Contents lists available at ScienceDirect

Future Generation Computer Systems

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

CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters

Ruben Laso [a,*], Oscar G. Lorenzo [a], José C. Cabaleiro [a,b], Tomás F. Pena [a,b], Juan Ángel Lorenzo [c], Francisco F. Rivera [a,b]

[a] CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain
[b] Departamento de Electrónica e Computación, Universidade de Santiago de Compostela, Santiago de Compostela, Spain
[c] ETIS, UMR 8051, CY Cergy Paris Université, ENSEA, CNRS, France

ARTICLE INFO

Article history:
Received 18 June 2021
Received in revised form 24 September 2021
Accepted 11 November 2021
Available online 25 November 2021

Keywords:
NUMA
Scheduling
Thread migration
Memory migration
Hardware counters

ABSTRACT

This paper introduces two novel algorithms for thread migrations, named CIMAR (Core-aware Interchange and Migration Algorithm with performance Record –IMAR–) and NIMAR (Node-aware IMAR), and a new algorithm for the migration of memory pages, LMMA (Latency-based Memory pages Migration Algorithm), in the context of Non-Uniform Memory Access (NUMA) systems. This kind of system has complex memory hierarchies that present a challenging problem in extracting the best possible performance, where thread and memory mapping play a critical role. The presented algorithms gather and process the information provided by hardware counters to make decisions about the migrations to be performed, trying to find the optimal mapping. They have been implemented as a user space tool that looks for improving the system performance, particularly in, but not restricted to, scenarios where multiple programs with different characteristics are running. This approach has the advantage of not requiring any modification on the target programs or the Linux kernel while keeping a low overhead.

Two different benchmark suites have been used to validate our algorithms: The NAS parallel benchmark, mainly devoted to computational routines, and the LevelDB database benchmark focused on read–write operations. These benchmarks allow us to illustrate the influence of our proposal in these two important types of codes. Note that those codes are state-of-the-art implementations of the routines, so few improvements could be initially expected. Experiments have been designed and conducted to emulate three different scenarios: a single program running in the system with full resources, an interactive server where multiple programs run concurrently varying the availability of resources, and a queue of tasks where granted resources are limited. The proposed algorithms have been able to produce significant benefits, especially in systems with higher latency penalties for remote accesses. When more than one benchmark is executed simultaneously, performance improvements have been obtained, reducing execution times up to 60%. In this kind of situation, the behaviour of the system is more critical, and the NUMA topology plays a more relevant role. Even in the worst case, when isolated benchmarks are executed using the whole system, that is, just one task at a time, the performance is not degraded.

1. Introduction

Current computer architectures feature complex computing and memory hierarchy structures. Extracting the maximum performance from these increasingly complex machines requires a big investment in time for programmers, who have to take care of different details. In particular, memory operations require special care due to their expensiveness, for which closing the distance, in terms of latency and bandwidth, between threads and data is critical. In this context, optimal mapping of the processes and their data plays a key role in performance [1,2]. This is a traditional challenge in High-Performance Computing and a continuously ongoing research work.

The main complexity resides in the fact that the behaviour can dynamically change when multiple processes from different tasks are in execution, each with a different number of threads and different resource requirements that may vary with time. Additionally, the number of possible combinations for mapping and scheduling increases with the number of cores and threads

* Corresponding author.
E-mail address: r.laso@usc.es (R. Laso).

https://doi.org/10.1016/j.future.2021.11.008

present in the system. Therefore, decisions in runtime are important to improve performance by adapting to the changing behaviour. This problem is already challenging when a single parallel task is executed, but it is even more in multitasking environments where more benefits could be obtained. The problem can be formulated as an optimisation problem, being the system performance the target function to be optimised. Unlike other optimisation problems, there is no room for testing solutions while searching for optimal placement. Each wrong solution tested causes an overhead that cannot be recovered later, so algorithms have to find a complex balance between being conservative, to make sure that things do not get worse, and performing thread and memory migrations to improve the performance of the system.

In NUMA systems, this challenge is particularly critical. The distance between threads and memory, in terms of NUMA nodes, can change substantially the latency of the memory operations, affecting the performance as a consequence. The latency of a thread accessing data residing in the same NUMA node (local access) is lower than when accessing remote nodes. So, closing the distance between threads and data is preferable in most cases, even though the reality is much more complex and other phenomena might take place, like memory contention and interconnect congestion [3].

To implement a thread or memory migration algorithm, certain criteria based on performance must be followed. Several models have been proposed to understand the performance of a code running on a given system [4–7]. The Roofline Model (RM) [8] is the most popular performance model among HPC researchers, since it offers a good balance between simplicity and descriptiveness. It uses just the number of FLOPS (Floating Point Operations per Second) and the OI (Operational Intensity), defined as the number of FLOPS per byte of DRAM traffic (FLOPS/B). The quantities can be extended to the more general OPS (Operations per Second) and OI defined as OPS/B. However, the traditional Roofline Model cannot draw the whole picture for NUMA servers, since it misses the memory latency, which is included in the 3DyRM [9]. These quantities can be measured through hardware counters since it is well known that reading their information implies very low overheads.

In this article, we propose a user space tool that, based on the 3DyRM performance model, implements several algorithms for the migration of threads and memory pages in NUMA systems. Using simple heuristics for keeping overhead low, the different migrations are considered and a score is given according to its likelihood to improve performance. Also, our tool is especially, but not exclusively, focused on multitasking environments, where several tasks are in execution concurrently and there are more chances to improve the performance and throughput of the system. Finally, it is important to note that the proposed migration tool does not require modifying the program or programs under inspection nor the Linux kernel, so any application might take immediate profit from it.

The rest of the paper is structured as follows: Section 2 includes a brief discussion on related work and other proposals found in the literature. Section 3 describes which performance information is acquired and how. The proposed migration strategies are introduced in Section 4. Section 5 contains the description of the experimental environment. Section 6 describes the different experiments performed and their motivation. Section 7 shows the results of the aforementioned experiments and their discussion. Finally, Section 8 presents the final conclusions and future work.

2. Related work

Tasks scheduling and memory mapping are extensive topics in computer science research work. Default policies are mainly focused on load-balancing, placing locality and affinity aside, which are of paramount importance in NUMA architectures. Different proposals can be found in the literature, each one with advantages and disadvantages.

A considerable amount of proposals aim for Linux kernel modifications or modules to improve memory or thread placement. Carrefour, introduced by Dashti et al. [10], proposes a modification of the kernel to prevent and alleviate memory congestion for NUMA systems, achieving improvements up to 3.6× compared to the original kernel and other popular modifications like AutoNUMA [11]. Carrefour uses hardware counters to measure performance, and, according to the authors, it takes global decisions to migrate memory pages, while letting the Operating System deal with the thread placement. In the work by Diener et al. [12] it is introduced kMAF (kernel Memory Affinity Framework), a kernel modification to improve thread and data affinity through an analysis in runtime of the shared and exclusive memory regions reducing runtime by 13% on average and up to 36%. MVAS (Multi-View Address Space) [13], by Di Gennaro et al. is a kernel module that changes the page-fault handler to improve the accuracy of per-thread memory working-set and migrate memory pages, hence improving system performance up to 40%. Works by Chiang et al. [14–16] use several modifications to the kernel to improve thread mapping, locality, and deal with memory congestion, obtaining important performance boosts in PARSEC 3.0 [17] benchmarks. Also, Gureya et al. [3] propose BWAP, an algorithm for memory pages placement based on asymmetric weighted page interleaving, combining an analytical performance model of the target system and online tuning. Again, thread mapping is entrusted to the OS. With this approach, BWAP improves up to 66% the performance of the system.

User space tools for thread and memory mapping is the least explored approach, but there are interesting works, like AsymSched, by Lepers et al. [18]. This tool implements a dynamic thread and memory placement algorithm in Linux to improve performance in non-symmetric NUMA systems. Using hardware counters information during runtime, AsymSched computes the best thread and memory location every second, just focusing on maximising the bandwidth for communicating threads. This approach obtains important performance improvements in single and multiple application workloads. The experiments are done in AMD processors, that lack some features present in Intel architectures, like the measurement of memory accesses latency, which should help in improving memory transactions. DeLoc [19] is a tool that computes the optimal mapping after an initial profile phase where the communication and memory data are gathered and recorded. The computed mapping tries to improve the locality and reduce memory congestion. With this approach, the authors claim that performance can be improved by 61% compared to the AutoNUMA Linux policy.

Finally, it should be mentioned that thread mapping is not limited to NUMA systems, since multi-core systems can benefit from this kind of strategies too. In [20], an affinity and architecture-aware thread mapping technique is proposed to optimise data reuse, remote communications, and cache coherency costs of multi-threaded applications. Application-specific data dependency signatures are created, collecting data from previous executions, and they are used to determine the appropriate thread mapping of application for a given architecture. The proposed framework is evaluated using the Phoenix benchmark suite on two different multicore architectures, achieving a 25% improvement in performance compared to the default Linux

Table 1
State of the art comparison. Acronyms HIB and LIB stand for "higher is better" and "lower is better", respectively.

	Carrefour	kMAF	MVAS	Chiang et al.	BWAP	AsymSched	DeLoc	Ours
Does not require kernel changes	✘	✘	✓	✘	✓	✓	✓	✓
Does not require loading kernel modules	✓	✓	✘	✓	✘	✓	✓	✓
Runs in user/kernel space	Kernel	Kernel	Kernel	Kernel	Kernel	User	User	User
Uses Hardware Counters	✓	✘	✘	✘	✓	✓	✓	✓
Does not require previous profiling	✓	✓	✓	✓	✘	✓	✘	✓
Handles threads	✘	✓	✘	✓	✘	✓	✓	✓
Handles memory pages	✓	✓	✓	✘	✓	✓	✓	✓
Max. speedup (HIB)	3.60×	1.56×	1.66×	1.51×	1.66×	2.90×	1.61×	2.50×
Runtime overhead (LIB)	<5%	<4%	?	<2%	<4%	?	?	<8%
Tested scenarios (HIB)	2	1	1	3	1	1	1	3

scheduler. Reinforcement learning [21] has been also proposed to improve hardware-level thread migration. By utilising the recent history of memory access locations as input, each thread learns to recognise the relationship between prior access patterns and future memory access locations. Results using the Snipersim multicore simulator [22] on a set of SPLASH-2 [23] and PARSEC [17] benchmarks show a reduction of on-chip data movement and energy consumption by 41% while reducing execution time by 43% when compared to no thread migration.

In this work, we present new algorithms that aim to improve the performance of NUMA systems through user space migration of both threads and memory pages. These algorithms decide in runtime the best migrations to be performed by gathering and processing the information of relevant hardware counters. A score is assigned to each candidate migration involving threads in such a way that a higher score means a more likely improvement in performance. For memory pages, an analysis of the latency of every memory page and the average latency of the system is performed with a double purpose: first, look for NUMA nodes that present high latencies (considering them too busy) to balance workload, and to determine if a memory page can be placed in another node that lowers the latency of its accesses. This approach introduces low overhead and does not need previous profiling phases, nor modification of the Linux kernel or the target code.

