Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms

Eduardo Henrique Molina da Cruz,
Marco Antonio Zanata Alves,
Alexandre Carissimi,
Philippe Olivier Alexandre Navaux
PPGC Graduate Program in Computer Science
Institute of Informatics
UFRGS Federal University of Rio Grande do Sul
Porto Alegre, RS, Brazil
Email: {ehmcruz, mazalves, asc, navaux}@inf.ufrgs.br

Christiane Pousa Ribeiro,
Jean-François Méhaut
INRIA Mescal Research Team
LIG Laboratory
University of Grenoble, France
Email: {Christiane.Pousa, Jean-Francois.Mehaut}@imag.fr

***Abstract*—In parallel programs, the tasks of a given application must cooperate in order to accomplish the required computation. However, the communication time between the tasks may be different depending on which core they are executing and how the memory hierarchy and interconnection are used. The problem is even more important in multi-core machines with NUMA characteristics, since the remote access imposes high overhead, making them more sensitive to thread and data mapping. In this context, process mapping is a technique that provides performance gains by improving the use of resources such as interconnections, main memory and cache memory. The problem of detecting the best mapping is considered NP-Hard. Furthermore, in shared memory environments, there is an additional difficulty of finding the communication pattern, which is implicit and occurs through memory accesses. This work aims to provide a method for static mapping for NUMA architectures which does not require any prior knowledge of the application. Different metrics were adopted and an heuristic method based on the Edmonds matching algorithm was used to obtain the mapping. In order to evaluate our proposal, we use the NAS Parallel Benchmarks (NPB) and two modern multi-core NUMA machines. Results show performance gains of up to 75% compared to the native scheduler and memory allocator of the operating system.**

***Keywords*-Process Map; Memory Affinity; Parallel Architectures; High Performance Computing.**

I. Introduction

On shared memory parallel platforms with a hierarchical memory sub-system, the communication (data sharing) time between threads of a parallel program may be different, depending on how the processors or cores are interconnected through the memory hierarchy, the interconnections used and the memory coherence protocol [1]. This difference is even more relevant on parallel machines with non-uniform memory access characteristics (NUMA), since the latency of remote access are high.

In this context, process mapping helps to improve system performance by mapping groups of threads which shares high amounts of data to cores or processors that shares some level of cache or DRAM memory. Mapping parallel programs on NUMA becomes harder [2] and more expensive, as usually there are more levels on the memory hierarchy to be explored. Furthermore, the problem to find the best mapping is considered NP-Hard [3] and, in shared memory environments, there is the additional difficulty to find the communication pattern, which is implicit and occurs through memory accesses.

In this paper, we propose and evaluate a technique of process mapping to bind threads of a given application on cores and allocate their data on DRAM memories, reducing the overhead of communication among the threads which shares data. To perform the static mapping, the proposed model is based on the analysis of memory traces and does not require any prior knowledge of the application. We use two metrics (amount of shared memory and access performed in the shared memory) to identify the data sharing pattern between threads and an heuristic algorithm to map threads and data on the machine. This heuristic is based on the Edmonds matching algorithm, which provides a well suited thread and memory affinity for a selected application. Thus, the main objective is to reduce the data sharing time on multi-core machines with NUMA characteristics, improving the overall system performance.

In comparison to related works, our approach differs in at least two aspects. First of all, we provide an heuristic that can be adopted in different shared memory architectures. Secondly, we consider application and hardware characteristics in order to map threads and data with the most suited memory policy. We have evaluated our proposal by performing experiments with HPC benchmarks on two multi-core NUMA platforms. The results have been compared to the Linux standard thread and data mapping and to the worst affinity for an application.

This paper is organized as follows. Section II presents

the general description of the applications used to evaluate the proposed technique. Section III introduces the method proposed for process mapping, it also presents some analysis of the workload used. The platforms used to validate the mapping technique is presented on Section IV. The experimental results are presented on Section V. Some related works are presented and compared on Section VI. On Section VII, the conclusions and future works are described.

II. Selected Benchmarks

In this section, we present the **NAS Parallel Benchmark (NPB)** workload used to evaluate the performance of our thread and data mapping method.

The NPB has its applications derived from computational fluid dynamics (CFD) codes and it is composed by applications and kernels [4]. NPB applications and kernels perform representative computation and data communication of CFD codes. These characteristics allow us to better evaluate the impact of both threads and data mapping on multi-threaded programs over multi-core machines. NPB has been implemented on different languages, using different strategies for code parallelization. In this work, we have used the OMNI compiler group implementation of NPB version 2.3, where all the applications are written in C and have been parallelized with OpenMP (based on the standard Pthreads interface). This version of NPB was used because Minas requires that the applications are written in the C language.

