Improved Scalability by Using Hardware-Aware Thread Affinities

Sven Mallach[1] and Carsten Gutwenger[2]

[1] Universität zu Köln, Germany
[2] Technische Universität Dortmund, Germany

Abstract. The complexity of an efficient thread management steadily rises with the number of processor cores and heterogeneities in the design of system architectures, e.g., the topologies of execution units and the memory architecture. In this paper, we show that using information about the system topology combined with a hardware-aware thread management is worthwhile. We present such a hardware-aware approach that utilizes thread affinity to automatically steer the mapping of threads to cores and experimentally analyze its performance. Our experiments show that we can achieve significantly better scalability and runtime stability compared to the ordinary dispatching of threads provided by the operating system.

1 Introduction

Today, multicore processors have become an ultimate commodity in private and scientific computing allowing multiple levels of parallelism. Namely one can achieve parallelism by superscalarity, SIMD instructions, multiple cores, and simultaneous multithreading using only a single processor. Under these conditions cache/memory considerations become more complex and, when combining two or more processors within one system, complexity rises extremely if an implementation shall scale on different architectures.

However, even within the omnipresent products of the x86 processor market, systems have significant differences concerning cache and memory subsystem design, which has a high impact on the overall performance of an implementation. This heterogeneity will considerably increase when multicore processors will offer 64, 128 or even more cores on a single chip. In such a scenario not all cores can have uniform access to all caches (or cache levels) and memory in terms of latencies. Therefore the choice of the "right" cores to share data will be of great importance. At the same time, other heterogeneous designs arise. E.g., the Cell processor [3] shipped with every Playstation 3 has lead to a community using the offered parallel floating point computation power for scientific purposes. Again, exploiting this potential requires new and non-standard programming techniques as well as knowledge about hardware issues.

Taking a look at current designs in the x86 processor market, we consider NUMA to become the dominant type of multiprocessor systems. Despite the

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 29–41, 2010.

never ending discussion whether explicit (hand-tuned) or implicit (compiler-driven) parallelism is the better strategy, hardware details have to be taken into account in order to achieve good scalability. This fact makes it hard for parallelized implementations to perform reasonably in general.

In this paper, we present a *hardware-aware* thread management using *thread affinity* as a key concept and analyze its performance. Our goal is to show that the effort to investigate the underlying system topology and properties in combination with a steered mapping of threads to execution units is worthwhile. When discussing parallel implementations, it is often argued that they lead to less "determinism" in terms of running times. In contrast to that we will show in our experimental analysis that the controlled dispatch of threads results in much more stable and reliable scalability as well as improved speedups.

Thread affinity is not an entirely new idea. Studies which propose its use within OpenMP-driven programs can be found, e.g., in [10]. The authors in [9] use affinities to prevent interrupt handling routines to be executed by different cores. Further, autopin [5] is an application that already binds threads to execution units in order to optimize performance. However, autopin asks the user to specify possible bindings which are then successively applied to find out the superior one. This requires detailed knowledge about the underlying hardware. Moreover, the implementation of the hardware performance counters used for evaluation is not yet standardized. Reading them out needs modified kernels and their values can be influenced by other processes. Additionally, the time needed for the optimization process is high compared to the runtime of typical small algorithmic tasks on relevant input sizes as we will show here. In contrast to that, the presented approach tries to exploit operating system (OS) calls as well as processor instructions to retrieve topology information automatically.

The remainder of this paper is organized as follows. In Sect. 2, we describe the differences in common memory subsystem and processor designs in detail. Sect. 3 presents our hardware-aware thread management. We explain how the respective strategies are implemented using thread affinity. In Sect. 4, we evaluate the sustained performance when applying the proposed strategies and compare them to the performance achieved when leaving the full responsibility to schedule threads to the OS. We will conclude on our approach in the last section.

