivity E_{App} for 64,4 nodes and two page sizes.

Table I
MEMORY BALANCE METRICS B_{Pages} AND B_{Acc} (IN %).

	Metric	BT	CG	EP	FT	IS	LU	MG	SP	UA	Blackscholes	Bodytrack	Facesim	Ferret	Freqmine	Raytrace	Swaptions	Fluidanimate	Vips	X264	Canneal	Dedup	Streamcluster	**Average**
FirstT.	Pages	80	99	0	99	99	81	92	64	99	6	30	2	39	93	0	73	35	98	40	0	47	0	**54**
	Acc	46	96	31	98	26	47	73	48	47	1	9	5	36	52	69	36	40	68	35	4	46	1	**42**
Local.	Pages	80	99	99	99	99	76	99	78	99	44	71	50	54	95	10	85	83	92	26	4	75	44	**71**
	Acc	79	99	62	99	90	86	87	81	95	80	60	97	22	81	79	75	81	74	66	49	40	24	**73**
Inter.	Pages	99	99	99	99	99	99	99	99	99	99	99	99	99	99	99	96	99	99	99	99	99	99	**99**
	Acc	95	99	89	99	83	86	87	99	90	70	88	90	97	94	93	87	84	86	86	86	95	91	**89**
Balan.	Pages	81	99	64	99	99	76	99	85	98	53	49	53	45	91	48	69	63	69	58	28	56	83	**71**
	Acc	99	96	90	99	89	99	87	99	95	81	96	99	99	99	99	96	99	99	83	82	99	99	**95**
Mixed	Pages	83	99	99	99	99	90	91	81	98	44	88	50	57	98	22	86	82	99	43	97	78	96	**81**
	Acc	80	99	83	99	90	78	82	84	90	79	80	97	30	90	80	89	83	93	59	68	42	88	**80**

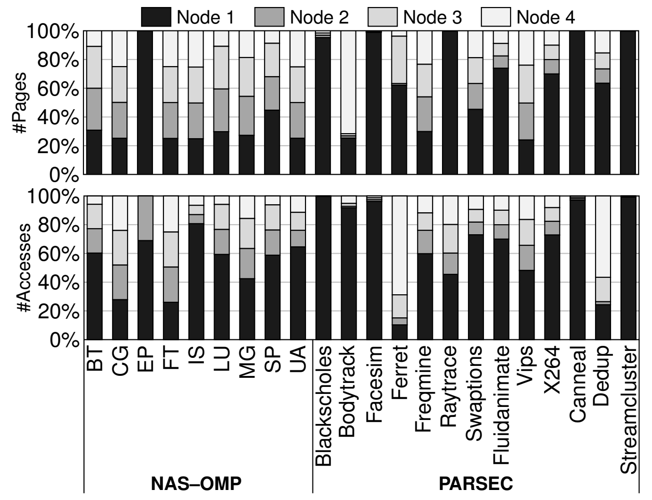

Figure 2. Page and memory access balance for the first-touch policy.

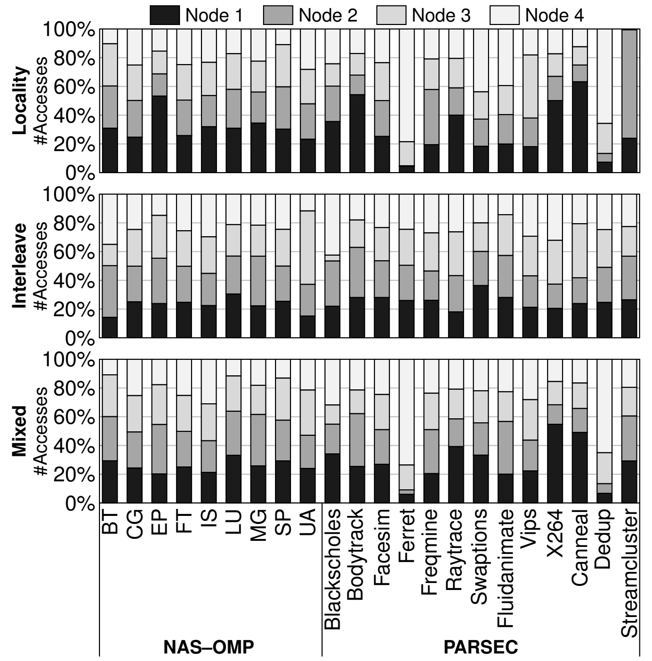

Figure 3. Memory access balance for locality, interleave and mixed.