A comparison of the most relevant proposals with ours is included in Table 1 showing the most important features.

2.1. Kernel and user space migrations

There are two ways of implementing thread and memory migration strategies, by either using kernel or user space. Both have advantages and disadvantages that are worth discussing.

Algorithms in kernel space benefit from having all the information of the system available, like used CPU-time, the current location of memory pages, number of page faults, etc. Also, all the processes of the system are in disposition to have their threads or memory pages migrated, with no restriction on permissions. This gives full control over the system, and with great power comes great responsibility. Nevertheless, those algorithms require modifications of the Linux kernel, implying some disadvantages. Modifying the kernel might be an expensive task in terms of programming time to make the algorithms work, solving bugs and so on, thus slowing down the research work. Also, for final users, those patches might not be available due to compatibility issues (like different kernel versions) or, mainly, restrictions in permissions to install and use those patches or modules.

User space tools, on the other hand, work with a limited amount of information, mostly restricted to what is available in `/proc` directory and the information available through performance profiling. They are also limited to migrating user-level threads and their memory pages due to how Linux permissions work. Note that this should not be a big issue since kernel threads are light processes in terms of computational demands. Also, user space tools are a step behind the operating system since the migration tool can reallocate resources only after the initial allocation is decided by the OS. This causes two problems: initial resources allocation might be important for certain applications, like those running in real-time or with restricted response time, and user space migrations cause more overhead due to the required reallocation. Development time is a great advantage of user space migration tools, so researchers can expend more time on improving the algorithms rather than debugging. Finally, the biggest advantage of user space tools is that they do not need any special permissions to be executed, so any user can easily download the software, compile it and make it run.

3. Performance measurement

Several metrics or combinations of metrics can be used for characterising the performance of a process or thread. The most popular performance model in the HPC field is the Roofline Model [8], which utilises FLOPS and OI. FLOPS gives the amount of floating-point operations per second. OI (Operational Intensity) measures the number of operations performed per byte of data, indirectly, measuring the efficiency of use of the cache memory. Although FLOPS and OI are often enough in conventional systems, NUMA servers require extra metrics since thread and memory placement play a key role in memory access latency. To improve the accuracy of the Roofline Model, the average latency of memory operations should be included. Thus, as mentioned before, we consider that the 3DyRM [9] model is more appropriate for this kind of systems. Furthermore, we have used the more generic OPS (Operation Per Second) instead of FLOPS, since not all applications make computations with floating-point arithmetic.

For measuring 3DyRM metrics, we have used Intel PEBS [24] (Intel Processor Based Samples) and Perfmon [25]. PEBS is a feature available in the Intel processor that allows the direct recording of samples from specific hardware counters into a designated memory region. Perfmon provides an interface to simplify the retrieval of information from these hardware counters. Some other APIs like PAPI [26] have been considered, but they do not provide the low-level information required by the algorithms introduced in this work.

In particular, these are the performance counters that have been used:

- `MEM_TRANS_RETIRED:LATENCY_ABOVE_THRESHOLD`: memory operations for which latency is above a given threshold. Threshold value is given as an option to the migration tool, by default is 1, so every transaction can be sampled.
- `OFFCORE_REQUESTS:ALL_DATA_RD`: number of read requests off-core, that is, to data allocated in DRAM memory.
- `INST_RETIRED`: total number of instructions retired.

- `FP_ARITH:SCALAR_DOUBLE:SCALAR_SINGLE`: scalar single and double precision floating point operations executed.
- `FP_ARITH:128B_PACKED_DOUBLE`: 128-bit vector double precision floating point operations executed.
- `FP_ARITH:128B_PACKED_SINGLE`: 128-bit vector single precision floating point operations executed.
- `FP_ARITH:256B_PACKED_DOUBLE`: 256-bit vector double precision floating point operations executed.
- `FP_ARITH:256B_PACKED_SINGLE`: 256–bit vector single precision floating points operations executed.

4. Migration algorithms

In this work, we introduce several algorithms for thread and memory migrations. These algorithms have been implemented in a migration tool that runs in user space named Thanos.[1] This approach has two main advantages: no instrumentation of the target application is required and no Linux kernel modifications have to be done. This tool is launched by just executing the command without being a privileged user in the system:

```
$ thanos [options] <target>
```

where `target` is the program whose threads and memory pages will be controlled by the migration tool. If you want to include more than one program, they must be included in a shell script that will serve as `target`. Note that the tool will only migrate the threads and memory pages of the target program or script. The rest of the threads and memory pages in the system are not affected nor taken into account by this tool.

The only requirement for this tool to work resides in the content of the file `perf_event_paranoid`, which must be set to a negative integer value or zero to allow the collection of the hardware counters samples.

4.1. Problem formulation

To explain the migration algorithms, some notation and formal concepts should be introduced. A NUMA server is composed by N_{nodes} nodes. The kth node, ν_k with $1 \le k \le N_{\text{nodes}}$, has C_k cores named ζ_{kl}, $1 \le l \le C_k$. At any given time, there is a set of p processes running, which we define as $\Pi = \{\pi_1, \dots, \pi_p\}$. Each process π_i consists of a set of h_i threads named θ_{ij}, $1 \le j \le h_i$. These processes and threads use data distributed across m memory pages, which we will denote as $\Psi = \{\psi_1, \dots, \psi_m\}$.

Every T seconds, the migration algorithm is executed, allowing us to define the sequence of time intervals $\tau_1, \tau_2 \dots$

In the decision-making process, in each time interval τ_t, several metrics are considered, which are represented by the following functions:

- $A\left(\theta_{ij}, \nu_n, \tau_t\right)$ gives the number of memory accesses performed by the thread θ_{ij} to data in the node ν_n.
- $\hat{A}\left(\psi_i, \nu_n, \tau_t\right)$ is the number of accesses performed by all threads running in the node ν_n to data located in the memory page ψ_i.
- $L\left(\theta_{ij}, \nu_n, \tau_t\right)$ is the average latency of the memory operations performed by thread θ_{ij} while running in the node ν_n.
- $\hat{L}\left(\psi_i, \nu_n, \tau_t\right)$ is the average latency of the accesses performed by all threads running in the node ν_n to data located in the memory page ψ_i.
- $\hat{L}\left(\nu_n, \tau_t\right)$ is the average latency of the accesses performed by all threads running in the node ν_n to data located in any memory page.
- $\hat{L}\left(\psi_i, \tau_t\right)$ is the average latency of the accesses performed by all threads running in the system to data in the memory page ψ_i.
- $\hat{L}\left(\tau_t\right)$ is the average latency of the accesses performed by all threads in the system to all memory pages.
- $O\left(\theta_{ij}, \nu_n, \tau_t\right)$ is the number of operations performed by thread θ_{ij} while running in the node ν_n.
- $I\left(\theta_{ij}, \nu_n, \tau_t\right)$ is the operational intensity for thread θ_{ij} while running in the node ν_n. This operational intensity is computed as

$$I\left(\theta_{ij}, \nu_n, \tau_t\right) = \frac{O\left(\theta_{ij}, \nu_n, \tau_t\right)}{A\left(\theta_{ij}, \nu_n, \tau_t\right) \cdot S_{\text{cache}}}, \tag{1}$$

where S_{cache} is the number of bytes of a cache line.

Using these functions, we can introduce the concept of the preferred node for threads and memory pages. The preferred node, ν_{pref}, for a thread θ_{ij} is the node in which has performed most of its memory operations. Formally, it is the NUMA node ν_r which produces the higher value for $A\left(\theta_{ij}, \nu_r, \tau_t\right)$, that is,

$$\nu_{\text{pref}} = \max_{\nu_r} A\left(\theta_{ij}, \nu_r, \tau_t\right). \tag{2}$$

Similarly, the preferred node for a memory page ψ_i is the NUMA node that hosts the threads that produce the most memory operations. That is,

$$\nu_{\text{pref}} = \max_{\nu_r} \hat{A}\left(\psi_i, \nu_r, \tau_t\right). \tag{3}$$

Using these metrics, a score (Q) is given to each considered migration according to the likelihood of improving the system performance. Details on how this Q is computed are given in Sections 4.2 and 4.3.

Therefore, the set of possible thread migrations in each time interval is defined as $M = \{M_1, M_2, \dots\}$. Each possible thread migration is denoted by the tuple $M_i = \left[\vec{\Theta}, \vec{C}, Q\right]$, where $\vec{\Theta}$ is the list of threads to be migrated, $\vec{C}$ their destination cores, and Q is the summation of their scores. In the algorithms presented in this work, the number of threads to be moved in each migration M_i are one or two, corresponding to a single migration or an interchange, respectively. For a single migration, $\vec{\Theta} = \left[\theta_{ij}\right]$ and $\vec{C} = [\zeta_{kl}]$, so thread θ_{ij} would be migrated to core ζ_{kl}. For an interchange, $\vec{\Theta} = \left[\theta_{ij}, \theta_{i'j'}\right]$ and $\vec{C} = [\zeta_{kl}, \zeta_{k'l'}]$, so θ_{ij} would be migrated to ζ_{kl} and $\theta_{i'j'}$ to $\zeta_{k'l'}$.

Alternatively, threads can be moved to nodes, leaving it up to the OS to choose the specific core within the node on which the thread will run. In this scenario, $M_i = \left[\vec{\Theta}, \vec{N}, Q\right]$, where $\vec{N}$ are the destination nodes for threads in $\vec{\Theta}$, in the same way as we defined $\vec{C}$ for cores.

Migrations are considered for memory pages too. In that case, we would have $M_i = \left[\vec{\Psi}, \nu_n\right]$, where all the pages in the subset $\vec{\Psi}$ would be migrated to the NUMA node ν_n if M_i is performed.

4.2. CIMAR

In this work, we propose CIMAR (Core-aware Interchange and Migration Algorithm with performance Record –IMAR–), a thread migration algorithm built upon IMAR[2] (Interchange and Migration Algorithm with performance Record and Rollback) [27], with the objective of improving its stability and consistency. There are three main differences between both algorithms: the scoring system of the candidate migrations, the selection of migrations to be performed, and the removal of the rollback.

CIMAR algorithm is executed every T_{CIMAR} seconds and it uses the 3DyRM information to guide the decision process. A function,

[1] Software available at https://gitlab.citius.usc.es/ruben.laso/migration.

defined as $P\left(\theta_{ij}, \nu_n, \tau_t\right)$, gives the performance of the thread θ_{ij}, while running in a core of the NUMA node ν_n during the time interval τ_t. According to [27], a good definition for the performance function is

$$P\left(\theta_{ij}, \nu_n, \tau_t\right) := \frac{O\left(\theta_{ij}, \nu_n, \tau_t\right) \cdot I\left(\theta_{ij}, \nu_n, \tau_t\right)}{L\left(\theta_{ij}, \nu_n, \tau_t\right)}. \quad (4)$$

A set of candidate threads for migration, $\hat{\Theta}$, is selected using per-process relative performance,

$$\hat{P}\left(\theta_{ij}, \nu_n, \tau_t\right) = \frac{P\left(\theta_{ij}, \nu_n, \tau_t\right)}{\bar{P}\left(\pi_i, \tau_t\right)}, \quad (5)$$

where $\bar{P}\left(\pi_i, \tau_t\right)$ is the function returning the average performance of the threads of process π_i.