Considering the OMNI implementation details, all the benchmarks have a parallel initialization of data in order to make sure all data is touched and to reduce variable startup costs. Table I provides a description of the selected set of NPB. Most of them are memory bound applications, except EP, which is CPU bound. NPB has several standard inputs(from smallest to greatest) S, W, A, B, C, and the ones used for this paper were W, A and B. The W was used to discover the sharing pattern and A and B to perform the experiments.

The BT, IS, SP and UA applications, which are also part of the NPB benchmark, were not used in this paper. Regarding BT, we have observed it does not scale for more than 12 threads with the W input, therefore, the mapping obtained is not suitable for the A and B inputs and, consequently, no performance gains are expected for this benchmark in our experiments. UA is not implemented in the NAS version used, and IS and SP present implementation problems.

III. Mapping Threads and Data

The static mapping proposed in this paper is performed in five steps (Figure 1): monitor the memory access to generate the traces, analysis of the trace to generate the sharing matrix, calculate the thread affinity with the memory hierarchy, map threads and data with the Minas framework and, finally, execute the application with the best mapping found.

Table I
Selected Applications from NPB.

Name	Description
FT	Computational kernel of a 3D Fast Fourier Transform (FFT) method. FT performs three 1D FFT, one for each dimension.
MG	Multigrid V-cycle method used to solve the 3D scalar Poisson equation. The algorithm works between coarse and fine grids. It exercises both short and long distance data movement.
LU	Simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to the system resulting from finite-difference discretization of Navier-Stokes equations in 3D by splitting into block Lower and Upper triangular systems.
CG	Conjugate Gradient method used to compute the smallest eigenvalue of a large, sparse, unstructured matrix. Exercising unstructured grid computations and communications.
EP	Embarrassingly Parallel benchmark, which generates pairs of Gaussian random. Aiming to establish the reference point for peak performance of a given platform.

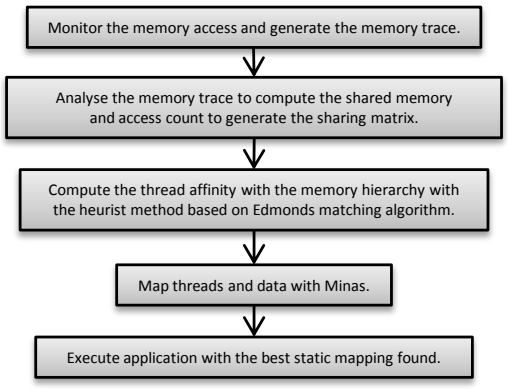


Figure 1. Methodology used to perform the static mapping.

A. Monitoring the Memory Access

To achieve the static mapping, a preliminary analysis of the application is required to obtain the information that is used to compose the sharing matrix. In shared memory environments, it is necessary to monitor all the memory access of the applications, which was accomplished by executing them inside the Simics [5] simulator, registering memory access information such as the moment when the access happened, the identifier of the thread that generated the access, the memory address, the operation type (read, write or instruction fetch) and its size.

To instrument Simics to register every memory access, the event *Core_Breakpoint_Memop* can be triggered. Nevertheless, as the triggers are implemented in the Python language, they have a high overhead, decreasing the simulation speed. Therefore, it was developed a module in the C language that is dynamically linked to Simics, and when it is plugged to a processor in the simulation environment, it monitors all the memory access performed by it.

However, it is necessary to filter the access to be registered, so that only the memory access performed by the evaluated application are stored in the trace file. Although Simics API implements tools to determine which task is

running in each processor, it is unstable when the number of processors simulated is high. To overcome this issue, the Linux kernel inside Simics was modified to warn the simulator about which task was being scheduled to run.

Another possibility to generate the memory traces is through dynamic binary instrumentation, using tools such as Pin [6] or Valgrind [7]. Although dynamic binary instrumentation is easier and faster than simulation, it alters the application behavior. For instance, the Valgrind tool serializes the threads in parallel applications, which may led to different communication patterns.