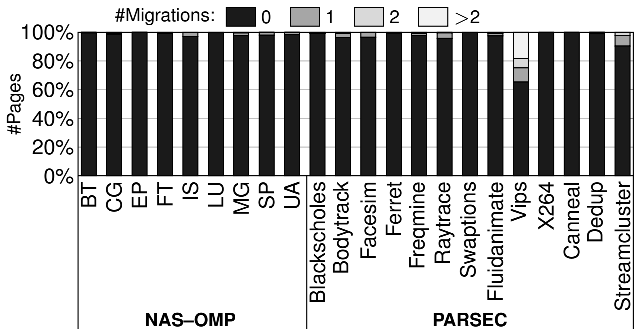

Figure 4. Number of page migrations for 1 s time slices, in order to maintain every page on the node with the most accesses to it on each slice.

access balance for the first-touch policy. For all NAS-OMP benchmarks except EP, as well as some PARSEC benchmarks, the policy distributes the pages well, with high values for B_{Pages}. However, comparing the results for the page and memory access balance shows that a high page balance does not necessarily result in a high memory access balance. For example, UA with first-touch has a page balance of more than 99%, but a memory access balance of only 47%, with 60% of memory accesses coming from NUMA node 1.

The other mapping policies result in a much higher memory access balance, as shown in Table I and Fig. 3. It is important to mention that the locality policy already results in a much better memory access balance than the first-touch policy. For example, the memory access balance of UA increases from 47% with first-touch to 95% with locality. The interleave and balanced policies result in the highest values for the balance metrics for most benchmarks, as expected. The mixed policy results in a balance that is between the interleave and locality policies.

3) Dynamic Behavior: The dynamic memory access behavior of the applications in terms of page migrations during the execution as described in Section II-C is shown in Fig. 4. We use a time slice length of 1 s, below which no improvements from page migrations can be expected. In the figure, we depict how often the pages need to be migrated in order to maintain each page on the NUMA node with the most accesses to it on each time slice. The results show that most of the applications have a very static behavior. Only Vips has a significant number of migrations, although even in this application more than 75% pages need to be migrated only zero or one times. In the other applications, almost no pages need to be migrated. For this reason, mapping mechanisms for most applications do not need to take the dynamic behavior into account.

V. Performance Evaluation of Data Mapping

In this section, we evaluate the performance improvements that can be achieved with the data mapping policies.

A. Methodology of the Performance Experiments

We execute the benchmarks on three NUMA machines: *Itanium*, *Xeon* and *Opteron*. The *Itanium* machine represents a traditional NUMA machine based on the SGI Altix 450 platform. It consists of 2 NUMA nodes, each with 2 dual-core Intel Itanium 2 processors and a custom SGI interconnection. The *Xeon* machine represents a newer generation NUMA machine with high-speed interconnections. It consists of 4 NUMA nodes with 1 eight-core Intel Xeon processor each and a QPI interconnection. The *Opteron* machine represents a recently introduced generation of NUMA machines with multiple on-chip memory controllers. It consists of 4 AMD Opteron processors, forming 8 NUMA nodes in total. The Itanium machine can execute up to 8 threads concurrently, while Xeon and Opteron can execute 64 threads concurrently. The NUMA factors of the machines, defined as the ratio of the latency for a remote access and a local access, were measured with Lmbench [16]. Their values are 2.1, 1.5 and 2.8 for Itanium, Xeon and Opteron, respectively. All machines run version 3.8 of the Linux kernel. A summary of the machines is shown in Table II.