Let $\hat{\Theta}$ be the set of m threads with the lowest value of $\hat{P}$, verifying that $\hat{P}\left(\theta_{ij}, \nu_n, \tau_t\right) \leq \delta_{\text{perf}}, \forall \theta_{ij} \in \hat{\Theta}$, where δ_{perf} serves as a threshold to avoid migrations in scenarios where all threads have similar performance and their mapping is, probably, optimal or near to optimal. By default, we set $\delta_{\text{perf}} = 0.8$.

For each thread $\theta_{ij} \in \hat{\Theta}$, running in the core ζ_{kl} of the node ν_k, all the cores but those in ν_k are considered as destination. Cores in node ν_k are discarded since it is assumed that every core in a node will have similar behaviour and the same latency for the same memory accesses.

For every combination of threads, $\theta_{ij} \in \hat{\Theta}$, and destination cores, $\zeta_{k'l'}$, a score is assigned. Also, if $\zeta_{k'l'}$ currently hosts other threads, an interchange is considered instead of a single migration and the score of those threads to be migrated to ζ_{kl} is computed. Points are given according to the following criteria:

- q_1 points are granted if destination core $\zeta_{k'l'}$ was not hosting threads during τ_t. By default, q_1 is set to 2.
- q_2 points are assigned to the cores according to the NUMA distance to the preferred node of θ_{ij}. For the destination core $\zeta_{k'l'}$, and the preferred node ν_{pref},

$$q_2 = \frac{\hat{q}_2 \cdot d\left(\nu_{k'}, \nu_{k'}\right)}{d\left(\nu_{k'}, \nu_{\text{pref}}\right)}, \quad (6)$$

  where $d\left(\nu_i, \nu_j\right)$ corresponds to the distance between nodes as returned by the system call `numa_distance(i, j)`, and we set $\hat{q}_2 = 4$ by default.
- q_3 points are given according to the previous performance of θ_{ij} in the considered destination core $\zeta_{k'l'}$. Performance during τ_t is compared with the last performance measure obtained by θ_{ij} when running in a core in node $\nu_{k'}$. If the previous performance was better, q_3 is set to 4, if it was worse, $q_3 = 1$, and if there is no information or it has not changed, $q_3 = 2$.
- q_4 points are given if a swap is considered between θ_{ij} and a thread $\theta_{i'j'}$ currently running in core $\zeta_{k'l'}$, and it holds that $\hat{P}\left(\theta_{i'j'}, \nu_{k'}, \tau_t\right) < \delta_{\text{perf}}$. By default, we set $q_4 = 3$.

Additionally, the score of keeping the thread in its current location is computed. If it is higher than any possible migration of that thread, it is kept in its current core. This is done to prevent migrations to destinations that are unlikely to improve performance.

Once that the score of candidate migrations have been computed, the m migrations with the highest score are selected to be performed.

Finally, rollback has been removed from this algorithm. Rollback was introduced in IMAR^2 to fix migrations to bad locations, which are very costly. With the additional condition in CIMAR that a migration can only take place if its score is higher than keeping threads still, bad migrations are highly prevented. This also results in a more conservative algorithm, reducing the exploration of new placements. For example, a migration with a low score might be beneficial. With the IMAR^2 algorithm it could be explored, but not with CIMAR. This is a matter of a trade-off between exploring new migrations and avoiding the bad ones.

Algorithm 1 shows the pseudocode of the migration selection process of CIMAR.

4.3. NIMAR

NIMAR (Node-aware IMAR) algorithm is a variation of CIMAR, introduced to improve a particular flaw of the latter. CIMAR shows some problems regarding work balance, where two computationally intensive threads can share and stress a CPU even if there are other CPUs less loaded. To solve this, NIMAR does migrations to NUMA nodes instead of cores, yielding the work balance within nodes to the OS. Therefore, it is the Operating System that decides the particular core in which a thread will run. Work balance in OS is a well-studied field, so its reliability for this task is high since it has better information to do it correctly by working on kernel space. Thus, this algorithm presents a hybrid approach between user-space and kernel-space thread scheduling.

NIMAR is executed every T_{NIMAR} seconds and selects the set of threads to be migrated, $\hat{\Theta}$, in the same way as CIMAR. For each thread $\theta_{ij} \in \hat{\Theta}$, running in node ν_n, the rest of the nodes are considered for the destination, and a score is given to that migration. If the destination node $\nu_{n'}$ is hosting C_n or more threads, an interchange is considered, and for every thread $\theta_{i'j'}$ running in $\nu_{n'}$, the score of the migration of $\theta_{i'j'}$ to ν_n is computed. Points are given in a similar way to CIMAR:

- q_1 points are granted if destination node ν_n was hosting less than C_n threads during τ_t, it has free cores. By default, we set $q_1 = 2$.
- q_2 points are assigned according to the NUMA distance of the destination node $\nu_{n'}$ to the preferred node of θ_{ij} in the same way as shown in Eq. (6).
- q_3 points are given according to the previous performance of θ_{ij} in the considered destination node $\nu_{n'}$. Performance during τ_t is compared with the last performance measure obtained by θ_{ij} when running in a core in node $\nu_{n'}$. If the previous performance was better, q_3 is set to 4, whereas if it was worse, $q_3 = 1$. Otherwise, $q_3 = 2$.
- q_4 points are given if a swap is considered between θ_{ij} and a thread $\theta_{i'j'}$ currently running in the node $\nu_{n'}$, and $\hat{P}\left(\theta_{i'j'}, \nu_{n'}, \tau_t\right) < \delta_{\text{perf}}$. By default, we set $q_4 = 3$.

Also, migrations with a lower score than the one obtained keeping the thread in their current location are discarded. Finally, the m migrations with the best score are performed.

Algorithm 2 shows the pseudocode of NIMAR.

4.4. LMMA

LMMA (Latency-based Memory pages Migration Algorithm) is a migration algorithm whose target is to move memory pages across the nodes to improve the performance of the system. This algorithm is executed periodically every T_{LMMA} seconds. Between executions, in the period τ_t, memory samples are processed. Each sample contains information about, among others, the address of the memory region accessed to, the node from which it was accessed, a timestamp, the number of memory operations, and the average latency of those operations.

An ageing factor is applied to every received sample according to

$$f\left(t_{\text{mig}}\right) = \frac{1}{1 + t_{\text{mig}}}, \quad (7)$$

Algorithm 1 CIMAR migration strategy.

```
Input:   Processes $\Pi = \{\pi_1, \pi_2, \ldots, \pi_p\}$.
         Threads $\Theta = \{\Theta_{ij}, i = 1, \ldots, p, j = 1, \ldots, h_i\}$.
         Cores $Z = \{\zeta_{kl}, k = 1, \ldots, N_{\text{nodes}}, l = 1, \ldots, C_k\}$.
         Relative performance threshold $\delta_{\text{perf}}$.
         Maximum number of migrations $m$.
Output:  Migrations to perform $M = \{M_1, M_2, \ldots\}$.

1:  procedure CIMAR($\Pi, \Theta, Z, \delta_{\text{perf}}, m$)
2:      $\hat{\Theta} = \{\theta_{ij} \in \Theta \mid P(\theta_{ij}, \nu_n, \tau_t)/\bar{P}(\pi_i, \tau_t) < \delta_{\text{perf}}\}$    ▷ Select $m$ threads with worst rel. performance and such that $\hat{P} < \delta_{\text{perf}}$.
3:      $\hat{M} = \varnothing$    ▷ Candidate migrations is an empty set at the beginning.
4:      for each $\theta_{ij} \in \hat{\Theta}$ do    ▷ Compute candidate migrations.
5:          $\zeta_{kl}$ := core hosting $\theta_{ij}$
6:          $Q_{\text{ref}} = \text{Score}(\theta_{ij}, \zeta_{kl})$
7:          for each $\zeta_{k'l'} \in Z \mid k' \neq k$ do    ▷ For each core in a different node, search for candidate migrations.
8:              if $\zeta_{k'l'}$ is free then
9:                  $Q = \text{Score}(\theta_{ij}, \zeta_{k'l'})$    ▷ Compute score for migration of $\theta_{ij}$ to $\zeta_{k'l'}$.
10:                 if $Q > Q_{\text{ref}}$ then    ▷ If it is better to migrate than keeping $\theta_{ij}$ still...
11:                     $\hat{M} = \hat{M} \cup \{[\theta_{ij}], [\zeta_{k'l'}], Q\}$    ▷ Add a single migration of $\theta_{ij}$ to $\zeta_{k'l'}$ to the set of candidates.
12:             else
13:                 for each $\theta_{i'j'}$ running in $\zeta_{k'l'}$ do
14:                     $Q = \text{Score}(\theta_{ij}, \zeta_{k'l'}) + \text{Score}(\theta_{i'j'}, \zeta_{kl})$    ▷ Compute score for migrations of $\theta_{ij}$ and $\theta_{i'j'}$.
15:                     if $Q > Q_{\text{ref}} + \text{Score}(\theta_{i'j'}, \zeta_{k'l'})$ then    ▷ If it is better to migrate than keeping $\theta_{ij}$ and $\theta_{i'j'}$ still...
16:                         $\hat{M} = \hat{M} \cup \{[\theta_{ij}, \theta_{i'j'}], [\zeta_{k'l'}, \zeta_{kl}], Q\}$    ▷ Swap, $\theta_{ij}$ would be moved to $\zeta_{k'l'}$, and $\theta_{i'j'}$ to $\zeta_{kl}$.
17:     $M$ := $m$ migrations in $\hat{M}$ with highest $Q$
18:     return $M$
```

