IADIS International Journal on Computer Science and Information Systems
Vol. 7, No.1, pp. 79-93
ISSN: 1646-3692

EVALUATING CPU AND MEMORY AFFINITY FOR NUMERICAL SCIENTIFIC MULTITHREADED BENCHMARKS ON MULTI-CORES

Christiane P. Ribeiro, Márcio Castro, Vania Marangozova-Martin,
Jean-François Méhaut. *Nanosim team, Laboratoire d'Informatique de Grenoble (LIG),CNRS, INRIA,CEA, University of Grenoble. ZIRST 51, avenue Jean Kuntzmann, 38330 Montbonnot Saint Martin, France.*

Henrique C. Freitas, Carlos A. P. S. Martins. *Department of Computer Science – Pontifical Catholic University of Minas Gerais (PUC Minas). Av. Dom José Gaspar, 500, Belo Horizonte, MG, Brazil.*

ABSTRACT

Modern multi-core platforms feature complex topologies with different cache levels and hierarchical memory subsystems. Consequently, thread and data placement become crucial to achieve good performance. In this context, CPU and memory affinity appear as a promising approach to match the application characteristics to the underlying architecture. In this paper, we evaluate CPU and memory affinity strategies for numerical scientific multithreaded benchmarks on multi-core platforms. We use and analyze hardware performance event counters in order to have a better understanding of such impact. Indeed, the results obtained on different multi-core platforms and Linux kernels show that important performance improvements (up to 70%) can be obtained when applying affinity strategies that fit both the application and the platform characteristics.

KEYWORDS

Performance, Affinity, NAS Benchmarks, Multi-core Platforms.

1. INTRODUCTION

Modern multi-core platforms are designed with a hierarchical memory topology to reduce the communication latency and to increase bandwidth. Depending on the architecture design decisions, the shared main memory can be either composed of a single memory bank (UMA - Uniform Memory Access) or distributed in several memory banks (NUMA - Non-Uniform

Memory Access) [Asanovic, 2006]. In addition to the main memory topology, such platforms also feature multiple levels of cache, which can be disposed in many different ways. Usually, cores on the same chip share cache whereas cores on different chips do not.

In this paper we argue that the memory architecture has a great impact on applications' performance and that it should be explicitly taken into account. Indeed, when dealing with parallel applications whose major performance metrics include the speed of execution and the speedup [Foster, 1995], non uniform memory accesses may break the load balancing and thus slow down the application. On the other hand, knowledge and intelligent usage of the cache management strategies may prevent data from moving around and thus accelerate the computations.

To respond to the variability of architectural characteristics and ensure good application performance, it is necessary to efficiently manage thread and data locality. In this context emerges the concept of *affinity*, which can be divided into two types: *CPU affinity* and *memory affinity*. *CPU affinity* forces a thread to run on a specific core or a subset of cores. The idea is to take advantage of the fact that the data accessed by the thread may remain in the processor cache. However, in order to do so, the default behavior of the operating system (OS) scheduler has to be changed [Mei, 2010] [Castro, 2012].

Memory affinity [Terboven, 2008] is ensured when data is efficiently distributed over the machine memory. Such distribution can either reduce the number of remote accesses (latency optimization), which may be very significant on NUMA platforms, or the memory contention (bandwidth optimization), which can be present on both UMA and NUMA platforms.

In this context, some important questions arise: how do different affinity strategies impact scientific parallel applications? Are CPU-bound applications affected by different CPU affinity strategies? How can memory affinity strategies impact the performance of a memory-bound application?

In this work we focus on evaluate the impact of CPU and memory affinity on UMA and NUMA multi-core platforms. Our methodology is based on the following major aspects:

Choice and control of the used multi-core hardware architectures. We experiment with two different platforms, respectively one UMA and one NUMA platform. We know all the parameters characterizing their hardware architecture, namely the type and the number of the cores, their interconnection and their cache and memory organization. Moreover, we control all possible operating system parameters.

Use of representative parallel applications. We are interested in evaluating numerical scientific multithreaded benchmarks, which exhibit significant memory and processing power usage, data-sharing and different memory access patterns. This is the reason why we have chosen to work with the NAS Parallel Benchmarks (NPB) [Haoqiang, 1999]. NPB is a benchmark derived from computational fluid dynamics (CFD) codes. It is composed of a set of applications and kernels that are examples of memory-bound and CPU-bound programs.

Rigorous performance evaluation and analysis. All experiments have been executed multiple times and the average values have been used in order to minimize the measurement error. During experiments, the platforms have been reserved and dedicated to the running application i.e. there has been no outside interference. In order to evaluate the impact of affinity strategies on NPB, we have studied performance in a top-down manner. We start with the standard performance metrics for parallel applications i.e. speedup and scalability. In order to understand furthermore the obtained results, we have observed thread migrations, as well as hardware performance counters. For the latter, we have typically observed the cache accesses and cache misses.

Experimentation with different affinity strategies. We have experimented with different CPU and memory affinity strategies. For CPU affinity, we have tried two strategies in which threads are forced to run on a specified set of CPU/cores. The first strategy maps threads to a single node of cores thus allowing cache sharing. The second one prevents cache sharing by distributing the threads on different nodes. As for memory affinity, we replaced the Linux default data placement strategy with two alternative ones. The idea is to allocate data on specified banks of memory aiming at optimizing the latency or the bandwidth. Our experiments show that affinity strategies have a significant impact on applications' performance and that their adapted usage can lead up to 70% performance improvement.

