Dynamic Thread Pinning for Phase-Based OpenMP Programs

Abdelhafid Mazouz, Sid Touati, Denis Barthou

Dynamic Thread Pinning for Phase-Based OpenMP Programs

Abdelhafid MAZOUZ[1], Sid-Ahmed-Ali TOUATI[2], and Denis BARTHOU[3]

[1] University of Versailles Saint-Quentin-en-Yvelines, France
[2] University of Nice Sophia Antipolis, France
[3] University of Bordeaux, France

Abstract. Thread affinity has appeared as an important technique to improve the overall program performance and for better performance stability. However, if we consider a program with multiple phases, it is unlikely that a single thread affinity produces the best program performance for all these phases. If we consider the case of OpenMP, applications may have multiple parallel regions, each with a distinct inter-thread data sharing pattern. In this paper, we propose an approach that allows to change thread affinity dynamically (thread migrations) between parallel regions at runtime to account for these distinct inter-thread data sharing patterns. We demonstrate that as far as cache sharing is concerned for SPEC OMP01, not all the tested OpenMP applications exhibit a distinct phase behavior. However, we show that while fixing thread affinity for the whole execution may improve performance by up to 30%, allowing dynamic thread pinning may improve performance by up to 40%. Furthermore, we provide an analysis about the required conditions to improve the effectiveness of the approach.

Keywords: OpenMP, thread level parallelism, thread affinity, multicores.

1 Introduction

Multicore architectures are nowadays the state of the art in the industry of processor design for desktop and high performance computing. With this design, multiple threads can run simultaneously exploiting thread level parallelism and consequently, improve overall program performance of the system. Unfortunately, the growing gap between processor performance and memory performance has led manufacturers to propose highly hierarchical machines to alleviate this problem. The common architectural design consists of two or more cores sharing some levels of memory caches, memory buses, prefetchers or memory nodes. As the memory hierarchy of these machines is becoming increasingly complex, achieving better program performance of parallel applications on these modern architectures is more challenging.

A hierarchy of memory caches allows to exploit data sharing between threads running on such platforms. Of course, to exploit that, a multi-threaded application has to meet two conditions. First, threads have to access common data. Second, the reuse distance has to be short enough to effectively exploit these shared data across multiple threads. In this context, thread affinity has appeared as an important technique to exploit data sharing and to accelerate program execution times [15, 12, 6, 18, 9]. Another advantage of fixing thread affinity is for better performance stability [9].

Using thread affinity enhances inter-thread data locality. If two threads make extensive accesses to common data in memory, it is better to place them on cores sharing the same L2/L3 cache, or the same NUMA node. Doing so, we decrease the number of cache misses. Indeed, if one thread brings a data element to some cache level, the second thread accessing the same data element will avoid unnecessary cache misses. Furthermore, binding threads to cores by considering the machine architecture may help hardware prefetching of frequently accessed shared regions.

In this paper, we focus on cache sharing, and study the impact of a phase-based or dynamic thread pinning. It it based on the *control flow graph* (CFG) of a parallel execution in a given program. A *node* in this CFG can be defined using different granularities: a sequence of some instructions, a function call, etc. Since OpenMP programs may implement multiple parallel regions which are called multiple times iteratively, we consider the CFG as a graph representing a sequence of calls to distinct parallel regions in an OpenMP program. This also means that we define an OpenMP phase as the execution of a parallel OpenMP region. In this study, we consider that the parallel region represents a good trade-off between a better sharing patterns identification accuracy and a low overhead incurred by a smaller number of thread migrations.

A previous research study [9] has showed that fixing thread affinity during the whole execution provides better performance improvements on NUMA machines than on SMP ones. Nevertheless, we think that it is possible to further enhance the performance gain using thread affinity by exploiting phase-based behavior in OpenMP programs. We made an extensive performance evaluation of multiple thread pinning strategies on four distinct machines. Among them, four strategies are application dependent: they rely on the characteristics (data sharing) of the application, and they set a distinct thread placement for each parallel region. Three strategies are application independent: they apply the same thread affinity for the whole execution and for each application. We show that dynamic thread pinning can improve performance by up to 40% compared to the Linux OS scheduler. Furthermore, we show the amount of inter-thread data sharing and the granularity of the parallel regions are main factors influencing most the effectiveness of dynamic thread pinning.