Algorithm 2 NIMAR migration strategy.

```
Input:   Processes $\Pi = \{\pi_1, \pi_2, \ldots, \pi_p\}$.
         Threads $\Theta = \{\Theta_{ij}, i = 1, \ldots, p, j = 1, \ldots, h_i\}$.
         Nodes $N = \{\nu_n, n = 1, \ldots, N_{\text{nodes}}\}$.
         Relative performance threshold $\delta_{\text{perf}}$.
         Number of threads to be migrated $m$.
Output:  Migrations to perform $M = \{M_1, M_2, \ldots\}$.

1:  procedure NIMAR($\Pi, \Theta, N, \delta, m$)
2:      $\hat{\Theta} = \{\theta_{ij} \in \Theta \mid P(\theta_{ij}, \nu_n, \tau_t)/\bar{P}(\pi_i, \tau_t) < \delta_{\text{perf}}\}$    ▷ Select $m$ threads with worst relative performance and such that $\hat{P} < \delta_{\text{perf}}$.
3:      $\hat{M} = \varnothing$    ▷ Candidate migrations is an empty set at the beginning.
4:      for each $\theta_{ij} \in \hat{\Theta}$ do    ▷ Compute candidate migrations.
5:          $\nu_{n'}$ := node hosting $\theta_{ij}$
6:          $Q_{\text{ref}} = \text{Score}(\theta_{ij}, \nu_n)$
7:          for each $\nu_{n'} \in N \mid n' \neq n$ do    ▷ For each different node, search for candidate migrations.
8:              if $\nu_{n'}$ has free cores then
9:                  $Q = \text{Score}(\theta_{ij}, \nu_{n'})$    ▷ Compute score for migration of $\theta_{ij}$ to $\nu_{n'}$.
10:                 if $Q > Q_{\text{ref}}$ then    ▷ If it is better to migrate than keeping $\theta_{ij}$ still...
11:                     $\hat{M} = \hat{M} \cup \{[\theta_{ij}], [\nu_{n'}], Q\}$    ▷ Add a single migration of $\theta_{ij}$ to $\nu_{n'}$ to the set of candidates.
12:             else
13:                 for each $\theta_{i'j'}$ running in $\nu_{n'}$ do
14:                     $Q = \text{Score}(\theta_{ij}, \nu_{n'}) + \text{Score}(\theta_{i'j'}, \nu_n)$    ▷ Compute score for migrations of $\theta_{ij}$ and $\theta_{i'j'}$.
15:                     if $Q > Q_{\text{ref}} + \text{Score}(\theta_{i'j'}, \nu_{n'})$ then    ▷ If it is better to migrate than keeping $\theta_{ij}$ and $\theta_{i'j'}$ still...
16:                         $\hat{M} = \hat{M} \cup \{[\theta_{ij}, \theta_{i'j'}], [\nu_{n'}, \nu_n], Q\}$    ▷ Swap, $\theta_{ij}$ would be moved to $\nu_{n'}$, and $\theta_{i'j'}$ to $\nu_n$.
17:     $M$ := $m$ migrations in $\hat{M}$ with highest $Q$
18:     return $M$
```

where t_{mig} is the number of seconds until the next execution of the migration algorithm. In that way, more recent memory operations have more influence in the decision-making process considering that, according to locality principle, more recent accessed data is more likely to be accessed again. For example, when processing a sample corresponding to the page ψ_n with

$t_{\text{mig}} = 0.25$, its contribution to the different functions $\hat{A}$ and $\hat{L}$ would be weighted by 0.8.

Once the period τ_t is over and all memory samples are processed, several operations are performed to decide which pages will be migrated and their destinations. First, every node is evaluated deciding if it is busy or not. We say that the node ν_n is busy when

$$\frac{\hat{L}(\nu_n, \tau_t)}{\hat{L}(\tau_t)} > \delta_{\text{busy}}. \tag{8}$$

By default, $\delta_{\text{busy}} = 1.3$. Also, we calculate the least busy node, ν_{alt}:

$$\nu_{\text{alt}} = \min_{\nu_n} \frac{\hat{L}(\nu_n, \tau_t)}{\hat{L}(\tau_t)}. \tag{9}$$

Memory pages are migrated according to the following criteria. For each page $\psi_i \in \Psi$, we compare its average latency to the global average latency. If

$$\frac{\hat{L}(\psi_i, \tau_t)}{\hat{L}(\tau_t)} > \delta_{\text{lat}}, \tag{10}$$

the page will be considered for migration, since its latency is considerably higher than the average. By default, we choose $\delta_{\text{lat}} = 1.3$. Given the case, two possible destinations are considered for ψ_i, its preferred node or the least saturated node. If the preferred node, ν_{pref}, is not busy, then $\nu_{\text{dest}} = \nu_{\text{pref}}$, else, $\nu_{\text{dest}} = \nu_{\text{alt}}$. The least saturated node will only be its destination when the preferred node is noted as busy.

Similarly to a cache prefetcher, LMMA tries to move consecutive memory pages to anticipate future memory operations. The main target of this part of the algorithm is to choose a destination for those memory pages for which there is no sampling information. Once LMMA has a destination node for ψ_i, up to S_{preload} next consecutive pages might be migrated to ν_{dest}. By default, $S_{\text{preload}} = 8$. Until LMMA finds a page ψ_{i+j} with a different preferred node, or j reaches S_{preload}, pages $\psi_i, \ldots, \psi_{i+j-1}$ will be migrated to the node ν_{dest} including those pages for which no information is available. For example, if the page ψ_{i+3} has another preferred node, only pages ψ_i, ψ_{i+1} and ψ_{i+2} will be migrated to ν_{dest}.

Finally, it should be noted that the Linux procedure for memory pages migrations is known to be inefficient [18,28]. Given that, this algorithm is not expected to produce a big impact on raw performance, but it should improve the stability of the system by reducing memory congestion.

Algorithm 3 shows the pseudocode of LMMA.

5. Experimental environment

The experiments described in this section have been carried out using two NUMA servers with different topologies and different values for memory access latencies and memory bandwidth. Latency and bandwidth matrices shown in Tables 2 and 3 were obtained with Intel Memory Latency Checker [29]:

- Server HPL (High Penalty on Latency): A Debian GNU/Linux 9, kernel version 5.1.15 composed of four nodes with Intel Xeon E5-4620 v4 processors with 10 cores each (40 in total), Broadwell-EP architecture, 25 MB L3 cache, 2.1 GHz–2.6 GHz, and 256 GB of RAM. Server topology is shown in Fig. 1a, and all available memory channels are used for each node. In this server, remote accesses have a latency more than 3× higher than local accesses (see Table 2a), while its bandwidth is reduced by 79% (see Table 2b). Note that memory operations requiring 2-hop communications (for example, between nodes 0 and 2) have a slightly higher latency than the other remote accesses.
- Server LPL (Low Penalty on Latency): A Debian GNU/Linux 9, kernel version 4.18.0 composed of four nodes with Intel Xeon Gold 6248 with 20 cores each (80 in total), Cascade Lake architecture, 27.5 MB L3 cache, 2.50 GHz-3.9 GHz, and 1 TB of RAM. Server topology is shown in Fig. 1b. All memory channels are in use and all memories are interconnected with each other, so all remote accesses have similar latencies. As reported by Table 3a and b, remote accesses have about 1.7× higher latency, while the bandwidth is decreased by 78% approximately.

Given the characteristics of both servers, we can state that data and thread placement is more critical in Server HPL than LPL, and therefore more susceptible to improving its performance using our algorithms.

6. Experiments description

For the experimental validation of our proposal, we have designed three experiments with the NAS parallel benchmarks [30] version 3.4.1 and the LevelDB benchmark [31] version 1.0.2 included in the Phoronix Test Suite [32]. With these benchmarks, we measured performance improvements in two extremely important areas, namely HPC and databases. This is the list of used benchmarks:

- **BT**: **B**lock **T**ri-diagonal solver based on a CFD pseudo-application.
- **CG**: **C**onjugate **G**radient solver based on a CFD pseudo-application.
- **DC**: Arithmetic **D**ata **C**ube computations focused on data movement across cores.
- **EP**: **E**mbarrassingly **P**arallel. Kernel designed to provide an estimate of the upper achievable limits of floating-point performance.
- **FT**: 3-D Fast **F**ourier **T**ransform computations with all-to-all communication.
- **IS**: **I**nteger **S**ort algorithm with random memory accesses.
- **LU**: **L**ower-**U**pper Gauss–Seidel solver factorisation.
- **MG**: **M**ulti-**G**rid on a sequence of meshes, long- and short-distance communication. Memory intensive benchmark.
- **SP**: **S**calar **P**enta-diagonal solver based on a CFD pseudo-application.
- **UA**: **U**nstructured **A**daptive mesh with dynamic and irregular memory accesses.
- **LevelDB**: database application developed by Google. This benchmark includes several read and write operations, namely: Rand Delete, Fill Rand, Fill Seq, Fill Sync, Overwrite, Hot Read, Read Rand, and Seek Rand.

Fig. 2 shows the roofline information obtained with Intel Advisor [33] for all the benchmarks used in the experiments with their proportion of last level cache (LLC) misses obtained with `perf` [34]. With this information, we can classify the benchmarks according to their rate of LLC misses:

- Low (0%–20%): BT, EP, CG, and IS.
- Medium (20%–40%): FT, MG, UA, and LevelDB Rand Delete, Fill Rand, Fill Seq, Fill Sync, and Overwrite.
- High (40% or more): DC, LU, SP, and LevelDB Hot Read, Read Rand, and Seek Rand.

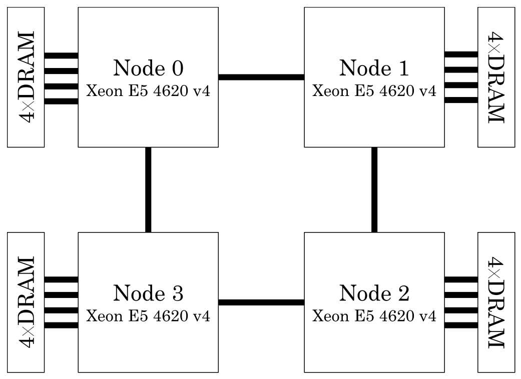
Three different scenarios have been considered for the experiments and validation tests:

- Experiment A: Servers are used with one application at a time, which can use all available resources, all cores and all available memory. This experiment is the most complicated in terms of improving performance. Since the selected