This paper is an extension from previous research work [Ribeiro, 2011]. We added new results and findings related to the impact of CPU and memory affinity on parallel application on the newest version of the Linux kernel. We highlight that, although the Linux affinity management is more aware of multi-core machines, there is still need to carefully place processes, threads and memory data to achieve better performance.

This paper is organized as follows. In Section 2, we present the platforms and benchmarks used in this work. We report the overall performance of NPBs in Section 3. The investigation of the CPU and memory affinity is discussed in Section 4. Section 5 presents the performance of NPBs when affinity strategies are applied. In Section 6, we present the same evaluation of affinity strategies on a newer version of the Linux kernel. Related works are discussed in Section 7. Finally, concluding remarks and future works are pointed out in Section 7.

2. EXPERIMENTAL ENVIRONMENT

In order to conduct our experiments, we have selected two representative multi-core platforms. **Intel UMA**: a multiprocessor based on four six-core Intel Xeon X7460. Each group of two cores shares a L2 cache (3MB) and each group of six cores shares a L3 cache (16MB). **Intel NUMA**: a Dell PowerEdge R910 equipped with 4 eight-core Intel Xeon X7560 processors. Each core has a private L1 (32KB) and L2 (256KB) caches and all cores on the same socket share a L3 cache (24MB).

Table 1 summarizes the hardware characteristics of these machines. Memory bandwidth (obtained from Stream - Triad operation [McCalpin, 1995]) and NUMA factor (obtained from BenchIT [Molka, 2009]) are also reported in this table. The NUMA factor is obtained through the division of remote read latency by local read latency. NUMA factors are shown in intervals, meaning the minimum and maximum penalties to access a remote DRAM in comparison to a local DRAM. Both machines run Linux operating system 2.6.32 and 3.2.0 with GCC (GNU C Compiler) 4.4.4. In order to control CPU and memory affinity, we have used the *numactl tool* version 2.0.4 [Kleen, 2005].

Table 1. Overview of the multi-core platforms.

Platform	#cores	#sockets	#nodes	Clock (GHz)	LLC (MB)	DRAM (GB)	Bandwidth (GB/s)	NUMA factor
UMA Intel	24	4	-	2.66	16	64	6.39	-
NUMA Intel	32	4	4	2.27	24	64	35.54	[1.36-3.6]

We have selected the NAS Parallel Benchmarks (NPB) [Haoqiang, 1999] to evaluate the performance impact of affinity on multi-core machines. NPB is a benchmark derived from

computational fluid dynamics (CFD) codes. It is composed of a set of applications and kernels [Haoqiang, 1999] that are examples of memory-bound and CPU-bound programs. In this work we use the OMNI compiler group implementation of NPB version 2.3. We consider the following benchmarks: FFT, MG, LU, CG, BT, EP and SP. For all selected applications, we use three classes, which define the size of the problem: A (small), B (medium) and C (large). The IS and UA applications, which are also part of the NPB benchmark, were not used in this paper. UA was not implemented in this version whereas IS presented some problems during the execution.

3. OVERALL PERFORMANCE OF NAS PARALLEL BENCHMARKS

In this section, we present the performance of the original version of NPB on each machine presented in Section 2. We use the speedup as the performance metric to compare all results. We vary the number of threads considering the available cores in each platform, using one thread per core. All applications were executed using the problem classes A (small), B (medium) and C (large). We executed each experiment several times, obtaining a maximum standard deviation of 5%.

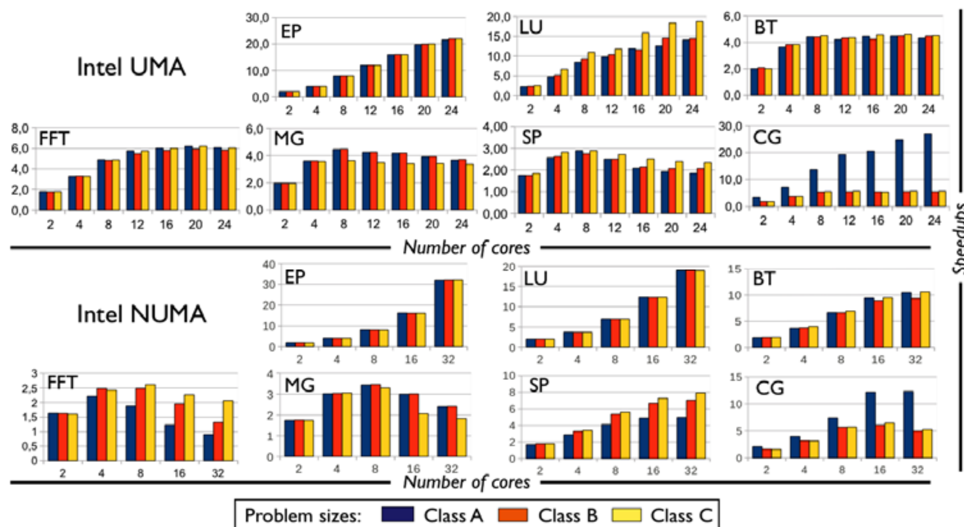

Figure 1. NAS parallel benchmarks: scalability results on Linux 2.6.32.

Figure 1 shows speedups obtained for the NAS Parallel Benchmarks on the selected machines. EP and LU have presented great scalability due to the high ratio between computation and communication. Furthermore, on LU benchmark, data allocation has been optimized for NUMA machines. A parallel initialization of all data was performed by the team of threads in order to ensure data locality and to avoid remote accesses. On the contrary, BT, CG, FFT, SP and MG have presented poor scalability. Indeed, on the Intel NUMA the result is

mainly related to the machine characteristics (NUMA effects) and to the way the benchmarks have been developed, with a NUMA-unaware data allocation and initialization.