Table II
Overview of the machines used in the evaluation.

Name	Nodes	Processors	Page size
Itanium	2	4x Intel Itanium 2 9030, 2 cores	16 KByte
Xeon	4	4x Intel Xeon X7550, 8 cores, SMT	4 KByte
Opteron	8	4x AMD Opteron 6386, 8 cores, SMT	4 KByte

We implemented our mapping mechanisms in a Linux kernel module. On application startup, the module reads the previously generated data mapping from a file and maps each page to the assigned NUMA node on the first access to the page. Pages are not migrated during execution. Since the mappings were generated with the help of the memory tracer presented in Section IV-A2, we require that the page addresses do not change between executions. Therefore, all static memory allocations, and dynamic memory allocations in the sequential phase of the benchmarks (for example, during initialization) are supported, which is true for most of the applications from the NAS-OMP and PARSEC benchmark suites. We selected 13 benchmarks from these two suites for the performance experiments due to space limitations. All benchmarks were executed with the maximum number of threads that each machine can execute in parallel (8 threads for Itanium and 64 threads for Xeon/Opteron).

We evaluate the data mapping policies and compare them to the baseline, the first-touch policy of Linux. Threads were pinned to the execution cores with the mechanism discussed in Section IV. We show the average execution time of 10 executions for each benchmark. Since the mappings are static, the standard deviation was very low for all experiments (less than 1% of the execution time).

B. Performance Results

The results for the execution time (normalized to the baseline, the first-touch policy of Linux) for the three evaluated architectures are shown in Fig. 5, 6 and 7.

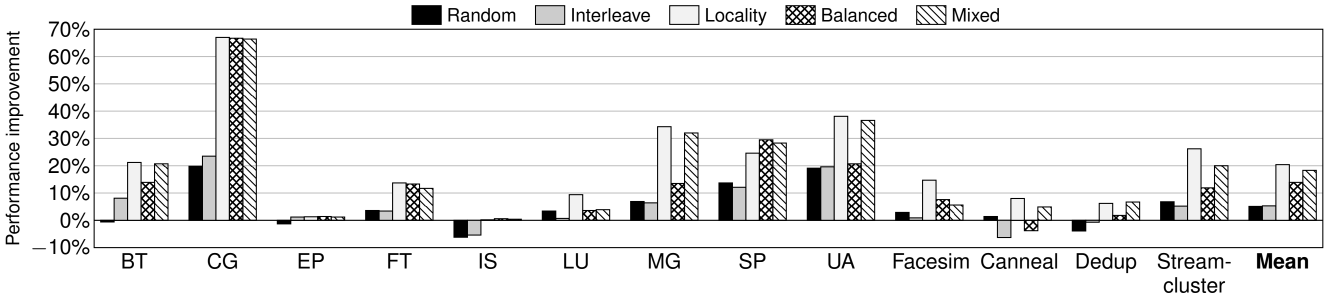

Figure 5. Performance improvements on ***Itanium*** with different data mapping policies, compared to the first-touch mapping.