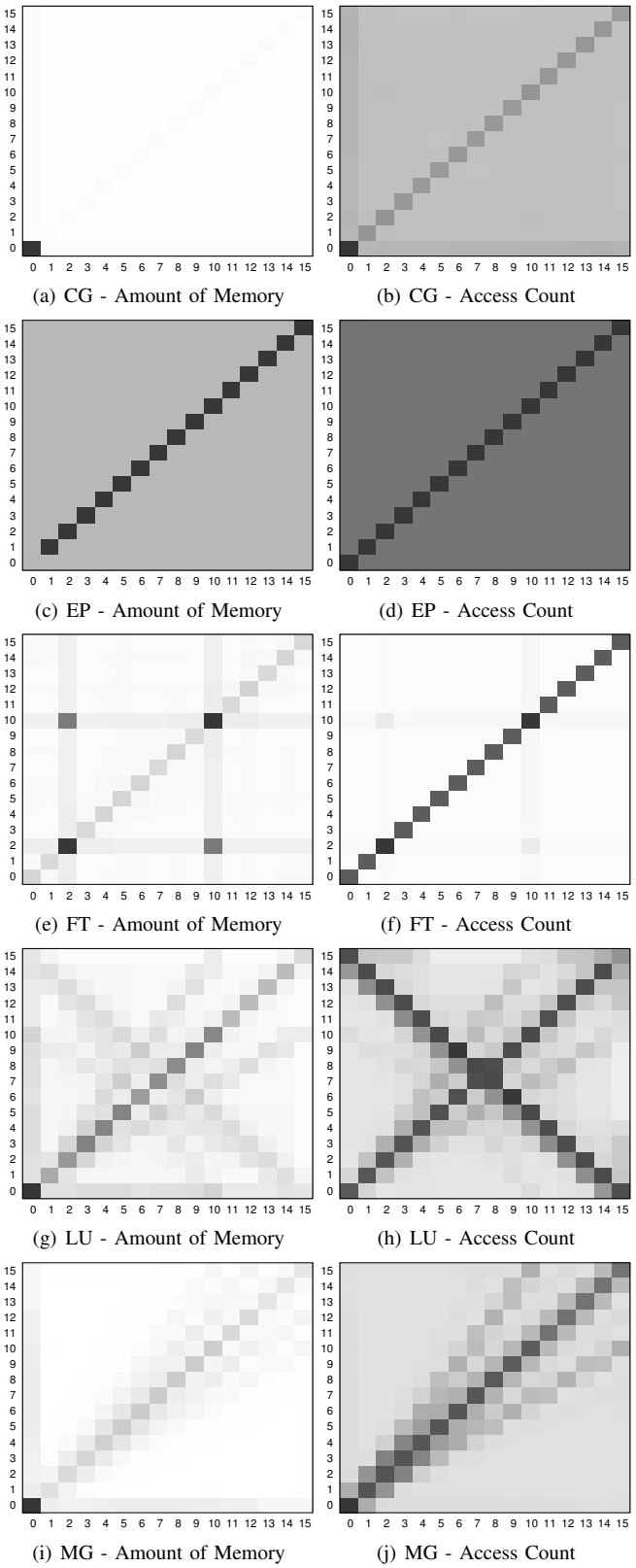
B. Generating the Sharing Matrix

The memory traces alone are not enough to guide the process mapping. The traces must be analyzed to discover the communication pattern, which depends on the metric adopted. For this work, two metrics were considered separated to evaluate the communication: the amount of memory shared by threads and the amount of access performed to a block of memory that is shared. Problems like false sharing were taken into account, as they are very common in multiprocessor architectures.

The shared memory can be analyzed in different groups of threads. To calculate how much memory is being shared between any group of threads, the amount of space necessary rises exponentially, discouraging the use of process mapping. Therefore, the shared memory was evaluated only between pairs of threads, generating a sharing matrix. Although this decreases the accuracy of the results, it reduces the space complexity to $\Theta(N^2)$, where N is the number of threads, and allows a faster processing of the information.

Figure 2 contains the communication pattern of the applications of NPB with the W input size for both metrics. Each cell *(i, j)* represents the communication between threads i and j. When i equals j, it represents the amount of access to the private area or its size. As darker the cell is, higher is the amount of communication. It is important to notice that CG, with the amount of memory metric, shows a pattern whereas there is almost no communication between threads and that thread 0 dominates and performs most of its communication with its private area. However, with the access count metric, although there is still a predominance of thread 0, the others present a bigger share of the total communication. LU and MG behave similar to CG. This indicates that, in these applications, thread 0 probably does some initialization or post-checking of the data.

Another analysis of the figure exhibits that MG has a communication pattern that the nearby threads communicate more among themselves. LU is the opposite, as the most distant threads communicate more. Both of them can be explored by mapping due to this heterogeneous pattern. On the other hand, EP is the one with the most homogeneous communication pattern, hence it may not benefit from process mapping like the other benchmarks.



(a) CG - Amount of Memory (b) CG - Access Count

(c) EP - Amount of Memory (d) EP - Access Count

(e) FT - Amount of Memory (f) FT - Access Count

(g) LU - Amount of Memory (h) LU - Access Count

(i) MG - Amount of Memory (j) MG - Access Count

Figure 2. Communication patterns of NPB applications.

C. Thread Affinity with the Memory Hierarchy

After the generation of the sharing matrix, it is necessary to map threads and its data. The mapping problem is considered NP-Hard, consequently, finding the optimal solution becomes infeasible when the number of tasks grows. Thus, heuristic algorithms must be employed to determine the mapping in reasonable time, with results as similar as possible to the perfect mapping. Methods like the Dual Recursive Bipartitioning produces good results, and are available on the software Scotch [8]. However, for this work, a different method was used to obtain the mapping, based on the maximum weight perfect matching problem for complete weighted graphs, as presented in [9].