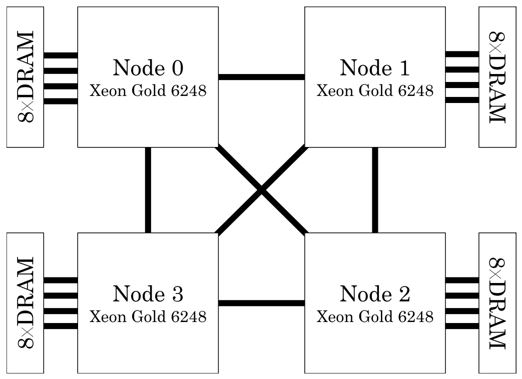
Algorithm 3 LMMA migration strategy.

```
Input:   Memory pages $\Psi = \{\psi_1, \psi_2, \dots\}$.
         Nodes $N = \{\nu_n, n = 1, \dots, N_{\text{nodes}}\}$.
         Thresholds $\delta_{\text{busy}}, \delta_{\text{lat}}$.
Output:  Migrations to perform $M = \{M_1, M_2, \dots\}$.

 1: procedure LMMA($\Psi, N, \delta_{\text{busy}}, \delta_{\text{lat}}$)
 2:     $N_{\text{busy}} := \left\{\nu_n \in N \mid \hat{L}(\nu_n, \tau_t)/\hat{L}(\tau_t) > \delta_{\text{busy}}\right\}$      ▷ Evaluate busy nodes.
 3:     $\nu_{\text{alt}} = \min_{\nu_r} \hat{L}(\nu_r, \tau_t)/\hat{L}(\tau_t)$      ▷ Pick the least busy node.
 4:     $M = \varnothing$      ▷ Migrations to perform is an empty set at the beginning.
 5:     for each $\psi_i \in \Psi \mid \psi_i \notin M$ do      ▷ Compute migrations.
 6:         if $\hat{L}(\psi_i, \tau_t)/\hat{L}(\tau_t) > \delta_{\text{lat}}$ then      ▷ If the latency of $\psi_i$ is high, search for a migration.
 7:             $\nu_{\text{pref}} = \max_{\nu_r} \hat{A}(\psi_i, \nu_r, \tau_t)$
 8:             if $\nu_{\text{pref}} \in N_{\text{busy}}$ then      ▷ Check if the preferred node is busy.
 9:                 $\nu_{\text{dest}} = \nu_{\text{alt}}$
10:             else
11:                 $\nu_{\text{dest}} = \nu_{\text{pref}}$
12:             $\vec{\Psi} := [\psi_i]$
13:             for $j = 1, \dots, S_{\text{preload}}$ do      ▷ Compute preload.
14:                 $\nu'_{\text{pref}} = \max_{\nu_r} \hat{A}\left(\psi_{i+j}, \nu_r, \tau_t\right)$
15:                 if $\nu'_{\text{pref}} = \nu_{\text{dest}}$ or $\nu'_{\text{pref}} = \varnothing$ then      ▷ If preload conditions are matched...
16:                     $\vec{\Psi} := \vec{\Psi} + \psi_{i+j}$      ▷ Add $\psi_j$ to the set of pages to migrate.
17:                 else      ▷ Else, no more pages available for preload.
18:                     $M = M \cup \left[\vec{\Psi}, \nu_{\text{dest}}\right]$      ▷ Pages already in $\vec{\Psi}$ will be migrated to $\nu_{\text{dest}}$.
19:                     End preload and continue on line 5.      ▷ Repeat process with the next memory page.
20:     return $M$
```



(a) Server HPL.



(b) Server LPL.

Fig. 1. Network topologies of Server HPL and Server LPL.

Table 2
Latency and bandwidth matrices for server HPL.

(a) Latency matrix (in ns) for server HPL.

	Node 0	Node 1	Node 2	Node 3
Node 0	85.8	254.7	271.4	256.2
Node 1	254.9	86.0	253.2	271.5
Node 2	271.4	252.7	86.0	254.7
Node 3	255.1	271.9	254.3	85.7

(b) Bandwidth matrix (in MB/s) for server HPL.

	Node 0	Node 1	Node 2	Node 3
Node 0	59,497	12,709	12,002	12,368
Node 1	12,153	59,528	12,210	12,007
Node 2	12,028	12,395	59,506	12,689
Node 3	12,371	11,992	12,704	59,487

Table 3
Latency and bandwidth matrices for Server LPL.

(a) Latency matrix (in ns) for server LPL.

	Node 0	Node 1	Node 2	Node 3
Node 0	88.3	145.1	141.3	144.4
Node 1	143.3	82.9	142.1	140.4
Node 2	139.0	141.2	83.1	144.8
Node 3	144.8	138.4	143.3	83.4

(b) Bandwidth matrix (in MB/s) for server LPL.

	Node 0	Node 1	Node 2	Node 3
Node 0	76,251	17,148	17,141	17,127
Node 1	17,141	76,286	17,141	17,100
Node 2	17,153	17,157	76,306	17,152
Node 3	17,141	17,146	17,140	76,210

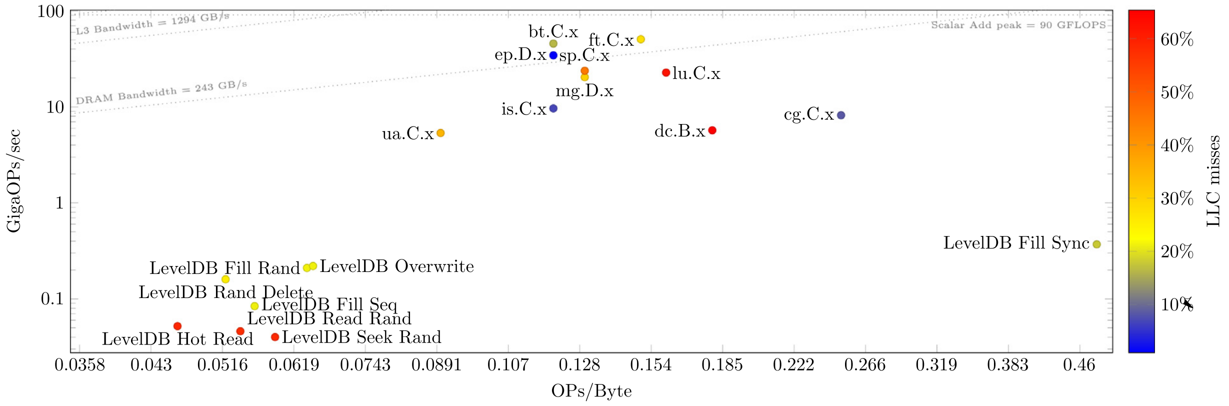


Fig. 2. Roofline model of the used benchmarks obtained with Intel Advisor in Server HPL with colour gradient of the rate of LLC misses obtained with `perf`. Red means more misses.

Table 4
Start times in seconds for each task and server in Experiment B.

	Server HPL	Server LPL
lu.C.x-1	0	0
bt.C.x-1	18	9
cg.C.x-1	137	68.5
sp.C.x-1	190	95
lu.C.x-2	224	112
bt.C.x-2	244	122
cg.C.x-2	269	134.5
sp.C.x-2	316	168

benchmarks are well-known among HPC researchers, and well-programmed according to locality principles, little improvement, if any, is expected. However, this experiment is interesting to give an idea of the potential overhead of our migration tool in general and the different algorithms in particular.

- Experiment B: Servers are used interactively, that is, users can send little tasks at any time. This experiment is a simulation of an interactive system based on anonymous data of the use of computing nodes in CESGA (Centro de Supercomputación de Galicia, https://www.cesga.es). The time at which each task starts to execute is fixed, so the objective is to reduce the execution time of each task. This experiment draws a scenario where the number of concurrent tasks changes with time, giving more chances for improving the performance with migrations than in Experiment A. The start time of each task is shown in Table 4. Note that start times are scaled-down ($\times 0.5$) in Server LPL to achieve a similar number of concurrent tasks to HPL.
Each task consists of 8 and 16 threads for Servers HPL and LPL, respectively. At peak, it is possible to have more threads in execution than cores. An example of an execution of this experiment is shown in Fig. 3.
- Experiment C: Servers are used with a queue of tasks, like Slurm [35], where users send tasks and only a fraction of the resources are available. We simulate four users that have sent tasks based on NAS benchmarks [30]. The objective is to reduce the time to complete all tasks. There are granted only 10 and 20 threads per user for Server HPL and LPL, respectively. At peak, there are as many threads as cores. An example of an execution of this experiment is shown in Fig. 4. This is the experiment where the potential improvement due to thread and memory migrations is higher. Reducing the execution time of a task implies launching the following tasks earlier. Also, the chances of improving performance are greater due to the high number of processes in the system.

Different conditions and configurations have been considered in the execution of the benchmarks. The first three configurations in the following list are commonly used in the literature as a reference for validation purposes. The others correspond to the proposed algorithms, used individually or combining thread and page migration.

- Baseline: Thread and memory mapping are under the control of the operating system. Linux default policies are used with AutoNUMA enabled, so the Linux option `numa_balancing` is set to 1.
- Direct: Each task is granted a memory node, so its threads are directly mapped using the `numactl` command [36]. This way, each benchmark will allocate its running threads in the same node as its data, as long as the memory cell is large enough. This is a common option used by experienced users who know the limits and behaviour of their parallel applications [37,38].
- Interleave: Memory pages are distributed evenly across NUMA nodes using the `numactl` utility. Thread migration is under the responsibility of the OS. Like direct mapping, this is another popular option when executing programs in NUMA servers.
- CIMAR: Thread mapping is controlled by the algorithm shown in Section 4.2, with up to $m = 5$ migrations per iteration and $T_{\text{CIMAR}} = 1$ second, so up to 5 migrations are performed every second. The rest of the parameters are set to their default value. The initial thread mapping is determined by the OS, which also controls all memory mappings.
- NIMAR: Thread mapping is controlled by the algorithm described in Section 4.3, with $m = 5$ and $T_{\text{NIMAR}} = 1$ second. The rest of the parameters are set to their default value. As in the previous case, the OS sets the initial thread mapping and controls all memory mappings.
- LMMA: Memory mapping is controlled by the algorithm described in Section 4.4, with $T_{\text{LMMA}} = 1$ second. The rest of the parameters are set to their default value. Now, the OS establishes the initial memory mapping and controls all thread mappings.
- CIMAR + LMMA: both algorithms are enabled with the aforementioned parameters.
- NIMAR + LMMA: both algorithms are enabled with the aforementioned parameters.

The values of the parameters used by CIMAR, NIMAR and LMMA have been selected after several experiments, not included in this paper, for finding those which fit well in different servers and scenarios. The most critical parameters are the number of

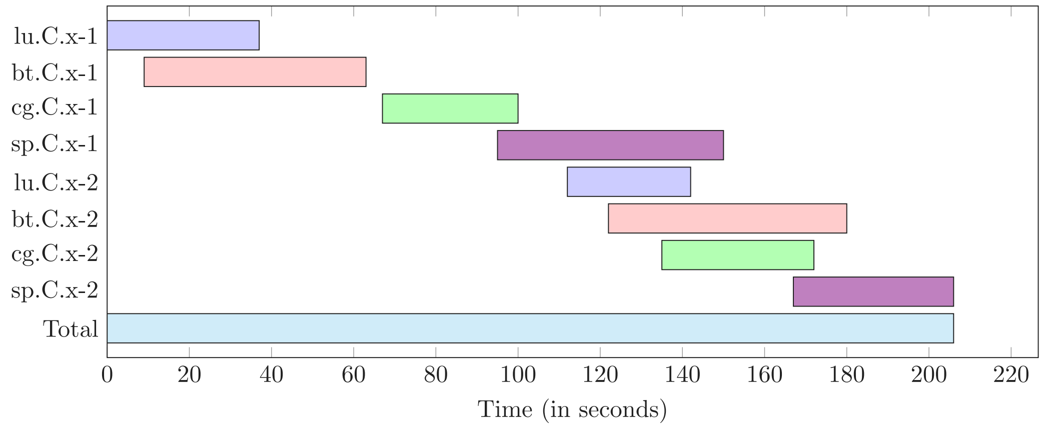


Fig. 3. Example of a time trace for the Experiment B.