2 Hardware Preliminaries

The central difference between todays multiprocessing designs is their way of accessing the memory. Besides uniform systems, like SMP (*symmetric multiprocessing*), the number of *non-uniform memory architectures* (NUMA) steadily increases. In SMP systems (see Fig. 1(a)), we have a single amount of physical memory shared equally by all execution units, and one memory controller that (sequentially) arbitrates all incoming requests. Not necessarily, but in general, this implies that all processors are connected to the controller via a single bus system called *front side bus* (*FSB*). The main advantage of this design is that data can be easily shared between processors at equal cost. The disadvantage is that it does not scale for arbitrary numbers of processors, since memory

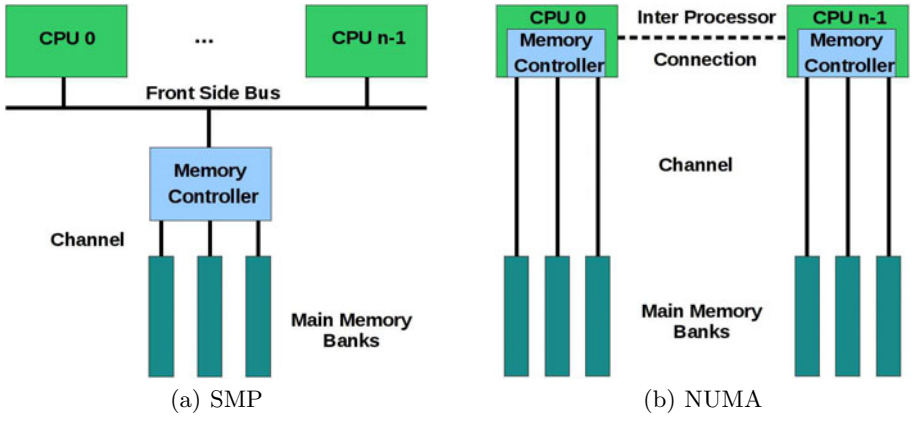


Fig. 1. Typical memory system topologies

bandwidth quickly becomes a bottleneck if multiple processors require access simultaneously. In contrast to that, scalability is the most important design target of NUMA systems; see Fig. 1(b). Each processor has its own memory banks, its own memory controller, and its own connection. The drawback shows up if one processor needs access to data stored in a memory bank controlled by a different processor. In this case considerably slower connections between the processors have to be used to transfer the data. This can lead to a significant performance loss if an algorithm or thread management is not able or not designed to keep data local to processors.

3 Hardware-Aware Thread Management

3.1 Hardware Awareness

We consider now SMP and NUMA systems with multicore processors, i.e., we have not only parallelism between processors but also *within* each processor. For simplicity of presentation, we use the terms *core* and *execution unit* as synonyms, even though in case of simultaneous multithreading hardware-threads would represent multiple execution units on one core. Due to the different extent of locality we have to treat SMP and NUMA in different ways if we are to share data and to synchronize threads. Therefore we are interested in controlling which thread runs on which core, and—at any time—to be able to pick the best processor core for the current job that is to be processed. For this purpose, our hardware-aware thread management tries to gain as much topology information as possible, e.g., how many cores exist and which of them are placed on the same processor chip—possibly sharing caches. For x86 processors the corresponding mapping is obtained by determining the APIC IDs of their cores [2]. Similar functionality is extracted from the *numactl*-API [4] to investigate the node structure of the underlying multiprocessing system.

In this paper, we focus on the typical use case where a sequence of memory (e.g., a container data structure) has been initialized sequentially (e.g., by reading some input) and will now be worked on in parallel. Due to the *first touch strategy* applied by many operating systems, the data or part of it is only stored within the cache(s) belonging to the core which executed the main thread, while all other caches in the system remain *cold*. Since the end of the data structure was initialized last, we can also assume that the probability of remaining cache data increases towards the end of it.

A thread running on the same core (or cores with access to the same cache-level) will perform well and threads running on cores located on the same chip with access to a coherent higher-level cache or at least local memory will incur only small delays. But threads on cores of other processors that cannot exploit any locality will experience high latencies. Therefore, especially for small tasks or only once traversed data, page migration on NUMA systems is often not worthwhile compared to the reuse of a subset of the local cores while keeping remote ones idle.