This problem consists of, given a complete weighted graph $G = (V, E)$, it must be found a subset M of E in which every vertex of V is met by exactly one edge of M, and the sum of the weights of the edges of M is maximized. According to [10], this problem can be solved by the Edmonds matching algorithm in polynomial time, and a parallel algorithm can solve the problem with a time complexity of $O(\frac{N^3}{P} + N^2 \lg N)$, where N is the vertex number and P is the number of processors.

To model process mapping as a matching problem, the vertices represent the tasks and the edges the amount of communication, which depends on the metric used. Thus, a complete graph is obtained, and it is processed by the matching algorithm, which gives as result the pairs of tasks so that the amount of communication is maximized. This information is extremely relevant, since, in general, there are few processor cores connected to one same cache. For instance, on many architectures, there is only 2 cores sharing the same L2 cache, therefore, map tasks to them with the matching algorithm works well.

Nevertheless, there are architectures in which more than 2 processors share the same cache, or there are more levels of memory hierarchy to be explored, such as NUMA machines. In these cases, the matching algorithm by itself fails. To overcome this issue, another communication matrix, containing the communication between pairs of pairs of threads, is given as input to the algorithm. This matrix was generated by the following heuristic function:

$$H_{(x,y),(z,k)} = M_{(x,z)} + M_{(x,k)} + M_{(y,z)} + M_{(y,k)}$$

where (x, y) e (z, k) are the matchings found at the previous step, and $M_{(i,j)}$ is the amount of communication between threads i and j. Although this does not guarantee that the result will contain the pairs of pairs with the most amount of communication, it is a reasonable approximation and keeps the time and space complexity polynomial.

D. Mapping with Minas Framework

In order to apply the mapping technique introduced on the previous subsection to applications on real platforms, it is necessary some support on the operating system to ensure thread and memory affinity.

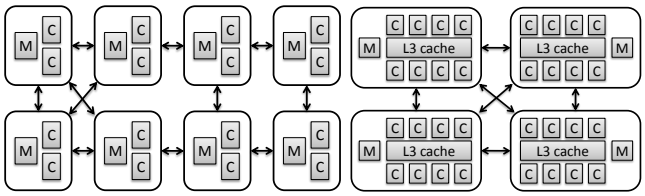
Linux operating system provides an interface named **libnuma** that allows developers to manage affinity on applications [11]. However, this interface, which is a wrapper layer over Linux system calls, provides a limited set of thread mapping strategies and memory policies, which are used to distribute data on memory banks. Additionally, **libnuma** obligates developers to explicitly select nodes and cores that must be used, demanding a large set of hand coded modifications on source code. Because of these constraints, we have searched for other solutions that provide us more transparent mechanisms to control thread and data placement on Linux based machines [12], [13].

Minas is a framework that allows developers to manage affinity of parallel applications on large scale multi-core platforms [14]. This framework avoids any manual application source code modification, since it implements an automatic memory affinity mechanism that controls threads and data mapping. Minas is composed of three modules: numArch library, MApp preprocessor and MAi interface. Numarch extracts all information about the target platform that are necessary to pin threads to cores and place data on memory banks. MApp is a preprocessor that implements a mechanism to extract information of application variables at compile time. MAi, which is the core of Minas, is responsible for implementing several memory policies that deal with data allocation and placement at runtime [12].

Using the Minas Framework, the original NAS Parallel Benchmarks source codes have been modififed to consider threads and data mapping. The code transformation process is divided into three steps. Firstly, Minas retrieves the platform characteristics, using the information extracted with the numArch module (e.g. NUMA factor[1], number of nodes and memory subsystem). Secondly, Minas scanned the application source code with MApp to obtain information about variables (e.g. access mode, parallel section scope, type). During the third step, the information gotten by MApp and numArch are used by Minas to automatically transform the source code by changing the arrays declaration and including MAi specific functions for thread mapping, data allocation and data placement.

The modification performed in NAS Parallel Benchmark source codes guarantee suitable memory policies for data mapping (how memory pages will be distributed over the platform). However, the number of nodes to be used and where to map data depends on the set of cores where threads are running. On Minas, the standard thread mapping mechanism is based on the machine topology. It maximizes cache sharing between threads, since it considers the cache memory levels to map threads on cores. However, in this paper, we aim at use the information generated by the

[1]NUMA factor is the ratio between remote latency and local latency



(a) Machine with Opteron 875. (b) Machine with Xeon X7560.

Figure 3. Evaluated Multi-cored NUMA Platforms.

mapping heuristic (Section III-A) to pin threads on cores.