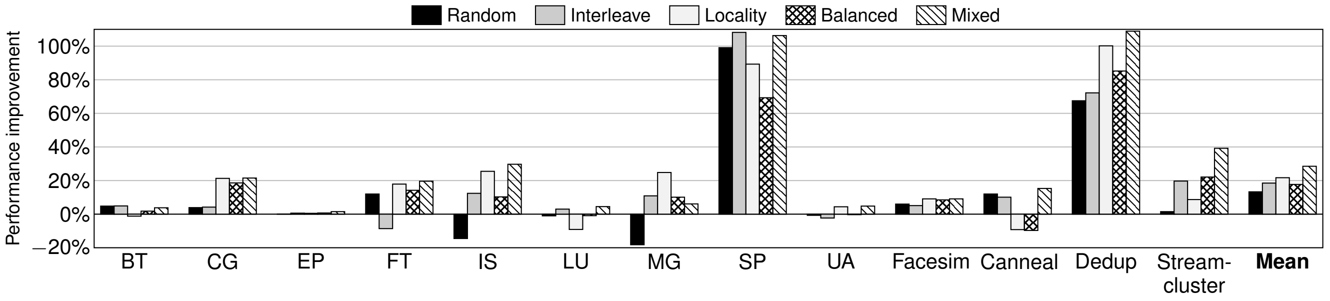

Figure 6. Performance improvements on ***Xeon*** with different data mapping policies, compared to the first-touch mapping.

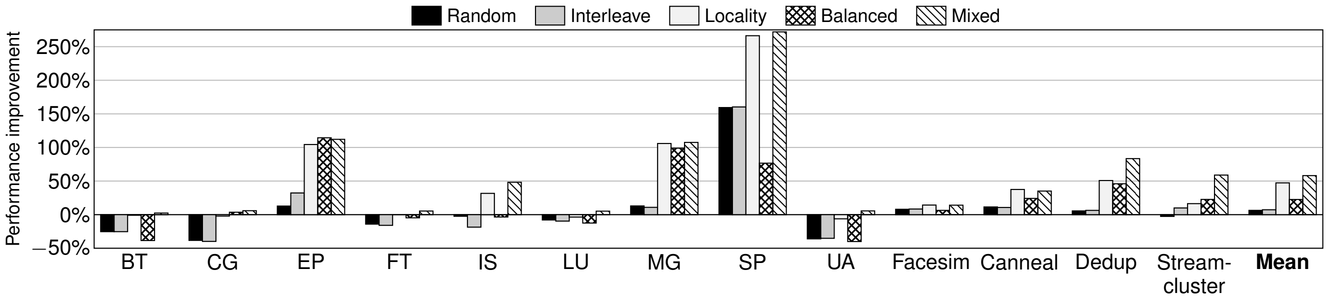

Figure 7. Performance improvements on ***Opteron*** with different data mapping policies, compared to the first-touch mapping.

1) Itanium: Fig. 5 shows the results for the Itanium machine. The highest improvements were achieved for the CG benchmark, of up to 67% for locality, balanced and mixed. Due to Itanium's relatively slow interconnection, the interleave policy achieves generally lower improvements than the locality and mixed policies. Since it still takes locality into account, the balanced policy achieves higher improvements than Interleave. As the machine has only a small number of NUMA nodes (2), even the random policy improves performance in many cases, as it has a higher chance of mapping the page to the local node. The average performance improvements are: 5.2% with random, 5.7% with interleave, 20.9% with locality, 14.4% with balanced and 19.4% with mixed. The fact that the improvements with mixed are lower than the improvements with locality (even when using a 99% minimum exclusivity for mixed) shows that on this machine, locality is more important than balance.

2) Xeon: Fig. 6 shows the results for the Xeon machine. Due to the higher number of threads and NUMA nodes, performance improvements from data mapping are higher than on Itanium, despite Itanium's slower interconnection. For the same reason, the balance policies achieve improvements closer to the locality policies. The SP benchmark achieved the highest improvements, even for the random policy, of up to 108%. For the other benchmarks, the random policy can not reduce execution time significantly or even increases it (up to 37% for MG). On average, performance was improved by 8.7% with random, 15.8% with interleave, 18.5% with locality, 14.6% with balanced and 23.7% with mixed. The results show that improving only locality is not enough to achieve optimal improvements. For several benchmarks (BT, LU and Canneal), the locality policy actually reduces performance compared to first-touch.

3) Opteron: Fig. 7 shows the results for the Opteron machine. Due to its large number of NUMA nodes (8), the highest improvements were achieved on this machine. In many cases, the random and interleave policies reduce performance. Balancing memory accesses is still important though, as shown by the higher improvements for mixed than for locality. As before, the locality policy increases execution time slightly compared to the baseline for some benchmarks (CG, LU, and UA). Similar to the Xeon machine, the highest improvements were achieved for SP, up to 272% with the mixed policy. On average, performance was improved by 6.3% with random, 7.2% with interleave, 50.0% with locality, 23.9% with balanced and 61.8% with mixed.