Even though there exist routines in multithreading libraries that look like they start a bunch of threads at once, behind the scenes, the respective number of threads has to be started one by one. This means every thread t_i has a start time T_i^{start} and an end time T_i^{end} that varies with the choice of the executing core. Due to synchronization needs, the overall performance of the parallel computation depends on the "slowest" thread, i.e., for t threads in the parallel section the running time ΔT is $max_{i=1}^{t} T_i^{\text{end}} - \min_{j=1}^{t} T_j^{\text{start}}$. Therefore our goal is to minimize ΔT by starting potentially slow threads as early as possible and finding an optimal mapping of threads (e.g., processing certain array intervals) to cores.

Under the assumptions made, it is reasonable to start threads on "distant" cores processing the front-end of the memory sequence first, since they will be the "slowest" ones. There is only a small chance that the data resided in a cache, and on a NUMA system there is a memory transfer needed to get the data to the executing processor. Coming closer to the end of the sequence, the probability that it can be obtained from a cache of the processor executing the main thread rises. It is therefore promising to use cores as local as possible to the core executing the main thread for processing these intervals. The last interval might then even be processed by the main thread itself.

3.2 Thread Affinity

While the concept of *thread pooling*, i.e., keeping once used threads alive for fast re-use without creation overhead, is a well known and widely applied concept (even in compiler libraries like OpenMP), *thread affinity* has not yet gained the focus it deserves. This is especially surprising since thread affinity can easily be combined with pooling leading to much more reliable parallel execution.

If we consider a multiprocessing system with k cores, our usual assumption is that, if we create k threads, these will be scheduled one-by-one on the k available units. In many cases this assumption is wrong and operating systems

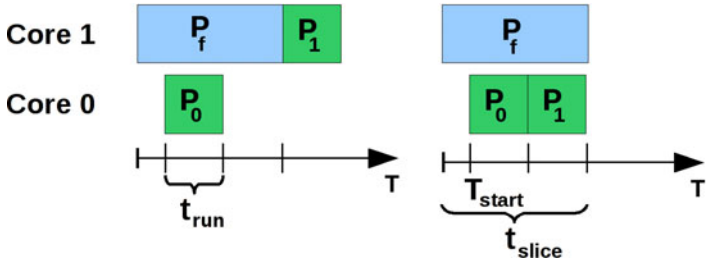


Fig. 2. Worst case scenario: using thread affinity (left); with flexible scheduling (right)

tend to schedule more than one thread on one core, or need some time slices to move threads to a less loaded one. The concept of *thread affinity* allows us to take control over the spreading of threads by explicitly configuring which thread shall run on which execution unit. Together with knowledge about the topology and memory design of the system, we use it as a powerful instrument to exploit algorithmic properties and significantly improve runtime latencies. Nearly all strategies that derive from the observations in Sect. 3 depend on the possibility to control the mapping of threads to execution units for exploiting data localities and available memory bandwidth.

The usual way thread affinity is implemented is by using OS functions. The Unix scheduler offers the function `sched_setaffinity()` and when using POSIX threads [1], the function `pthread_setaffinity()` can also be applied. Windows offers the API function `SetThreadAffinityMask()` for this purpose. Solaris also knows ways to implement thread affinity, but with the drawback that they all need privileged execution rights.

Despite all advantages, there exists a pathological example where the application of thread affinity may have negative impacts if other compute-intensive processes load one or more of the cores where threads are bound to. As a simple example for a dual-core processor, imagine some program P that can be trivially partitioned into two parts P_0 and P_1. Suppose its parallel execution shall start at some point of time T_{start} and time t_{run} is needed to execute P_0 or P_1. Now assume that at T_{start}, there is already another process P_{f} scheduled on core 1, as shown in the left part of Fig. 2. Since the thread that shall execute part P_1 is bound to core 1, the scheduler is forced to wait until the end of the current time slice. If we abstract from the costs of context switches and assume that P_0 can be shortly re-scheduled for synchronization, the running time of the program will be $t_{\text{slice}} + t_{\text{run}} - T_{\text{start}}$. If the binding could be released, as is indicated in the right part of Fig. 2, a running time of $2 \cdot t_{\text{run}}$ would suffice. In contrast to the left scenario, this would mean no loss compared to sequential execution of P and with more cores speedup could still be achieved. To enable the release of a binding we could relax the restriction of a thread's affinity mask to only one core, but this would lead to non-optimal scheduling in other cases. One solution could be to augment the mask only if necessary—at the cost of additional overhead

for testing the respective conditions. Experiments how to realize such a strategy efficiently are just in their infancy.

