Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of SPEC OMP applications on intel architectures

Abdelhafid Mazouz, Sid Touati, Denis Barthou

Performance Evaluation and Analysis of Thread Pinning Strategies on Multi-Core Platforms: Case Study of SPEC OMP Applications on Intel Architectures

Abdelhafid Mazouz
Univ. of Versailles St-Quentin en Yvelines
Abdelhafid.Mazouz@prism.uvsq.fr

Sid-Ahmed-Ali Touati
Univ. of Versailles St-Quentin en Yvelines
Sid.Touati@uvsq.fr

Denis Barthou
Univ. of Bordeaux
Denis.Barthou@labri.fr

ABSTRACT

With the introduction of multi-core processors, thread affinity has quickly appeared to be one of the most important factors to accelerate program execution times. The current article presents a complete experimental study on the performance of various thread pinning strategies. We investigate four application independent thread pinning strategies and five application sensitive ones based on cache sharing. We made extensive performance evaluation on three different multi-core machines reflecting three usual utilisation: workstation machine, server machine and high performance machine. In overall, we show that fixing thread affinities (whatever the tested strategy) is a better choice for improving program performance on HPC `ccNUMA` *machines compared to OS-based thread placement. This means that the current Linux OS scheduling strategy is not necessarily the best choice in terms of performance on* `ccNUMA` *machines, even if it is a good choice in terms of cores usage ratio and work balancing. On smaller* `Core2` *and* `Nehalem` *machines, we show that the benefit of thread pinning is not satisfactory in terms of speedups versus OS-based scheduling, but the performance stability is much better.*

KEYWORDS: OpenMP, Thread Level Parallelism, Thread Affinity, Operating Systems, Multi-Cores.

1. INTRODUCTION

Parallel programming is an old and still active research topic for high performance computing. Multiple parallel programming paradigms exist: dataflow, synchronous programming, shared memory model (pthread, OpenMP), distributed memory model (MPI, PVM), etc. In this article, we focus on OpenMP applications running on a shared memory machine.

The introduction of multi-core processors did not fundamentally change parallel computing paradigms. Any parallel application can be run safely on multi-core processor as if it is run on a classical multi-processor machine. The operating system (OS) considers every core as a distinct processor. If we have a processor with, say 8 cores, the OS sees 8 homogeneous processors that are capable of executing concurrent threads, processes or jobs. However, in terms of performance tuning, we cannot consider the cores as homogeneous because they share common micro-architectural resources: L2 or L3 shared caches, shared memory buses, etc. Consequently, the placement of threads on the cores, called *thread pinning*, is of high importance. For instance, if two threads make extensive accesses to common data in memory, it is better to place them on adjacent cores sharing the same L2 or L3 cache, or the same NUMA node. Data locality and reuse are not the unique performance factors that influence the program execution times in case of concurrent applications. Other factors may play influence: memory bus bandwidth, non uniform memory access (NUMA) effects, OS (synchronisation costs, Input/Output, thread scheduling), etc. Our article focuses the study on cache sharing only.

In [1] we did multiple runs of various parallel applications. We studied performance variations when we execute an OpenMP application multiple times (with the same data input) in a batch mode. We demonstrated the following conclusions:

- Work balancing is satisfactory because threads are scheduled and placed to optimise the usage of all cores.
- However, in terms of performance, the execution times exhibit high variations. For every new run of the parallel application (with the same data input), the OS may decide for a different thread placement that has an important impact on performance. In addition, threads

may be migrated from one core to another to improve work balancing and core utilisation ratio.

- When thread affinity is fixed, performance variations are greatly reduced, but are still present in few cases.

Work balancing is a classical performance criteria in parallelism and task scheduling, targeted by OS, distributed systems, grid computing, etc. However, its direct impact on code performance is not guaranteed. The reason is that work balancing aims to keep all cores busy. While all cores are busy (executing threads), the performance may still be poor because of bad harmony between the code and the micro-architecture: the OS may see that all cores are busy while threads are spending most of their times servicing cache misses, doing pipeline stalls and branch mispredictions. Consequently, work balancing may improve synchronisation costs by making all threads to reach the synchronisation barrier conjointly, but with poor performance.