This article is composed as follows. Section 2 a synthetic example aiming to show the effectiveness of using per-parallel regions thread affinity within OpenMP programs. Section 3 presents the method we use to compute a distinct thread affinity for each parallel region. Section 4 describes our experimental setup and methodology (test machines, running methodology, statistical significance analysis). Section 5 shows our experimental results and analysis. Related work is presented in Sect. 6, then we conclude.

2 Motivation and Problem Description

We define an OpenMP phase as a unique and distinct OpenMP parallel region. In OpenMP, each structured code started by the construct `#pragma omp parallel` in C/C++ or `!$omp parallel` in Fortran is a new parallel region. That is, each OpenMP parallel region translates into distinct OpenMP phases.

To illustrate the benefit of changing thread pinning between consecutive OpenMP parallel regions, we use a synthetic micro-benchmark. This benchmark implements two OpenMP parallel regions, each with a distinct sharing pattern. The benchmark uses a

single large rectangular (the width is much greater than the height) matrix which is subdivided into equal parts among all the intervening threads. The benchmark is designed so that in parallel region 1, data sharing is between (T_1, T_5), (T_2, T_6), (T_3, T_7) and (T_4, T_8) thread pairs. Similarly, in parallel region 2, data sharing is between (T_1, T_2), (T_3, T_4), (T_5, T_6) and (T_7, T_8) thread pairs. Cache lines sharing between threads is implemented by allowing for each pair of threads to access common cells from the portion of the array that has been assigned to them. For each assigned portion from the array, each thread performs simple computations like additions and multiplications.

Each thread accesses to the same amount of data. Moreover, the amount of shared data blocks is equal between each pair of threads, of course with different sharing patterns across the two parallel regions. To analyze how the amount of inter-thread data sharing can influence the effectiveness of allowing thread migrations across parallel regions, we fix the same inter-thread data sharing in the first parallel region, and vary the amount of data sharing in the second. We consider in these experiments the 0%, 25%, 75% and 100% amounts of data sharing cases in the second parallel region.

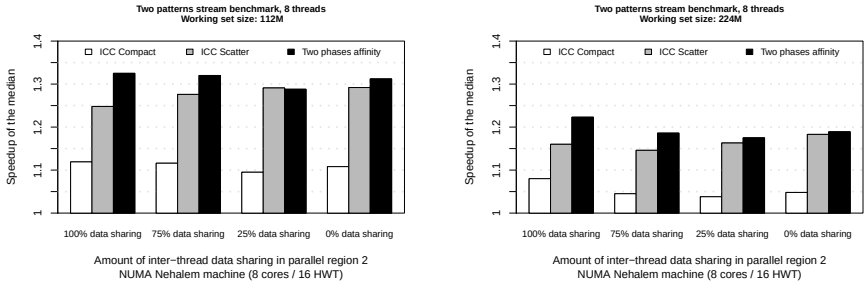


Fig. 1: Speedup of the median of the tested thread affinities for the synthetic benchmark using multiple matrix sizes and running with 8 threads on the `Nehalem` machine.

We run the micro-benchmark multiple times and using multiple thread pinnings on top of an 8 cores Intel `Nehalem` machine[4]. Figure 1 shows the obtained speedups. We consider the `no affinity` strategy as the base comparison configuration. Speedups are reported according to the amount of data sharing in the second parallel region (four configurations) and the tested thread affinities. Reading from left to right, the first group represents the case of 100% data sharing, the second represents the case of 75% data sharing, the third represents the case of 25% data sharing and the last group represents the case of 0% data sharing.

First, it is clear that using a per parallel region thread affinity helps to improve performance compared to application-wide thread pinning strategies. Second, we observe that as the amount of data sharing in the second parallel region is reduced, the performance of the `icc scatter` and the per parallel region thread affinity strategies are close. This is due to two reasons: 1) since these two strategies are able to exploit the data sharing of the first parallel region, the obtained performance for that parallel region is similar, and 2) when there is no sharing at the second parallel region, the precise thread pinning is not important, so we obtain these observed program performance. From this

[4] More details about the machine can be found in Sect. 4

simple experiment, we can conclude that changing the affinity between OpenMP parallel phases is beneficial and can lead to non negligible performance improvement over a fixed affinity for the whole program or a `no affinity` strategy.

3 Parallel OpenMP Phases Extraction and Thread Pinning

We focus on data sharing to compute effective thread pinnings. Instead of computing an *application-wide* thread affinity (apply the same affinity for the whole execution), we compute a thread pinning for each distinct parallel region in the OpenMP program. We use a profile guided method which consists of multiples steps that we detail below.