In order to do so, we have included on Minas the support for input configuration files. Such support allows users to specify a file with the set of cores that must be used to map threads of an application. Using the thread affinity information obtained with the mapping heuristic, we generate a configuration file that is later used by Minas. Using a function from MAi, Minas maps threads to the cores specified in the configuration file at runtime. After that, Minas retrieves threads nodes and maps their data using the chosen memory policy.

IV. Multi-cored NUMA Platforms Evaluated

In order to evaluate the proposed method, we have selected two representative parallel machines, and their topology is present in Figure 3. The Opteron machine is composed by eight NUMA nodes, each node with one dual core processor. It has no shared cache memories and each core has two private cache levels. In Xeon, there are four NUMA nodes, each node with one eight core processor. In this machine, each core has private L1 and L2 cache, and all cores share the same L3 cache (24MB). Both machines are cache coherent Non-Uniform Memory Access (ccNUMA) architectures. However, they have different implementations of the cache coherence protocol, the Opteron machine uses the MOESI protocol whereas the Xeon machine uses the MESIF one [15]. All machines run Linux operating system (kernel 2.6.32) with NUMA support and GCC (GNU C Compiler 4.4.3). For the remainder of the paper, we will name the machines Opteron and Xeon respectly.

Table II
Hardware Details of Multi-core Machines used in our Study.

Characteristic	Opteron	Xeon
Number of Cores	16	32
Number of Processors	8	4
NUMA Nodes	8	4
Clock (GHz)	2.22	2.27
Last Level Cache (MB)	1	24
Total DRAM Size (GB)	32	64
NUMA Factor	1.2 to 1.5	1.20 to 3.6

The main differences between the platforms presented above is their memory subsystem and interconnection design, which gives different memory access costs. All the machines have hardware support and on-chip memory controllers to provide a global shared memory. Such shared memories is actually physically distributed over the machine nodes that are interconnected by an efficient network (e.g. HyperTransport for AMD and QuickPath Interconnection for Intel). The interconnection network gives different memory latencies for remote access by nodes of the platform. Table II summarizes the characteristics of these machines.

V. Experimental Results

In this section, we present the experimental evaluation of the mapping method introduced in this paper. In our evaluation, we have used execution time as performance metric. Considering the benchmark execution, we have used all the cores of each given machine presented on section IV with one thread per core.

Results have been obtained through the average of several executions with exclusive access to the machine. The analysis of the results is per machine and it compares the performances obtained with our solution (best mapping) to the ones obtained with Linux standard mapping control (Operating System) and to the worst mapping. The worst mapping was also calculated using the Edmonds matching, but with the minimum cost perfect matching [16] instead of the maximum cost perfect matching. The maximum standard deviation was 20% for Linux Default, 3.5% for best mapping and 16% for worst mapping, for the results considering the total amount of shared memory. Regarding the total number of shared access, the results had 8% and 14.6% maximum standard deviations for best and worst mapping, respectively. High standard deviations are expected for the Linux Default, since it can map threads to different cores during each execution. Both the amount of shared memory and access count metrics were evaluated.

Linux schedules threads considering their access to memory hierarchy. During the application execution, if Linux notices that a thread is generating more cache misses, it will re-schedule this thread to a new core in order to try to reduce cache misses in the future. However, some times Linux fails in the prediction and such re-scheduling can generate even more cache misses. Additionally, since the evaluated platforms are NUMA, the re-scheduling can led to remote accesses, which decreases the overall performance. Considering data placement, Linux uses the first-touch memory policy, in which only the first access by threads on data are considered for data placement on the DRAM memories. Contrary to this strategy, our method allow us to observe data sharing between threads and place them considering these patterns.

Figure 4 shows the execution time obtained with the benchmarks on the Opteron machine. This machine do not have shared cache memories, so data sharing between