Thread affinity has quickly appeared to be one of the most important factors to accelerate program execution times on multi-core processors. Still now, it is not clear how to decide for the best thread placement that considers all the possible performance factors (data locality, memory bus bandwidth, OS synchronisation overhead, NUMA effects, etc.). This article makes an exhaustive empirical study of nine thread placement strategies on three distinct machines. Among them, four strategies are application independent: they apply the same thread placement decision whatever the application. Five strategies are dependent on the application: they fix the thread placement after a profile-guided analysis. For popularity in the HPC community and availability for our work, the three test machines are based on Intel X86 (64 bits) with three distinct designs (to be precised later). Other multi-core architectures may be tested with exactly the same methodology.

This article is composed as follows. Sect. II describes our experimental setup and methodology (test machines, running methodology, statistical significance analysis). Sect. III defines all the thread pinning strategies tested in this study. Sect. IV shows a synthesis of the experimental results and analysis. Related work and discussion are presented in Sect. V, then we conclude

2. EXPERIMENTAL SETUP AND METHODOLOGY

Our experiments have been conducted using all the SPEC OMP benchmarks, executed on three distinct machines, using both `ref` and `train` data inputs. 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 35 times and special care has been taken to limit any external interference on performance measures. Our experiments lasted more than two full years to be prepared, accomplished, collected and analysed carefully. Below we describe the detailed experimental setup and methodology.

2.1. Test Machines Description

1. **Desktop workstation (8 cores)** that we name the *Core2 machine*. It is a single SMP machine with two `Clovertown` sockets (Intel Xeon E5345 with the `Core2` micro-architecture). Each processor has 4 cores, each pair of cores have a shared L2 cache. The platform has two L2 4MB caches per socket, for both instructions and data. The core frequency is 2.33 GHz. Each core has a separate 32KB L1 data and instruction caches. The main memory size is 4 GB RAM. The front-side bus has a clock rate of 1.33 GHz.

2. **Server (8 cores)** that we name the *Nehalem machine*. It is a single SMP machine with two `Gainestown` sockets (Intel Xeon X5570 with the `Nehalem` micro-architecture). Each processor has 4 cores with a shared L3 cache. The platform has two L3 caches of 8 MB, one on each chip, for both instructions and data. The core frequency is 2.93 GHz. Each core has a private 256KB L2 cache unified for data and instructions. In addition, each core has a separate L1 data and instructions caches with 32 KB each. The main memory size is 24 GB. Each chip in the platform features an integrated memory controller.

3. **High performance computer (96 cores)** that we name the *ccNUMA machine*. It is an IBM System X3950M2 `ccNUMA` shared memory machine with four compute nodes. Each node has four `Dunnington` sockets (Intel Xeon X7460 with the `Core2` micro-architecture). Each socket (chip) has 6 cores, where each pair of cores have a shared 3MB L2 cache. The 6 cores of a chip have a shared 16MB L3 cache. The L2 and L3 caches are unified for data and instructions. Each core has a separate L1 data and instruction caches of size 32 KB. The core frequency is 2.66 GHz. Each node has a quad 1066 MHz FSBs (one per socket) and a 47 GB RAM memory domain (188 GB in total). In this platform, 256 MB of virtual cache per node is used for interprocessor communications between nodes, to keep data in synchronization (this amounts to as much as 1 GB in total taken from main memory).

The benchmarks have been compiled using three different compilers (gcc 4.1.3, gcc 4-3.2, icc 11.1) with flag `-O3`

`-openmp`. In this article, we report the performance numbers obtained using the vendor compiler icc 11.1 because it provided the best performance in overall. The tested Linux kernel is X86_64 2.6 patched with `perfmon` kernel 2.81.

2.2. Running and Performance Measurement Methodology

The test machines were entirely dedicated during the experiments to a single user. No more than one application was executed at a time. The execution of each benchmark was repeated 35 times for each software configuration and machine. This high number of runs allows us to report statistics with a high confidence level [2, 3]. The successive executions are performed sequentially in a back-to-back way (the termination of an execution launches the following repetitive run). We unset all the shell environment variables that were inessential, to avoid starting stack address bias [4]. We also deactivated the randomisation of the starting address of the stack (this is an option in the Linux kernel versions since 2.6.12). The dynamic voltage scaling was disabled to avoid core frequency variation. We used the build system and scripts of SPEC OMP2001 to compile and optimise the applications, launch them, measure the execution times, check validity of the results and report the performance numbers. Finally, the applications have been executed with various numbers of threads (4, 6, 8 on `Core2` and `Nehalem` machines, 16 and 96 on the `ccNUMA`). Because of performance scalability reasons on OpenMP programs, we fixed an upper limit of the number of threads as the number of available cores (8 or 96 depending on the test machine).