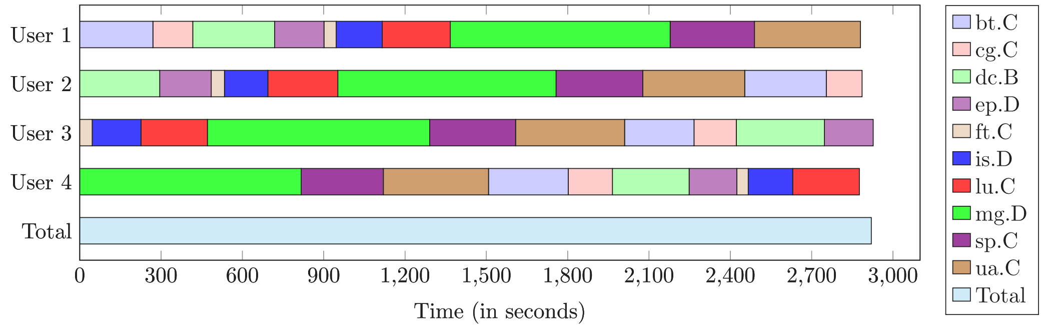


Fig. 4. Example of a time trace for the Experiment C.

threads to be migrated and the time between migrations. Regarding the number of threads to be migrated, it is possible to state that the larger value of m, the more aggressive the algorithms will be. This might help find an optimal mapping sooner but might also incur non-optimal placements with a high overhead due to the high number of migrations. About T_{CIMAR}, T_{NIMAR} and T_{LMMA}, a balance should be found with the information available. On one hand, it is needed to wait some time to gather enough samples so decisions are based on solid measurements. On the other hand, a higher frequency in migrations might address performance issues more quickly.

7. Experimental results

The results shown in this section correspond to the normalised execution time against the OS baseline, shown in parentheses in the figures. For the LevelDB benchmarks, the normalised performance in μs/Op or MB/s is shown.

7.1. Experiment A

Figs. 5 and 6 show results for the Server HPL, while Figs. 7 and 8 show the results for Server LPL.

As mentioned before, little improvement can be expected in this experiment, since the benchmarks used are state-of-the-art implementations of their respective routines. In general terms, little overhead is introduced with the migration tool, about 8% of the total execution time. This overhead is compensated in most cases, even for LMMA which was expected to be ballasted by the inefficient Linux memory migration mechanism. Only the CIMAR algorithm produces significant performance losses in this scenario. Together with the NAS benchmarks, other Linux commands like `time` and scripts for automating the tests are running. Although these scripts imply negligible computation time, it is enough to trigger the flaws of the CIMAR regarding work-balance, and so increasing the execution time of the target code. With the NIMAR algorithm, those problems are alleviated, and results are generally better and more stable. Though, CIMAR makes important improvements in those tests with LevelDB databases with more LLC cache misses, that also rely massively on L1 cache hits, more than 90% of L1 cache accesses are valid. Since CIMAR pins the threads to the individual CPUs, the cache memory is kept in a "hot" state, producing a massive improvement when the L1 cache is used in such an intensive way. In NIMAR, the OS is free to move threads within a node, so whenever a thread is moved, the content of the L1 is no longer valid and some time is wasted retrieving back this information. In addition, read operations are further benefited because the hardware counters available on Intel platforms focus on data read transactions. LMMA does not improve execution times for NAS benchmarks, though it reduces the standard deviation of the results frequently, so execution times are more consistent across different runs. This is achieved by closing the distance between pages and threads, but also by balancing the memory workload when nodes are considered busy.

Note that NIMAR and LMMA algorithms are the more balanced options, producing little overhead while working, and even improving performance for some benchmarks like the BT, MG and SP, which can be considered memory-intensive codes.

7.2. Experiment B

We can see two differentiated behaviours in this experiment. Attending to the results of Server HPL in Fig. 9, migration algorithms increase the system performance. Those tasks that coincide with others, like the sp.C.x-1, lu.C.x-2, are the most benefited.

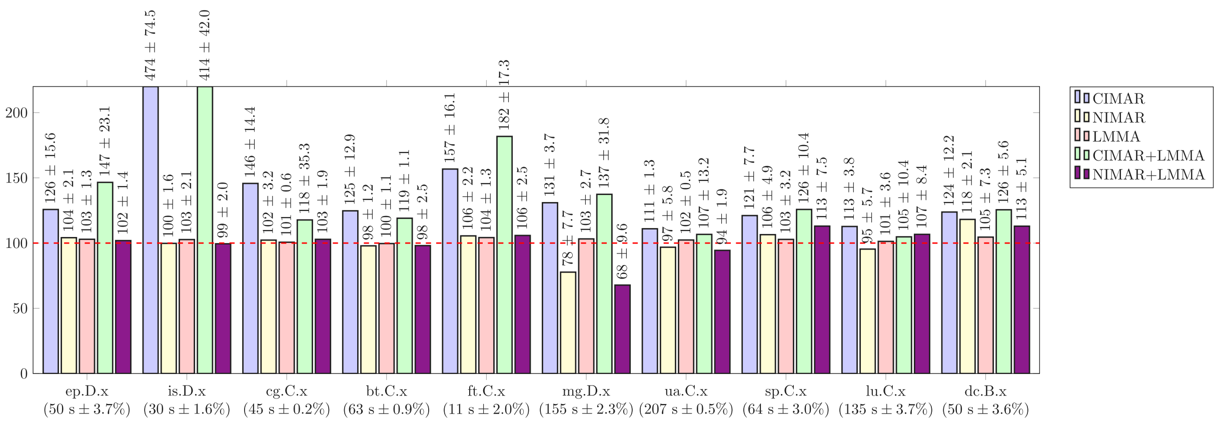


Fig. 5. Normalised execution time for NAS tests in Experiment A in Server HPL. Lower is better. Baseline in parentheses.

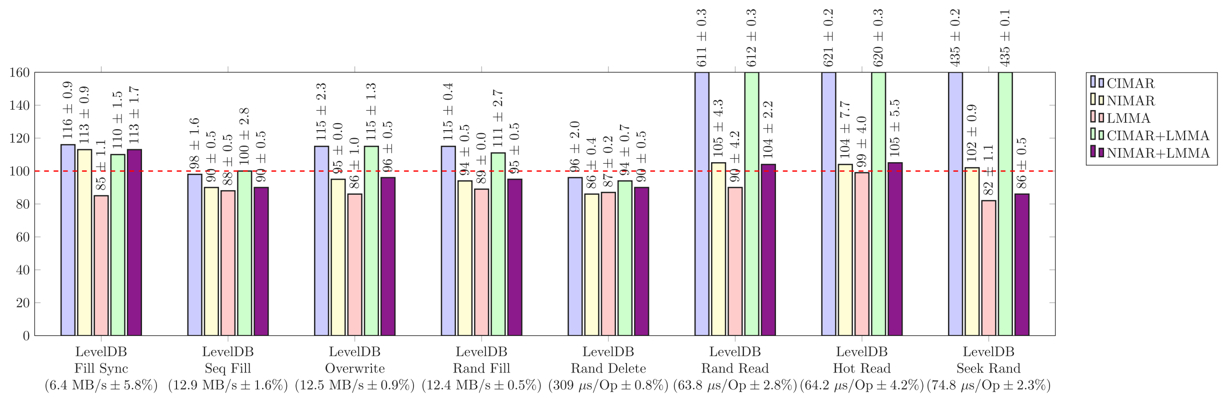


Fig. 6. Normalised performance for LevelDB tests in Experiment A in Server HPL. Higher is better. Baseline in parentheses.

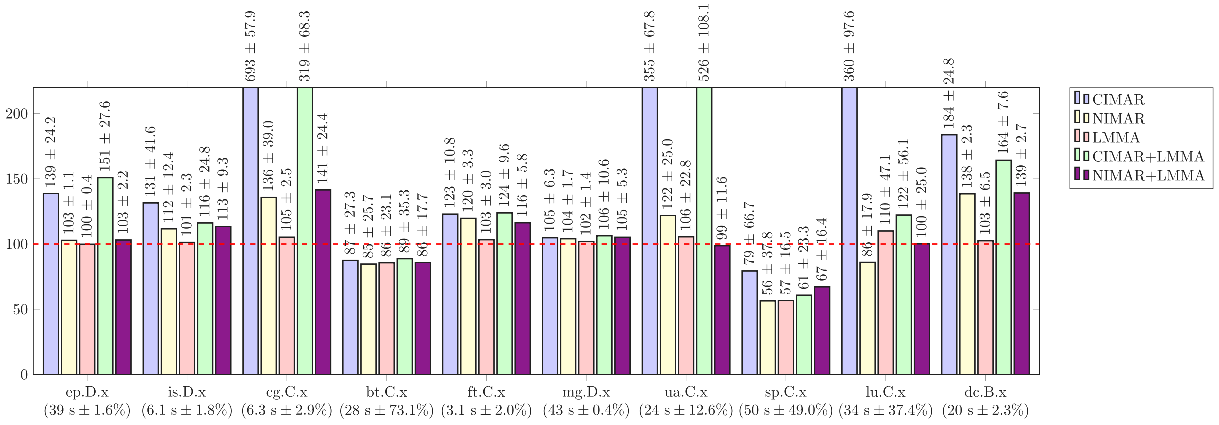


Fig. 7. Normalised execution time for NAS tests in Experiment A in Server LPL. Lower is better. Baseline in parentheses.

Note that the Direct mapping does not perform well in this scenario, increasing execution times for most tasks. Interleaving improves for some tasks, but it is still outperformed by CIMAR, NIMAR and LMMA. Comparing the migration algorithms, CIMAR and NIMAR achieve similar results for most tasks, being NIMAR more stable. This stability is further improved, the deviation is reduced, when the memory pages migrations are enabled with LMMA, resulting in more consistent execution times.

For Server LPL (see Fig. 10), results are different. For most tasks, there are little differences between the baseline policies and the best result among alternatives. Note that the CG tasks are the only ones that improve substantially their execution time, and only while using Direct mapping. For the rest of the tasks, migration algorithms results are comparable to those obtained with the best particular policy.

7.3. *Experiment C*

Figs. 11 and 12 show the results for the Servers HPL and LPL, respectively.

The impact of migrations is higher in the Server HPL since the differences in memory latency are also higher for this server. The Direct and Interleave mapping options do not improve overall performance, obtaining similar results to those obtained with the

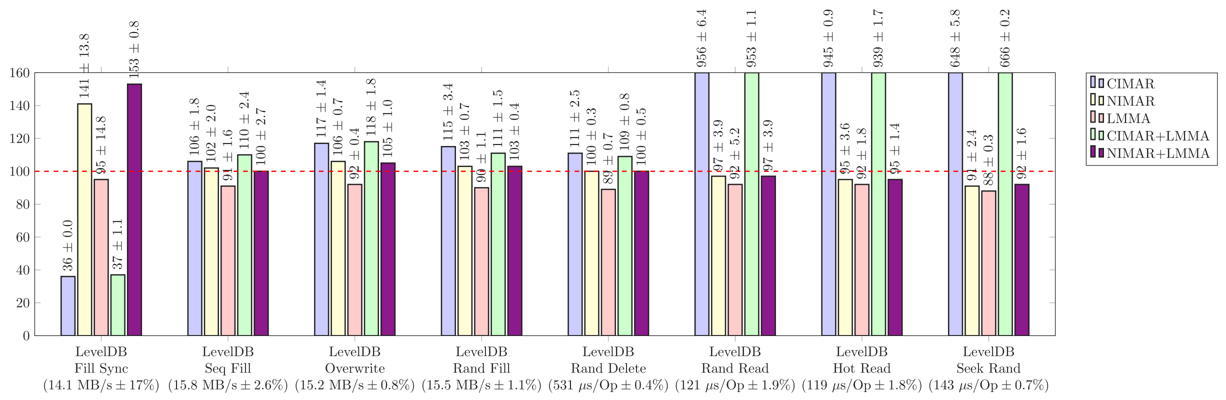


Fig. 8. Normalised performance for LevelDB tests in Experiment A in Server LPL. Higher is better. Baseline in parentheses.

