International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 2, Number 1, pages 97–116, January 2012

Memory-aware Thread and Data Mapping for 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
{ehmcruz, mazalves, asc, navaux}@inf.ufrgs.br

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

Received: July 25, 2011
Revised: October 30, 2011
Accepted: December 15, 2011
Communicated by Akihiro Fujiwara

Abstract

In parallel programs, the threads 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, thread and data mapping are techniques that provide 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. Our mechanism provides static mapping on NUMA architectures which does not require any prior knowledge of the application by the programmer. To obtain the mapping, different metrics were adopted and an heuristic method based on the Edmonds matching algorithm was used. In order to evaluate our proposal, we use the NAS Parallel Benchmarks (NPB) running on two modern multi-core NUMA machines. Results show performance gains of up to 75% compared to the native Linux scheduler and memory allocator.

1 Introduction

On shared memory parallel platforms with a hierarchical memory sub-system, the communication time spent between threads to accomplish data sharing in parallel programs may be different, depending on how the processors or cores are interconnected through the memory hierarchy, the

interconnections used and the cache coherence protocol [14]. This difference is even more relevant on parallel machines with non-uniform memory access characteristics (NUMA), since the latency and the memory bandwidth to get data vary depending where the processors are in the topology.

In this context, thread and data mapping help to improve system performance by optimizing the usage of resources such as interconnections, main memory and cache memory. Thread mapping helps by placing groups of threads which shares high amounts of data to cores that shares some level of cache memory, reducing unnecessary data replication and invalidation due to coherence protocols. Data mapping is more suitable for NUMA Machines, and consists of placing memory pages in DRAM memory banks which are close to the cores that are accessing the page. Mapping parallel programs on NUMA becomes harder [32] and more expensive, because usually there are more levels on the memory hierarchy to be explored. Furthermore, the problem to find the best mapping is considered NP-Hard [11] 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 to map the threads of a given application on cores and allocate their data on DRAM memories. Our main objective is to reduce the overhead imposed by data sharing on multi-core machines with NUMA characteristics, improving the overall system performance. To perform the static mapping, our proposed model is based on the analysis of memory traces and does not require any prior knowledge of the application by the programmer. We use two metrics to identify the data sharing pattern between threads. The first metric is the amount of memory shared between the threads, and the second metric is the number of accesses performed to a memory region shared between the threads.

To generate the memory traces, we instrumented the Simics simulator [17] to monitor all the memory accesses and save them on trace files. These memory traces are analyzed to detect the communication pattern by an algorithm that calculates the thread affinity with the memory hierarchy. To avoid exponential time complexity, this algorithm is heuristic, based on the Edmonds matching algorithm [23], and provides a well suited thread affinity for a selected application. The detected thread affinity is used by the Minas framework, which is responsible for mapping the threads and data.

In comparison to related works, our approach differs in at least two aspects. First, we provide an heuristic that can be adopted in different shared memory architectures. Second, we consider both 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 2 introduces the method proposed for thread and data mapping. Section 3 presents the general description of the applications used to evaluate the proposed technique. The platforms used to validate the mapping technique are presented in Section 4. The experimental results are presented in Section 5. Some related works are presented and compared in Section 6. Section 7 presents our conclusions and future works.

2 Mapping Threads and Data

The static mapping proposed in this paper is performed in five steps, as described in Figure 1: monitor the memory accesses to generate the traces; analysis of the trace to generate the sharing matrix; calculate the thread and data affinity with the memory hierarchy; map threads and data with the Minas framework [25]; and execute the application with the best mapping found.

2.1 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, which stores the amount of communication between each pair of threads. In shared memory environments, it is necessary to monitor all the memory accesses of the applications, which was accomplished by executing them inside the Simics

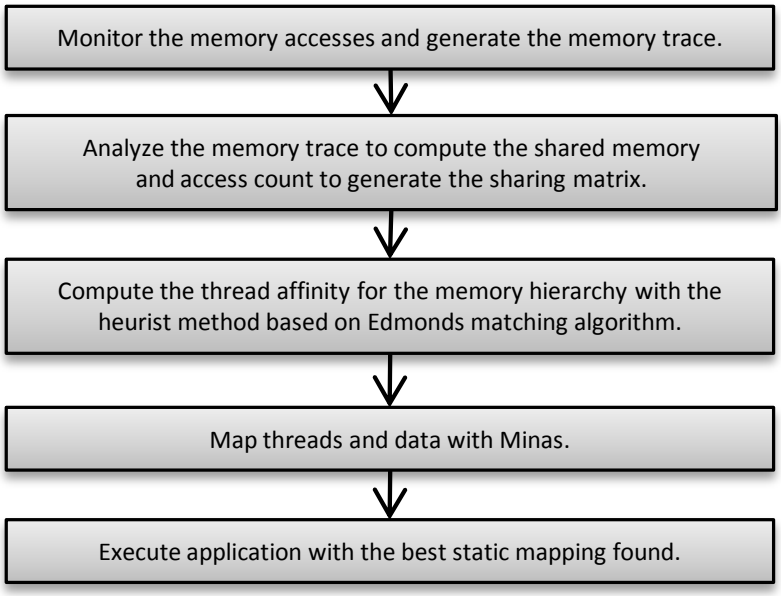


Figure 1: Methodology used to perform the static mapping.

[17] simulator. Simics was instrumented [3] to register 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 accesses, the event *Core_Breakpoint_Memop* can be triggered. However, as the triggers are implemented in the Python language, they have a high overhead, decreasing the simulation speed. Therefore, it was developed a memory trace 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 accesses performed by the processor.

It is necessary to filter the accesses to be registered, so that only the memory accesses 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 becomes 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. This way, the memory trace module is able to detect if a memory access was performed by the application being analyzed.

Another possibility to generate the memory traces is through dynamic binary instrumentation, using tools such as Pin [2, 5] or Valgrind [21]. Although dynamic binary instrumentation is easier and faster than simulation, in some cases it can alter the application behavior. For instance, the Valgrind tool serializes the threads in parallel applications, which may lead to different communication patterns.

