A New Hardware Counters Based Thread Migration Strategy for NUMA Systems

Oscar García Lorenzo[1](✉), Rubén Laso Rodríguez[1], Tomás Fernández Pena[1], Jose Carlos Cabaleiro Domínguez[1], Francisco Fernández Rivera[1], and Juan Ángel Lorenzo del Castillo[2]

[1] CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain
oscar.garcia@usc.es

[2] Quartz Research Lab - EISTI, Pau, France

Abstract. Multicore NUMA systems present on-board memory hierarchies and communication networks that influence performance when executing shared memory parallel codes. Characterising this influence is complex, and understanding the effect of particular hardware configurations on different codes is of paramount importance. In this paper, monitoring information extracted from hardware counters at runtime is used to characterise the behaviour of each thread in the processes running in the system. This characterisation is given in terms of number of instructions per second, operational intensity, and latency of memory access. We propose to use all this information to guide a thread migration strategy that improves execution efficiency by increasing locality and affinity. Different configurations of NAS Parallel OpenMP benchmarks running concurrently on multicore systems were used to validate the benefits of the proposed thread migration strategy. Our proposal produces up to 25% improvement over the OS for heterogeneous workloads, under different and realistic locality and affinity scenarios.

Keywords: Roofline model · Hardware counters · Performance · Thread migration

1 Introduction

Current multicores feature a diverse set of compute units and on-board memory hierarchies connected by increasingly complex communication networks and protocols. For a parallel code to be correctly and efficiently executed in a multicore system, it must be carefully programmed, and memory sharing stands out as a *sine qua non* for general purpose programming. The behaviour of the code depends also on the status of the processes currently executed in the system. A programming challenge for these systems is to partition application tasks, mapping them to one of many possible thread-to-core configuration to achieve a desired performance in terms of throughput, delay, power, and resource consumption, among others [11]. The behaviour of the system can dynamically change when multiple processes are running with several threads each. The number of mapping choices increases as the number of cores and threads do. Note that, in general purpose systems, the number of multithreaded processes can be large and change dynamically. Concerning architectural features, particularly those that determine the behaviour of memory accesses, it is critical to improve locality and affinity among threads, data, and cores. Performance issues that are impacted by this information are, among others, data locality, thread affinity, and load balancing. Therefore, addressing these issues is important to improve performance.

A number of performance models have been proposed to understand the performance of a code running on a particular system [1,4,6,17]. In particular, the roofline model (RM) [18] offers a balance between simplicity and descriptiveness based on the number of FLOPS (Floating Point Operations per Second) and the operational intensity, defined as the number of FLOPS per byte of DRAM traffic (flopsB). The original RM presented drawbacks that were taken into account by the 3DyRM model [14], which extends the RM model with an additional parameter, the memory access latency, measured in number of cycles. Also, 3DyRM shows the dynamic evolution of these parameters. This model uses the information provided by Precise Event Based Sampling (PEBS) [8,9] on Intel processors to obtain its defining parameters (flopsB, GFLOPS, and latency). These parameters identify three important factors that influence performance of parallel codes when executed in a shared memory system, and in particular, in non-uniform memory access (NUMA) systems. In a NUMA system, distance and connection to memory cells from different cores may induce variations in memory latency, and so the same code may perform differently depending on where it was scheduled, which may not be detectable in terms of the traditional RM.

Moving threads close to where their data reside can help alleviate memory related performance issues, especially in NUMA systems. Note that when threads migrate, the corresponding data usually stays in the original memory module, and they are accessed remotely by the migrated thread [3]. In this paper, we use the 3DyRM model to implement strategies for migrating threads in shared memory systems and, in particular, multicore NUMA servers, possible with multiple concurrent users. The concept is to use the defining parameters of 3DyRM as objective functions to be optimised. Thus, the problem can be defined in terms of a multiobjective optimisation problem. The proposed technique is an iterative method inspired from evolutionary optimisation algorithms. To this end, an individual utility function to represent the relative importance of the 3DyRM parameters is defined. This function uses the number of instructions executed, operational intensity, and average memory latency values, for providing a characterisation of the performance of each parallel thread in terms of locality and affinity.