In OpenMP programs, computing a thread affinity for each parallel region requires to detect the entry and exit events of that region. For this purpose, we use the `OPARI` [10] instrumentation tool. The objective of `OPARI` is to provide a performance and measurement interface for OpenMP. It is a source-to-source translation tool which automatically adds function calls to a `POMP` runtime measurement library. This library is used to collect runtime performance data for OpenMP applications. `OPARI` supports C/C++ and Fortran programming languages. The idea behind the concept is to detect each OpenMP pragma/directive and add function calls to the `POMP` library. This method allows us to be compiler and runtime independent. In our approach, we do not use the `POMP` library for performance measurement. Instead, we have made changes in order to achieve dynamic thread pinning for each parallel region.

After the `OPARI` instrumentation, we make a memory tracing of OpenMP applications using the `PIN` [8] binary instrumentation framework. We fix a number of threads per application, and we collect for every thread and for each distinct parallel region (PR) all the accesses to all memory addresses (which are transformed to accesses to memory cache lines). In addition, we are able to deduce the parallel regions control flow graph `PRCFG`. It is a directed valued graph where the vertices represent the distinct PRs of the program and the edges represent the predecessor and the successor relationship between them. As reported before, an edge between a PR_i and PR_j is valued by the number of times the execution of the PR_i is followed by the execution of PR_j.

The collected memory trace profile is used to build an *affinity graph* for each parallel region in the program. Each *affinity graph* in the application is an undirected valued graph $G_p = (\mathcal{T}, \mathcal{E}, \alpha) \quad \forall p \in \mathcal{P}$. $\mathcal{T}$ is the set of application threads, $\mathcal{E} = \mathcal{T} \times \mathcal{T}$, $\alpha : \mathcal{E} \mapsto \mathbb{N}$ is a gain function applied to every pair of threads and $\mathcal{P}$ is the set of parallel regions implemented in the application.

The gain function $\alpha(T_i, T_j)$ models the attraction factor between two threads. Since we rely on data sharing between threads to compute an affinity graph, the gain function represents the number of common accesses to common memory caches lines accessed by both the T_i and T_j threads for a given parallel region. Let us precisely define α for an application with a fixed number of threads $n = \|\mathcal{T}\|$ and for a given parallel region $p \in \mathcal{P}$. The collected memory trace profile contains the information $A_p(T_i, b)$ which is the number of accesses of thread T_i to data block b at parallel region p. If we consider $B^p_{i,j}$ as the set of all data blocks accessed by the pair of threads (T_i, T_j) at parallel region p, then we can compute $\alpha(T_i, T_j)$ using Equation 1. We call this method the simple model or SM.

$$\alpha_p(T_i, T_j) = \sum_{b \in B^p_{i,j}} \min\left(A_p(T_i, b), A_p(T_j, b)\right) \tag{1}$$

We define another method to compute α. We call it the read/write model or RWM. We added this method because we consider that from the performance perspective, it is important to separate read and write accesses. The reason for that is that we consider a shared region of data wherein accesses are dominated by reads will have less impact on performance than a shared region of data wherein the read and write accesses are balanced. In fact, when the shared data are accessed only in a read mode, duplicating these data on multiple caches may not harm the performance in a great extent. Since we distinguish between reads and writes, then we exactly have $RD_p(T_i, b)$ and $WR_p(T_i, b)$ which is the number of reads and writes respectively performed by thread T_i to data block b and where $A_p(T_i, b) = RD_p(T_i, b) + WR_p(T_i, b)$. Given these constraints, Equation 2 defines the function $\alpha(T_i, T_j)$ for the read/write model.

$$\begin{aligned} \alpha_p(T_i, T_j) = \sum_{b \in B^p_{i,j}} \Big(& \min\left(RD_p(T_i, b), WR_p(T_j, b)\right) \; + \\ & \min\left(WR_p(T_i, b), RD_p(T_j, b)\right) \; + \; \min\left(WR_p(T_i, b), WR_p(T_j, b)\right) \Big) \end{aligned} \tag{2}$$