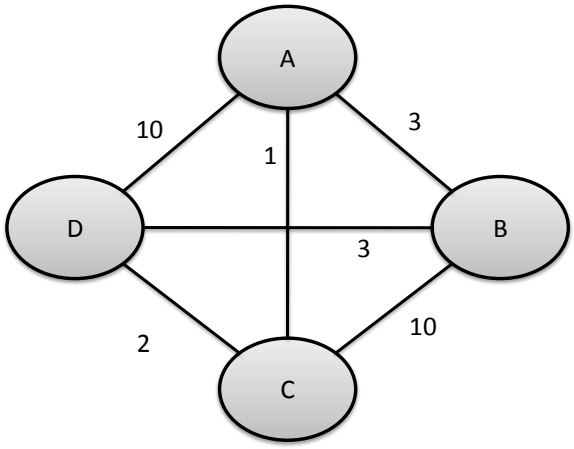
2.2 Generating the Sharing Matrix

The memory traces alone are not enough to guide the thread mapping. The traces must be analyzed to discover the communication pattern, which depends on the metric adopted. For this work, two metrics were separately considered to evaluate the communication: the amount of memory shared by threads and the number of accesses performed to a block of memory that is shared.

The amount of memory shared by threads metric is more suitable to applications in which the number of accesses to the shared memory is insignificant when compared to the number of accesses to the private memory. On the other hand, the amount of accesses to the shared memory metric are better to describe the behavior of applications that present a huge amount of accesses to the shared memory.

	A	B	C	D
A		3	1	10
B	3		10	3
C	1	10		2
D	10	3	2	

(a) Sharing Matrix



(b) Graph

Figure 2: Sharing Matrix and the corresponding Communication Graph.

Problems like false sharing were taken into account, as they are very common in multiprocessor architectures. The tool we developed to analyze the memory traces organizes the memory address space in blocks of memories in the same way the cache organizes the memory into cache lines. By setting the block size to the value of the cache line size of the target architecture, the tool generates the data sharing information considering falsely shared addresses.

The shared memory can be analyzed grouping different number of threads. To calculate how much memory is being shared between any group of threads size, the amount of space necessary rises exponentially, discouraging the use of thread mapping. Therefore, the shared memory was evaluated only between pairs of threads, generating a sharing matrix. Although this pair analysis 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.

2.3 Thread Affinity with the Memory Hierarchy

After the generation of the sharing matrix, it is necessary to map the threads and their data. The mapping problem is considered NP-Hard, consequently, finding the optimal solution becomes infeasible when the number of threads grows. Thus, heuristic algorithms must be employed to determine the mapping in reasonable time, with results as close as possible to the perfect mapping. Methods like the Dual Recursive Bipartitioning produces good results, and are available on the software Scotch [30]. 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 [23], 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 thread mapping as a matching problem, the vertices represent the threads and the

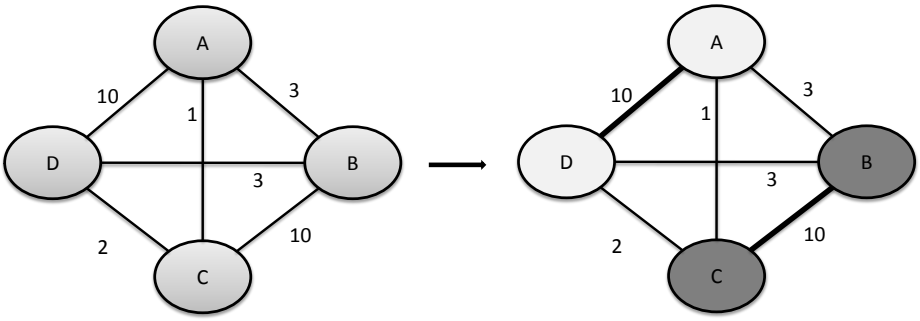


Figure 3: The Matching Problem.

edges represent the amount of communication. A complete graph is obtained directly from the sharing matrix, as exemplified in Figure 2. The graph is processed by the matching algorithm, that outputs the pairs of threads so that the amount of communication is maximized, which is an extremely relevant information, since, in general, there are few processor cores connected to one same cache. Figure 3 shows the results that the matching algorithm would produce for the given graph, in which the threads with the same color should be mapped together.

On many architectures, there are only 2 cores sharing the same L2 cache, therefore, mapping threads to them with the matching algorithm is straightforward. 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 is insufficient. To overcome this issue, another communication matrix, containing the communication between pairs of pairs of threads, is given as input and the algorithm is re-executed. This matrix was generated by the following heuristic function:

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

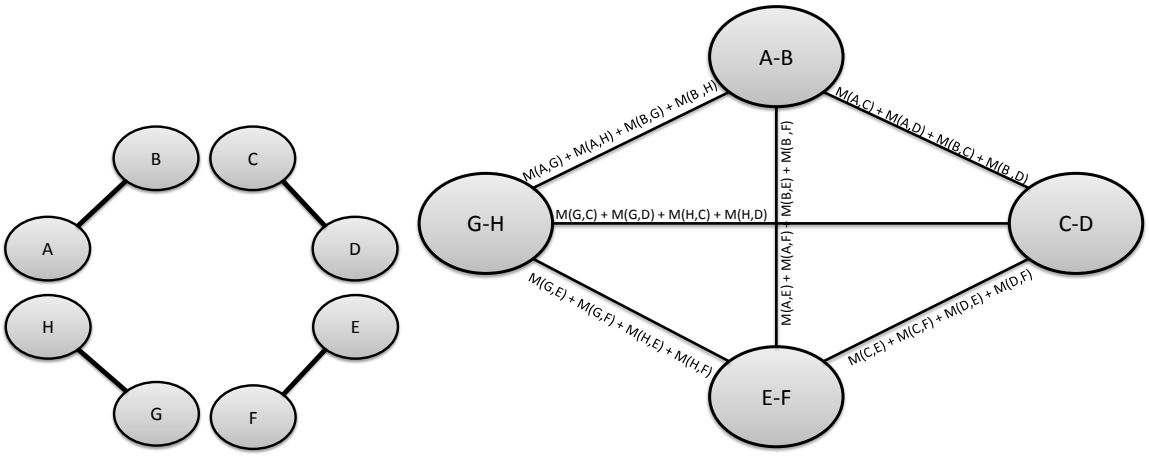
where x, y, z and k are thread ids, (x, y) and (z, k) are the matchings found at the previous step, and $M_{(i,j)}$ is the amount of communication between threads i and j. The result obtained with the heuristic function is represented in Figure 4. Although this does not guarantee that the result will contain the pairs of pairs with the most amount of communication, as the sharing matrix does not provide sharing information about groups with more than 2 threads, it is a reasonable approximation and keeps the time and space complexity polynomial.