3.3 General Pitfalls

When leaving the responsibility completely to the OS, it often happens that threads are dispatched on the same core that also executes the main thread. In the worst case, if the assigned tasks have running times in the order of a few time slices, every thread performs its tasks one-by-one instead of being moved to another core. Hence, a more or less sequential execution takes place. Even if this is not the case, a considerable performance penalty due to synchronization overhead results. Consider a thread that is to be awakened. No matter which implementation of threads is used, some time before it will have acquired a mutex lock and called the `wait()` function of a condition variable, which releases the lock again. If now the main thread wants to wake up the thread, it locks the mutex and calls `signal()` for the corresponding condition variable. Afterwards it would unlock the mutex. Experiments [6] have shown that the signaled thread may receive a time slice before the main thread has reached the atomic part of the unlock, like shown in Fig. 3. As returning from `wait()` results in a try to re-lock the mutex, the time slice will be spent with waiting, until, in the next time slice assigned to the main thread, the unlock completes.

In Sect. 4 we will see that it can even be profitable to use less than the maximum available execution units for a given task. As shown in [6], the execution time needed to perform a task on a given input size by a single thread taking part in the parallel execution can increase when the overall number of participating threads rises. This is the case if the memory controller is saturated or if inter-processor communication has to be used in a NUMA system. Even the assumption that the sustainable memory bandwidth for one processor is sufficient to serve all its cores is wrong as our and other [5] results demonstrate. Especially if we decide to use only a subset of the available threads, it is important not to start with the most "distant" ones, but to use those that are most local to the core which initialized the data. Employing this strategy we can be considerably faster than if we would have spread the work randomly.

3.4 Implementation of Dispatch Strategies

In order to facilitate flexible strategies for different subsystem designs and scenarios by means of thread affinity, we implemented a thread-pool that keeps

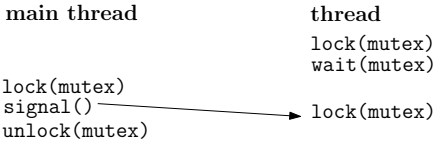
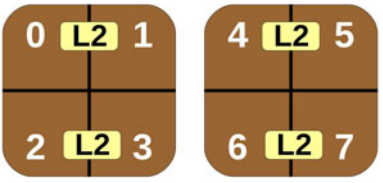


Fig. 3. Synchronization process between the main thread and a computation thread

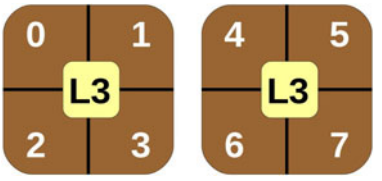
threads sorted according to their topology in the system. To avoid the issues described above, we also control which core executes the main thread and bind no other thread to it. Hence, now every sequential initialization will be performed on that known core that we now refer to as c_{main}. We implement dispatch strategies optimized for different memory subsystem designs, tasks, and numbers of participating threads by deciding to group threads on, e.g., the processor comprising c_{main}, or to alternate between the available processors in order to balance load and optimize bandwidth. For this purpose, the functions that realize the selection of particular threads mainly operate on the sorted array mentioned above.

In case of a *compute-bound* operation with dynamic load balancing, two major properties have to be taken into account, namely the memory architecture and the number of threads to use. Due to our assumption to work on sequentially initialized data, we emphasize on exploiting locality on NUMA systems. That is, the dispatching function will not alternate between the nodes, but try to use as many cores from the processor comprising c_{main} as possible. In other scenarios with threads allocating a lot of memory themselves, a bandwidth-optimizing distribution strategy would be more appropriate. On an SMP architecture, incorporating cores of another processor does not incur higher delays. Additionally, if the processors have their own bus connections to the memory controller (like in our Penryn test system; see Sect. 4) it is worthwhile to use cores on distinct ones even if only two threads have to be dispatched, since using only one of them would not fully exploit the sustainable bandwidth of the controller.

