A library for portable and composable data locality optimizations for NUMA systems

A Library for Portable and Composable Data Locality Optimizations for NUMA Systems

Zoltan Majo
Department of Computer Science
ETH Zurich, Switzerland

Thomas R. Gross
Department of Computer Science
ETH Zurich, Switzerland

Abstract

Many recent multiprocessor systems are realized with a non-uniform memory architecture (NUMA) and accesses to remote memory locations take more time than local memory accesses. Optimizing NUMA memory system performance is difficult and costly for three principal reasons: (1) today's programming languages/libraries have no explicit support for NUMA systems, (2) NUMA optimizations are not portable, and (3) optimizations are not composable (i.e., they can become ineffective or worsen performance in environments that support composable parallel software).

This paper presents TBB-NUMA, a parallel programming library based on Intel Threading Building Blocks (TBB) that supports portable and composable NUMA-aware programming. TBB-NUMA provides a model of task affinity that captures a programmer's insights on mapping tasks to resources. NUMA-awareness affects all layers of the library (i.e., resource management, task scheduling, and high-level parallel algorithm templates) and requires close coupling between all these layers. Optimizations implemented with TBB-NUMA (for a set of standard benchmark programs) result in up to 44% performance improvement over standard TBB, but more important, optimized programs are portable across different NUMA architectures and preserve data locality also when composed with other parallel computations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; C.4 [*Performance of Systems*]: Performance attributes, Measurement techniques

Keywords NUMA, scheduling, data placement

1. Introduction

Recent parallel systems are often realized with a non-uniform memory architecture (NUMA). In NUMA systems the latency and the bandwidth of memory accesses to last-level caches and to DRAM memory varies depending on the target of the memory access: *local memory accesses* (accesses that remain within the boundaries of a processor) have much lower memory access latency than *remote memory accesses* (accesses that are transferred between processors). E.g., on an Intel Nehalem, local accesses to the last-level cache (DRAM) take 38 (190) cycles, whereas remote accesses take 186 (310) cycles. Similar slowdowns (4.9x for cache accesses, 1.6x for DRAM) can also be observed on comparable AMD systems [15].

Due to the large performance penalty of cross-chip memory accesses, performance optimizations for NUMA systems typically target improving data locality, i.e., the reduction (or even elimination) of remote memory accesses [5, 6, 8, 21, 24, 25, 28, 32–34]. Optimizations are often automatic, that is, the runtime system (e.g., the OS, the VM, or the compiler) profiles the memory accesses of programs and then, based on the profiles, it automatically adjusts the distribution of data and/or the scheduling of computations.

Automatic optimizations for NUMA systems can be highly effective; however, for some programs (e.g., programs with complex memory access patterns) profiles do not convey enough information to enable the runtime system to carry out optimizations successfully. In these cases high-level information about programs (e.g., program data dependences) is needed. As this type of information is likely to be available to the programmer, several projects consider making the development toolchain NUMA-aware. E.g., recent profilers like MemProf [18], Memphis [22], HPCToolkit [19], and DProf [27] present information about a program's memory behavior to the developer, who can then change the code to improve performance.

Profilers pinpoint code locations with inefficient usage of the memory system. In practice, however, programs are rarely optimized for NUMA systems as commonly used parallel languages and libraries like OpenMP or Intel Threading Building Blocks (TBB) are geared towards exploiting the lower levels of the memory system (i.e., L1 and L2 caches), if at all, and have no support for NUMA systems. More specifically, existing parallel programming frameworks have three main limitations.

First, existing frameworks usually require memory-system-aware code to consider many details of the memory hierarchy's layout, thus optimized programs are *not portable*. Second, NUMA-aware code is *not composable*. Mapping data and computations depends on the hardware resources (i.e., cores/processors) available to the program. However, in frameworks with support for composable parallel software (i.e., parallel software composed of multiple, concurrently executing parallel computations [26]) the amount of resources available to a computation can change over time, and therefore memory-system-aware programs are required to adapt the mapping at runtime. Existing frameworks provide the programmer little information about the program's runtime configuration, thus optimizations often simply assume that all hardware resources (i.e., all cores/processors) are continuously available. As a result, the advantages of memory system optimizations are annulled as soon as the optimized computation is composed with other parallel computations. Finally, existing parallel programming frameworks

PPoPP'15, , February 7–11, 2015, San Francisco, CA, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

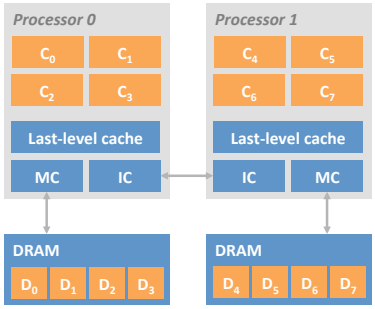


Figure 1. Computation optimized for data locality.