Once all the affinity graphs are constructed for an application and for a given number of threads, we can use them to investigate multiple thread pinning strategies. The idea is based on graph partitioning methods [5]. The affinity graphs must be decomposed into disjoint subsets, named a partition. A partition $V = \{V_1, V_2, \cdots, V_k\}$ has the property that $\bigcup_{1 \leq l \leq k} V_l = \mathcal{T}$ and $V_l \cap V_m = \emptyset$, where $l \neq m$ and $l, m \in [1, k]$. Every subset $V_l \in V$ contains a set of nodes representing threads that have to be placed on adjacent cores sharing the same cache level (L2 or L3, depending on the target machine). If we have k shared caches on the system, then we compute a partition with k subsets [5]. The global objective function is to maximize $\sum_{(T_i, T_j) \in V_l \times V_l} \alpha(T_i, T_j)$ the sum of the gains between threads belonging to the same partition. This optimization problem is a classical NP-complete problem, so we have to use heuristics such as in [5]. Fortunately, we have a special polynomial case. Indeed, if we are given a machine architecture where a cache level is shared between *two* adjacent cores, then the problem becomes to seek for partitions with a size equal to 2 ($\|V\| = 2$). It is easy to see that in the case of seeking partitions of size 2 the problem is equivalent to computing a set of thread pairs sharing a common cache while maximizing a global gain. In this special case, the optimization problem can be solved with a simpler maximum-weight matching in general graphs [2]. Precisely, it can be polynomially and optimally solved thanks to the algorithm of Edmonds in $O(\|\mathcal{T}\|^2.\|\mathcal{E}\|)$ [2].

On a parallel machine with a memory hierarchy, the graph partitioning problem can be applied to reflect data reuse at each level of shared caches. We define two application dependent thread pinning strategies, corresponding to the application of heuristics for solving the graph k-partitioning problems:

1. `LPGP` strategy. After an initial step of optimal computation of thread pairs, we proceed by a graph k-partitioning [5]. It is a hierarchical strategy, where threads are

first paired and pinned on shared L2 or L3 cache then thread pairs are partitioned and placed on the different sockets according to their affinity.
2. `GPLP` strategy. It is a hierarchical strategy. It starts by an initial graph k-partitioning to fix threads on sockets, then perform an optimal polynomial algorithm to compute thread pairs sharing L2 or L3 cache levels.

In addition to the application dependent strategies presented above, we consider in our evaluation, the following application independent strategies:

1. `No affinity`. This strategy lets the OS decide about thread placement. This strategy allows thread migration between cores during application execution.
2. `icc compact`. This strategy assigns successive OpenMP threads to cores as close as possible in the topology map of the platform.
3. `icc scatter`. This strategy distributes OpenMP threads as evenly as possible across the entire sockets (one thread per socket if possible).

4 Experimental Setup and Methodology

Our experiments have been conducted using all SPEC OMP01 [14]. We used the `ref` data input with SPEC OMP01[5] whether for memory tracing or for performance evaluation. We tested multiple numbers of threads for every application according to the available number of cores. We tested various thread placement strategies for every application, thread number, input data set. For statistical significance, each measure was repeated 31 times and special care has been taken to limit any external interference on performance measures. The benchmarks have been compiled using Intel compiler (`icc` 11.1) with flag `-O3 -openmp`. To set a per parallel region thread affinity, we focus only on *hot parallel regions* which dominate the total execution time. This methodology helps to avoid setting a thread affinity for infrequently called or too short parallel regions, thus lowering the number of unnecessary thread migrations. No more than one application was executed at a time. The execution of each benchmark was repeated 31 times for each software configuration and machine. This high number of runs allows us to report statistics with a high confidence level [11, 17]. The dynamic voltage scaling was disabled to avoid core frequency variation. Depending on the test machine, we run each benchmark with 8, 16 or 32 threads. When we plot speedups, only statistically significant ones are reported.

We conducted all our experiments on four platforms:

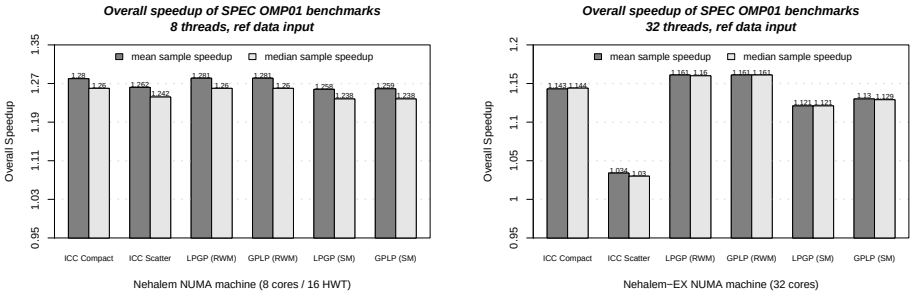
1. The `Nehalem` (8 cores) machine. It is an Intel `NUMA` machine with 2 processors. Each processor (`Nehalem` micro-architecture) has 4 cores (2 hardware threads per core) sharing an inclusive L3 cache of 8 MB. The core frequency is 2.93 GHz.
2. The `Nehalem-EX` (32 cores) machine. It is an Intel `NUMA` machine with 4 processors. Each processor (`Nehalem` micro-architecture) has 8 cores sharing an inclusive L3 cache of 18 MB. The core frequency is 2.0 GHz.
3. The `Shanghai` (8 cores) machine. It is an AMD `NUMA` machine with 2 `Opteron` processors. Each processor (`K10` micro-architecture) has 4 cores sharing an exclusive L3 cache of 6 MB. The core frequency is 2.4 GHz.