In case of a *memory-bound* operation things become more complicated because, for small inputs, cache considerations decide whether a dispatch strategy is a good one or not. If the input entirely fits into the cache, it is generally difficult to be faster than sequential execution, since other cores have cold caches. To react on this, our dispatching function takes into account the amount of memory that will be worked on by a single thread as a third parameter. If we have small inputs and a coherent cache level for some or all cores on one processor, we try to dispatch *all* threads on the processor with core c_{main}. This leads to a trade off: Using less cores than available means a weaker parallelization, but these fewer threads now work on hot caches and receive much faster memory access. On SMP systems we can expect only little effects resulting from that, but on NUMA this strategy avoids the high latencies that would be incurred by a page migration to other processors. If the input size is bigger than the cache, local and remote cores will equally have to read from main memory. The cost for inter-processor communication can therefore be amortized by a stronger parallelization, so it is worthwhile to use cores on other processors, too. Similarly, if available, we can get more bandwidth by using multiple bus connections of SMP systems. The dispatching function starts the threads on cores of the other processors first as they will have the highest delays and then moves on to the processor with the main thread. If only a subset of the available cores is used, only as many "distant" cores as necessary are involved. Fig. 4 shows the



(a) A 2-processor system with 4 cores and two cores sharing an L2-cache



(b) A 2-processor system with 4 cores and a shared L3-cache each

scenario / core	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
NUMA (a) and (b), 4 threads, out-of-cache					0	1	2	3
SMP (a) and (b), 4 threads, out-of-cache	0	1					2	3
NUMA and SMP (a), 4 threads, in-cache							0/2	1/3
NUMA and SMP (b), 4 threads, in-cache					0	1	2	3

(c) Example dispatchs for the above system architectures

Fig. 4. Treatment of different architectures by our thread management

resulting thread dispatch strategies for some example scenarios that can be mapped one-to-one to the test systems used in the evaluation.

4 Experimental Evaluation

A good indicator for the performance of a thread management is its memory throughput. Due to the constantly growing gap between computational and memory throughput, most algorithms will not be able to scale linearly as long as they do not comprise intensive computations on a relatively small amount of data. Otherwise speedup can only increase to a certain factor well known as *memory wall* [8]. This factor cannot be superseded even by use of arbitrary numbers of cores, since every execution unit spends its time on waiting for memory transfers and additional units may even worsen the pollution of the memory bus. Another important measure is the speedup for small inputs. The advantages of efficient dispatching strategies, synchronization and small latencies achieved by a thin management layer can be visualized here. For our purposes, this focus is straight-forward, as we want to demonstrate the benefits of a hardware-aware thread management rather than present a competition between algorithms implemented on top of different backends which have their own impact on the measured performance.

We consider two scenarios to demonstrate the advantages when using thread affinity. We begin with a shallow copy (the C function `memcpy()`) as an examination of a *memory bound* directive which can be trivially parallelized and therefore theoretically achieve a linear speedup only bounded by the sustainable memory throughput. With this example we gain insight about the current state of the memory gap mentioned above, sensitize for the dependencies between speedup and memory throughput, and demonstrate the effects of our cache-aware thread management. We intentionally do not use established benchmarks

like, e.g., STREAM [7] for comparison because they initialize data in parallel. As a second experiment, we consider a more computationally intensive operation which is not memory bound. For this purpose, we choose the function `partition()` from the Standard Template Library, which is part of the C++ standard. It has been parallelized using dynamic load balancing [6,11] and is now executed on top of our thread-pool. Being a basis for many sorting and selection problem algorithms, it is an adequate example to demonstrate the improved scalability and stability achieved.