However, the number of cores sharing a cache may not be 2^x, where x is an integer. In this case, the matching algorithm is not able to cluster all the threads properly. Considering an architecture with 6 cores and 2 caches, where each cache is shared by 3 cores, for some given graph, the matching algorithm would produce the result shown in Figure 5(a). As can be seen, the resulting graph contains 3 disconnected graphs, but the target architecture has only 2 caches. To overcome this issue, we sort the pairs found according to the edge weight, and group only the ones that maximizes the total amount of communication, as in Figure 5(b). A bipartite graph is then generated, as exposed in Figure 5(c), and the matching algorithm can be applied again to group the threads for the target architecture, as show in Figure 5(d). The graph must be bipartite in order to prevent matchings between threads already clustered, and between threads that were not clustered yet. This process can be modified to map any number of threads.

2.4 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 data distribution over the machine.

Linux operating system provides an interface named *libnuma* that allows developers to manage affinity on applications [15]. However, this interface, which is a wrapper layer over Linux system



(a) Matching from previous step.

(b) Graph obtained after applying the heuristic function.

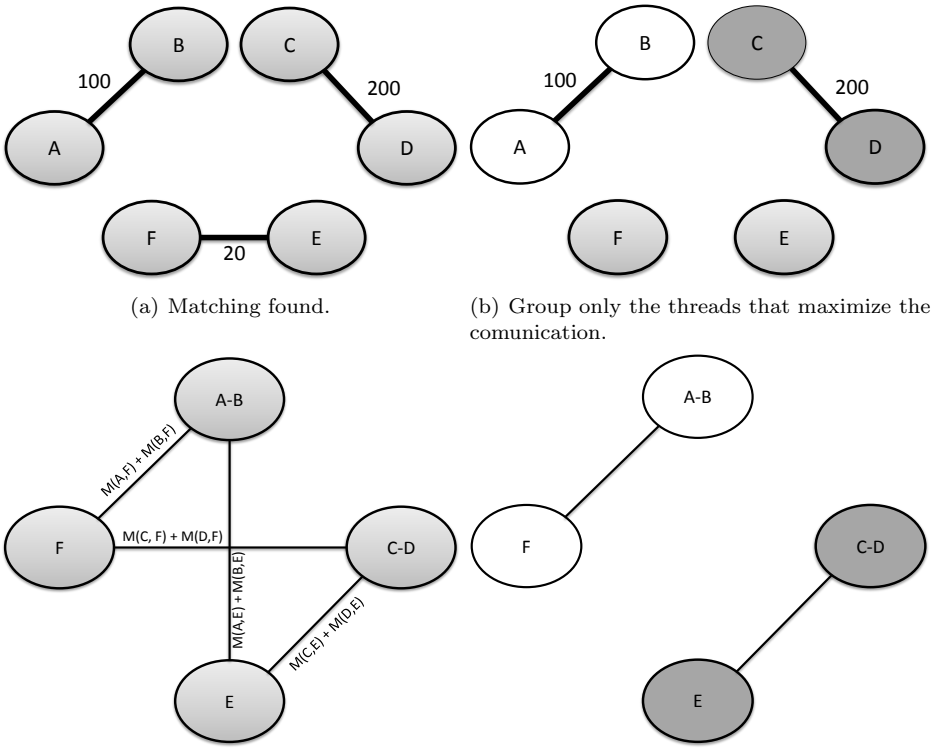
Figure 4: Heuristic used to generate new communication graphs (or sharing matrix) from previous matching.

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 [25, 6].

Minas [27] is a framework that allows developers to manage affinity on parallel applications for large scale multi-core platforms with NUMA design [25]. It provides a fine control of memory accesses for application data and similar performance on different NUMA platforms. Additionally, Minas allows architecture and compiler abstraction and none or minimal modifications to the source code of the applications. Minas is composed of three components: MAi, MApp and numArch.

MAi is a high level API that is responsible for implementing the explicit thread and data placement mechanism of Minas. It provides simple and high level functions that can be called in the application source code to perform data allocation, placement and migration [25, 28]. MAi functions can be divided in three groups: allocation and memory policies. Allocation functions are responsible for reserving space for application data on heap, similar to a standard malloc function. Memory policy functions are used to physically distribute data among the memory banks of the machine. MAi implements the memory two types of policies that can be used to optimize memory access on NUMA platforms taking into account the latency and the bandwidth. Regarding thread placement, MAi implements the classical compact thread placement mechanisms that is used to better manage memory affinity.

The MApp preprocessor implements an automatic mechanism to place data by considering the application and platform characteristics at compile time. MApp retrives application information using a two level parser. The first one extracts information of variables whereas the second one extracts information of the parallel constructions of the programming interface. The parser responsible to extract information of variables is written with the Lex/Yacc tools and it is called CUIA. It aims at providing variables information of the parallel application. CUIA parses C code and returns, for each of the variables: (i) the name and type; (ii) the lexical scope, and (iii) the name of the file where it has been declared. When the variable is a static array, CUIA also obtains the number of dimensions of the array and the number of entries in each one of the dimensions; (iv) a list of the access modes made on the variable: Read, Write, or both, and (v) the location in the program where these accesses occur. The second level of the parser is implemented inside MApp and it retrieves information of how data is distributed for worker threads (work sharing) and the sharing type of a static array (i.e. shared vs private). Using the information generated by the parsers, MApp changes the static array declaration to use the memory allocation provided by MAi. It also includes for each



(a) Matching found.

(b) Group only the threads that maximize the comunication.

(c) Re-generate the graph.

(d) Final result.

Figure 5: Steps when the number of cores sharing a cache is 3.

static array a memory policy from MAi considering the access mode on them.

The last module, numArch, extracts several information about the target platform, which is then used by the MAi and the MApp components. These informations are combined together to represent the machine topology and the impact of the non-uniform memory access on application performance. Therefore, numArch describes how processing elements share memories and how they are related to each other. Considering Minas objectives and scope, the following information is necessary: number of NUMA nodes, number of cpus/cores, number of sockets, number of caches, size of cache memories, set of cores that share a cache memory, memory banks size, free memory and relation between nodes and cpus/cores. To retrieve such information, numArch parses the topology-related */sys/devices/* and */proc/PID/* file system of the Linux operating system. From the file system, numArch gets the information of the hardware of the machine parsing some text files. From these text files , numArch extracts the information of the nodes (number and physical id), of the cores (number, physical id and siblings cores) and of the cache memory hierarchy (number of levels, size and sharing among cores). After parsing step, the obtained information is stored in temporary files on the */tmp/* of the machine. In the initialization of Minas, these files on */tmp/* are loaded to dynamic structures (e.g. queues, hash tables and matrices) that can be later accessed using the numArch interface.