2 Characterisation of the Performance of Threads

The main bottleneck in shared memory codes is often the connection between the processors and memory. 3DyRM relates processor performance to off-chip memory traffic. The Operational Intensity (OI) is the floating operations per byte of DRAM traffic (measured in flopsB). OI measures traffic between the caches and main memory rather than between the processor and caches. Thus, OI incorporates the DRAM bandwidth required by a processor in a particular computer, and the cache hierarchy, since better use of cache memories would mean less use of main memory. Note that OI is insufficient to fully characterise memory performance, particularly in NUMA systems. Extending RM with the mean latency of memory access provides a better model of performance. Thus, we employ the 3DyRM model, which provides a three dimensional representation of thread performance on a particular placement.

PEBS is an advanced sampling feature of Intel Core based processors, where the processor directly records samples from specific hardware counters into a designated memory region. The use of PEBS as a tool to monitor a program execution was already implemented in [15], providing runtime dynamic information about the behaviour of the code with low overhead [2], as well as an initial version of a thread migration tool tested with a toy examples. The migration tool presented in this work continuously gathers performance information in terms of the 3DyRM, i.e. GFLOPS, flopsB, and latency, for each core and thread. However, the information about floating point operations provided by PEBS may sometimes be inaccurate [9] or difficult to obtain. In addition, accurate information about retired instructions can be easily obtained, so giga instructions per second (GIPS) and instructions retired per byte (instB) may be used rather than GFLOPS and flopsB, respectively. For this reason, GIPS and instB are used in this work.

3 A New Thread Migration Strategy

We introduce a new strategy for guiding thread migration in NUMA systems. The proposed algorithm is executed every T milliseconds to eventually perform threads migrations. The idea is to consider the 3DyRM parameters as objective functions to be optimised, so increasing GFLOPS (or GIPS) and flopsB (or instB), and decreasing latency in each thread improve performance in the parallel code. There is a close relation between this and multiobjective optimisation (MOO) problems, which have been extensively studied [5]. The aim of many MOO solutions is to obtain the Pareto optimality numerically. However, this task is usually computationally intensive, and consequently a number of heuristic approaches have been proposed.

In our case, there are no specific functions to be optimised. Rather, we have a set of values that are continuously measured in the system. Our proposal is to apply MOO methods to address the problem using the 3DyRM parameters. Thread migration is then used to modify the state of each thread to simultaneously optimise the parameters. Therefore, we propose to characterise each thread using an aggregate objective function, P, that combines these three parameters.

Consider a system with N computational nodes or cores in which, at certain time, multiple multithreaded processes are running. Let P_{ijk} be the performance for the i-th thread of the j-th process when executed on the k-th computational node. We define the aggregate function as

$$P_{ijk} = \frac{\mathrm{GIPS}_{ijk} \cdot \mathrm{intsB}_{ijk}}{\mathrm{latency}_{ijk}}, \tag{1}$$

where GIPS_{ijk} is the GIPS of the thread, and instB_{ijk} and $\mathrm{latency}_{ijk}$ are the instB and average latency values, respectively. Note that, larger values of P_{ijk} imply better performance.

Initially, no values of P_{ijk} are available for any thread on any node. On each time interval, P_{ijk} is computed for every thread on the system according to the values read by the hardware counters. In every interval some values of P_{ijk} are updated, for those nodes k where each thread was executed, while others store the performance information of each thread when it was executed in a different node (if available). Thus, the algorithm adapts to possible behaviour changes for the threads. As threads migrate and are executed on different nodes, more values of P_{ijk} are progressively filled up.

To compare threads from different processes, each individual P_{ijk} is divided by the mean P_{ijk} of all threads of the same process, i.e. the j-th process,

$$\widehat{P}_{ijk} = \frac{P_{ijk}}{\sum_{m=1}^{n_j} P_{mjh}/n_j}, \tag{2}$$

where n_j is the number of threads of process j, and h is, for each thread m of the j-th process, the last node where it was running.

Every T milliseconds, once the new values of P_{ijk} are computed, the thread with the worst current performance, in terms of P_{ijk}, is selected to be migrated. Thus, for each process, those threads with $\widehat{P}_{ijk} < 1$ are currently performing worse than the mean of the threads in the same process, and the worst performing thread in the system is considered to be the one with the lowest $\widehat{P}_{ijk}$, i.e., the thread performing worse when compared to the other threads of its process. This is identified as the migration thread, and denoted by Θ_m.