2.3. Statistical significance analysis

When faced to variations of observed execution times, we must use rigorous statistics to study the validity of our empirical conclusions. Empirical conclusions must not rely on sample metrics such as sample means or averages [2], we must rely on statistical tests. This is done thanks to the Speedup-Test methodology described in [3]. Declaring a statistical significance of a speedup (either mean or median execution times) follows a formal protocol:

- Comparing between the average execution times of two samples (two thread placement strategies) is done thanks to the one-sided Student t-test.
- Comparing between the median execution times of two samples (two thread placement strategies) is done thanks to the one-sided Wilcoxon-Mann-Whitney test.

3. TESTED THREAD PINNING STRATEGIES

We classify the thread pinning strategies into two main families. 1) Application insensitive thread pinning strategies place threads on cores independently of the characteristics of the program: threads are placed exactly in the same way for any application. 2) Application sensitive thread pinning strategies place threads on cores according to the characteristics of the program. In this article, we focus on data sharing between threads to decide about the best thread placement. Below, we detail the two main thread pinning families.

3.1. Application Insensitive (Independent) Affinity

The application independent class consists of the following thread affinity strategies:

1. Run the application without affinity (`No affinity`), this means we let the OS to decide about the thread placement on the cores. The OS scheduler tries to improve work balancing between the cores. This strategy allows thread migration between cores during the execution of the application.
2. `Random` strategy. The application threads are placed at launch time randomly through the cores of the machine. Every repeated run corresponds to a new random affinity. Affinity between threads is fixed during the whole run, there is no thread migration during a single run.
3. `Compact icc` strategy. The `icc` compiler uses this strategy to assign successive (in order of their creation) OpenMP threads to cores as close as possible in the topology map of the platform (filling all the cores of a single socket before starting to fill the next socket). This strategy is convenient for applications that have a high data reuse between threads, so they can profit from shared caches inside sockets.
4. `Scatter icc` strategy. The `icc` compiler uses this strategy to distribute the OpenMP threads as evenly as possible across the entire sockets (one thread per socket if possible). This strategy is convenient for applications that have a high data locality inside each thread, so they can profit from a large private cache inside a socket without sharing it with other threads.

3.2. Application Sensitive (Dependent) Affinity

This family of thread pinning strategies rely on an *affinity graph* for each application. It is a directed valued graph $G = (\mathcal{V}, \mathcal{E}, \alpha)$. $\mathcal{V}$ is the set of application threads, $\mathcal{E} = \mathcal{V} \times \mathcal{V}$ and $\alpha : \mathcal{E} \mapsto \mathbb{N}$ is a gain function applied to every pair of threads. The gain function models the attraction factor between two threads. For instance, since we rely on data reuse between threads to compute an affinity, the gain function $\alpha(T_i, T_j)$ represents the number of common accesses to common memory caches lines, accessed by both the threads T_i and T_j.

Now the question is how to compute α for an application. Currently, since we are faced to complex SPEC OMP application, we use a profile guided method instead of a static code analysis. We first fix n a number of threads per application ($\|\mathcal{V}\| = n$). Then we use the `Pin` tool to achieve a memory tracing analysis of the applications. We can collect for every thread all the accesses to all memory addresses (which are transformed to accesses to memory cache lines). Then, we aggregate this information among all the threads in order to compute $\alpha(T_i, T_j)$ for every pair of threads.

Once an affinity graph constructed for an application for a given number of threads, we can use it to investigate multiple thread pinning strategies. The idea is based on graph partitioning methods [5]. The affinity graph must be decomposed into disjoint subsets, named a partition. A partition $V = \{V_1, V_2, \cdots, V_k\}$ has the property that $\bigcup_{1 \leq l \leq k} V_l = \mathcal{V}$ 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 reprsenting 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 maximise $\sum_{(T_i,T_j) \in V_l \times V_l} \alpha(T_i, T_j)$ the sum of the gains between threads belonging to the same partition. This optimisation problem is a classical NP-complete problem, so we have to use a heuristics such as [5]. Fortunately, we have a special polynomial case. Indeed, if we are faced to a machine architecture where a cache level is shared between *two* adjacent cores (such as in the `Core2` machine), 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 maximising a global gain. In this special case, the optimisation problem can be solved with a simpler maximum-weight matching in general graphs [6]. Precisely, it can be polynomially and optimally solved thanks to the algorithm of Edmonds in $O(\|\mathcal{V}\|^2.\|\mathcal{E}\|)$ [6].