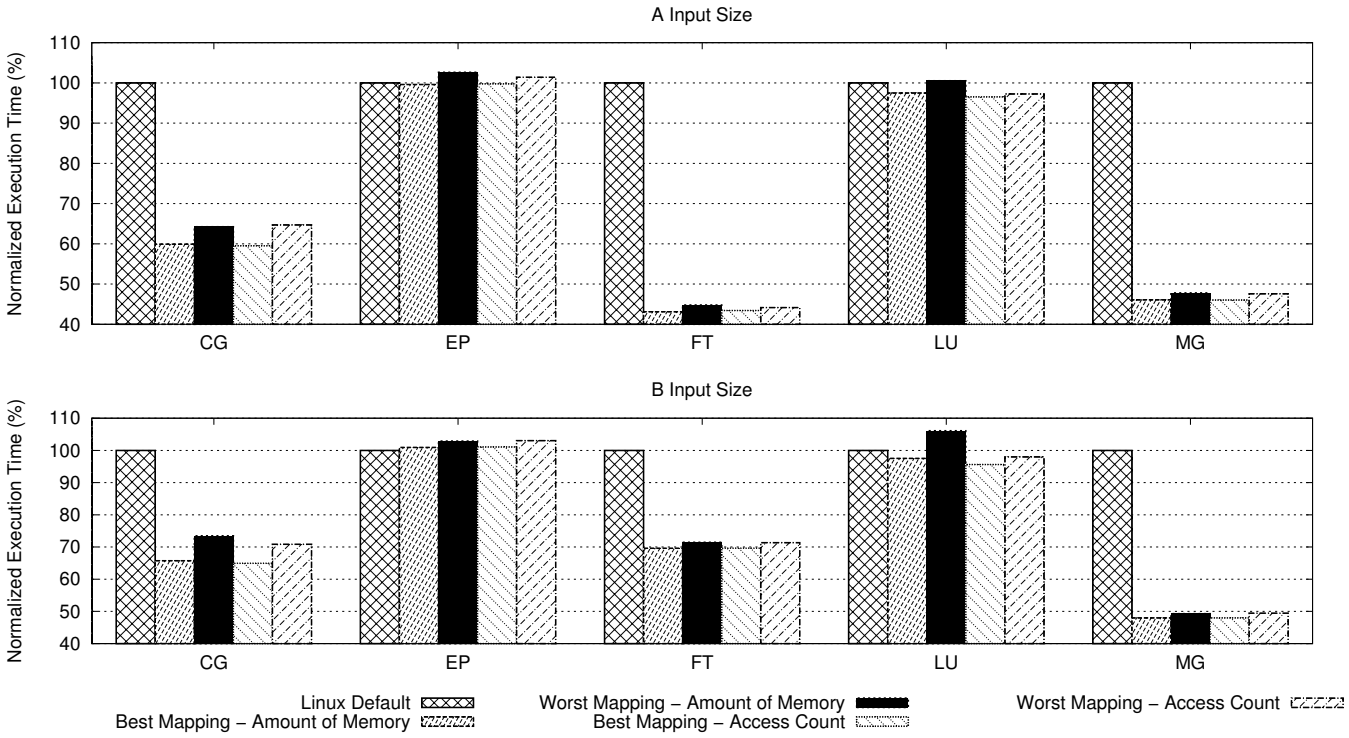


Figure 4. Execution time of the applications on the Opteron platform.

threads is related to DRAM memories. The EP benchmark has not presented any speedup. However, this is the expected result, since EP is CPU-bound and its threads perform independent computation on their private data. Thus, any improvement gains can be archived for this benchmark when using our method.

We have significant performance gains with A input in CG (up to 40%), FT (up to 55%) and MG (up to 55%) benchmarks when compared to results obtained with the operating system on the Opteron machine. As these benchmarks are more sensitive to memory access (more shared data), and considering the characteristics of the machine, placing threads that share some data closer reduces the number of remote access. On the other hand, although LU presents an heterogeneous communication pattern, gains only up to 4% were obtained. The reason for this difference is that, in CG, FT and MG, some application data is initialized by one thread, and, in LU, each thread initializes all data that they need, therefore, Linux first touch policy works well only for LU.

The knowledge of how threads access data allow us to perform some optimization on data allocation and placement by using Minas memory policies. These memory policies allow us to ensure memory affinity, reducing the NUMA penalties such as load balancing, memory contention and remote access. In the case of CG, FT and MG, we guarantee load balancing and less memory contention by using all the DRAM memories available on the machine, whereas the operating system has placed more data on some restrict DRAM memories.

Figure 5 reports the execution time obtained with the benchmarks on the Xeon machine. Similar to the results obtained with the Opteron machine, we observe important gains for CG, FT and MG benchmarks and none for EP. No gains were obtained with CG using the A input. The reason for this behavior is that with 32 threads it executes so fast that the overhead of the mapping is greater than the benefits. Considering LU benchmark, for this machine, no significant gains were obtained. As mentioned, Linux first touch policy works well for LU, so the improvement on performance are similar to that obtained in UMA machines, which are less than 2% for this application according to [9].

Regarding CG, FT and MG benchmarks, gains are up to 75%, 50% and 70% respectively with B input when compared to the operating system and worst mapping. On this machine, the operating system mapping is similar to the worst mapping generated by our technique. As mentioned on above paragraphs, these benchmarks have more shared data and different memory access patterns. Because of this, our method allow us to better map threads and their shared data over the machine. Considering the CG and MG benchmarks, their main characteristic is the indirect access by threads on some arrays. Due to this, Linux can not perform an efficient thread and data mapping for them.