[5] We also tested NAS Parallel Benchmarks (NPB) [3]. For lack of space, we limit our analysis to OMP01.

4. The `Barcelona` (16 cores) machine. It is an AMD `NUMA` machine with 4 `Opteron` processors. Each processor (`K10` micro-architecture) has 4 cores sharing an exclusive L3 cache of 2 MB. The core frequency is 1.9 GHz.

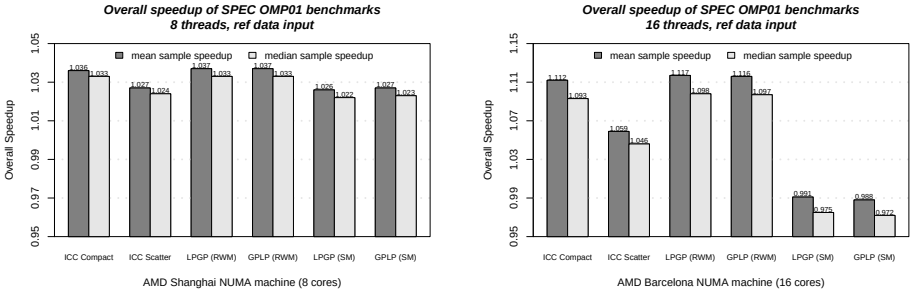
5 Experimental Evaluation of Phase-Based Thread Pinning

This section presents a performance evaluation and analysis about the effectiveness of the per parallel region thread affinity strategy for SPEC OMP01 benchmarks. We used four NUMA machines: `Nehalem`, `Nehalem-Ex`, `Shanghai` and `Barcelona`. Each benchmark has been executed with 8, 16 and 32 threads with respect to the maximal number of physical cores. We report the obtained speedups using the `icc compact`, `icc scatter`, `LPGP(RWM)`, `GPLP(RWM)`, `LPGP(SM)` and `GPLP(SM)` strategies compared to the default `no affinity`. Figures 2 and 3 show the overall sample speedups of every tested thread pinning strategy on the `Nehalem`, `Nehalem-EX`, `Shanghai` and `Barcelona` NUMA machines using bar plots. We report the speedups of the average and median execution times of all SPEC OMP01 applications running with 8, 16 and 32 threads.



(a) (b)

Fig. 2: Overall sample speedups of the tested thread affinities with SPEC OMP01 benchmarks running on the Intel `Nehalem` and `Nehalem-EX` NUMA machines.



(a) (b)

Fig. 3: Overall sample speedups of the tested thread affinities with SPEC OMP2001 benchmarks running on the AMD `Shanghai` and `Barcelona` NUMA machines.

On the Barcelona and Nehalem-EX machines, running OMP01 with 16 and 32 threads respectively with thread affinity enabled leads to non-negligible speedups and slowdowns (Barcelona). On the Shanghai machine, fixing thread affinity for OMP01 running with 8 threads leads to marginal speedups. On Nehalem, when running OMP01 with 8 threads, we observe non-negligible speedups for all the tested strategies. On Nehalem, the experiments were performed with HT enabled. This option increases the number of OS scheduling possibilities. Moreover, the number of involuntary thread migrations is increased which lead to the observed poor performance.

Even if the difference in terms of speedups is not significant, we observe that the LPGP(SM) and the GPLP(SM) perform slightly worse than LPGP(RWM) and GPLP(RWM) strategies. As a reminder, the SM strategies are computed from affinity graphs that do not consider the read/write model. Regarding the test machines, we do not observe any important difference, in terms of speedups, between the tested thread affinity strategies. This situation may suggest that there in no benefit of enabling a per parallel region thread affinity. Moreover, it is possible to conclude that this approach is not effective for SPEC OMP01 benchmarks.