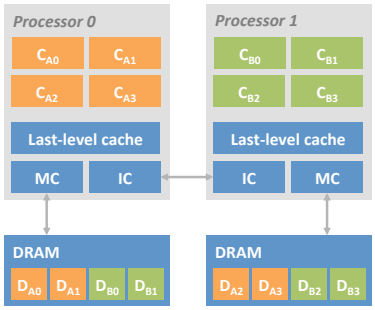


Figure 2. Shared threads: Unfortunate mapping.

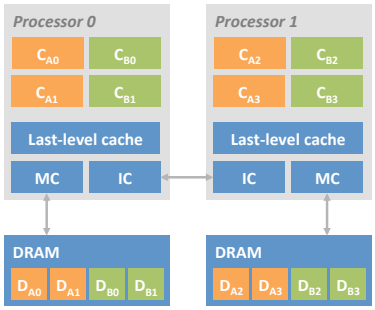


Figure 3. Shared threads: Appropriate mapping.

have *no support for explicit mapping* of data and computations, i.e., the programmer is required to be aware of runtime/compiler/library internals to be able to set up a mapping. As a result of these limitations, even if a programmer conceptually knows how to optimize a program, implementing optimizations is difficult with existing frameworks.

2. Motivation and Goals

These limitations impact the toolchains for NUMA systems: source-level optimizations for NUMA systems are rare in practice, and the performance potential of NUMA systems is often unexploited. To fill in the gap in the development toolchain for NUMA systems, this paper presents TBB-NUMA, a parallel programming library for programming NUMA systems, to demonstrate how portable and composable data locality optimizations can be supported for NUMA systems. We present the library as an extension to TBB to demonstrate that these ideas can be exploited in the context of a widely used platform, but the concepts and ideas are not limited to this software platform.

2.1 Principles of Data Locality Optimizations

Data locality optimizations for NUMA systems have traditionally targeted the co-location of data and computations. To understand the principles of these optimizations, let us consider an example multithreaded program that is parallelized for a 2-processor 8-core NUMA system (the system in Figure 1). The program consists of a set of concurrently executing computations (tasks) $C_0, C_1, \ldots, C_7$; each computation accesses a subset $D_0, D_1, \ldots, D_7$ of the total data used by the program.

To achieve good data locality, the programmer must go through a series of steps. First, the programmer must parallelize the algorithm (i.e., define the computations C_i) so that the data subsets D_i overlap as little as possible. Second, the programmer must distribute data subsets among processors. The final step is to schedule computations so that each computation C_i executes at the same processor as where its data D_i is placed at.

Figure 1 shows the mapping of the example program onto the 2-processor 8-core NUMA system. In the figure each computation executes at the processor where its data is allocated. If data subsets do not overlap (small adjustments often suffice to adjust overlap in multi-threaded computations [35]), this mapping is beneficial for both caching and DRAM performance: (1) The cache capacity available to the computation is maximized (data subsets are disjoint; as a result, each piece of data is present in only one cache); (2) the bandwidth available to the program is increased (and the contention on the memory interfaces is reduced) as all paths to memory are utilized when data is placed at all DRAM modules, and (3) as each computation accesses locally placed data, the program does not encounter any remote memory accesses.

2.2 Enforcing Data Locality in Practice

Data locality optimizations are simple in theory but difficult to implement in practice. Although data distribution is well supported in recent OSs (e.g., Linux supports per-processor memory allocation through the `libnuma` library and memory migration through the `move_pages()` system call), scheduling computations at appropriate processors is problematic in today's parallel languages and libraries. In commonly used parallel frameworks the scheduling of computations at processors depends on two components. First, as most parallel frameworks operate with thread pools, computations (tasks) must be first mapped to threads in the pool (we thus use the term *setting task-to-thread affinities* for mapping tasks to threads). Second, threads from the pool must be *pinned to processors* of the system to ensure that computations execute where intended. If both mappings are set up appropriately, the system guarantees data locality. In the following we discuss problems related to both components.