4.1 System Setup

The test systems used in our experimental evaluation are summarized in Table 1. For our benchmarks we use the C function `clock_gettime()`, measuring *wall clock time* with a resolution of 1 ns on all test systems, and the `g++-4.4.1` compiler with optimization flag `-O2`. We compare the arithmetic average of the running times (or throughputs) of 1000 calls to the sequential and parallel function on *different* input data, thus wiping out cache effects between successive function calls. This simulates real world applications, which usually have cold caches before the start of a parallel computation. The pseudo-random inputs are equal for every number of used threads (by using seeds) and stored in containers of type `std::vector`. For the throughput benchmarks these are 64-bit floating point numbers, and for `partition()` we use 32-bit floating point numbers in the range $[0, MAX\_RAND]$ with $MAX\_RAND/2$ as pivot element. For our test systems MAX_RAND is $2^{32} - 1$.

4.2 Results

Fig. 5 shows the results for the `memcpy()` function. We first notice the typical progression of the line representing the sequential memory throughput. On all platforms we have a very high throughput at the beginning stemming from full in-cache data, which steadily decreases with the proportion of data that fits into the cache. When no data from the initialization is reusable, it becomes stagnant.

Table 1. Systems used for benchmarks

	Nehalem	**Shanghai**	**Penryn**
CPU	Intel Core i7 940	AMD Opteron 2378	Intel Xeon E5430
# CPUs / Frequency	1 / 2.93 GHz	2 / 2.4 GHz	2 / 2.66 GHz
Cores per CPU	4 (8 Threads)	4	4
Memory Architecture	Single	NUMA	UMA / SMP
L1 Data / Instr. (Core)	32 KB / 32 KB	64 KB / 64 KB	32 KB / 32 KB
L2	256 KB per Core	512 KB per Core	2×6 MB per CPU
L3 (CPU)	8 MB	6 MB	-
Main memory	12 GB	16 GB	8 GB
Linux (x86-64) Kernel	2.6.28-14-generic		2.6.24-25-generic