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

1. `GP` strategy. Apply a graph k-partitioning only to place threads on sockets. For instance, on the `Nehalem` machine, we compute two partitions since we have two shared L3 caches (one L3 per socket).
2. `LP compact` strategy. After using the polynomial method to optimally compute a set of thread pairs (algorithm of Edmonds), this strategy assigns successive thread pairs to cores with shared caches as close as possible.
3. `LP scatter` strategy. After using the polynomial method to optimally compute a set of thread pairs, it distributes the thread pairs as evenly as possible across the entire set of sockets of the machines (one thread pair per socket if possible).
4. `LPGP` strategy. After an initial step of optimal computation of thread pairs (algorithm of Edmonds), we proceed by a graph k-partitioning [5]. It is a hierarchical bottom-up 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.
5. `GPLP` strategy. It is a hierarchical top-down 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.

4. EXPERIMENTAL RESULTS AND ANALYSIS

Every SPEC OMP application has been executed 35 times on every machine, with different number of threads and on two different data sets (`train` and `ref`), according to nine thread pinning strategies. The amount of performance data and figures to analyse is huge. Three synthetic tables reflect the speedups obtained through the thread pinning strategies with respect to the default "no affinity" strategy of the OS scheduler. Tab. I shows the overall sample speedup of every thread pinning strategy on the desktop machines (`Core2` and `Nehalem` machines, having 8 cores each). We show the speedups of the average and the median execution times of all SPEC OMP applications executed with 4, 6 and 8 threads. Tab. II and Tab. III illustrate the same performance metrics on the HPC `ccNUMA` machine where the benchmarks have been executed with 16 and 96 threads. When only 16 threads are used (Tab. II), the LPGP and GPLP strategies may have some variants which are the number of sockets used for executing 16 threads (described Tab. II). In all the tables, we also report the minimal and maximal observed variances of the program execution times in order to study the performance stability. Below we give our experimental conclusions and analyses:

1. On the `Core2` and `Nehalem` machines (8 cores), we can observe that fixing a thread affinity leads to marginal speedups and slowdowns. This means that in terms of average or median execution times, letting the OS decide about thread placement is not a poor strategy. However, the performance variation is high (up to 82.24 for the `Core2` machine). Consequently, if performance stability is an additional quality criteria, it is

better to fix a thread affinity (check the maximal observed variances in Tab. I except for the random affinity strategy).
2. On the `ccNUMA` machine (Tab.. II and Tab. III), we observe speedups for all thread affinity strategies (no slowdown). This means that using Linux thread scheduler is not a good choice in terms of performance.
3. When 96 threads are used on the `ccNUMA` machine (Tab. III), the speedups are more significant. The reason is that the OS thread scheduler gives higher priority to work balancing compared to NUMA latencies: while the Linux kernel is able to distinguish between the latencies of distinct NUMA nodes, it still prefers to schedule threads to free available cores (to optimise work balancing by keeping all cores busy) even if such work balancing increases the cost of memory accesses (remote access to NUMA nodes). Consequently, a poor overall performance is observed if no affinity is fixed because some cores access data to remote memory nodes.
4. We do not observe any important difference, in terms of speedups, between application independent and application dependent strategies. This means that the price of profile guided methods is not easy to justify compared to cheap and easy-to-use icc scatter or compact strategies. One of the explanations is that fixing an affinity does not allow any thread migration during the execution of the application. Since any parallel application code may have different phases, it would be only by luck that the same thread placement gives the optimum for all phases. This favours to investigate other affinity solution based on thread migrations.

5. RELATED WORK AND DISCUSSION

Most of the thread affinity studies on multi-core architectures focus on data locality and cache sharing in parallel applications. Zhang *et al.* [7] have conducted a measurement analysis to study the influence of CMP cache sharing on multi-threaded performance applications using the PARSEC [8] benchmark suite. 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.* [9] proposed threads clustering to schedule threads based on data sharing patterns detected on-line using hardware performance monitoring units. The mechanism was implemented inside a Linux operating system. Iteratively, they attempt to group threads with high degree of data sharing in the same socket. The mechanism is based on cross-ship communication performance impact.

Klug *et al.* [10] proposed 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 named `autopin` requires that the user provides an initial set of good thread placements. Kazempour *et al.* [11] examined the performance effect of exploiting cache affinity on multi-core multiprocessors and uniprocessors. They demonstrated that performance improvement from exploiting cache affinity on multi-core multiprocessors is significant. Terboven *et al.* [12] 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 similar to the ones studied in this article. 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 optimised thread placements. They also introduce an analytical model to estimate the cost of running an affinity-based thread schedule. While their affinity graph is based on the number of addresses shared among threads, our affinity graph is built upon the number of accesses to common cache lines. This reflects the real cache activity.