Figure 6 shows a scheme of the Minas approach to enhance memory affinity. The original application source code can be modified by either using the explicit mechanism (gray arrows) or the automatic one (black arrows). In the case of the explicit mechanism the programmer has to change the application source code to manually improve memory affinity. In contrast to this approach, in the automatic mechanism the application source code is automatically changed by Minas. The decision between automatic and explicit mechanisms depends on the developer's knowledge about the target application and the characteristics of the application source code. Since MApp preprocessor current version only deals with C language and static arrays, it is possible to use it only on aplication with these characteristics [24, 26].

Using the Minas Framework, we modify the original source code to consider threads and data

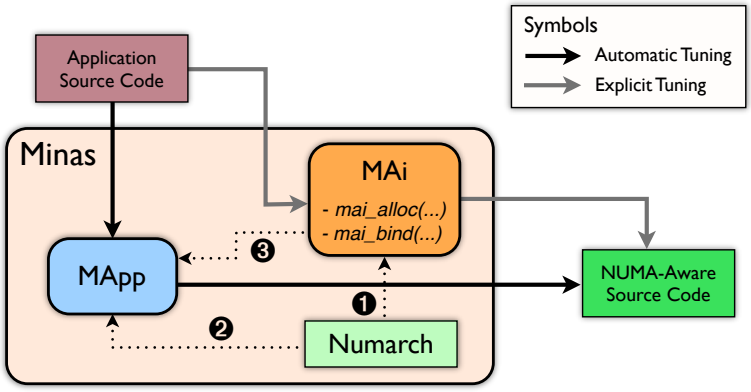


Figure 6: Overview of Minas Framework.

mapping. The code transformation process is divided into three steps. Firstly, we use Minas to retrieve the platform characteristics, using the information extracted with the numArch module (e.g. NUMA factor[1], number of nodes and memory subsystem). Secondly, Minas scan the application source code with MApp to obtain information about variables. During the third step, the information gotten by MApp and numArch are used to manually (dynamic variables) or automatically transform the application source code (static variables). To do so, the source code is changed to include MAi specific functions for thread mapping, data allocation and data placement.

The modifications performed in the source code ensure 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 use the information generated by the mapping heuristic (Section 2.1) 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 the 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.

3 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 [13]. 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 language and are parallelized with OpenMP.

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 1 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,

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

Table 1: 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.

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.

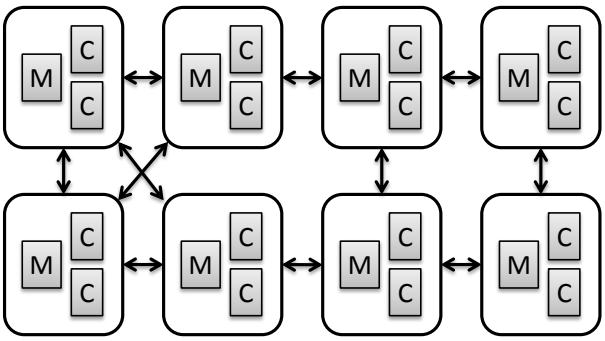
The BT, IS, SP and UA applications, which are also part of the NPB benchmark, were not used in this paper. Regarding BT, we have observed it does not scale for more than 12 threads with the W input, therefore, the mapping obtained is not suitable for the A and B inputs and, consequently, no performance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

4 Multi-core NUMA Platforms Evaluated

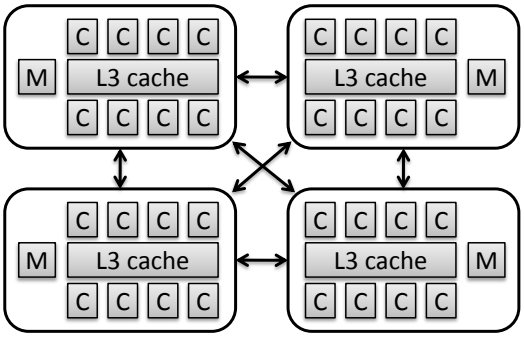
In order to evaluate the proposed method, we selected two representative parallel machines. The first is a multi-core machine based on eight dual-core AMD Opteron Processor 875 and the second is a Dell PowerEdge R910 equipped with 4 eight-core Intel Xeon X7560. All machines run Linux operating system (kernel 2.6.32) with NUMA support and GCC (GNU C Compiler 4.4.3). For the remainder of the paper, we will refer to these machines as Opteron and Xeon respectly.

Figure 7 shows the topology of evaluated machines. The Opteron machine is composed by eight NUMA nodes, each node with one dual core processor, as shown in Figure 7(a). It has no shared cache memories and each core has two private cache levels. The Xeon machine has four NUMA nodes, each node with one eight core processor, as shown in Figure 7(b). Each core has private L1 cache (32KB) and L2 (256KB) and all cores share the same L3 cache of 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 [20]. These two protocols have different number of hops and messages to guarantee the cache coherence on the platform.

Regarding thread and data placement, 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 are 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 2 summarizes the characteristics of these machines.



(a) Machine with Opteron 875.



(b) Machine with Xeon X7560.

Figure 7: Evaluated Multi-cored NUMA Platforms.