We have also observed that the memory access patterns are irregular on some of these benchmarks (e.g., CG and MG). Indeed, some of them use sparse matrices and indirect matrices access, which means that a number of threads probably compete for the same memory bank. In such case, memory contention is generated because the interconnection network becomes overloaded, reducing the overall performance. On the Intel UMA, the poor scalability is mainly related to its hierarchical cache memory subsystem, in which some data accesses can generate expensive inter processor communications such as the long distant communication of FFT and CG, and the non-continuous communication of SP and LU. We can see that the speedups of most applications are significantly distant from the ideal ones. We have found that the main scalability issues were due to the way threads access data, cache sharing issues and to the way of data are placed over the NUMA nodes. In order to have a better understanding of the archived performances, we have made a more detailed analysis of two benchmarks, EP and MG with very distinct characteristics. The former is a CPU-bound application whereas the latter is a memory-bound application.

4. IMPACT OF AFFINITY: MG AND EP CASE STUDIES

In this section, we evaluate the impact of affinity strategies on the performance of NPB applications on the selected machines with Linux 2.6.32. Specifically, we have used MG and EP as our case studies to perform such analysis. We have considered the following metrics: execution time, speedup and hardware counters.

4.1 CPU Affinity

There are two ways of dealing with CPU affinity in most of operating systems. The first one, named soft affinity, relies on the traditional OS scheduler in which processes/threads remain on the same CPU as long as possible. However, in some situations the OS scheduler may migrate processes/threads to another processor, even when it is not necessary, impacting the overall performance of the system. The second way of dealing with CPU affinity is called hard affinity, which delegates the processes/threads locality to the user. In this case, the user may define on which processor/core each process/thread must run. In this paper, we mean by CPU affinity the latter definition (hard affinity).

In order to observe how the Linux scheduler behaves, we have performed some experiments with the two selected benchmarks (EP and MG) on both Intel machines. On those experiments, we traced the locality of all threads at the beginning of every iteration. After analyzing the traces obtained from EP, we concluded that the Linux scheduler assigned each thread to a core and did not migrate them. This occurs because EP is CPU-bound, so all cores are always performing computations and little time is spent accessing the memory.

We have conducted the same experiments with MG but we notice a difference in the behavior: the locality of threads constantly changed during the execution on both machines. To demonstrate such behavior, we have picked the trace information obtained from a single thread during the execution with 24 threads on the Intel UMA (Figure 2a) and Intel NUMA (Figure 2b) platforms. As it can be seen, the Linux scheduler has considerably varied the

locality of the thread. On the Intel NUMA machine, we observe thread migrations between different NUMA nodes whereas on the Intel UMA, there were many migrations between processors and sockets. All other threads have also presented analogous behaviors.

We now analyze the impact of applying different CPU affinity strategies in comparison to the OS scheduler (soft affinity). To study the influence of cache sharing on the Intel UMA, we have compared two CPU affinity strategies: **intra-socket** (threads are bound to sibling cores, sharing cache) and **inter-socket** (threads are bound to cores on different sockets and do not share cache). We have used four threads in these experiments, since with more than 4 threads there would be no interesting non-sharing case to compare. In order to implement the CPU affinity strategies, we have used the numactl tool considering the machine topology with the following parameters: (i) *--physcpubind=0,12,4,16* and *--physcpubind=0,4,8,12*, for the intra-node strategy on the Intel UMA and NUMA respectively and (ii) *--physcpubind=0,1,2,3*, for the inter-node strategy on both machines.

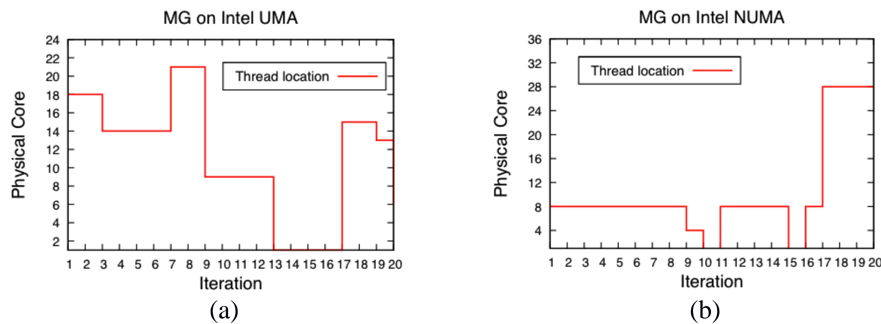

Figure 2. Thread scheduling of Linux 2.6.32 on MG.

Figure 3 presents some event counters obtained during the execution of both benchmarks on Intel UMA. The value shown by a bar corresponds with the result of the normalization between an event counter of a CPU affinity strategy (hard affinity) and an event counter of the OS scheduler (soft affinity). Thus, bars higher than 1 mean that the CPU affinity has increased the number of events whereas bars lower than 1 indicate the contrary.