Some studies have addressed the data cache sharing at the compiler level. They focused on improving the data locality in multi-cores based on the architecture topology. Chu *et al.* [14] proposed a profile guided method for partitioning memory accesses across distributed data caches. The difference with our work is that they focused on fine grain parallelism in single-threaded applications. Lee *et al.* [15] proposed a framework to automatically adjust the number of threads in an application to optimise system efficiency. The work assumes a uniform distribution of the data between threads. Kandemir *et al.* [16] discussed a compiler directed code restructuring scheme for enhancing locality of shared data in multi-cores. The scheme distributes the iterations of a loop to be executed in parallel across the cores of an on-chip cache hierarchy target.

Our current article is not intended to propose a yet another thread pinning strategy based on data locality and sharing information. We oriented our work to an empirical exhaustive study to test multiple thread placements methodologies that are various and precise enough to cover the large spectrum of the existing ones. Three main points charac-

Table 1. Overall Sample Speedups of the Tested Thread Affinities with SPEC OMP2001 Benchmarks Running on the `Core2` and the `Nehalem` SMP Machines. The baseline thread placement strategy is the OS free affinity. Each benchmark is executed repeatedly 35 times, using each run the `train` input dataset and with 4, 6, and 8 threads. The minimal and maximal observed performance variances of the OS free affinity are [0.01 ; 82.24] on `Core2` machine and [0.00; 0.52] on `Nehalem` machine.

		Overall speedup (mean)		Overall speedup (median)		Min and max observed performance variances	
Thread pinning strategy		Core2 SMP	Nehalem SMP	Core2 SMP	Nehalem SMP	Core2 SMP	Nehalem SMP
Application independent	Random	0.995	0.995	0.992	1.001	[0.00 ; 82.24]	[0.00 ; 12.25]
	icc compact	0.952	0.995	0.944	0.996	[0.00 ; 0.22]	[0.00 ; 0.05]
	icc scatter	1.013	0.998	1.004	0.999	[0.00 ; 0.25]	[0.00 ; 0.31]
Application dependent	LP compact	0.952	0.995	0.945	0.996	[0.00 ; 0.11]	[0.00 ; 0.32]
	LP scatter	1.019	1.007	1.011	1.008	[0.00 ; 0.11]	[0.00 ; 0.27]
	LPGP	1.022	1.032	1.014	1.032	[0.00 ; 0.06]	[0.00 ; 0.04]
	GP	-	1.012	-	1.013	-	[0.00 ; 0.03]

Table 2. Overall Sample Speedups of the Tested Thread Affinities with SPEC OMP2001 Benchmarks Running on the `ccNUMA` Machine. The baseline thread placement strategy is the OS free affinity. Each benchmark is executed repeatedly 35 times, using for each run the `ref` input dataset and 16 threads. The minimal and maximal observed performance variances of the OS free affinity are [0.02 ; 12015.19]

Thread pinning strategy		Overall speedup (mean)	Overall speedup (median)	Min and max observed performance variances
Application independent	Random	1.053	1.046	[0.05 ; 3329.47]
	icc compact	1.025	1.018	[0.06 ;12.62]
	icc scatter	1.159	1.153	[0.08 ; 44.61]
Application dependent	LP compact	1.024	1.016	[0.08 ; 5.7]
	LP scatter	1.165	1.155	[0.10 ; 60.45]
	LPGP(4 sockets)	1.086	1.078	[0.03 ; 6.29]
	LPGP(8 sockets)	1.211	1.203	[0.01 ; 14.57]
	LPGP(16 sockets)	1.165	1.157	[0.02 ; 53.33]
	GPLP(4 sockets)	1.083	1.075	[0.05 ; 10.63]

terise our contribution: 1) We take into account the performance variability (repeat the run of each application 35 times), analysed through a rigorous statistical protocol [3]; 2) We use real complex applications (SPEC OMP) not synthetic benchmarks or small kernels; Finally 3) Our work is not limited to find the best thread placement against the default one of the application, instead, we investigate how the overall performances of a given multi-threaded applications behave under a set of predominant thread affinity schedules. The last point is important because we showed that simple application independent thread pinning strategies such as `compact` or `scatter` perform very well for many OpenMP applications.