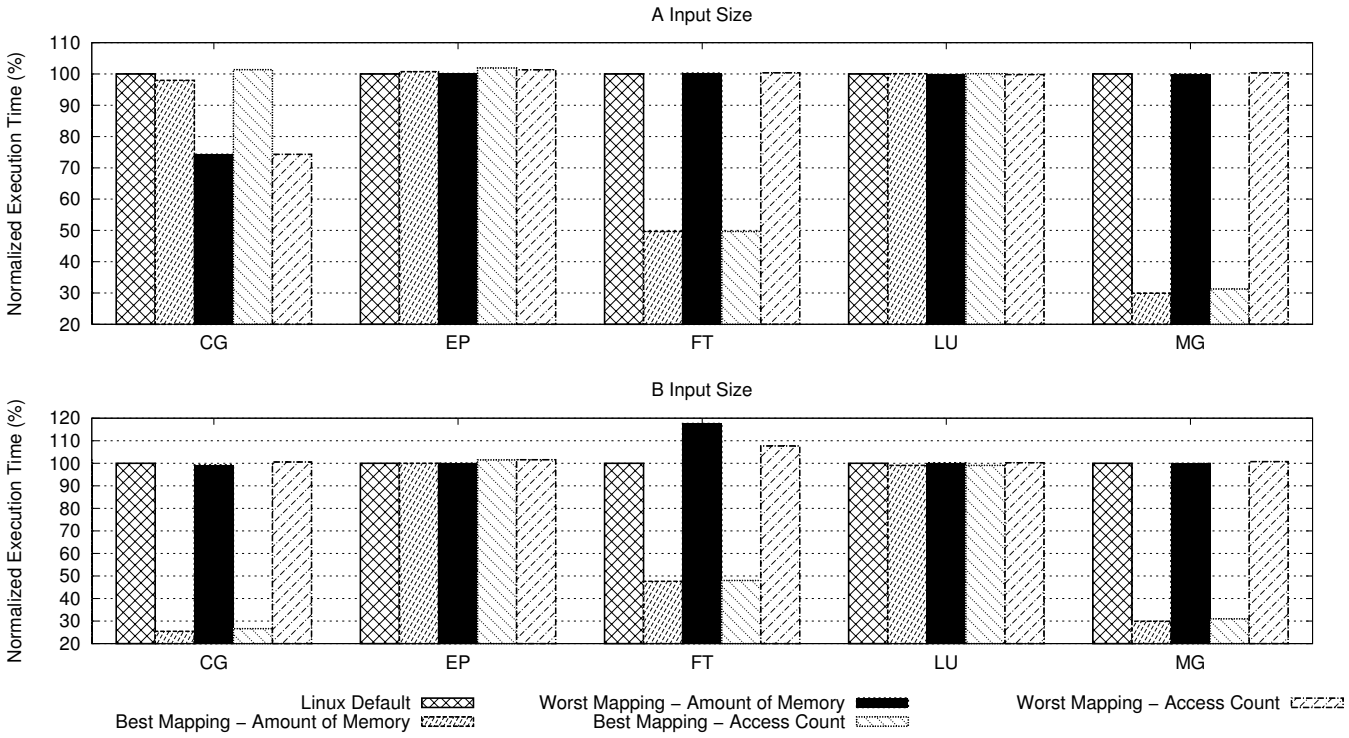


Figure 5. Execution time of the applications on the Xeon platform.

One important thing to notice is that the difference between the best mapping and worst mapping is much lower than when compared to Linux default in the Opteron machine, but for the Xeon, an opposite behavior is accomplished. The main reason for this is that the Xeon machine has more cores per NUMA node to be explored by mapping than the Opteron machine, therefore, there is a greater latency in the communication time between the best and worst mapping.

VI. Related Work

In order to reduce the necessary efforts to control thread and data mapping on OpenMP applications, transparent data distribution has been object of study of several OpenMP implementations [17], [13], [18].

Dynamic task and data placement for OpenMP applications have been proposed in [13]. In these works, researchers have proposed an NUMA-aware runtime for OpenMP, named ForestGOMP. This proposal relies on hints provided by developers and extracts information about the architecture to better distribute them over the NUMA machine. Such runtime is an extension of to the GNU OpenMP GNU library, which restricts its usage. Additionally, some modifications on the application source codes are needed to provide some hints to the runtime system (e.g. data distribution, which variables to consider).

In [17], researchers have compared runtime and manual data distribution for OpenMP over NUMA platforms. They have shown that automatic data distribution algorithms (e.g. *first-touch*) are easy to use, but have generated worser results than manual data distribution. These researchers have concluded that it is important to select data distribution strategy considering the target application. Thus, it may be interesting to have a solution that combines compile time application information with runtime one to better distribute data for OpenMP applications on NUMAs.

Static process mapping of MPI applications has been evaluated in [18], where performance gains of up to 9.16% were obtained when mapping the groups of threads that send more data among them to nearby processors. Although performance gains were achieved, they were not as great as expected, since the experiments were performed on a cluster, which imposes a high latency remote access and has high potential for process mapping. Besides, as the parallel programming paradigm used was messaging passing, discovering the communication pattern is straightforward when compared to shared memory, and it was accomplished by adding wrappers to the MPI functions that register informations about the messages sent.