2.2.1 Component 1: Setting Task-to-Thread Affinities

Commonly used parallel frameworks operate with *implicit* task-to-thread affinities, i.e., with these frameworks the programmer has no direct control on how to map computations to threads. OpenMP static loop partitioning is an example of implicit computation scheduling. For statically partitioned parallel loops the OpenMP runtime assigns a well-defined chunk of the iteration space to each thread. If programmers are aware of the internals of static partitioning and know which pieces of data are touched by each loop iteration, they can distribute data among processors so that each thread accesses data locally. With other OpenMP work-division schemes (e.g., dynamic partitioning), however, the distribution of loop iterations between threads is not deterministic [1], thus the programmer cannot assume much about the data accesses of the program and, as a result, data locality is not controllable.

Setting up task-to-thread affinities is not easy in case of systems based on task parallelism either. E.g., in Intel TBB each task can be assigned a special value; the value defines the affinity of that task to a thread in the pool. The TBB Reference Manual [2] states the following about the values of a task's affinity:

> "A value of 0 indicates no affinity. Other values represent affinity to a particular thread. Do not assume anything about non-zero values. The mapping of non-zero values to threads is internal to the Intel TBB implementation."

Such hints would require the programmers to reverse-engineer the TBB implementation if they wanted to set up a mapping between tasks and threads. Due to this obstacle, it is difficult to implement NUMA data locality optimizations in TBB.

Finally, defining task-to-thread affinities depends on the number of threads available to the program (a value that can change at runtime) but the distribution of data is expressed depending on the number of processors in the system. To ensure data locality on any system and in any runtime configuration the programmer must consider both parameters, an impediment that makes writing NUMA-aware programs with current systems even more cumbersome.

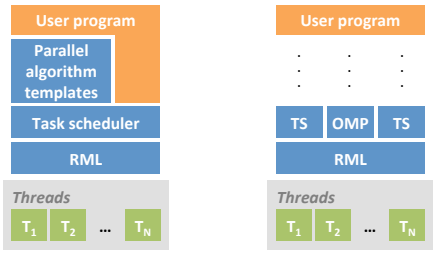
2.2.2 Component 2: Pinning Threads to Processors

The second component of mapping computations to processors is pinning threads to processors. Unless threads are pinned, the OS scheduler may freely move threads around in the system. OS reschedules can result in remote memory accesses (or costly data migrations if data follows the computations using it, e.g., in systems with automatic data migration [5]; this paper assumes a standard OS without automatic data migration).

Some OpenMP implementations allow (although not required by the OpenMP standard [1]) pinning thread pool threads to processors. Pinning threads requires understanding the memory system for every new machine the program is to be run on. If threads are pinned to processors, and the programmer has distributed data and has also set up task-to-thread affinities (e.g., by relying on the properties of static loop partitioning, as discussed before), each piece of data will be accessed at a well-defined processor and the program has thereby good data locality.

Pinning threads to processors works well only as long as only one parallel computation uses a thread pool at a time. Modern runtime systems, however, support composable parallel software, i.e., programs that contain nested parallelism, programs that reuse functionality from parallelized libraries, or programs that are parallelized using different parallel languages/libraries [26]. For these programs the thread pool of the runtime is shared by multiple parallel computations and the runtime distributes threads between all computations that are registered with it.

To illustrate the problems composability causes for programs optimized for NUMA systems, Figure 2 shows an example where two parallel computations, C_A and C_B, execute in parallel on the example 2-processor 8-core system. Computation C_A is composed of subcomputations $C_{A_0} \ldots C_{A_3}$; each subcomputation C_{A_i} accesses a different data subset D_{A_i}. Similarly, each subcomputation C_{B_i} of C_B accesses a distinct data subset D_{B_i}. The programmer optimized both computations for NUMA, thus data used by the computations is distributed across processors (according to the principles discussed in Section 2.1). The programmer has set up task-to-thread affinities as well, but as the runtime is not aware of the programmer's intentions, it can allocate threads to computations in several ways. Figure 2 shows an unfortunate allocation that cancels the optimization intended by the programmer: C_A is mapped to threads executing at Processor 0 and C_B is mapped to threads executing at Processor 1. Thus, both computations access some of their data remotely ($D_{A_2}, D_{A_3}, D_{B_0}, D_{B_1}$). Figure 3 shows an appropriate assignment of threads to computations: In this case each computation is assigned threads from both processors so that each computation can access data locally and exploits all caches of the system.



(a) TBB layers. (b) Composed program.

Figure 4. TBB architecture.