Note that the migration can be to any core in a node other than the current node in which Θ_m resides. A weighted random process is used to choose the destination core, based on the stored performance values. In order to consider all possible migrations, all P_{ijk} values have to be taken into account. Therefore, it is important to fill as many entries of P_{ijk} as possible.

A lottery strategy is proposed in such a way that every possible destination is granted a number of tickets defined by the user, some examples are shown in Table 1, according to the likelihood of that migration improving performance. The destination with the larger likelihood has a greater chance of being chosen. Migration may take place to an empty core, where no other thread is currently being executed, or to a core occupied with other threads. If there are already threads in the core, one would have to be exchanged with Θ_m. The swap thread is denoted as Θ_g, and all threads are candidates to be Θ_g. Note that, although not all threads may be selected to be Θ_m (e.g. a process with a single thread

Table 1. List of tickets.

Ticket	Description	Default value
`MEM_CELL_WORSE`	Previous data show worst performance in a given node	1
`MEM_CELL_NO_DATA`	No previous data in a given memory node	2
`MEM_CELL_BETTER`	Previous data show better performance in a given node	4
`FREE_CORE`	It is possible to migrate a thread to a free core	2
`PREF_NODE`	It is possible to migrate a thread to a core located in the node in which it makes most of its memory accesses	4
`THREAD_UNDER_PERF`	It is possible to interchange a thread with another whose relative performance in under a determined threshold	3

would always have $\widehat{P}_{ijk} = 1$ and so never be selected), they may still be considered to be Θ_g to ensure the best performance for the whole system. When all tickets have been assigned, a final destination core is randomly selected based on the awarded tickets. If the destination core is free, a simple migration will be performed. Otherwise, an interchanging thread, Θ_g, is chosen from those currently being executed on that core. Once the threads to be migrated are selected, the migrations are actually performed.

Migrations may affect not only the involved threads, Θ_m and Θ_g, but all threads in the system due to synchronisation or other collateral relations among threads. The total performance for each iteration can be calculated as the sum of all P_{ijk} for all threads. Thus, the current total performance, Pt_{current}, characterises a thread configuration, independently of the processes being executed. The total performance of the previous iteration is stored as Pt_{last}. On any interval, Pt_{current} may increase or decrease relatively to Pt_{last}. Depending on this variation, decisions are made regarding the next step of the algorithm.

Our algorithm dynamically adjusts the number of migrations per unit of time by changing T between a given minimum, $T_{\min}$, and maximum, $T_{\max}$, doubling or halving the previous value. To do that, a ratio, $0 \leq \omega \leq 1$ is defined for $Pt_{\text{current}}/Pt_{\text{last}}$, to limit an acceptable decrement in performance. So, if a thread placement has a lower total performance, more migrations should be performed to try to get a better thread placement, because they are likely to increase performance ($Pt_{\text{current}} \geq \omega Pt_{\text{last}}$). This way, T is decreased to perform migrations more often and reach optimal placement quicker. However, if current thread placement has high total performance, migrations have a greater chance of being detrimental. In this case, if $Pt_{\text{current}} < \omega Pt_{\text{last}}$, T is increased. Additionally, a rollback mechanism is implemented, to undo migrations if they result in a significant loss of performance, returning migrated threads to their former locations. Summarising, the rules guiding our algorithm are:

- If $Pt_{\text{current}} \geq \omega Pt_{\text{last}}$ then the total performance improves, so, migrations are considered productive, T is halved ($T \to T/2$), and a new migration is performed according to the rules indicated previously.
- If $Pt_{\text{current}} < \omega Pt_{\text{last}}$ then the total performance decreases more than a given threshold ω, so, migrations are considered counter-productive, T is doubled ($T \to 2 \times T$), and the last migration is rolled back.

This algorithm is named Interchange and Migration Algorithm with Performance Record and Rollback (IMAR^2). To simplify notation, IMAR^2 and its parameters are denoted as IMAR$^2[T_{\min}, T_{\max}; \omega]$.