5 Experimental Results

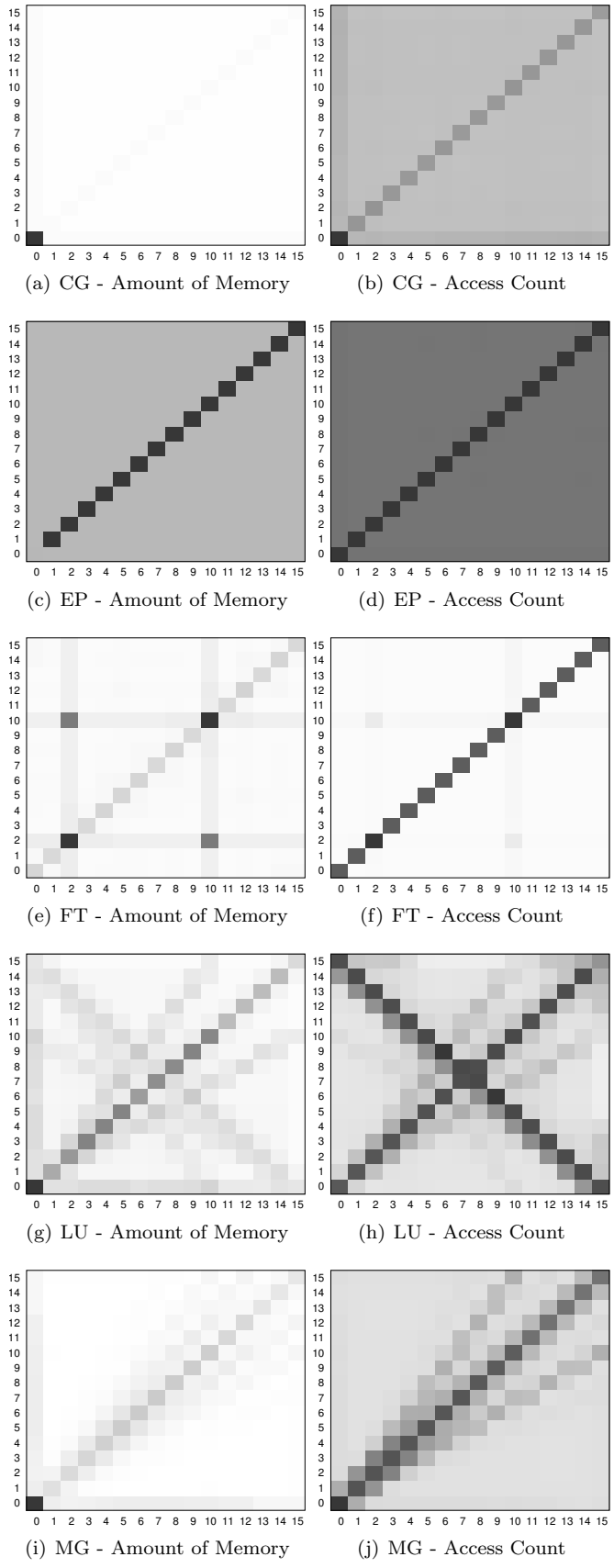
In this section we evaluate the impact of our mapping mechanism on the selected benchmarks. For the evaluation, we have used execution time as performance metric. Considering the benchmark execution, we have used all the cores of each given machine presented on section 4 with one thread per core. All the results are based on an arithmetic mean of several runs. First, we present and discuss the shared matrices generated for the selected benchmarks. After that, we evaluate the performance impact of the proposed method in the two selected NUMA platforms. Then, more details from two benchmarks are presented in order to perform a deeper analysis of the performance improvements.

5.1 Shared Matrices for NPB

Figure 8 contains the communication pattern of the applications of NPB with the W input size for the amount of memory and access count 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. Darker cells represent more 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. This happens because, in some steps of the initialization, thread 0 accessed almost all the data. 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. The difference between the results obtained with the metrics indicates that each address in the shared data is accessed several times. Otherwise, both metrics would exhibit the same behavior.

The Figure also shows that MG has a communication pattern where the nearby threads communicate more among themselves. This is very common in parallel applications based on domain decomposition, where most of the shared memory is located at the border of each sub-domain. Both



(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 8: Sharing Matrices showing the communication patterns of NPB applications.

Table 2: Hardware details of multi-core machines used in our evaluation.

Characteristic	Opteron	Xeon
Number of Cores	16	32
Number of Processors	8	4
NUMA Nodes	8	4
Clock Frequency (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

metrics indicates this sharing behavior. However, with the access count metric, the domain decomposition pattern is more evident. As with CG, this indicates that each address in the shared data is accessed several times. However, the sharing pattern of MG is more heterogeneous than CG, as there are more shared memory regions regarding a subset of threads. Therefore, MG presents more potential for performance improvement with mapping than CG.

The application LU, besides the domain decomposition pattern, also presents huge amount of communication between the most distant threads. With the amount of memory metric, the amount of communication between the most distant threads and nearby threads is about the same. However, the access count metric makes the communication between the most distant threads more evident than the communication between nearby threads. This shows that the amount of shared data between the nearby threads is equivalent to the amount of shared data between the most distant threads, but the most distant threads present more access to the shared data.

EP is the one with the most homogeneous communication pattern, hence it shall not benefit from mapping like the other benchmarks. In FT, the amount of shared memory metric shows more communication than the access count metric. This happens because the number of access to the private area overwhelms the number of access to the shared memory.

5.2 Performance Evaluation

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 results 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 algorithm. Both the amount of shared memory and access count metrics were evaluated. 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.

Linux schedules threads considering their access to memory hierarchy. During the application execution, if Linux notices that one given thread has high cache miss rate, it will re-schedule this thread to a new core in order to try to reduce cache misses in the future. However, sometimes 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 lead to remote memory 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 9 shows the execution time obtained with the benchmarks on the Opteron machine. This machine do not have shared cache memories, so data sharing between threads is related to DRAM

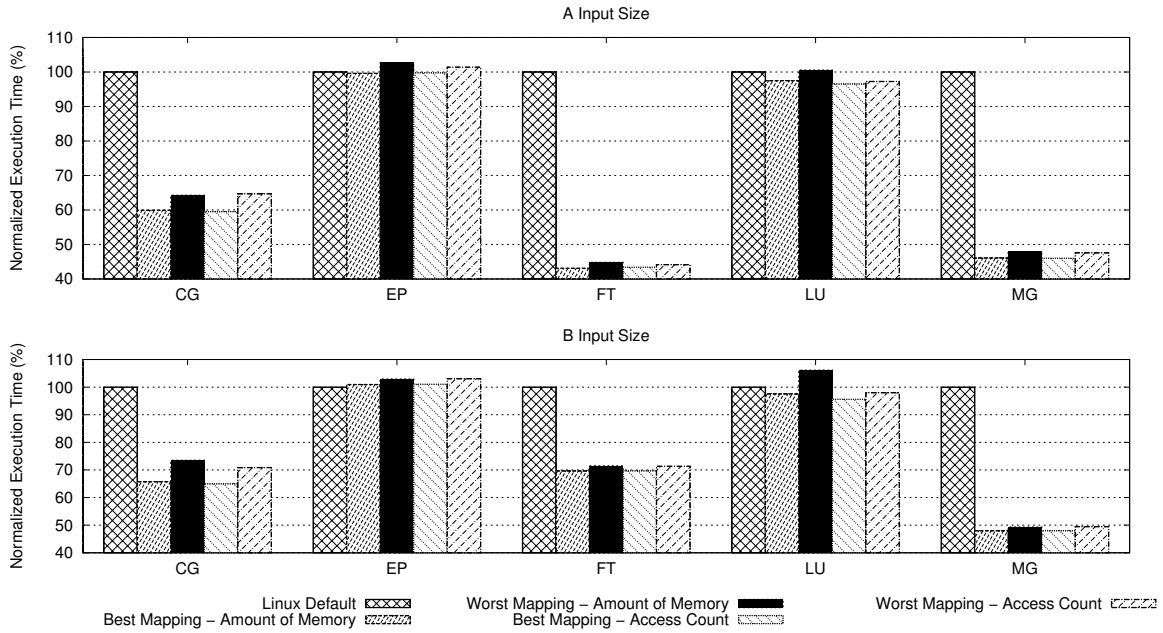


Figure 9: Execution time of the applications on the Opteron platform.

memories. The EP benchmark has not presented any speedup. However, this result is expected, since EP is CPU-bound and its threads perform independent computation on their private data. Thus, no performance improvement can be archived for this benchmark using our method.