2.3 A Practical Solution

An approach to programming NUMAs that aims to support performance, portability, and composability must address three concerns:

Explicit mapping The programmer can define the distribution of data among processors and, in addition, can also express the preferred schedule of computations (e.g., in the form of hints to a work-stealing scheduler) without being required to understand runtime system internals. The scheduler honors these hints unless there are idle resources; in this case a task may be moved by the scheduler to a different processor in an attempt to balance the load (as in current systems incurring the overhead of remote execution is preferable to idling processing resources).

Portability Programmers are not required to have information about the exact hardware layout but should target a generic NUMA system with P processors. The parallel programming framework must automatically determine the remaining details of pinning threads to appropriate processors so that the optimized programs are portable.

Composability The framework manages its thread pool so that the advantages of data locality optimizations are preserved even if only a fraction of all system resources are available for an optimized computation. This setup allows optimized programs to be included as part of libraries (or reuse functionalities from libraries already parallelized) and to utilize the memory system appropriately at the same time.

This paper describes TBB-NUMA, a parallel library for NUMA systems. We describe TBB-NUMA as an extension to Intel TBB so that we can leverage prior work. To describe the innovation of TBB-NUMA (i.e., defining the semantics of task affinities so that portable and composable parallel programs can be written) we start with the architecture of standard TBB (Section 3) and then (Section 4) highlight the differences between standard TBB and TBB-NUMA in terms of locality-aware programming. Finally, Section 5 presents an evaluation of the performance, composability, and portability of data locality optimizations implemented with TBB and TBB-NUMA for a set of well-known benchmark programs.

3. Anatomy of TBB

Standard TBB has a layered architecture (shown in Figure 4(a)[1]). This section describes the layers top-down, that is, the discussion starts with the layer closest to the programmer and ends with the farthest layer (the layer closest to the hardware).

[1] TBB has an additional layer, the *task arena*, below the task scheduler layer. Threads are registered with the task arena, the task scheduler implements only scheduling. To simplify the discussion we refer to the task scheduler and task arena layers as one layer (task scheduler), see [2] for exact details.

3.1 User Programs

There are two ways to implement parallelism with standard TBB: programmers can either use the library's Cilk-style work-stealing scheduler [12] directly, or they can reuse parallel algorithm templates from a set of templates defined by the library. Templates hide the complexity of the work-stealing scheduler from the programmer, but they still use the work-stealing scheduler internally.

3.2 Parallel Algorithm Templates

TBB supports loop parallelism through the `parallel_for` algorithm template (and also variations of it, e.g., `parallel_reduce`, `parallel_do`). TBB also supports pipeline-parallelism (through the `pipeline` template). In this paper we concentrate on two algorithm templates, `parallel_for` and `pipeline`, because they are widely used and they represent two significantly different ways of approaching parallelism. In standard TBB both templates are optimized for better utilizing L1 and L2 caches. The `parallel_for` template preserves cache locality if it is given an `affinity_partitioner` object as a parameter. We briefly discuss this optimization in Section 3.6 (see [3] and [30] for details about the principles and implementation of this optimization, respectively). Parallel pipelines are optimized for better L1 and L2 cache locality through the way they generate the task tree corresponding to a pipeline computation (see Section 4.4.3 for details).

3.3 Task Scheduler

Similar to Cilk [12], the TBB task scheduler interface exposes library functions to spawn and join tasks (implemented in TBB by the `spawn()` and `wait_for_all()` methods and variations of them). TBB allows but does not guarantee parallelism, thus the task scheduler can have multiple threads (but must have at least one thread) at any given point of time. Each thread has a local deque where spawned tasks are inserted. A thread removes tasks for execution from its local deque in LIFO order and, if the local deque is empty, steals tasks from other threads' deques in FIFO order.

The task scheduler has a set of mailboxes. Each thread in the task scheduler is connected to a (different) *mailbox*. A task can be assigned a special value that specifies the affinity of that task to a mailbox. Thus, the definition of task affinities provided by the TBB Reference Manual (see Section 2.2.1 and [2]) is misleading. According to the manual, a task affinity implies that a task is associated with a *particular thread*, yet an affinity value associates a tasks *only with a mailbox*. During the lifetime of a program, possibly different threads (but only one thread at a time) can be connected to a mailbox. Therefore, affinity values provided by the standard TBB implementation guarantee only that a task is associated with a mailbox, but not with any particular thread. We further discuss the implications of task-to-mailbox affinities in Section 3.6.