4) Summary and Discussion: Several important conclusions can be drawn from our performance results.

First-touch often has a negative impact on performance. Compared to most other policies, the first-touch policy of modern operating systems often has a negative impact on the performance of parallel applications. In many cases, even a random assignment of pages to NUMA nodes outperforms first-touch. The reason for this behavior is that in many parallel applications, one thread initializes the data and forces page allocation on a single NUMA node, leading to an increased number of memory accesses to that node, while a random policy can balance the memory access load more equally among the nodes.

Locality is still more important than balance. Regarding the importance of locality and balance, results depend on the hardware architecture. On traditional NUMA systems with a relatively slow interconnection, such as our Itanium machine, memory access locality is the most important metric for performance improvements. For modern NUMA architectures, the importance of balancing the memory accesses between memory controllers is becoming more important, although locality is still the more important metric to optimize, as evidenced by the results of the interleave and balanced policies compared to the locality policy. An important reason for this result is that improving locality also improves balance for most applications.

Mixed policies can provide the highest improvements. Neither locality nor balance are able to improve performance in all cases and actually reduce performance significantly in some instances. Taking both locality and balance into account (as done by our mixed policy) when mapping pages to NUMA nodes achieves the highest results and also avoids the performance decrease of the other policies. In this way, highly exclusive pages can benefit from the locality, while shared pages are distributed among the nodes to balance the memory accesses between memory controllers.

VI. Related Work

Traditional data mapping strategies, such as *first-touch* and *next-touch* [8], have been used by operating systems to allocate memory on NUMA machines. In the case of first-touch, a memory page is mapped to the node of the thread that causes its first page fault. This strategy affects the performance of other threads, as pages are not migrated during execution. When using next-touch, pages are migrated between nodes according to memory accesses to them. However, if the same page is accessed from different nodes, next-touch can lead to excessive data migrations. Problems also arise in the case of thread migrations between nodes [17]. The *interleave* policy mentioned before is available via the `numactl` tool [11]. Autonuma [18] performs a simple next-touch page migration by introducing page faults during the execution of the application to migrate a page to the NUMA node that causes the next page fault.

The Carrefour mechanism [7] profiles parallel applications during execution and migrates pages based on their access pattern. Memory accesses are sampled using hardware counters available on AMD architectures. Due to the runtime overhead of the mechanism, it is enabled only for a very small subset of pages (30,000), which limits the improvements that Carrefour can achieve. Using a similar machine (Opteron), we achieved higher performance improvements, especially for the NAS-OMP benchmarks. kMAF [10] uses page faults of parallel applications to determine their memory access behavior during execution, and migrates threads and data to optimize locality, but not balance.

Marathe et al. [9], [19] use the performance monitoring unit (PMU) of the Itanium 2 processor to sample memory addresses accessed by the running application. The samples are taken from long latency load operations and TLB misses. On the software level, the samples are captured to generate a memory trace. This trace is analyzed to map the pages to the NUMA nodes such that the locality is maximized. Similarly, Tikir et al. [20] and Zhou et al. [21] use TLB statistics to guide data mapping. These profiling mechanisms consider only a small sample of the memory accesses, which decreases the accuracy of the mapping, and do not take the load of the memory controllers into account. Awashti et al. [5] present a page migration mechanism that uses queuing delays and row-buffer hit rates from memory controllers. This work only considers the load balance between the memory controllers and does not improve locality to reduce the number of remote memory accesses.

The Memory Affinity Interface (MAi) [17] allows data mapping on NUMA platforms. Each application must be specifically programmed for MAi. Minas [2] is a framework for data mapping that uses MAi to control data mapping, and implements a preprocessor that analyzes the source code of an application to determine a suitable data mapping for a target architecture. Afterwards, instrumentation code to perform data mapping is inserted into the application. Other proposals suggest changing the application source code to optimize it for NUMA systems [4].