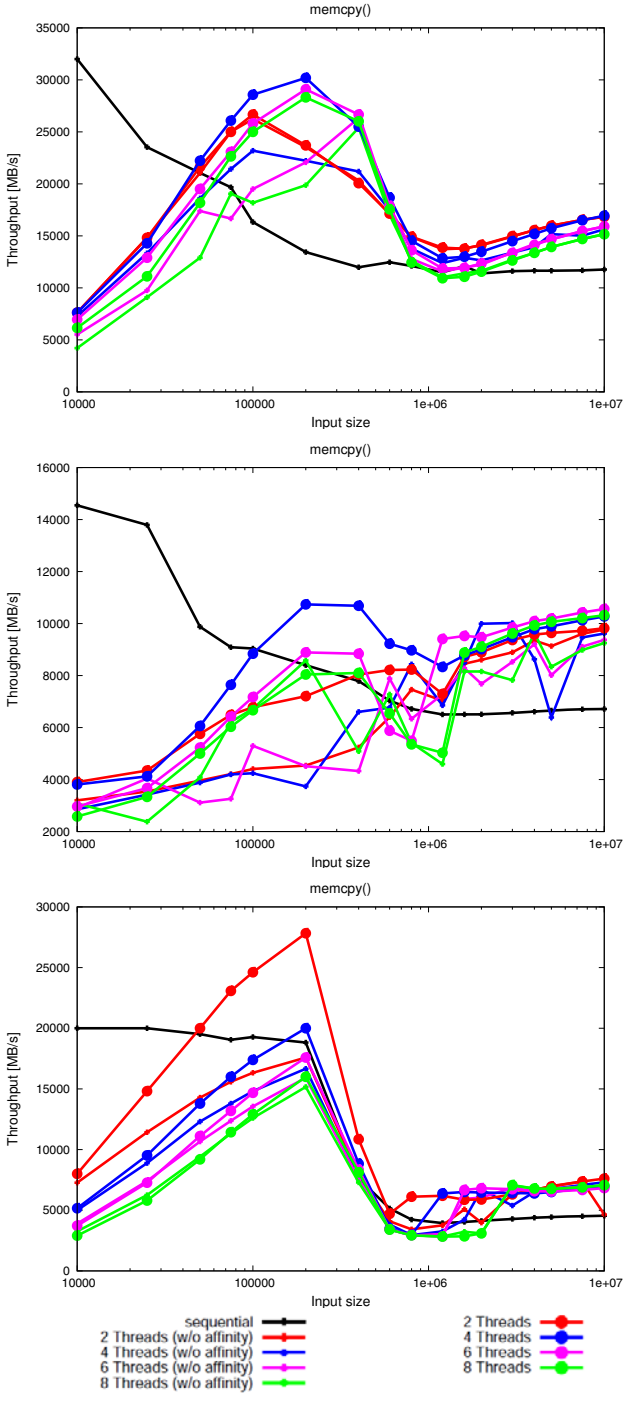


Fig. 5. Throughput for `memcpy()`: Nehalem (top), Shanghai (middle), Penryn (bottom)

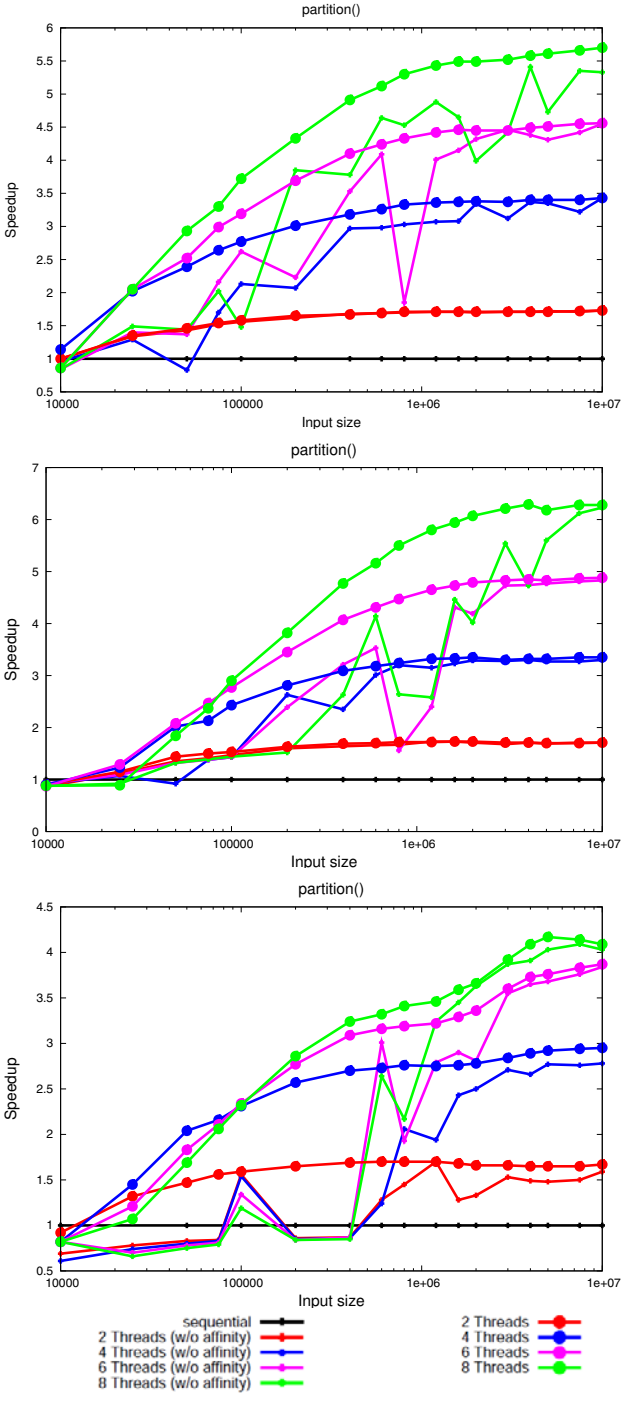


Fig. 6. Speedup for `partition()`: Nehalem (top), Shanghai (middle), Penryn (bottom)