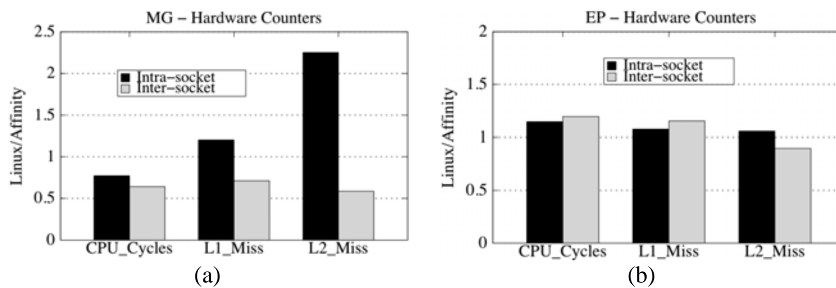

Figure 3. Intra-socket vs. Inter-socket thread placements on Intel UMA (Linux 2.6.32).

Figure 3a shows the results obtained from MG. We can notice that both strategies have reduced considerably the number of CPU cycles, since the OS scheduler does not spend too much cycles to deal with thread migrations. However, by applying the intra-socket strategy,

we can observe a considerable growth on the number of cache misses. This is due to the fact that the total memory allocated by MG does not fit on a single L3 cache, which is shared by the four threads. Thus, the amount of data moving between the main memory and the L3 cache is increased. A different behavior is observed with EP in Figure 3b. Since it is CPU-bound application, there is no important impact on the cache hierarchy for the analyzed event counters. Thus, EP performances obtained with intra- and inter-node strategies are very similar with those obtained with Linux.

An analogous analysis of some performance event counters on the Intel NUMA is presented in Figure 4. Since it is a NUMA machine, we are interested in the following event counters: last level cache misses (L3_Miss), number of accesses on local and remote DRAM (L_DRAM and R_DRAM), number of accesses on local cache and remote cache (L_cache and R_cache). The intra-node strategy assures that all four threads are placed on the same node whereas the inter-node strategy places one thread per node.

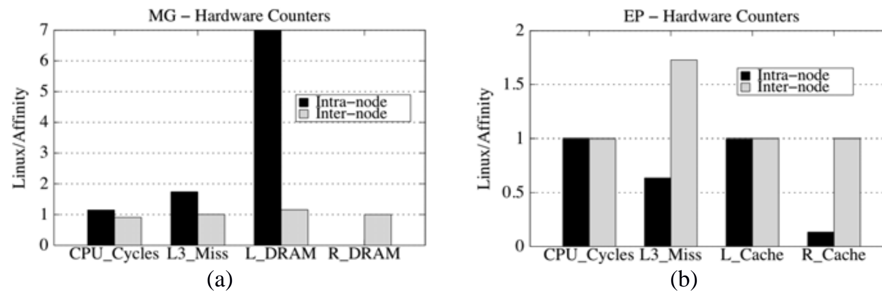

Figure 4. Intra-node vs. Inter-node thread placements on Intel NUMA (Linux 2.6.32).

Figure 4a presents the event counters obtained from MG. The intra-node strategy does not generate any access to the remote DRAM, all memory access are done on the local DRAM. Consequently, the memory contention is increased due to the fact that all data allocated by MG is placed on the same node. On the contrary, the inter-node strategy has presented very similar results to the Linux scheduler, reducing the memory contention and improving cache L3 usage by distributing the data among the nodes.

Event counters obtained from EP are shown in Figure 4b. Both strategies have reported similar number of CPU cycles when compared to Linux scheduler. Since EP is not memory-bound, placing threads on different nodes does not impact on memory sub-system performance. However, we can observe that cache L3 misses and remote cache responses have been smaller in intra-node strategy. By placing threads in a single node, we then avoid data distribution in several nodes. Additionally, the fact of placing all threads in the same node allows data to be prefetched into cache memories.

4.2 Memory Affinity

Operating systems must ensure memory affinity on applications, in order to optimize memory allocation and placement on NUMA multi-core machines. Memory affinity is guaranteed when a compromise between threads and data is achieved reducing latency costs or increasing bandwidth for memory accesses [Ribeiro, 2009].

In the case of Linux, *first-touch* is the strategy used to guarantee memory affinity. This policy places data on the node that first accesses it [Joseph, 2006]. Therefore, applications

with a regular memory access patterns can benefit from this strategy. However, this strategy will only present performance gains if applied on applications with a regular data access pattern. In case of irregular applications, first-touch will probably result in a high number of remote accesses and memory contention.

On Linux, it is possible to change this default memory affinity strategy by using the NUMA API [Kleen, 2005]. In this work, we used *numactl* tool to modify data placement for both EP and MG benchmarks on the Intel NUMA machine. We have studied two strategies that we named *bind* and *interleave*. The first strategy places memory of an application on a restricted set of memory banks. Interleave strategy spreads data over memory banks of a NUMA machine. We used the following parameters on numactl: (i) *--membind=0,1,2,3*, for the bind strategy and (ii) *--interleave=all*, for the interleave strategy. The difference between these strategies is that the former optimizes latency while the latter optimizes bandwidth. On both strategies, we have used all cores of the machine and we have pinned each thread to a core (inter-node CPU affinity) to avoid any impact of thread scheduling.

We measured some hardware counters on Intel NUMA machine, to study the impact of the different memory affinity strategies on EP and MG benchmarks. The selected counters are total CPU cycles, cache L3 miss (L3_misses), number of accesses to local and remote caches (L_cache and R_cache) and, number of accesses to local and remote DRAM access (L_DRAM and R_DRAM). The value shown by a bar corresponds with the result of the normalization between an event counter of a memory affinity strategy and an event counter of the OS strategy. Thus, bars higher than 1 mean that the strategy has increased the number of events whereas bars lower than 1 indicate the contrary.