Significant performance gains were achieved with the 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 of 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 10 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 previous paragraphs, these benchmarks have more shared data and different memory access patterns. Thus, 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

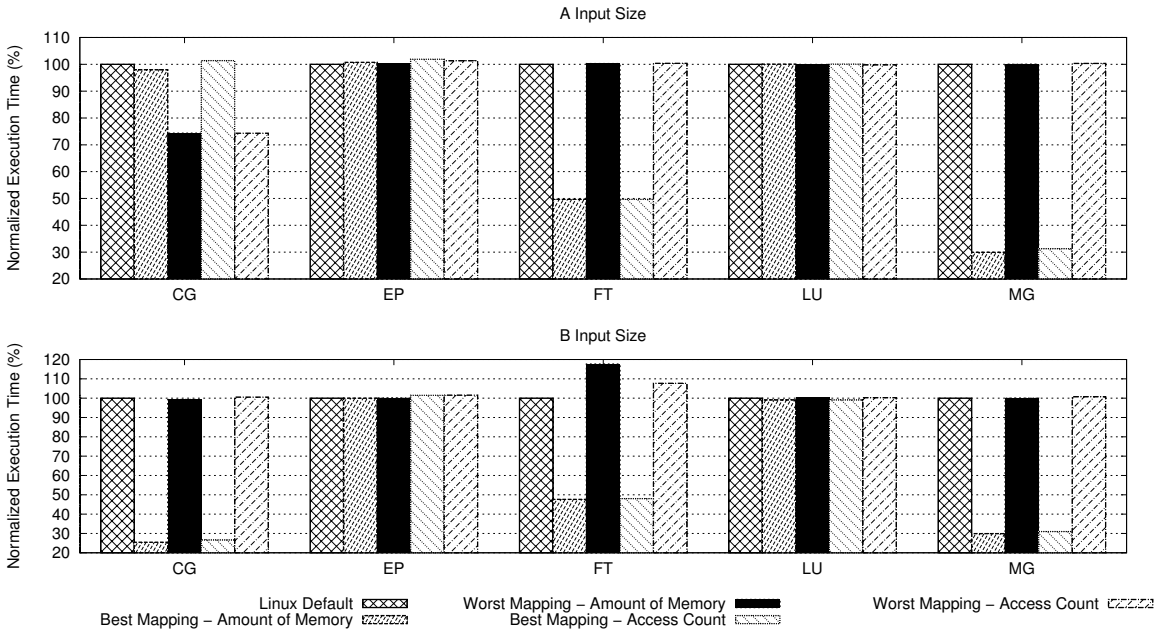


Figure 10: Execution time of the applications on the Xeon platform.

on some arrays, in such way that Linux can not perform an efficient thread and data mapping.

It is important to note that, for the Opteron machine, the difference between the best mapping and worst mapping is much lower than the difference between the best mapping and the operating system scheduler. For the Xeon, an opposite behavior is observed, the performance of the worst mapping is close to the performance of the operating system. This difference between the behaviors on the Opteron and Xeon shows that the performance improvement of our mechanism does not depend just on the behavior of the applications, but also on the characteristics of the architecture.

Considering data mapping, the Xeon machine has more cores per NUMA node than the Opteron machine, so our mapping mechanism is able to decrease the number of remote memory accesses more effectively on the Xeon. Additionally, the NUMA factor of the Xeon machine is much greater than the NUMA factor of the Opteron machine, which makes the overhead generated by the remote memory accesses more evident on the Xeon machine. Since the worst mapping causes lots of remote memory accesses, it is expected that the worst mapping present worse results with the Xeon than the Opteron.

Considering thread mapping, there is only one level of the memory hierarchy to be exploited by thread mapping in the Opteron machine: the NUMA node. However, the Xeon machine also presents a shared cache memory on each processor. Therefore, it is expected the performance improvement of any mapping technique to be more similar on the Opteron machine than on the Xeon machine. About the two different metrics that we evaluated, the sharing matrices present some differences. However, our mapping algorithm based on graph matching generated similar thread affinities for both metrics. This implies that the performance results of both metrics are also expected to be similar.

5.3 Understanding Performance

In order to have an insight about the reason of the thread and data placement impact in the benchmarks performance, we have selected two benchmarks that presented clearly different results: EP and MG, both with input size B. These benchmarks are very distinct: EP is CPU-bound while MG is memory-bound. Considering these benchmarks, we have used the vTune tool to obtain access to some performance hardware counters while executing the original version of the benchmarks on the Xeon machine. The Xeon machine is used for the performance hardware counters because it is

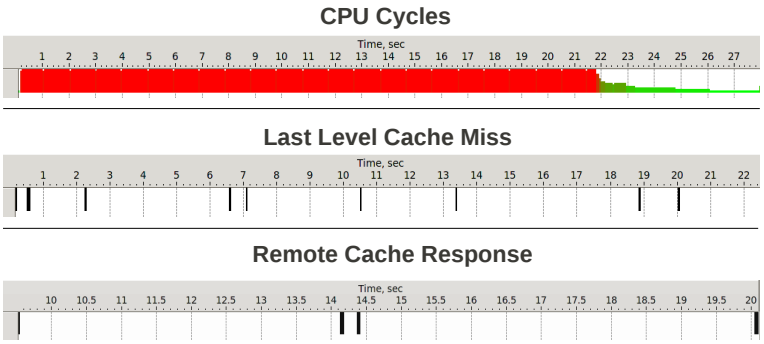


Figure 11: Event Counters of EP on Xeon.