For the parallel execution, we firstly observe a very poor performance due to the sequential initialization of the data, which is therefore only available in coherent or shared on-chip caches. Therefore and due to less overhead, execution with 2 (Penryn) or 4 threads (Nehalem and Shanghai) performs slightly better than with more threads. This effect dominates and leads to peak throughputs on all test systems until the input size runs ahead the cache size, e.g., at 10^6 (8 MB) for the Nehalem system. Using the presented hardware-aware approach we generally achieve a higher throughput than without thread affinity in this interval. Afterwards, the throughput becomes solely dependent on the bandwidth to the memory where the input was stored after initialization. As claimed before, all platforms cannot even serve their execution units with twice the sequential throughput. When the input size reaches a certain threshold, the gap between our multithreading strategy and the performance of the OS becomes smaller, since tasks now run long enough to be successively spread over all cores and smaller latencies gained by a smart dispatch order do not further dominate the overall execution time.

Fig. 6 gives the results for the `partition()` function. We observe very similar results on the Nehalem and Shanghai systems with excellent scaling for the highest input sizes measured. The performance of the Penryn system is reasonable, despite that the additional speedup using 6 and 8 threads is not comparable to the other systems. While the proposed hardware-aware implementation begins to scale already for small inputs, the lines representing execution without affinity have a less steady and sometimes even chaotic progression. Again and due to the same reasons, for the highest input sizes applied the respective lines tend to close up to each other.

5 Conclusion

We have presented a hardware-aware approach for efficient thread management. We pointed out how thread affinity can be used to improve speedup as well as stability of parallel executions and showed the effect of these improvements using two practical examples, namely a memory bound copy scenario and a function for partitioning a range of numbers. We can conclude for both scenarios that the use of thread affinity in connection with knowledge about the underlying processor topology and memory subsystem has lead to better and more reliable memory throughput and scalability. As claimed in the introduction, we confirmed that affinity-based execution is especially worthwhile for small inputs and showed that the interval where improvements make up a considerable difference is small. As a consequence, there is no scope to try out different strategies at runtime and the presented thread management is designed to automatically obtain the information needed to find a good setup.

We also sensitized that already today, and even in NUMA systems, generally not all cores can be simultaneously served with transferred data. As the number of cores steadily increases, this ratio will even worsen in the near future. Consequently, the effort of thread management paying respect to this fact will

increase at the same time. Whereas in the past even small and simple data structures were stored in memory for reuse, it may soon make sense to recompute them instead of waiting for satisfied load requests. Though memory has become cheaper, a change in algorithm design could lead to applications being faster and considered superior as previous implementations by consuming less memory.

References

1. Drepper, U., Molnar, I.: The native POSIX thread library for linux. Technical report, Red Hat, Inc. (February 2005)
2. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual (April 2008)
3. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: Built for speed. IEEE Micro 26(3), 10–23 (2006)
4. Kleen, A.: A NUMA API for linux. Technical report, Novell Inc., Suse Linux Products GmbH (April 2005)
5. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: `autopin` — Automated optimization of thread-to-core pinning on multicore systems. In: Transactions on High-Performance Embedded Architectures and Compilers. Springer, Heidelberg (2008)
6. Mallach, S.: Beschleunigung paralleler Standard Template Library Algorithmen. Master's thesis, Technische Universität Dortmund (2008), `http://www.informatik.uni-koeln.de/ls_juenger/people/mallach/pubs/diplomarbeit.pdf`
7. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, Dezember (1995)
8. McKee, S.A.: Reflections on the memory wall. In: Proceedings of the 1st Conference on Computing Frontiers (CF), p. 162. ACM Press, New York (2004)
9. Scogland, T., Balaji, P., Feng, W., Narayanaswamy, G.: Asymmetric interactions in symmetric multi-core systems: analysis, enhancements and evaluation. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC 2008), pp. 1–12. IEEE Press, Los Alamitos (2008)
10. Terboven, C., Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in OpenMP programs. In: Proceedings of the Workshop on Memory Access on Future Processors (MAW), pp. 377–384. ACM Press, New York (2008)
11. Tsigas, P., Zhang, Y.: A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In: Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP), pp. 372–381. IEEE Press, Los Alamitos (2003)