Figure 5 shows the performance events counters for MG and EP benchmarks on Intel NUMA, compared to the Linux execution. We can observe that bind and interleave have presented similar number of cache L3 misses and accesses to DRAM. However, it can be noticed that interleave strategy has expressively reduced the total number of CPU cycles on MG. On the other hand, it has increased the number of cache L3 misses on EP.

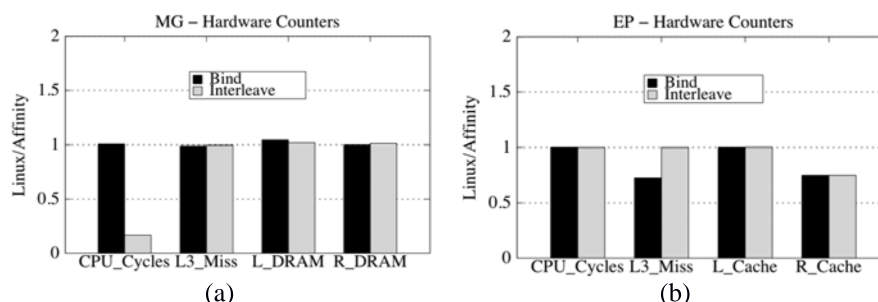

Figure 5. Bind vs. Interleave data placements on Intel NUMA (Linux 2.6.32).

In MG benchmark, computation is performed by memory zones with irregular accesses. Therefore, spreading data over all available DRAMs allows much more memory pages of a zone to be accessed by threads in the same interval time. Due to this, QPI interconnection link is more used to access data over different steps of MG benchmark. In the bind strategy, the application data is split in continuous memory blocks, forcing threads to access the same memory bank (DRAM) on MG steps. In the case of EP, we can noticed that bind and interleave strategies do not impact its overall performance. Considering the cache L3, the

interleave strategy have generated more misses because memory pages of the application are spread over all memory banks. Consequently, the first accesses generate some cache misses to get data for cores.

5. AFFINITY STRATEGIES ON NAS PARALLEL BENCHMARKS

In this section, we present the impact of the CPU and Memory affinity strategies on NPB. Figure 6 reports the speedup comparison between the best affinity strategy and the default Linux affinity management. The best affinity was chosen considering the application and platform characteristics.

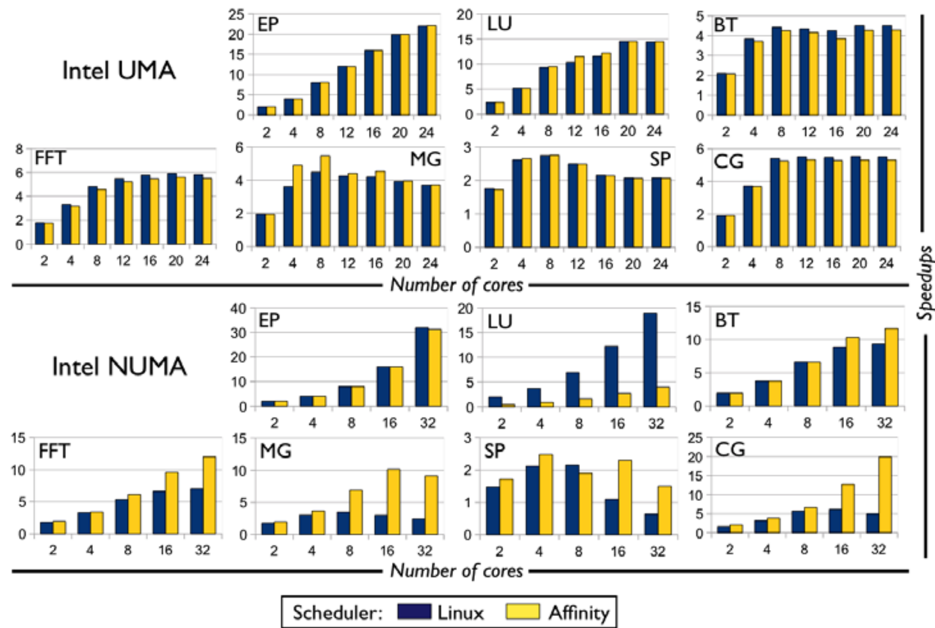

Figure 6. NAS Parallel Benchmarks affinity results (Linux 2.6.32).

We can observe that controlling the affinity has generally reported better results than Linux. One can also notice that the benefits of managing affinity are much lower on the Intel UMA than in the Intel NUMA. Indeed, on the Intel NUMA, the affinity strategy has an impact on both cache and on the main memory performances. Therefore, greater performance gains are expected on the NUMA machine.

On the Intel UMA, the best strategy was the inter-socket CPU affinity for all selected benchmarks. However, such strategy has not presented any improvement gains on FFT, CG, SP and BT. For these benchmarks, Linux has obtained better speedups since it migrates threads to reduce long distance and non-continuous accesses on memory. Considering MG and LU benchmarks, the inter-socket strategy has resulted in better cache sharing, improving their performances. On EP, we have observed that different CPU affinity strategies generate similar

performances. Since this benchmark perform independent work, both CPU and memory affinities do not have an impact on the performance.