As noticed earlier, from our experiments, we made two main observations which are highly related. First, the relative poor performance of strategies computed upon a model which does not consider the RWM. The observed overall sample speedup of the median for that strategies is in the range $[0.972 - 1.23]$. On the other hand, strategies that do consider the RWM have an overall sample speedup of the median in the range $[1.033 - 1.28]$. Second, the non clear benefit of using per parallel region thread affinity. If we compare the best overall sample speedup of the median obtained by application independent and application dependent strategies for each tested configuration (tested machine, number of threads), we observe that while application independent strategies have speedups in the range $[1.033 - 1.26]$, application dependent strategies have speedups in the range $[1.022 - 1.261]$. Indeed, regarding these speedups, it is possible to conclude that per parallel region affinity is not effective compared to application-wide (application independent) strategies.

To understand the presented experimental results, we first show in Table 1 the total number of times for which the computed per parallel region thread affinity (the LPGP and GPLP strategies computed using whether an aware or an unaware read/write model) consists of an application-wide thread affinity. This means, that the computed thread affinity is exactly the same for all the parallel regions in the program (Tables 2 reports the number of parallel regions in all the tested benchmarks), or at least for the detected most time consuming parallel regions. From Table 1, we can observe that using LPGP(RWM) and GPLP(RWM) strategies, at least 80% (on the Nehalem-EX, apsi and equake benchmarks run with a per parallel region thread affinity) of the benchmarks were executed with a single (application-wide) thread affinity. Moreover, the computed single application-wide strategies are similar to icc compact. On the other hand, the LPGP(SM) and GPLP(SM) strategies do not seem to reflect the same behavior. Indeed, for the SM strategies, we can observe that almost all the computed per parallel region thread affinity strategies have a thread affinity computed for at least two parallel regions.

In the light of the previous observations, we can say as a first conclusion, that thread affinity strategies computed from affinity graphs that do consider the RWM better cap-

#Threads	Machine	LPGP (RWM)	GPLP (RWM)	LPGP (SM)	GPLP (SM)
8	Nehalem	10/10	10/10	3/10	3/10
8	Shanghai	10/10	10/10	3/10	3/10
16	Barcelona	10/10	10/10	4/10	5/10
32	Nehalem-EX	8/10	9/10	4/10	4/10

Table 1: Number of benchmarks where the computed per-parallel region thread affinity consists of setting a single-global-wide thread affinity. Each benchmark is executed using 8 16 and 32 threads on the `Nehalem`, `Nehalem-EX`, `Shanghai` and `Barcelona` machines.

Benchmarks	#Parallel regions	#Iterations
wupwise	10	402
swim	8	1198
mgrid	12	18250
applu	22	50
galgel	32	117
equake	11	3334
apsi	24	50
fma3d	30	522
art	4	1
ammp	10	202

Table 2: Number of parallel regions in SPEC OMP01 benchmarks running with the `ref` data input. For each benchmark, the number of iterations of the first hot parallel region is reported.

Parallel region	Median execution time (seconds)			
	icc compact	icc scatter	LPGP (SM)	GPLP (SM)
PR 1	0.074634	0.074699	0.074282	0.07463
PR 2	0.069234	0.068776	0.068575	0.068885
PR 3	0.103967	0.103331	0.103097	0.103642
PR 4	18.08301	18.08452	19.33687	18.59277
PR 5	20.71075	20.72176	20.7149	23.32227
PR 6	4.972871	4.984532	4.974599	4.972682
PR 7	0.019681	0.019674	0.019683	0.030159
PR 8	23.53141	23.52575	43.52616	34.09414
Remote memory accesses ratio	3%	3%	55%	68%

Table 3: Observed median execution times at each parallel region of the `swim` benchmark on the `Nehalem-EX` machine.

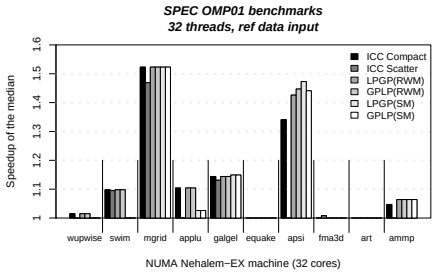


Fig. 4: The observed median speedups on the `Nehalem-EX` machine. Only statistically significant speedups are reported.

ture the sharing behavior of threads at the parallel region level than strategies that do not consider the RWM. Consequently, thread pinnings computed with the later strategies are likely to compute misleading thread affinity strategies which may hurt overall performance. Moreover, since almost all the per parallel region thread affinity strategies computed with a RWM tend to be application-wide strategies, explains why we observe that the performance of the RWM strategies is close to the performance of strategies like `icc compact` or `icc scatter`. Consequently, this observation suggests that SPEC OMP01 applications do not exhibit distinct phase behavior.