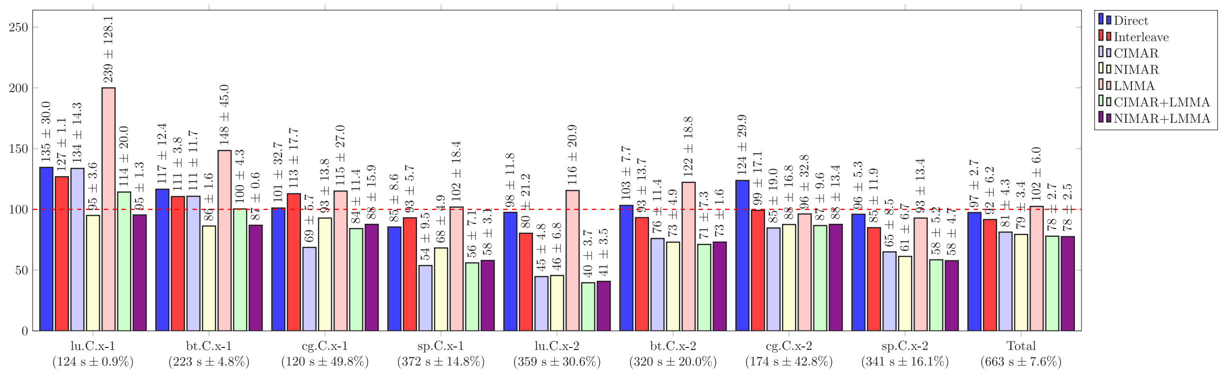


Fig. 9. Normalised execution times for each task and total test time for Experiment B in Server HPL. Lower is better. Baseline in parentheses.

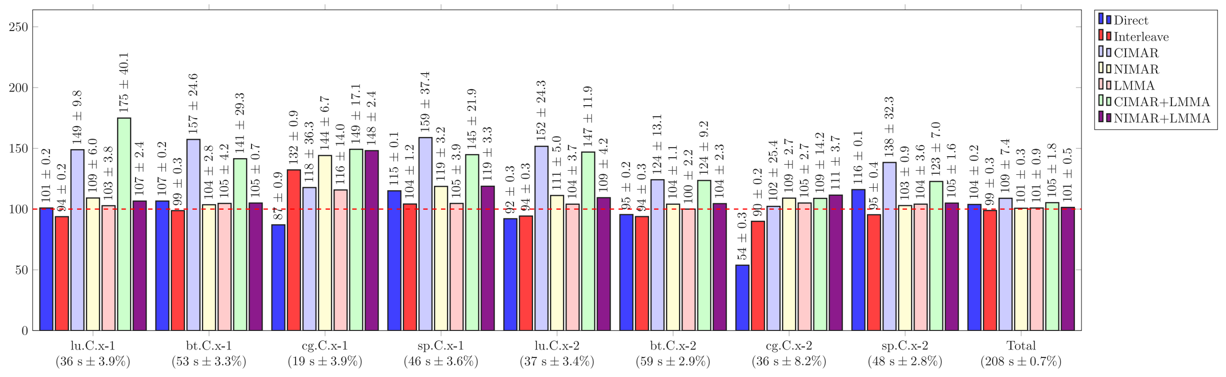


Fig. 10. Normalised execution times for each task and total test time for Experiment B in Server LPL. Lower is better. Baseline in parentheses.

OS default policies. However, our thread migration algorithms manage to improve up to 29% the test wall-time. Furthermore, some benchmarks like LU, MG and SP are significantly boosted, improving their execution time up to 49%. The UA benchmark, which is memory-intensive with irregular patterns, lowers its execution times significantly too. Generally, we can state that the benchmarks with more LLC misses and irregular memory patterns are the most benefited by our migration tool. In contrast, those which are intensive in cache use see little to no improvement.

It may seem that LMMA is not enough to improve system performance, with the exceptions of BT and LU benchmarks. Note that the standard deviation is reduced with the presence of this algorithm, making the system more stable in terms of performance and memory latency, so execution times are more consistent through different runs.

For the Server LPL, Direct mapping seems the best option attending to the individual tasks, especially for CG, IS, and DC, although it does not improve the total time compared to default OS policies. Interleaving is not an option, since it causes severe detriments for some tasks. Migration algorithms are near the baseline in terms of execution times, with slight improvements in the stability and performance for those programs with more LLC cache misses.

8. Conclusions

In this article, we presented three novel migration algorithms in the context of NUMA systems: CIMAR and NIMAR for threads, and LMMA for memory pages. These algorithms have been implemented in a user space tool that gathers the information from the hardware counters available in current processors. For the thread

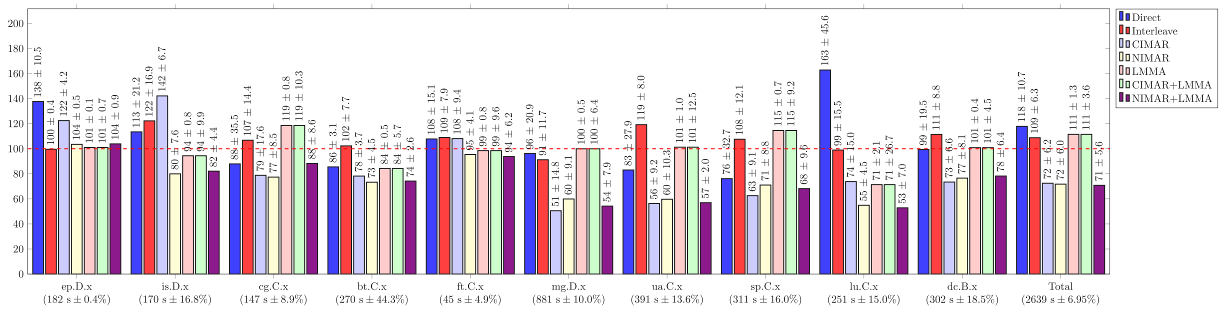


Fig. 11. Normalised execution times for each task and total test time for Experiment C in Server HPL. Lower is better. Baseline in parentheses.

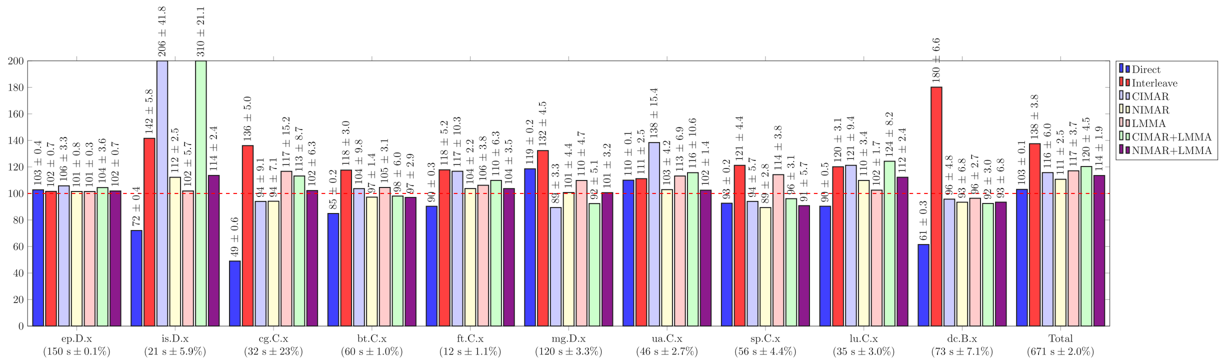


Fig. 12. Normalised execution times for each task and total test time for Experiment C in Server LPL. Lower is better. Baseline in parentheses.

migration algorithms, every possible migration is given a score according to the 3DyRM metrics and simple heuristics, and those with the highest score are performed. CIMAR pins the threads to individual cores, while NIMAR does the migration to a NUMA node, trusting the OS to decide the particular core. This way, the responsibility of keeping a proper work balance within a node resides on the OS, which is better fitted for the task because it can operate in kernel space. In the LMMA algorithm, memory pages are migrated according to the latency of the transactions in which they are involved. The objective of LMMA is to improve latency of memory accesses of the particular pages, and the system in general, by migrating them to the preferred or least busy nodes, according to the possible congestion.

Several experiments have been performed, to study different scenarios in terms of their different characteristics and the number of processes running in the system. The following lessons could be extracted from the results:

- Operating System with AutoNUMA patch performs well in most scenarios, being the fastest or being near the fastest alternative in most cases and has been proven a good compromise for general use.
- Interleave mapping is often slower than default OS policies, and might even cause significant slow-downs in systems with high local node bandwidths.
- Direct mapping is the theoretical best alternative and proved to be the best option in servers like LPL, where local node bandwidth is high. Otherwise, communication channels might get saturated, increasing the latency of memory operations and affecting the performance.
- Aggressive CPU pinning, like that performed by the CIMAR algorithm, is a good alternative, especially in multitasking environments and for applications that use L1 cache intensively. Unfortunately, it shows some flaws regarding work balance that might cause significant performance losses.
- NIMAR solves the problem with work balance by adopting pinning threads to NUMA nodes instead of particular cores. That way, it is the OS that decides which thread executes in which core within the given node. This is the preferred choice of the authors since it shows the best performance in most scenarios while keeping a low overhead.
- By migrating memory pages, LMMA manages to improve system stability, and even improve performance in some scenarios. Once that the aforementioned inefficiency of Linux memory migration [18,28] is improved, results of LMMA are expected to improve too.
- User space tools are a feasible option in terms of managing thread and memory mapping. Even though they introduce some overhead (under 8% in our tool), it can be compensated up to a point where system performance is improved, up to 60% in our tests.

Future work will be focused on the improvement of the memory migration algorithms. While performance has been improved in codes and systems in which latency is critical, there is room for improvements on those which require high bandwidth. The bandwidth of the system should be analysed and incorporated into the algorithms of memory migrations in order to achieve this objective.

CRediT authorship contribution statement

Ruben Laso: Conceptualization, Methodology, Software, Writing – original draft, Visualization. **Oscar G. Lorenzo:** Conceptualization, Software, Writing – original draft. **José C. Cabaleiro:** Conceptualization, Writing – original draft, Supervision. **Tomás F. Pena:** Conceptualization, Writing – original draft, Supervision. **Juan Ángel Lorenzo:** Conceptualization, Writing – original draft, Supervision. **Francisco F. Rivera:** Conceptualization, Writing – original draft, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research work has received financial support from the Ministerio de Ciencia e Innovación, Spain within the project PID2019-104834GB-I00. It was also funded by the Consellería de Cultura, Educación e Ordenación Universitaria of Xunta de Galicia (accr. 2019–2022, ED431G 2019/04 and reference competitive group 2019–2021, ED431C 2018/19).