On the contrary, we have found that there was a relation between the number of threads and the best affinity strategy on the Intel NUMA. For a small number of threads (up to 8), the best strategy was the intra-node (CPU affinity) combined with bind memory policy (memory affinity). With more than 8 threads, the best strategy was the inter-node (CPU affinity) combined with interleave memory policy (memory affinity).All affinity strategies have decreased the performance of LU on the Intel NUMA. Since this benchmark has a heterogeneous communication pattern, therefore threads perform many distant accesses during the whole execution. Consequently, when a CPU affinity strategy is applied, threads do not migrate and the number of distant accesses is increased. The best performance gains were obtained with FFT, CG, SP and MG due to their characteristics: indirect access, sparse matrix and centralized data initialization. On those benchmarks, affinity allowed better data distribution among the machine nodes and better cache sharing. Considering BT, we have observed that some threads do not share data, because BT does not scale with high number of threads. In that case, small performance gains are expected for this benchmark when affinity is applied.

6. OS EVOLUTION AND VALIDITY OF THE AFFINITY APPROACH

In the previous sections, all experiments were performed with Linux kernel version 2.6.32. This section presents results with Linux kernel version 3.2.0. This version has better support for multi-core machines with UMA and NUMA design has been included in the kernel. For instance, better scheduling and memory allocation for drivers [Linux kernel, 2012]. In order to validate our approach, we re-ran all the previously discussed experiments on the same machines using the kernel version 3.2.0.

6.1 CPU Affinity

Concerning CPU affinity, the first observation that we can make is that there is fewer or none thread migration on Linux 3.2.0 for both EP and MG benchmarks when executed on the selected machines. This is due to the changes on the scheduler of the operating system, which now keeps threads as long as possible in the same cores.

Figure 7 presents the hardware counters for the UMA platform with the Linux 3.2.0 when CPU affinity is applied on both EP and MG benchmark. On general, the intra-socket strategy has reduced the performance compared to the Linux default behavior. This has been already observed in the version 2.6.32 of the operating system. However, in this new version the impact of this strategy is much more important and performance is reduced for both benchmarks. Concerning the inter-socket strategy, it has presented similar results compared to Linux. We concluded that the Linux behavior for these benchmarks is similar to inter-socket strategy (CPU affinity).

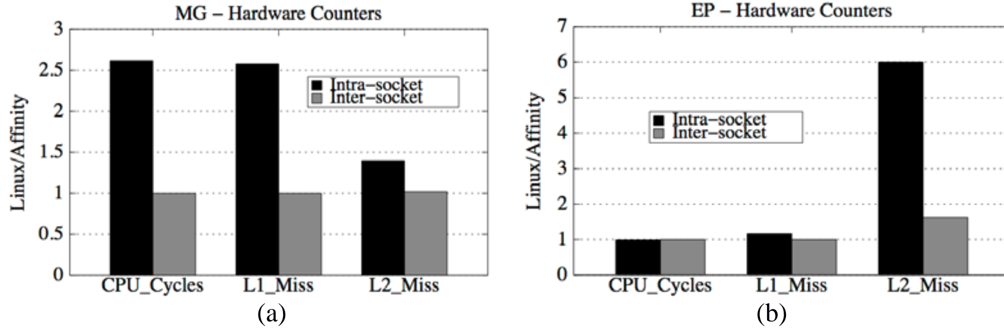

Figure 7. Intra-socket vs. Inter-socket thread placements on Intel UMA (Linux 3.2.0).

Figure 8 depicts the hardware counters for the NUMA platform with the Linux 3.2.0 when CPU affinity is applied. Although the cache misses are reduced for intra-node strategies on the newer version of the kernel, the CPU cycles are similar to the ones obtained with the version 2.6.32. Therefore, similar performances are achieved.

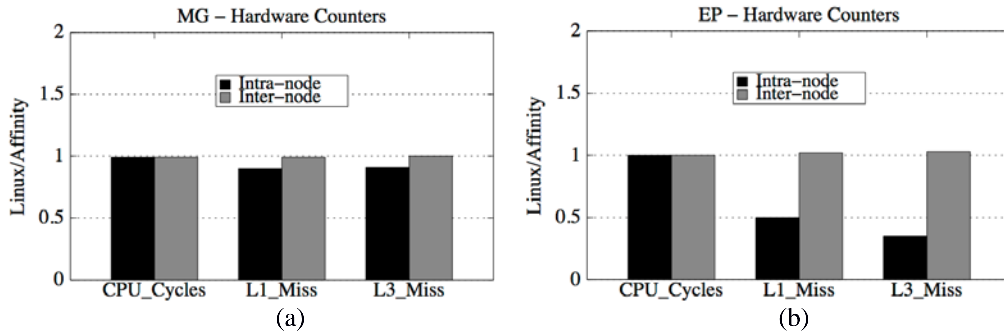

Figure 8. Intra-node vs. Inter-node thread placements on Intel NUMA (Linux 3.2.0).

6.2 Memory Affinity

In Figure 9, we can observe that the impact of using memory affinity on the benchmarks is still important for performance. Concerning MG benchmark, the performance is improved when interleave memory affinity is applied. However, this improvement is smaller than the one observed in the previous version of the kernel. In the case of EP, the results are similar to the ones obtained with the version 2.6.32 of the operating system, since this benchmark is not memory-bound.