Unlike all the majority of the tested benchmarks, the `apsi` application does exhibit distinct inter-thread sharing patterns across parallel regions (more than 30% of data sharing (`ammp` has also more than 40% of shared accesses, but in a single region.). Figure 4 reports the statistically observed median speedups on the `Nehalem-EX` machine for OMP01 running with 32 threads. We can observe for the `apsi` benchmark that while application-wide strategies achieve up to 30% performance improvement compared to the Linux `no affinity` strategy, per parallel region thread affinity strategies achieve up to 45% performance improvement.

Now, we have to understand why strategies that exhibit distinct thread pinnings for distinct parallel regions are less effective compared to application-wide strategies for the tested applications. There are mainly two reasons for this performance behavior. First, the poor inter-thread data sharing exhibited by the distinct parallel regions for the tested benchmarks. Thus, applying a dynamic thread affinity technique on OMP01

benchmarks is not effective. Unfortunately, this is true because of: 1) the uniform distribution of the working set between running threads and 2) the presence of non-uniform data sharing patterns is rare. Second, the ratio between the number of times each parallel region is called, and the elapsed execution time in a single iteration of a given parallel region is very low (as noticed before in Tables 2). This means that threads are frequently migrated across too short parallel regions. Consequently, the small granularity of the selected *hot* parallel regions leads to lower the benefit from that migrations. Moreover, the small amount of inter-thread data sharing can exacerbate in a non-negligible extent the performance degradation due to NUMA effects: unnecessary remote memory accesses.

To illustrate the influence of poor inter-thread data sharing and unnecessary thread migrations on the overall performance, we report in Table 3 the observed execution times at each parallel region of the `swim` benchmark running with 32 threads on the `Nehalem-EX` machine (we do not report execution times for the `LPGP(RWM)` and `GPLP(RWM)) because these strategies compute a thread pinning similar to icc compact`. Even if the `LPGP(SM)` and `GPLP(SM)` compute a phase-based thread pinning strategy (for parallel regions 4,5 and 8), we observe that they behave poorly compared to `icc compact` or `icc scatter`. Moreover, while the later strategies exhibit at most 3% of remote memory accesses, the former exhibit more than 50% of remote memory accesses. If we run `swim` with an application-wide strategy by considering only parallel region 8 or 5, then we observed that the obtained performance is similar to `icc compact`. In fact, this benchmark does not exhibit an important amount of inter-thread data sharing (less than 1%), an exact thread affinity is not important. Consequently, applying a phase-based technique on this benchmark leads to frequent thread migrations impacting negatively the locality of data (NUMA accesses), thus the observed poor program performance.

6 Related Work and Discussion

Most of the thread affinity studies on multicores focus on data locality and cache sharing in parallel applications. Zhang *et al.* [18] conducted a measurement analysis to study the influence of CMP cache sharing on multi-threaded performance applications using the PARSEC [1] benchmarks. Through measurement they suggest that cache sharing has very limited influence on the performance of the PARSEC applications. However, they do not conclude that cache sharing has no potential to be explored for multi-threaded programs. Tam *et al.* [15] proposed threads clustering to schedule threads based on data sharing patterns detected on-line using hardware performance monitoring units. The mechanism relies on cross-chip communication performance impact.

Klug *et al.* [6] proposed `autopin`, a framework to automatically determine at runtime the thread pinning best suited for an application based on hardware performance counters information. The work is achieved by evaluating the performance of a set of different scheduling affinities and select the best one. The tool requires that the user provides an initial set of good thread placements. Terboven *et al.* [16] examined the programming possibilities to improve memory pages and thread affinity in OpenMP applications running on ccNUMA architectures. They provided a performance analysis of some HPC codes which may suffer from ccNUMA architectures effects.

Song *et al.* [13] proposed an affinity approach to compute application-wide thread affinity strategies. It relies upon binary instrumentation and memory trace analysis to find memory sharing relationships between user-level threads. Like us, they build an affinity graph to model the data locality relationship. Then, they use hierarchical graph partitioning to compute optimized thread placements. While their affinity graph is based on the number of addresses shared among threads, our affinity graphs are built upon the number of accesses to common cache lines reflecting real cache activity.