4 Experimental Results

NPB-OMP benchmarks [10] were used to study the effect of the memory allocation. They are broadly used and their diverse behaviour when executed is well known. These benchmarks are well suited for multicore processors, although they do not greatly stress the memory of large servers. To study the effects of NUMA memory allocation, different memory stress situations were considered using the `numactl` tool [12], which allows the memory cell to store specific data, and threads to be pinned to specific cores or processors. Two servers were used to test NUMA effects. Both processors have one memory controller with four memory channels for connecting DIMM memory chips. In both systems node 0 has greater affinity with cell 0, node 1 with cell 1, and so on. Also, a NUMA aware Linux kernel was used. More specifically:

- Server A: An Ubuntu 14, with Linux kernel 3.10, NUMA server with four nodes, each has one octo-core Xeon E5-4620 (32 physical cores in total), Sandy Bridge architecture, 16 MB L3 cache, 2.2 GHz–2.6 GHz, and 512 GB of RAM. This server has memory chips connected in all four memory channels and may use all the available memory bandwidth.
- Server B: A Debian GNU/Linux 9, kernel version 5.1.15 composed by four nodes with Intel Xeon E5-4620 v4 with 10 cores each (40 in total), Broadwell-EP architecture, 25 MB L3 cache, 2.1 GHz–2.6 GHz, and 128 GB of RAM. Only one memory channel is used in this server, increasing the chances of memory congestion in remote accesses and increasing NUMA effects.

We designed experiments in which four instances of the NPB-OMP benchmarks are executed concurrently, and the placement of each can be controlled. Each benchmark instance was executed as a multi-threaded process with just enough threads to fill one node. We tested a representative set of memory and thread placements. The memory placements are:

- Free: No specific memory placement is selected, the OS decides where to place the data of each benchmark. This is the most common case for regular users.

– Direct: Each benchmark have its preferred memory set to a different cell. In the case of four benchmarks, each one have one memory cell for its data, as long as its memory is large enough. This is a common option used by experienced users who know the limits of their applications [13,16].
– Interleaved: Each benchmark have its memory set to interleaved, with each consecutive memory page set to a different memory cell in a round robin fashion. This is a common option used by experienced users who do not know the specific characteristics of their programs or want to use all the available bandwidth.

and the thread one's:

– OS: The OS decides where to place the threads, as well as their possible migrations. Note that the four benchmarks can not be initiated at exactly the same time, but only one at a time. This fact influences the initial thread placement. This is the most common case for regular users.
– Pinned: Each benchmark had its threads pinned to one node. When combined with the direct memory placement the same node is used for one benchmark. This is a common option used by experienced users [7].
– IMAR^2: The IMAR^2 algorithm is used to place and migrate the threads.

Different combinations of these memory and thread placements were tested. Results of four class C NPB-OMP codes were selected to be shown in this paper: `lu.C`, `sp.C`, `bt.C` and `ua.C`. Benchmarks were compiled with gcc and O2 optimisation. This selection was made according to three following criteria: First, these are codes with different memory access patterns and different computing requirements. The DyRM model was used to select two benchmarks with low flopsB (`lu.C` and `sp.C`) and two with high flopsB (`bt.C` and `ua.C`). Second, since the execution times of these codes are similar, they remain in concurrent execution most of the time. This helps studying the effect of thread migrations. Third, they are representative to understand the behaviour of our proposal.

Each test was executed on the four nodes, combined as four processes of the same code that produced four combinations, named **4 lu.C**, **4 sp.C**, **4 bt.C**, and **4 ua.C**, and four processes of different codes, that produced one combination named (**lu.C/sp.C/bt.C/ua.C**). Tables 2 and 3 show the results for servers A and B, respectively. The times for all benchmarks of **lu.C/sp.C/bt.C/ua.C** are shown, whereas only the times of the slowest instances are shown for the four equal benchmarks. A graphical comparison is shown in Fig. 1, where times of each test are normalised to the time of a normal OS execution, the free memory placement with OS thread placement, with times in the first column of Tables 2 and 3 are shown as a percentage. A percentage greater that 100 means a worse execution time, while a result under 100 shows a better execution time.

Table 2. Times for four NAS benchmarks in server A. When all benchmarks are of the same kind only the time of the slowest is shown. Best time on each row is remarked in **bold**. Best time for each memory policy is shown in *italics*.