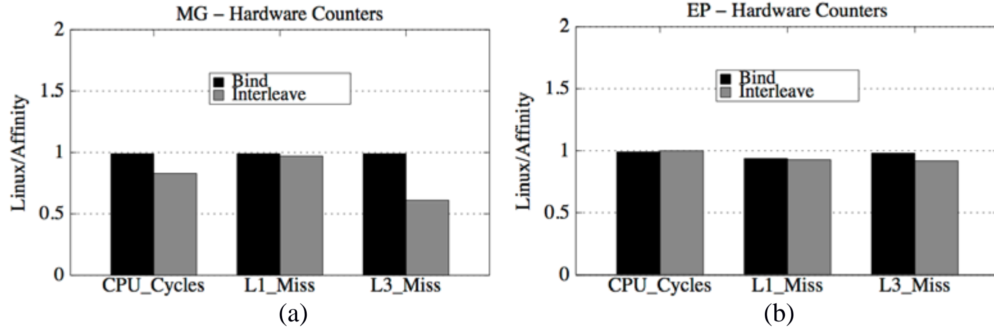

Figure 9. Bind vs. Interleave data placements on Intel NUMA (Linux 3.2.0).

6.3 Overall Performance with Thread and Memory Affinities

We present in this section the impact of the CPU and Memory affinity strategies on NPB on the current version of Linux operating system. Figure 10 reports the speedup comparison between the best affinity strategy and the default Linux affinity management. As for the version 2.6.32, the best affinity was chosen considering the application and platform characteristics.

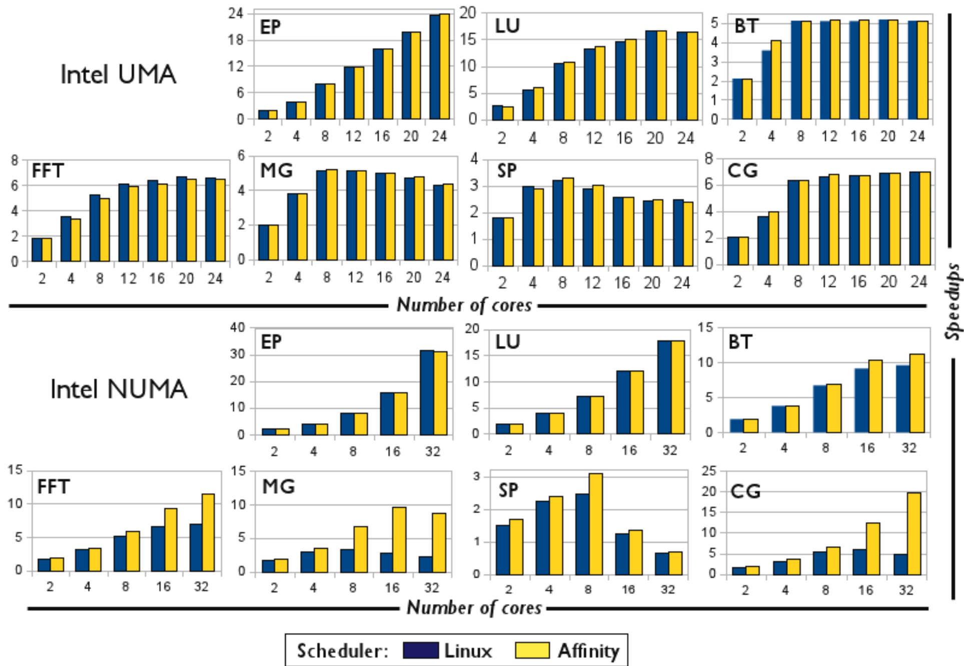

Figure 10. NAS Parallel Benchmarks affinity results (Linux 3.2.0).

Overall, we can observe that applying CPU and memory affinity has improved the performance of the selected benchmarks. However, the benefits of managing affinity are much

lower on the Intel UMA than in the Intel NUMA. This is due to the asymmetric latencies that NUMA machines has. As presented in Table 1, such asymmetry can generate latencies up to 3.6 times higher for some memory accesses. Therefore, managing affinities for NUMA machines can provide better performance by up to 70% for CG and MG benchmarks.

Similarly to the results presented with Linux 2.6.32, on the Intel UMA, the best strategy on average was the inter-socket CPU affinity for all selected benchmarks. On the Intel NUMA machine, the best strategy depends on the number of threads used to run the benchmark as the results presented for Linux 2.6.32.

Another interesting result is the differences on performance improvements observed with Linux 3.2.0. We can observe that on both machines, speedups for the benchmarks are better now. Considering LU benchmark, applying affinity to it on this operating system version does not slowdown the performance, but keeps it close to the ones obtained with Linux. Furthermore, for BT and SP benchmarks, speedups are slightly better with both Linux and affinities.

These results allows us to conclude that even though Linux operating system has being changed to better adapt to the current multi-core machines architecture, there is still place for some performance improvements related to CPU and memory affinity. Therefore, specialized strategies that can reflect the communication pattern of the parallel applications can provide better performances for applications, exploiting the hardware power of multi-core machines.

7. RELATED WORK

The complexity of multi-core multiprocessors systems has demanded better understanding on memory accesses, influences of data and thread placement over the machine and the impact of hierarchical topologies. Due to this, research groups have studied the performance and the behavior of several workloads over multi-core machines. In [DeFlumere, 2009], the authors have evaluated the performance of two multi-core platforms with scientific applications, focusing their analysis on the performance of memory and communication sub-systems. In [Zhang, 2010], the authors have studied the influence of cache sharing on different applications of the PARSEC benchmark suite. However, these works have only considered UMA multi-core platforms and they have focused only on CPU affinity. In [Alam, 2006], the authors have investigated the impact of multi-cores and processor affinity on hybrid scientific workloads based on Message Passing Interface (MPI). However, we use OpenMP, focusing on cache sharing problem, while they focus on the interconnection network issues present in MPI systems.. Secondly, we study the impact on different multi-core platforms (UMA and NUMA) with a significant number of cores per socket, while, they have used three platforms based on the same AMD Opteron processor with two cores per socket. Finally, we do a deeper study of the selected benchmarks through the analysis of performance event counters.