Some studies have addressed the data cache sharing at the compiler level. They focused on improving the data locality in multicores based on the architecture topology. Lee *et al.* [7] proposed a framework to automatically adjust the number of threads in an application to optimize system efficiency. The work assumes a uniform distribution of the data between threads. Kandemir *et al.* [4] discussed a compiler directed code restructuring scheme for enhancing locality of shared data in multicores. The scheme distributes the iterations of a loop to be executed in parallel across the cores of an on-chip cache hierarchy target.

Our work differs from the last efforts in two main points. First, we focus on the study of the impact on performance of dynamic thread pinning to exploit the inter-thread data sharing. Moreover, unlike other studies, we perform a statistical performance evaluation (running multiple times, we fix the experimental setup, data analysis through a rigorous statistical protocol [17]), we experiment multiple thread placement strategies and multiple machine architectures. Second, when it comes to compute a scheduling affinity, we rely on a profile-guided method. Using dynamic binary instrumentation, we fully analyze optimized binaries regardless of the compiler. Furthermore, we believe, that extracting all data dependencies and data sharing at compile time may not be sufficient, because these information depend on the working set which is known only at runtime.

7 Conclusion

We have presented an approach to exploit phase-based behavior in OpenMP programs using thread affinity. The presented technique relies on the *control flow graph* of the parallel OpenMP regions. The *control flow graph* gives for each parallel region its predecessor and successor in the execution flow. In other words, it is the graph representing the execution flow of distinct parallel regions. We have extended an existing tool to instrument the OpenMP constructs. Using a binary instrumentation tool, we build an *affinity graph* for each parallel region in the program. After that, we compute multiple thread pinning strategies for each parallel region.

References

1. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: Characterization and architectural implications. In: Proc. of the International Conference on Parallel Architectures and Compilation Techniques. PACT '08 (October 2008)
2. Edmonds, J.: Maximum matching and a polyhedron with 0-1 vertices. Journal Res. Nat. 69-B(1-22), 125–130 (1965)
3. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Tech. rep., NASA Ames Research Center (Oct 1999), http://www.nas.nasa.gov/Resources/Software/npb.html

4. Kandemir, M., Yemliha, T., Muralidhara, S., Srikantaiah, S., Irwin, M.J., Zhnag, Y.: Cache topology aware computation mapping for multicores. SIGPLAN Not. 45(6), 74–85 (2010)
5. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing 48, 96–129 (January 1998), http://dx.doi.org/10.1006/jpdc.1997.1404
6. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: autopin — automated optimization of thread-to-core pinning on multicore systems. Transactions on High-Performance Embedded Architectures and Compilers (2008)
7. Lee, J., Wu, H., Ravichandran, M., Clark, N.: Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In: Proc. of the annual international symposium on Computer architecture. pp. 270–279. ISCA '10, ACM, New York, NY, USA (2010)
8. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. of the ACM SIGPLAN conference on Programming language design and implementation. pp. 190–200. PLDI '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1065010.1065034
9. Mazouz, A., Touati, S.A.A., Barthou, D.: Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In: Proc. of IEEE International Conference on High Performance Computing & Simulation. pp. 273–279. HPCS '11, IEEE, Istanbul, Turkey (jul 4-8 2011)
10. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for openmp. The Journal of Supercomputing 23, 105–128 (August 2002), http://portal.acm.org/citation.cfm?id=603339.603347
11. Raj Jain: The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modelling. John Wiley and Sons (1991)
12. Song, F., Moore, S., Dongarra, J.: Feedback-directed thread scheduling with memory considerations. In: Proc. of the international symposium on High performance distributed computing. pp. 97–106. HPDC '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1272366.1272380
13. Song, F., Moore, S., Dongarra, J.: Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In: Proc. of the IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA. IEEE (2009)
14. Standard Performance Evaluation Corporation: SPEC CPU (2006), http://www.spec.org/
15. Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In: Proc. of theACM SIGOPS/EuroSys European Conference on Computer Systems 2007. pp. 47–58. EuroSys '07, ACM, New York, NY, USA (2007)
16. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in OpenMP programs. In: Proc. of the workshop on Memory access on future processors. pp. 377–384. MAW '08, ACM, New York, NY, USA (2008)
17. Touati, S.A.A., Worms, J., Briais, S.: The Speedup-Test: A Statistical Methodology for Program Speedup Analysis and Computation. To appear in the Journal of Concurrency and Computation: Practice and Experience (2012), http://hal.inria.fr/hal-00764454
18. Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In: Proc. of the ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 203–212. PPoPP '10, ACM, New York, NY, USA (2010)