6. CONCLUSION AND FUTURE WORK

This article investigates various cache-aware thread pinning strategies for SPEC OpenMP applications. We demonstrate that fixing an affinity provides statistically significant performance improvements compared the Linux OS strategy. However, the performance improvement is marginal in UMA `Core2` and `Nehalem` machines, but the performance stability is better. On the tested `ccNUMA` machine, the speedups of all thread pinnings are significant because the OS thread scheduler gives higher priority to work balancing among cores against NUMA sensitive scheduling. Interestingly enough, we demonstrated that application independent strategies (icc scatter and compact) provide equivalent performance gains compared to profile guided (application dependent) methods. However, we think that profile guided methods should be better if they consider program phases to decide about variable thread pinnings (migration). In the future, we will investigate thread pinning and migration strategies per parallel region.

ACKNOWLEDGEMENTS

We would like to thank the following colleagues for their hint on the algorithm of Edmonds: Sandrine VIAL*, Bertrand* LECUN*, Thierry* MAUTOR *and Franck* QUESSETTE.

Table 3. Overall Sample Speedups of the Tested Thread Affinities with SPEC OMP2001 Benchmarks Running on the `ccNUMA` Machine. The baseline thread placement strategy is the OS free affinity. Each benchmark is executed repeatedly 35 times, using for each run the `ref` input dataset and 96 threads. The minimal and maximal observed performance variances of the OS free affinity are [28.13 ; 3364.08]

Thread pinning strategy		Overall speedup (mean)	Overall speedup (median)	Min and max observed performance variances
Application independent	icc compact	1.557	1.562	[0.11 ; 2.18]
	icc scatter	1.426	1.428	[0.20 ; 1.89]
Application dependent	LP compact	1.556	1.559	[0.17 ; 2.17]
	LP scatter	1.42	1.422	[0.10 ; 2.33]
	LPGP	1.565	1.568	[0.10 ; 3.85]
	GPLP	1.566	1.569	[0.14 ; 3.99]

REFERENCES

[1] A. Mazouz, S.-A.-A. Touati, and D. Barthou, "Analysing the Variability of OpenMP Programs Performances on Multicore Architectures," in *Proc. of Programmability Issues for Heterogeneous Multicores, in conjunction with the HIPEAC conference*, Heraklion, Greece, Jan. 2011.

[2] Raj Jain, *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modelling*. John Wiley and Sons, 1991.

[3] S.-A.-A. Touati, J. Worms, and S. Briais, "The Speedup-Test," University of Versailles Saint-Quentin en Yvelines, Tech. Rep., Jan. 2010, http://hal.archives-ouvertes.fr/inria-00443839.

[4] Todd Mytkowicz and Amer Diwan and Peter F. Sweeney and Mathias Hauswirth, "Producing wrong data without doing anything obviously wrong!" in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[5] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, pp. 96–129, January 1998. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1997.1404

[6] J. Edmonds, "Maximum matching and a polyhedron with 0-1 vertices," *Journal Res. Nat.*, vol. 69-B, no. 1-22, pp. 125–130, 1965.

[7] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" in *PPoPP '10: Proc. of the ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2010, pp. 203–212.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[9] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors, booktitle = EuroSys '07: Proc. of theACM SIGOPS/EuroSys European Conference on Computer Systems 2007." New York, NY, USA: ACM, 2007, pp. 47–58.

[10] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "autopin — Automated Optimization of Thread-to-Core Pinning on Multicore Systems," *Transactions on High-Performance Embedded Architectures and Compilers*, 2008.

[11] V. Kazempour, A. Fedorova, and P. Alagheband, "Performance implications of cache affinity on multicore processors," in *Euro-Par '08: the international Euro-Par conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 151–161.

[12] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and thread affinity in OpenMP programs," in *MAW '08: Proc. of the workshop on Memory access on future processors*. New York, NY, USA: ACM, 2008, pp. 377–384.

[13] F. Song, S. Moore, and J. Dongarra, "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] M. Chu, R. Ravindran, and S. Mahlke, "Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures," in *MICRO 40: Proc. of the Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 369–380.

[15] J. Lee, H. Wu, M. Ravichandran, and N. Clark, "Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications," in *ISCA '10: Proc. of the annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2010, pp. 270–279.

[16] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. J. Irwin, and Y. Zhnag, "Cache topology aware computation mapping for multicores," *SIGPLAN Not.*, vol. 45, no. 6, pp. 74–85, 2010.