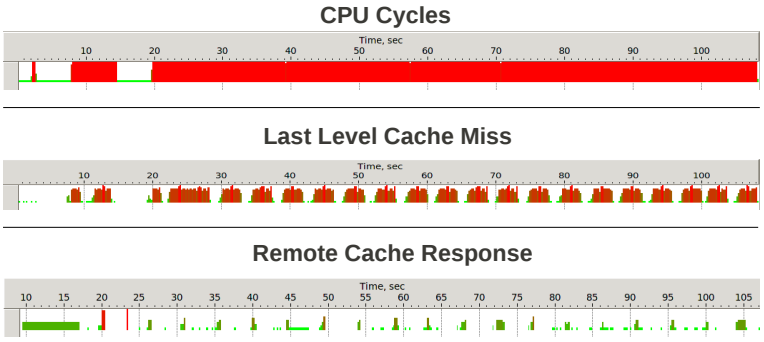


Figure 12: Event Counters of MG on Xeon.

the only one that supports the vTune software.

Figure 11 shows the CPU cycles, last level cache miss and remote cache response performance event counters for EP, whereas the Figure 12 shows the same event counters for MG. In these figures, the black color means few events, green some events and red several events. Since this machine has NUMA characteristics, it is important to investigate the ratio between remote accesses and CPU cycles on both benchmarks, to comprehend the importance of thread and data placement for the application. On this machine we have used vTune to extract information of accesses on local and remote last level caches. We can observe that the main difference between EP and MG is the number of cache accesses during the benchmark execution. EP has presented almost no access to the last level cache, while MG performs several accesses on the last level cache memory. Such results let us to conclude that MG is much more sensitive to memory placement than EP. Due to this, the performance improvements in MG are higher than in EP.

The proposed data and thread mapping method allows a better data locality for threads. Particularly, on applications that are memory bound the data mapping improves latency and bandwidth perceived by threads to get data. Due to this, threads take less cycles to access the needed data to compute their operations. Considering MG benchmark, we present in Table 3 the number of operations performed for each mapping strategy on MG benchmark. The highest performance is obtained with the best mapping using the access count metric. This is mainly due to the better data locality provided by the mapping, enhancing the cache and memory usage by threads. In Table 5.3, we can also observe that the results obtained with Linux are similar to the ones obtained with the worst mapping. Both strategies do not perform data placement and consequently, threads wait more to get the necessary data.

Table 3: Number of Operations Performed on MG for each Mapping in Mops/s (Millions of operations per second).

Mapping	Access Count	Amount of Memory
Linux	3506.98	3506.98
Worst	3479.91	3507.23
Best	11319.7	11700

6 Related Work

In order to reduce the necessary efforts to control thread and data mapping on parallel applications, transparent distribution of thread and data has been object of study of a number of researches [22, 18, 4, 6, 29, 34]. These studies have been designed at different levels and they can be grouped in: thread placement, data placement or a mix of both.

6.1 Thread and process mapping solutions

Static process mapping of MPI applications has been evaluated in [29], 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.

In [31], it is shown that hardware performance counters already present in current processors may be used to dynamically map parallel applications. They schedule threads by taking into account an indirect estimate of the sharing pattern based on stall cycles, cache miss counters and other hardware counters present in the Power5 processor. To decrease the overhead of the proposed mechanism, the mapping system is just enabled after the core stall cycles exceeds a given threshold. Performance was increased by up to 7%, and the number of memory accesses to remote cache memories were reduced by up to 70%. Our mechanism provides more reliable information to map the threads and data, while their mechanism indirectly estimates the communication pattern by relying on less accurate information about the sharing pattern. However, their method is more flexible and lightweight, because the mapping is performed by the operating system and do not require any profiling step.

A machine learning approach for OpenMP applications to map threads over the machine cores is presented in [34]. Using the machine learning approach, the proposed solution is capable of automatically predict the number of threads and the thread placement policy for an application. The thread affinity mechanism has several steps in order to find the best number of threads and the thread placement policy for an application. First, the mechanism has to train the machine learning model with some target applications using as input some characteristics of the application such as cache misses, loop iteration and branch miss rate. The output of the training step is a predictor for the best number of threads and scheduling strategy for a given application. The main limitation of this approach is that it can only predict correctly if the target applications have regular behaviors. If a novel application has to use this mechanism, it must have similar characteristics to the ones used in the training step. Otherwise, a new training must be performed to get the best thread placement for this application. In terms of flexibility, both this and our work present the same problem: the need of profiling steps. Their profiling present less overhead than ours, with the downside of relying on indirect information about the sharing pattern such as cache misses. In our mechanism, we are able to detect an accurate sharing pattern.

The Charm++ parallel system also provides an explicit and implicit support to thread mapping. In the case of explicit support, a command line option allows programmers to set the thread

placement for an application execution. The command line has to be used when launching the application, specifying which cores of the machine must be used to place the application threads [19]. In this explicit mechanism no abstraction of the machine is provided to the user, since the programmer has to explicitly specify the threads to cores mapping. The implicit support is provided by the use of a load balancer that schedules work over the machine cores. The load balancer in Charm++ can be employed as a plug-in, using the load balancing Charm++ framework to build them. Due to the simplicity of design and implementation, a number of load balancers have been proposed for Charm++. For instance, load balancers that consider constraints such as memory usage and threads communication [4, 12].

6.2 Data placement solutions

Data placement support for NUMA machines is now present in many operating systems, such as Linux, Windows and Solaris [15, 14]. *First-touch* is the default policy in Linux/Windows operating systems to manage data placement on NUMA. This policy places data on the node that first accesses it [14, 8]. To improve memory affinity using this policy, it is necessary to either execute a parallel initialization of all shared application data allocated by the master thread or allocate its data on each thread. However, this strategy will only present performance gains if it is applied on applications that have a regular data access pattern. In case of irregular applications, *first-touch* may result in a high number of remote accesses, since threads do not access the same data.

In [22], 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.

The work [18] presents a hardware-assisted page placement scheme based on automated profiling. The main objective of this solution is to reduce the execution time by placing memory pages closer to the most frequently requesting processor. The proposal relies on an automated profiling mechanism that extracts the application memory access patterns of both static and dynamic memory. Using such profiling information, some memory migration is performed to increase the number of local access. The proposed solution is implemented in the user space and it is independent of compiler, operating system and interconnection network but it relies on the machine providing the necessary hardware counters. This method has been evaluated using the NAS and SPEC OpenMP benchmarks and the results show that the mechanism can achieve performance improvements of up to 20%.

The recent integration of memory controllers inside processor chips has demanded a special attention when allocating data on machines based on such processors. Since memory controllers manage all access to physical memory, they can be a bottleneck and reduce the system performance. To deal with this problem, researchers have proposed in [1] two memory policies that can adapt to improve data locality. The memory policies make use of the hardware information (e.g. queuing delay of a memory controller, number of hops from a core to a memory controller) during the application execution to decide where to place a memory page. They have implemented the memory policies in the Virtutech Simics simulator and used some benchmarks to evaluate them. Results have shown gains of up to 35% on platforms with one memory controller per N number of processors and gains up to 5% on platforms with one memory controller per processor. The main disadvantage of this work is that it does not consider which threads are accessing the pages to choose where to migrate, leading to the increase of the remote memory accesses.