VII. Conclusions

In modern parallel architectures with NUMA characteristics, data mapping decisions have a high influence on the performance of parallel applications. The reason for this influence is the locality and balance of memory accesses over the NUMA nodes within the system. Several policies were proposed to handle the mapping of pages to NUMA nodes. Most traditional policies are very simple and do not consider the different characteristics of applications and architectures. It is therefore necessary to develop policies that take into account the memory access behavior of the application as well as the system properties.

In this paper, we introduced metrics to describe the memory access behavior of parallel applications and determine their suitability for data mapping. We developed a mixed mapping policy that takes both locality and balance into account. Several data mapping policies were evaluated with applications from the NAS-OMP and PARSEC benchmark suites on three different NUMA machines. Results show that large performance improvements can be achieved with data mapping in NUMA architectures, and that locality-based policies on average improve the performance more than policies that are based on memory balance. Generally, architectures with high latency remote memory accesses benefit more from locality than architectures in which the access time difference to local and remote memories is low. However, no previous mapping policy is the best to handle all applications and architectures and can actually result in a performance degradation. Our mixed policy presented the highest overall gains and never reduced performance compared to the default first-touch policy.

For the future, we intend to apply our mapping policies to mechanisms that perform an online migration of pages between NUMA nodes, which removes the need for a profiling step before application execution.

Acknowledgment

This work was partially supported by CNPq.

References

[1] C. McCurdy and J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[2] C. Ribeiro, M. Castro, J.-F. Mehaut, and A. Carrissimi, "Improving memory affinity of geophysics applications on NUMA platforms using Minas," in *High Performance Computing for Computational Science (VECPAR)*, 2010.

[3] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on NUMA architectures," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, 2014.

[4] Z. Majo and T. R. Gross, "(Mis)Understanding the NUMA Memory System Performance of Multithreaded Workloads," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[5] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers," in *Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[6] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A Case for NUMA-aware Contention Management on Multicore Systems," in *USENIX Annual Technical Conference*, 2010.

[7] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[8] H. Löf and S. Holmgren, "affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System," in *Supercomputing (SC)*, 2005.

[9] J. Marathe, V. Thakkar, and F. Mueller, "Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces," *Journal of Parallel and Distributed Computing*, vol. 70, no. 12, 2010.

[10] M. Diener, E. H. M. Cruz, P. O. A. Navaux, A. Busse, and H.-U. Heiß, "kMAF: Automatic Kernel-Level Management of Thread and Data Affinity," in *Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[11] A. Kleen, "An NUMA API for Linux," 2004. [Online]. Available: http://andikleen.de/numaapi3.pdf

[12] M. Diener, E. H. M. Cruz, and P. O. A. Navaux, "Communication-Based Mapping Using Shared Pages," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[13] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS Parallel Benchmarks and Its Performance," 1999.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

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

[16] L. McVoy and C. Staelin, "Lmbench: Portable Tools for Performance Analysis." in *USENIX Annual Technical Conference*, 1996.

[17] C. P. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L. G. Fernandes, "Memory Affinity for Hierarchical Shared Memory Multiprocessors," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2009.

[18] J. Corbet, "Toward better NUMA scheduling," 2012. [Online]. Available: http://lwn.net/Articles/486858/

[19] J. Marathe and F. Mueller, "Hardware Profile-guided Automatic Page Placement for ccNUMA Systems," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[20] M. M. Tikir and J. K. Hollingsworth, "Hardware monitors for dynamic page migration," *Journal of Parallel and Distributed Computing*, vol. 68, no. 9, Sep. 2008.

[21] J. Zhou and B. Demsky, "Memory management for many-core processors with software configurable locality policies," *ACM SIGPLAN Notices*, vol. 47, no. 11, 2012.