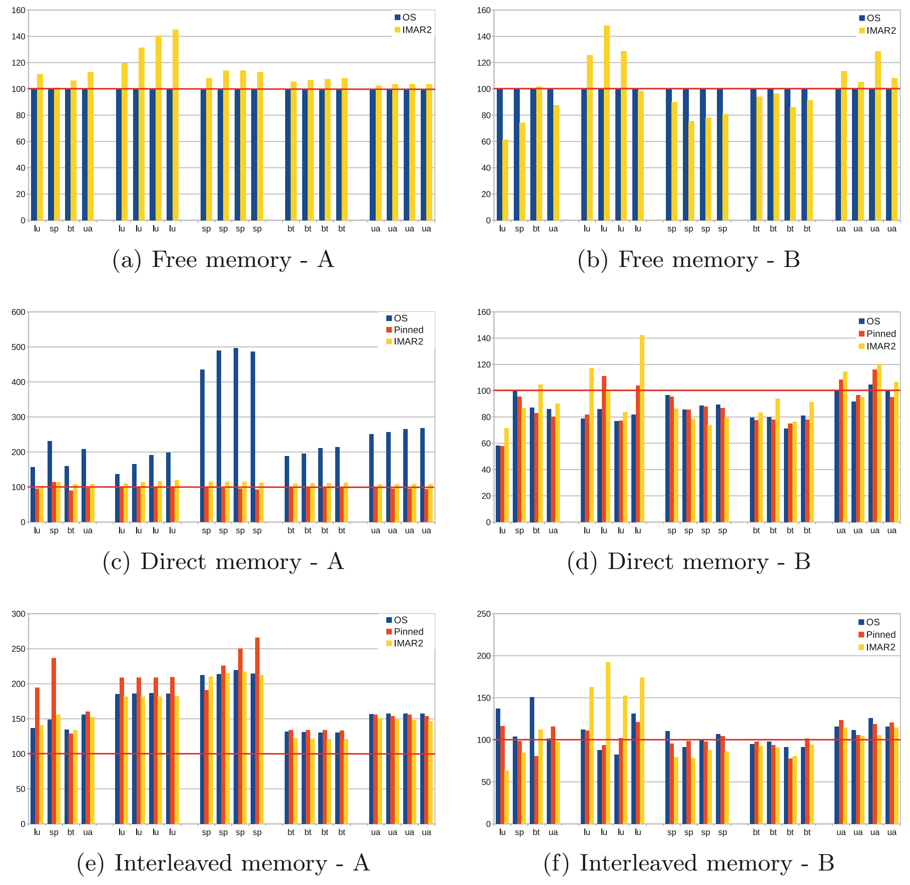
Test		Time (s)							
Benchmarks		Free		Direct			Interleaved		
		OS	IMAR^2	OS	Pinned	IMAR^2	OS	Pinned	IMAR^2
lu.C/sp.C bt.C/ua.C	lu	*220.24*	245.05	344.82	**210.00**	223.33	*300.55*	428.41	310.68
	sp	**235.53**	238.39	544.63	267.89	*267.86*	*350.73*	557.39	367.57
	bt	*201.69*	214.50	321.39	**180.77**	217.15	271.34	*260.46*	270.52
	ua	*197.03*	222.02	409.35	**190.26**	212.27	307.57	316.26	*299.89*
4 lu.C		*215.84*	313.24	428.85	**212.20**	258.43	401.49	452.15	*392.84*
4 sp.C		*287.49*	324.00	1397.28	**267.71**	323.59	616.40	763.88	*610.91*
4 bt.C		*185.37*	200.70	395.95	**182.29**	207.21	241.76	246.90	*223.57*
4 ua.C		*203.54*	211.21	545.63	**190.46**	220.65	319.67	313.59	*297.92*

Table 3. Times for four NAS benchmarks in server B. When all benchmarks are of the same kind only the time of the slowest is shown. Best time on each row is remarked in **bold**. Best time for each memory policy is shown in *italics*.

Test		Time (s)							
Benchmarks		Free		Direct			Interleaved		
		OS	IMAR^2	OS	Pinned	IMAR^2	OS	Pinned	IMAR^2
lu.C/sp.C bt.C/ua.C	lu	305.00	*187.00*	177.08	**176.37**	217.99	417.75	355.42	*194.01*
	sp	476.00	**354.95**	474.79	453.60	*412.59*	494.71	469.10	*402.60*
	bt	*276.75*	281.97	241.27	*229.83*	289.74	417.75	**222.39**	310.07
	ua	371.87	*326.74*	319.47	**298.33**	335.64	376.74	430.46	*363.81*
4 lu.C		*263.26*	341.69	**199.27**	259.40	326.85	*293.02*	317.60	449.19
4 sp.C		758.59	*592.73*	619.26	642.74	**569.48**	780.20	762.14	*627.22*
4 bt.C		322.58	*291.79*	**225.72**	232.30	267.85	305.67	299.00	*280.73*
4 ua.C		*316.93*	378.06	**297.95**	348.99	364.65	400.66	409.65	*358.03*