VII. Conclusions and Future Work

Future multi-core and many-core processors with tens of processing cores will require new techniques to control the computational resources available on the entire machine. This way, techniques for process mapping such as we presented will have a key role for future architectures.

This paper presented a technique to map threads and its data from parallel applications over multi-core machines with NUMA characteristics. We have used memory traces and an heuristic algorithm to estimate the most suited thread and data placement for each application in a given architecture. In order to evaluate our proposal, we have performed some experiments on two modern multi-core machines using NAS Parallel Benchmarks.

Results have shown performance improvements of up to 75% when compared to the Linux standard solution for thread and data mapping. Additionally, our results have shown that applications with homogeneous communication patterns, such as EP, may not benefit from mapping. Another important result is that, sometimes, the original scheduler of the operating system performs worse than the worst mapping, since it periodically migrates the threads.

As future work, we intend to develop a deeper mapping technique, which considers the different phases of the application to map threads and data. Additionally, the design of more efficient methods to retrieve the memory sharing and access patterns are also considered. Tools using dynamic binary analysis, like Pin, are being cogitated. Furthermore, we pretend to develop dynamic mechanisms to detect the sharing pattern and extend the presented techniques to be used in a dynamic scheduler.

ACKNOWLEDGMENT

This research has been partially supported by the CAPES-BRAZIL under grant 4874-06-4 and CNPq-BRAZIL.

REFERENCES

[1] A. Joseph, J. Pete, and R. Alistair, "Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport," in *High Performance Computing - HiPC*, 2006, pp. 338–352.

[2] C. Terboven, D. A. Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and Thread Affinity in OpenMP Programs," in *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*. ACM, 2008, pp. 377–384.

[3] M. Diener, F. Madruga, E. Rodrigues, M. Alves, J. Schneider, P. Navaux, and H.-U. Heiss, "Evaluating thread placement based on memory access patterns for multi-core processors," in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, 2010, pp. 491 –496.

[4] J. Y. Haoqiang Jin, Michael Frumkin, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," NAS System Division - NASA Ames Research Center, Tech. Rep. 99-011/1999, 1999. [Online]. Available: https://www.nas.nasa.gov/Research/Reports/Techreports/1999/PDF/nas-99-011.pdf

[5] P. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Computer Micro*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[6] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal, "Analyzing parallel programs with pin," *Computer*, vol. 43, no. 3, pp. 34 –41, 2010.

[7] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, 2007.

[8] Scotch, "Scotch," http://www.labri.fr/perso/pelegrin/scotch/, December 2010.

[9] E. H. Cruz, M. A. Alves, and P. O. Navaux, "Process mapping based on memory access traces," in *Computing Systems (WSCAD-SCC), 2010 11th Symposium on*, 2010, pp. 72 –79.

[10] C. Osiakwan and S. Akl, "The maximum weight perfect matching problem for complete weighted graphs is in pc," in *Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium on*, 9-13 1990, pp. 880 –887.

[11] A. Kleen, "A NUMA API for Linux," Tech. Rep. Novell-4621437, 2005. [Online]. Available: http://whitepapers.zdnet.co.uk/0,1000000651,260150330p,00.htm

[12] C. Pousa Ribeiro, M. Castro, L. G. Fernandes, A. Carissimi, and J.-F. Méhaut, "Memory Affinity for Hierarchical Shared Memory Multiprocessors," in *21st International Symposium on Computer Architecture and High Performance Computing*. São Paulo, Brazil: IEEE, 2009.

[13] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective," in *5th International Workshop on OpenMP*. Dresden, Germany: Springer, 2009, pp. 79–92.

[14] C. Pousa Ribeiro, I. S. Nicolas Maillard, and J.-F. Méhaut, "Compiling OpenMP Applications to Enhance Memory Affinity on Hierarchical Multi-Core Machines," in *23rd International Workshop on Languages and Compilers for Parallel Computing*. US: LNCS, 2010.

[15] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *18th International Conference on Parallel Architectures and Compilation Techniques*. USA: IEEE, 2009, pp. 261–270.

[16] V. Kolmogorov, "Blossom V: A new implementation of a minimum cost perfect matching algorithm," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 43–67, 2009.

[17] D. S. Nikolopoulos, E. Ayguadé, and C. D. Polychronopoulos, "Runtime vs. manual data distribution for architecture-agnostic shared-memory programming models," *Int. J. Parallel Program.*, vol. 30, no. 4, pp. 225–255, 2002.

[18] E. Rodrigues, F. Madruga, P. Navaux, and J. Panetta, "Multi-core aware process mapping and its impact on communication overhead of parallel applications," in *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*, 5-8 2009, pp. 811 –817.