6.3 Thread and data placement solutions

Dynamic task and data placement for OpenMP applications have been proposed in [6]. In these works, researchers have proposed an NUMA-aware runtime for OpenMP, named ForestGOMP. It is a an extension of GNU OpenMP library that relies on the *hwloc* framework [7], on the *Marcel* threading library [10] and on the *BubbleSched* framework [33]. 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. Their proposal does not require profiling steps. However, differently from our mechanism, ForestGOMP requires some modifications to the source code to provide informations about the program behavior, such as how the data is distributed, which variables to consider, among others.

7 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 thread and data 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 based on graph theory 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 experiments on two NUMA 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. This happens because the original scheduler periodically migrates the threads, which increases the cache miss rate, as the working set of the migrated thread must be loaded in the cache of the destination core.

As future work, we intend to develop a smarter 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 under grant 4874-06-4 and CNPq.

References

[1] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 319–330, New York, NY, USA, 2010. ACM.

[2] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, Chi-Keung Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with pin. *Computer*, 43(3):34 –41, 2010.

[3] N. Barrow-Williams, C. Fensch, and S. Moore. A communication characterisation of splash-2 and parsec. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 86 –97, 2009.

[4] Abhinav Bhatelé, Laxmikant V. Kalé, and Sameer Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 110–116, New York, NY, USA, 2009. ACM.

[5] C. Bienia, S. Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47 –56, sept. 2008.

[6] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective. In *5th International Workshop on OpenMP*, pages 79–92, Dresden, Germany, 2009. Springer.

[7] Francois Broquedis, Jerome Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, pages 180–186, 2010.

[8] Alexandre Carissimi, Fabrice Dupros, Jean-Francois Mehaut, and Rafael Vanoni Polanczyk. Aspectos de Programação Paralela em arquiteturas NUMA. In *VIII Workshop em Sistemas Computacionais de Alto Desempenho*, 2007.

[9] Eduardo H.M. Cruz, Marco A.Z. Alves, and Philippe O.A. Navaux. Process mapping based on memory access traces. In *Computing Systems (WSCAD-SCC), 2010 11th Symposium on*, pages 72 –79, 2010.

[10] Vincent Danjean and Raymond Namyst. Controlling kernel scheduling from user space: An approach to enhancing applications reactivity to i/o events. In Timothy Mark Pinkston and Viktor K. Prasanna, editors, *High Performance Computing – HiPC 2003*, volume 2913 of *Lecture Notes in Computer Science*, pages 490–499. Springer Berlin, 2003.

[11] M. Diener, F.L. Madruga, E.L. Rodrigues, M.A.Z. Alves, J. Schneider, P.O.A. 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*, pages 491 –496, 2010.

[12] Isaac Dooley, Chao Mei, Jonathan Lifflander, and Laxmikant Kalé. A study of memory-aware scheduling in message driven parallel programs. In *Proceedings of 17th Annual International Conference on High Performance Computing*, 2010.

[13] Jerry Yan Haoqiang Jin, Michael Frumkin. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report 99-011/1999, NAS System Division - NASA Ames Research Center, 1999.

[14] Antony Joseph, Janes Pete, and Rendell Alistair. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *High Performance Computing - HiPC*, pages 338–352. 2006.

[15] Andi Kleen. A NUMA API for Linux. Technical Report Novell-4621437, 2005.

[16] V. Kolmogorov. Blossom V: A new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1(1):43–67, 2009.

[17] P.S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer Micro*, 35(2):50–58, Feb 2002.

[18] Jaydeep Marathe and Frank Mueller. Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–99, New York, NY, USA, 2006. ACM.

[19] Chao Mei, Gengbin Zheng, Filippo Gioachin, and Laxmikant V. Kalé. Optimizing a parallel runtime system for multicore clusters: a case study. In *TG '10: Proceedings of the 2010 TeraGrid Conference*, pages 1–8, New York, NY, USA, 2010. ACM.

[20] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, USA, 2009. IEEE.

[21] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[22] Dimitrios S. Nikolopoulos, Eduard Ayguadé, and Constantine D. Polychronopoulos. Runtime vs. manual data distribution for architecture-agnostic shared-memory programming models. *Int. J. Parallel Program.*, 30(4):225–255, 2002.

[23] C.N.K. Osiakwan and S.G. 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*, pages 880 –887, 9-13 1990.

[24] Christiane Pousa Ribeiro, Márcio Castro, Alexandre Carissimi, and Jean-Francóis Méhaut. Improving Memory Affinity of Geophysics Applications on NUMA platforms Using Minas. In *9th International Meeting High Performance Computing for Computational Science, VECPAR*, US, 2010. LNCS.

[25] Christiane Pousa Ribeiro, Márcio Castro, Luiz Gustavo Fernandes, Alexandre Carissimi, and Jean-Francóis Méhaut. Memory Affinity for Hierarchical Shared Memory Multiprocessors. In *21st International Symposium on Computer Architecture and High Performance Computing*, São Paulo, Brazil, 2009. IEEE.

[26] Christiane Pousa Ribeiro, Ismael Stangherlini Nicolas Maillard, and Jean-Francóis Méhaut. Compiling OpenMP Applications to Enhance Memory Affinity on Hierarchical Multi-Core Machines. In *23rd International Workshop on Languages and Compilers for Parallel Computing - Poster*, US, 2010. LNCS.

[27] Christiane Pousa Ribeiro, Márcio Castro, Alexandre Carissimi, and Jean-François Méhaut. Improving Memory Affinity of Geophysics Applications on NUMA platforms Using Minas. In *9th International Meeting High Performance Computing for Computational Science, VECPAR*, US, 2010. LNCS.

[28] Christiane Pousa Ribeiro, Maxime Martinasso, and Jean-François Méhaut. NUMA Support for the charm++ Environment. 2010.

[29] E.R. Rodrigues, F.L. Madruga, P.O.A. 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*, pages 811 –817, 5-8 2009.

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

[31] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41:47–58, March 2007.

[32] Christian Terboven, Dieter A. Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and Thread Affinity in OpenMP Programs. In *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*, pages 377–384. ACM, 2008.

[33] Samuel Thibault. *Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications.* PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, December 2007. 128 pages.

[34] Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 75–84, New York, NY, USA, 2009. ACM.