4.1 Server A

Note that the direct memory placement with pinned threads gets the best execution time (it is below 100), while interleaved memory and pinned threads is not a good solution in this case. In Fig. 1(a) the results of using IMAR^2 with free memory placement are also shown and, in this case, the migrations do not improve, but actually decrease performance. This is due to the fact that IMAR^2 does not move memory, it depends on the OS for that, so it cannot reach as good results as the direct memory with pinned threads. Note that in this case the `sp.C` benchmark takes a longer time to execute, so it is favoured in the end by having the whole system for itself; both IMAR^2 and OS are able to take it into account and reach a similar end time. In Fig. 1(c), results with direct memory are shown. In this case the OS does not migrate threads or memory



(a) Free memory - A (b) Free memory - B

(c) Direct memory - A (d) Direct memory - B

(e) Interleaved memory - A (f) Interleaved memory - B

Fig. 1. Normalised execution times (in seconds, Y axis) against FREE - OS for all tests

taking into account that the benchmarks have their memory on just one node, so it results in worse performance; meanwhile the IMAR^2 migrations are able to move the threads to their correct location and performance does not suffer much. In Fig. 1(e), the results with INTERLEAVED memory are shown. Neither in this case is the OS able to fix the memory or thread placement, and results are not worse that leaving the OS alone; IMAR^2 migrations are able to improve the OS somewhat, but, since they cannot move the memory, the margin for improvement is low. In conclusion, in this system, while migrations may improve the OS in certain cases, the OS does a good work and there is little margin for improvement.

4.2 Server B

Figure 1(b) shows that global performance is greatly improved when different benchmarks run concurrently compared to the OS scheduling when memory policy is FREE. Improvements reach up to 38% the individual execution times, and up to 25% in total execution time. Note that, a migration strategy has more chances for improving performance when different processes are executed, as they may have different memory requirements. When a set of instances of the same benchmark is executed, results depend heavily on the behaviour of the code. As mentioned before, the influence of migrations is huge in **4 sp.C**, since memory latency is critical. The case for **4 bt.C** is similar, which improves too. For **4 lu.C** and **4 ua.C**, memory saturation makes almost impossible to improve the results, and even migrations cause a performance slowdown. When the memory is directly mapped to a node, see Fig. 1(d), OS outperforms the PINNED scheduling in many of the cases. Due to the work balance, OS mitigates the possible memory congestion caused when all the data is placed in a single memory node. Is must be noted that in this situation IMAR^2 improves the execution times of `sp.C`, the most memory intensive benchmark. Finally, when the INTERLEAVED strategy for memory is used, Fig. 1(f), IMAR^2 succeeds in achieving the best performance in the memory intensive benchmarks, thanks to a better thread placement through the cores of the server.

5 Conclusions

Thread and data allocation significantly influence the performance of modern computers, being this fact particularly true in NUMA systems. When the distribution of threads and data is far from being the optimum, the OS by itself is not able to optimise it. In this paper, a dynamic thread migration algorithm to deal with this issue is proposed. It is based on the optimisation of the operational intensity, the GIPS, and the memory latency, parameters that define the 3DyRM model. The proposed technique improves execution times when thread locality is poor and the OS is unable to improve thread placement in runtime.

In this paper, we define a product that combines the three 3DyRM parameters in a single value, which can be considered a fair representation of the whole performance of the system in terms of locality and affinity. To optimise this value, we propose a migration algorithm, named IMAR^2, based on a weighted lottery strategy. Hardware counters allow us to obtain information about the performance of each thread in the system in runtime with low overhead. IMAR^2 uses this information to quantify the 3DyRM parameters and then performs thread migration and allocation in runtime. Using benchmarks from the NPB-OMP, we evaluate IMAR^2 in a variety of scenarios. Results show that our algorithm improves execution time by up to 25% in realistic scenarios in terms of locality and affinity. Besides, only small performance losses were obtained in cases where the thread configuration was initially good. Rollbacks and changes in the time between migrations are mechanisms to adapt dynamically to the current behaviour of the system as a whole. These provide better results in cases where migrations are unnecessary, while still improving the performance in cases with low initial performance.