The work of [Tam, 2007] uses hardware counters in order to dynamically map the threads of parallel applications. The scheduling is done by estimating the hardware counters of the Power5 processor (stall cycles, cache misses, etc.) in order to deduct the sharing pattern. Even if performance has been increased bu up to 7%, this approach provides less reliable information for mapping the threads and data.

A work close to ours is the one of [Broquedis, 2009] in which the authors consider dynamic task and data placement for OpenMP applications. A NUMA-aware runtime for

OpenMP, named ForestGOMP, is developed as an extension of the GNU OpenMP library. It relies on the hwloc framework [Broquedis, 2010] and on the Marcel threading library [Danjean, 2003. This runtime uses hwloc to extract the target machine topology and then pin kernel threads on the machine cores. In order to provide more performance for OpenMP applications, the Marcel library is used to create user level threads within parallel sections and associates them to the kernel threads. The proposal does not require profiling steps. However, contrary to our mechanism, ForestGOMP requires some modifications to the source code to provide information about the program behavior, such as how the data is distributed, which variables to consider, among others.

8. CONCLUSION

Throughout this paper, we showed that multi-core machines with UMA and NUMA characteristics can present problems related to bandwidth, latency and performance scalability. In order to comprehend the impact of these problems on multi-core machines, we have selected Numerical Scientific multithreaded workloads as our case study. These workloads exhibit significant memory and processing power usage, data-sharing between threads and different memory access patterns.

We have conducted a series of experiments with different CPU and memory affinity strategies in order to observe how the performance behaves according to strategy used on such machines. The experimental results of this paper highlight the importance of the memory sub-system on the performance of such workloads on machines with a large number of cores. On different multi-core architectures (UMA vs. NUMA), the affinity has presented a significant influence on the performance of benchmarks that are memory bounded. The performance improvements are also present on newer version of Linux operating system, which has some modifications to support multi-core machines. Our future work will be related to an investigation of the same affinity on workloads developed with hybrid programming models (OpenMP/MPI) while considering the process mapping and the influence of memory and interconnection.

ACKNOWLEDGEMENT

This paper was supported by FAPEMIG, INRIA and CAPES (grant number 4874-06-4).

REFERENCES

Alam, S. R. et al, 2006. Characterization of Scientific Workloads on Systems with Multi-core Processors. *Proceedings of the IEEE International Symposium on Workload Characterization*. San Jose, USA, pp. 225-236.

Asanovic, K. et al, 2006. The Landscape of Parallel Computing Research: A view from Berkeley. *Technical report.* University of California, Berkeley.

Broquedis F. et al, 2009. Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective. *Proceeding of the 5th International Workshop on OpenMP*. Dresden, Germany, pp. 79–92.

Broquedis F. et al, 2010. libtopology: A Generic Framework for Managing Hardware Affinities in HPC Applications. *Proceedings of the Euromicro Conference on Parallel, Distributed, and Network-Based Processing.* Pisa, Italy, pp. 180–186.

Castro, M. et al 2012. Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications. *Proceedings of the International European Conference on Parallel and Distributed Computing*. Rhodes Island, Greece, pp. 465-476.

Danjean V. and Namyst R. 2003. Controlling Kernel Scheduling from User Space: An Approach to Enhancing Applications Reactivity to I/O Events. *Proceedings of the High Performance Computing*. Hyderabad, India, pp. 490–499.

DeFlumere, A. M. and Alam, S. R., 2009. Exploring Multi-core Limitations Through Comparison of Contemporary Systems. *Proceedings of the Richard Tapia Celebration of Diversity in Computing Conference*. Portland, Oregon, pp. 75-80.

Foster I., 1995. *Designing and Building Parallel Programs*. Addison Wesley, United Kingdom.

Haoqiang, J. and Michael Frumkin, J.Y, 1999. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. *Technical Report 99-011/1999*. NASA Ames Research Center.

Joseph, A. et al, 2006. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. *Proceedings of the High Performance Computing*. Bangalore, India, pp. 338-352.

Kleen, A, 2005. A NUMA API for Linux. *Technical Report Novell-4621437.*

Linux kernel, 2012. Kernel Coverage. *URL: http://lwn.net/Kernel/*

McCalpin, John D., 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA)*, No. 12, pp. 19-25**.**

Mei, C. et al, 2010. Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study. *Proceedings of the TeraGrid Conference*. New York, USA, pp. 1-8.

Molka, D. et al, 2009. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. Washington, USA, pp. 261-270.

Ribeiro, C. P. et al, 2011. Investigating the Impact of CPU and Memory Affinity on Multi-core Platforms: A Case Study of Numerical Scientific Multithreaded Applications. *Proceedings of the IADIS International Conference on Applied Computing.* Rio de Janeiro, Brazil, pp. 299-306.

Ribeiro, C. P. et al, 2009. Memory Affinity for Hierarchical Shared Memory Multiprocessors. *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*. São Paulo, Brazil, pp.59-66.

Tam D. et al, 2007. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. *Proceedings of the European Conference on Computer Systems*. Lisbon, Portugal, pp. 47-58.

Terboven, C. et al. 2008, Data and Thread Affinity in OpenMP Programs. *Proceedings of the Workshop on Memory Access on Future Processors*. New York, USA, pp. 377–384.

Zhang, E. Z. et al, 2010. Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs? *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, USA, pp. 203-212.

Zheng W. et al, 2009. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, USA, pp. 75–84.