Thanks to the people working at CESGA (https://www.cesga.es) for sharing with us anonymous data of the use of their systems.

References

[1] M. Ju, H. Jung, H. Che, A performance analysis methodology for multicore, multithreaded processors, IEEE Trans. Comput. 63 (2) (2014) 276–289, URL https://doi.org/10.1109/TC.2012.223.
[2] G.C. Chasparis, M. Rossbory, Efficient dynamic pinning of parallelized applications by distributed reinforcement learning, Int. J. Parallel Program. (2017) http://dx.doi.org/10.1007/s10766-017-0541-y.
[3] D. Gureya, J. Neto, R. Karimi, J.a. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano, V. Vlassov, Bandwidth-aware page placement in NUMA, in: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2020, pp. 546–556, http://dx.doi.org/10.1109/IPDPS47924.2020.00063.
[4] M. Schulz, B.R. de Supinski, PNMPI tools: A whole lot greater than the sum of their parts, in: SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, ACM/IEEE, 2007, pp. 1–10, http://dx.doi.org/10.1145/1362622.1362663.
[5] A. Cheung, S. Madden, Performance profiling with EndoScope, an acquisitional software monitoring framework, Proc. VLDB Endow. 1 (1) (2008) 42–53, URL https://doi.org/10.14778/1453856.1453866.
[6] M. Geimer, F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, B. Mohr, The scalasca performance toolset architecture, Concurr. Comput.: Pract. Exper. 22 (6) (2010) 702–719, URL https://doi.org/10.1002/cpe.1556.
[7] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, N.R. Tallent, Hpctoolkit: Tools for performance analysis of optimized parallel programs, Concurr. Comput.: Pract. Exper. 22 (6) (2010) 685–701, URL https://doi.org/10.1002/cpe.1553.
[8] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76, URL https://doi.org/10.1145/1498765.1498785.
[9] O.G. Lorenzo, T.F. Pena, J.C. Cabaleiro, J.C. Pichel, F.F. Rivera, 3DYrm: A dynamic roofline model including memory latency information, J. Supercomput. 70 (2) (2014) 696–708, URL https://doi.org/10.1007/s11227-014-1163-4.
[10] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, M. Roth, Traffic management: A holistic approach to memory placement on NUMA systems, SIGPLAN Not. 48 (4) (2013) 381–394, http://dx.doi.org/10.1145/2499368.2451157.
[11] C. Lameter, Local and remote memory: Memory in a linux/NUMA system, in: Linux Symposium, 2006, pp. 1–25.
[12] M. Diener, E.H. Cruz, P.O. Navaux, A. Busse, H.-U. Heiß, KMAF: Automatic kernel-level management of thread and data affinity, in: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, in: PACT '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 277–288, http://dx.doi.org/10.1145/2628071.2628085.
[13] I. Di Gennaro, A. Pellegrini, F. Quaglia, OS-based NUMA optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016, pp. 291–300, http://dx.doi.org/10.1109/CCGrid.2016.91.
[14] M.-L. Chiang, C.-J. Yang, S.-W. Tu, Kernel mechanisms with dynamic task-aware scheduling to reduce resource contention in NUMA multicore systems, J. Syst. Softw. 121 (2016) 72–87, http://dx.doi.org/10.1016/j.jss.2016.08.038, URL http://www.sciencedirect.com/science/article/pii/S0164121216301376.
[15] M.-L. Chiang, S.-W. Tu, W.-L. Su, C.-W. Lin, Enhancing inter-node process migration for load balancing on linux-based NUMA multicore systems, in: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Vol. 2, IEEE, 2018, pp. 394–399.
[16] M.-L. Chiang, W.-L. Su, S.-W. Tu, Z.-W. Lin, Memory-aware kernel mechanism and policies for improving internode load balancing on NUMA systems, Softw. - Pract. Exp. 49 (10) (2019) 1485–1508.
[17] C. Bienia, Benchmarking Modern Multiprocessors, (Ph.D. thesis), Princeton University, USA, 2011, AAI3445564.
[18] B. Lepers, V. Quema, A. Fedorova, Thread and memory placement on NUMA systems: Asymmetry matters, in: 2015 USENIX Annual Technical Conference (USENIX ATC 15), USENIX Association, Santa Clara, CA, 2015, pp. 277–289, URL https://www.usenix.org/conference/atc15/technical-session/presentation/lepers.
[19] M. Agung, M.A. Amrizal, R. Egawa, H. Takizawa, Deloc: A locality and memory-congestion-aware task mapping method for modern NUMA systems, IEEE Access 8 (2020) 6937–6953, http://dx.doi.org/10.1109/ACCESS.2019.2963726.
[20] H. Khaleghzadeh, H. Deldari, R. Reddy, A. Lastovetsky, Hierarchical multicore thread mapping via estimation of remote communication, J. Supercomput. 74 (3) (2018) 1321–1340, http://dx.doi.org/10.1007/s11227-017-2176-6.
[21] Q. Fettes, A. Karanth, R. Bunescu, A. Louri, K. Shiflett, Hardware-level thread migration to reduce on-chip data movement via reinforcement learning, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (11) (2020) 3638–3649.
[22] T.E. Carlson, W. Heirman, L. Eeckhout, Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–12.
[23] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, ACM SIGARCH Comput. Archit. News 23 (2) (1995) 24–36.
[24] Intel Corp, Intel 64 and IA-32 architectures software developer manuals, 2017, https://software.intel.com/articles/intel-sdm [Online; Dec. 2019].
[25] S. Eranian, Perfmon2: A Standard Performance Monitoring Interface for Linux, HP Labs, HP Labs, 2008, http://perfmon2.sf.net/perfmon2-20080124.pdf.
[26] D. Terpstra, H. Jagode, H. You, J. Dongarra, Collecting performance data with PAPI-c, in: M.S. Müller, M.M. Resch, A. Schulz, W.E. Nagel (Eds.), Tools for High Performance Computing 2009, Springer, Berlin, Heidelberg, 2010, pp. 157–173.
[27] R. Laso, O.G. Lorenzo, F.F. Rivera, J.C. Cabaleiro, T.F. Pena, J.A. Lorenzo, LBMA And IMAR^2: Weighted lottery based migration strategies for NUMA multiprocessing servers, Concurr. Comput.: Pract. Exper. 33 (11) (2021) e5950, http://dx.doi.org/10.1002/cpe.5950, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5950.
[28] J. Funston, M. Lorrillere, A. Fedorova, B. Lepers, D. Vengerov, J.-P. Lozi, V. Quéma, Placement of virtual containers on NUMA systems: A practical and comprehensive model, in: Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, in: USENIX ATC '18, USENIX Association, USA, 2018, pp. 281–293.
[29] Intel® memory latency checker v3.9, 2021, https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html (Accessed: 2021-04-21).
[30] H. Jin, M. Frumkin, J. Yan, The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance, Tech. Rep. Technical Report NAS-99-011, NASA Ames Research Center, 1999.
[31] S. Ghemawat, J. Dean, Leveldb, 2011.
[32] Phoronix test suite, 2021, https://www.phoronix-test-suite.com/ (Accessed: 2021-04-21).
[33] Intel® advisor, 2021, https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html (Accessed: 2021-06-07).
[34] A.C. De Melo, The new linux "perf" tools, in: Slides from Linux Kongress, Vol. 18, 2010, pp. 1–42.
[35] A.B. Yoo, M.A. Jette, M. Grondona, SLURM: Simple linux utility for resource management, in: D. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 44–60.
[36] A. Kleen, A NUMA API for Linux, Novel Inc, 2005.
[37] C. Lameter, NUMA (non-uniform memory access): An overview, ACM Queue 11 (7) (2013) 40, URL https://queue.acm.org/detail.cfm?id=2513149.
[38] A. Rane, D. Stanzione, Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems, in: Proc. of 10th LCI Int'L Conference on High-Performance Clustered Computing, 2009, pp. 1–10.

Ruben Laso obtained his B.Sc. in Computer Engineering in 2017 and his M.Sc. in Industrial Mathematics in 2019. Currently, he is a Ph.D. student at the University of Santiago de Compostela, working on the improvement of performance in NUMA servers via hardware counters. His research interests include computer architecture, particularly manycore and NUMA architectures, and the development of parallel algorithms in fields such as applied mathematics and numerical methods.

Oscar G. Lorenzo received the Ph.D. degree from the University of Santiago de Compostela in 2016, having previously obtained his B.Sc. in Computer Science Engineering in 2010 and M.Sc. in High Performance Computing in 2012, in said University. Currently he is a Post-Doc associate researcher at the Centro Singular de Investigación en Tecnoloxías Intelixentes (CITIUS).

José C. Cabaleiro received a Ph.D. degree from the University of Santiago de Compostela in 1994. Currently, he is an associate professor in the Department of Electronics and Computer Science at the University of Santiago de Compostela. Since 2010 he is a member of the Research Centre in IT (CiTIUS) of this University. His research interests include the architecture of parallel systems, the development of parallel algorithms for irregular problems and particularly for processing LiDAR data, and the prediction and improvement of the performance of parallel applications.

Tomás F. Pena got his Ph.D. in Physics in 1994 from the University of Santiago de Compostela (Spain). Since 1994, he is a professor in the Department of Electronics and Computer Science of the University of Santiago de Compostela. Since 2010, he is a senior researcher at the Research Center in IT (CiTIUS) of this University. His main research lines include high-performance computing in general, the architecture of parallel systems, the development of parallel algorithms for clusters and supercomputers, the optimisation of the performance in irregular codes and with sparse matrices, the prediction and improvement of the performance of parallel applications in general, and the use of Big Data technologies for scientific applications.

Juan Angel Lorenzo is an associate professor at CY Cergy Paris Université. After obtaining a M.Sc. degree in telecommunications engineering and a Ph.D. at the University of Santiago de Compostela, he worked at Hewlett-Packard Laboratories (Bristol, UK) as a researcher and cloud engineer. He later joined INRIA Bordeaux to carry out research in the field of resource allocation problems in cloud platforms. He has been a visiting researcher at the Edinburgh Parallel Computing Centre and the Department of Distributed and Dependable Systems at Charles University in Prague. His research interests lie in the development of strategies to improve resource locality in HPC, management of cloud infrastructures and profiling of large-scale architectures.

Francisco F. Rivera is a full professor at the University of Santiago de Compostela (Spain). Throughout his career, he has supervised researches and published extensively in the areas of computer-based applications, parallel processing and computer architecture. His current research interests include the compilation of irregular codes for parallel and distributed systems; the analysis and prediction of performance on parallel systems and the design of parallel algorithms; memory hierarchy optimisations, GIS, image processing, 3D point clouds computing, etc.