Several improvements might be considered as future work, like a precise measurement of FLOPS, including vector extensions, that could improve both performance estimation and migration decisions. Also, some modifications of the current objective function might be explored, like weighing its parameters or even testing different functions. Finally, other migration algorithms could be considered, maybe based on stochastic scheduling, optimisation techniques, or other state of the art approaches.

Acknowledgements. This work has received financial support from the Consellería de Cultura, Educación e Ordenación Universitaria (accreditation 2016-2019, ED431G/08 and reference competitive group 2019-2021, ED431C 2018/19) and the European Regional Development Fund (ERDF). It was also funded by the Ministerio de Economía, Industria y Competitividad within the project TIN2016-76373-P.

References

1. Adhianto, L., Banerjee, S., Fagan, M., et al.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput.: Pract. Exp. **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.1553
2. Akiyama, S., Hirofuchi, T.: Quantitative evaluation of intel PEBS overhead for online system-noise analysis. In: Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017, ROSS 2017, pp. 3:1–3:8. ACM, New York (2017). https://doi.org/10.1145/3095770.3095773
3. Chasparis, G.C., Rossbory, M.: Efficient dynamic pinning of parallelized applications by distributed reinforcement learning. Int. J. Parallel Program. **47**(1), 24–38 (2017). https://doi.org/10.1007/s10766-017-0541-y
4. Cheung, A., Madden, S.: Performance profiling with EndoScope, an acquisitional software monitoring framework. Proc. VLDB Endow. **1**(1), 42–53 (2008). https://doi.org/10.14778/1453856.1453866
5. Cho, J.H., Wang, Y., Chen, R., Chan, K.S., Swami, A.: A survey on modeling and optimizing multi-objective systems. IEEE Commun. Surv. Tutor. **19**, 1867–1901 (2017). https://doi.org/10.1109/COMST.2017.2698366
6. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurr. Comput.: Pract. Exp. **22**(6), 702–719 (2010). https://doi.org/10.1002/cpe.1556
7. Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N.: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. J. Supercomput. **50**(1), 36–77 (2009). https://doi.org/10.1007/s11227-008-0251-8
8. Intel Corp.: Intel 64 and IA-32 Architectures Software Developer Manuals (2017). https://software.intel.com/articles/intel-sdm. Accessed Nov 2019
9. Intel Developer Zone: Fluctuating FLOP count on Sandy Bridge (2013). http://software.intel.com/en-us/forums/topic/375320. Accessed Nov 2019
10. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, Technical Report NAS-99-011, NASA Ames Research Center (1999)
11. Ju, M., Jung, H., Che, H.: A performance analysis methodology for multicore, multithreaded processors. IEEE Trans. Comput. **63**(2), 276–289 (2014). https://doi.org/10.1109/TC.2012.223

12. Kleen, A.: A NUMA API for Linux. Novel Inc. (2005)
13. Lameter, C., et al.: NUMA (non-uniform memory access): an overview. ACM Queue **11**(7), 40 (2013). https://queue.acm.org/detail.cfm?id=2513149
14. Lorenzo, O.G., Pena, T.F., Cabaleiro, J.C., Pichel, J.C., Rivera, F.F.: 3DyRM: a dynamic roofline model including memory latency information. J. Supercomput. **70**(2), 696–708 (2014). https://doi.org/10.1007/s11227-014-1163-4
15. Lorenzo, O.G., Pena, T.F., Cabaleiro, J.C., Pichel, J.C., Rivera, F.F.: Multiobjective optimization technique based on monitoring information to increase the performance of thread migration on multicores. In: 2014 IEEE International Conference on Cluster Computing (CLUSTER), pp. 416–423. IEEE (2014). https://doi.org/10.1109/CLUSTER.2014.6968733
16. Rane, A., Stanzione, D.: Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems. In: Proceedings of 10th LCI International Conference on High-Performance Clustered Computing (2009)
17. Schulz, M., de Supinski, B.R.: PNMPI tools: a whole lot greater than the sum of their parts. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (2007). https://doi.org/10.1145/1362622.1362663
18. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009). https://doi.org/10.1145/1498765.1498785