An affinitized task (a task with a non-zero affinity value) is *sent* to the thread currently connected to the mailbox. Sending is realized by *inserting* the task into the mailbox. The thread connected to the mailbox *receives the task* by *removing* it from the mailbox. Furthermore, an affinitized task is inserted not only into the mailbox it is sent to, but also into the local deque of the thread that created it. Lastly, task affinities are by definition internal to TBB's implementation and only an `affinity_partitioner` uses them (internally).

TBB supports asynchronous operations through the `enqueue()` call. Tasks to be executed asynchronously are inserted into a FIFO queue shared between all threads (but are not inserted into any thread's local deque). Due to the multiple types of queues in the task scheduler, TBB defines a set of rules (Figure 5) that specify from where a thread is supposed to fetch the next task to be executed. The rules are listed in order of decreasing priority. If a high priority rule is unsuccessful, the scheduler tries the next rule.

1. The task returned by the current task `t`.
2. The successor of `t` (if all predecessors of `t` have completed).
3. The task removed from the thread's own deque (LIFO).
4. The task removed from the mailbox this thread is currently connected to.
5. The task removed from the task scheduler's shared queue.
6. The task removed from another (randomly chosen) thread's deque (FIFO) (steals).

Figure 5. Standard TBB: Rules to fetch next task.

3.4 Resource Management Layer

The number of threads in a task scheduler is determined by the Resource Management Layer (RML). (To avoid excessive OS scheduler overhead, the RML limits the number of threads available.) TBB is interoperable with other parallel frameworks (e.g., Intel OpenMP): If a program is composed of multiple computations (parallelized with possibly different parallel frameworks), all computations *register* with the same RML instance that assigns a subset of the available threads to each computation. Moreover, if the number of computations registered with an RML changes, threads are redistributed between computations. As a result, the number of threads assigned to a computation can vary over time.

Figure 4(b) shows a program composed of two TBB task scheduler-based computations (TS) and one OpenMP-based (OMP) computation (all are registered with the same RML that has N threads). (The example omits higher-level details about the program, e.g., the parallel algorithm templates it uses.) Upcoming examples consider only TBB task schedulers (but not OpenMP runtimes) to be registered with an RML. This simplifies the discussion but is not a real restriction of either standard TBB nor TBB-NUMA.

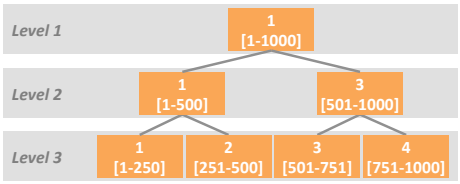
3.5 Threads

In addition to task schedulers, threads are registered and managed by the RML as well. The RML manages two types of threads: (1) The RML automatically creates $N - 1$ worker threads (N is the number of cores of the system); (2) master threads are created by the user program with a suitable system library (e.g., `pthreads`) and are registered the first time they use a parallel construct.

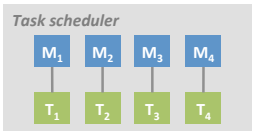
3.6 NUMA Issues in Standard TBB

In standard TBB, each task scheduler has a set of mailboxes, the number of mailboxes is usually set to the number of cores of the machine. When the RML assigns a thread to a task scheduler, the thread connects to a *randomly* chosen mailbox.

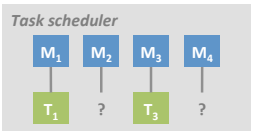
Figure 6(b) shows a task scheduler that is configured with four mailboxes ($M_1 \ldots M_4$); the task scheduler is allocated four threads by the RML ($T_1 \ldots T_4$). During its lifetime, a task scheduler can be allocated different numbers of threads. Moreover, even if the same set of threads is allocated to a task scheduler, each thread can be connected to a different mailbox during the lifetime of the task scheduler (e.g., if a thread leaves and then re-joins a scheduler, the thread can be assigned to a randomly chosen mailbox, and thus possibly not to the same mailbox it was connected to before it left the task scheduler). We refer to the combination of the number of threads in a task scheduler and the mailboxes used by these threads as a *task scheduler configuration*. Figure 6(b) shows the task scheduler in a fully populated configuration where



(a) Task tree with task-to-thread affinities (top) and partitioning (bottom).



(b) TS fully populated.



(c) TS partially populated.

Figure 6. Mailboxing (standard TBB).

a thread is connected to each mailbox; in this configuration each thread T_i is connected to mailbox M_i. In contrast, Figure 6(c) shows the task scheduler in a partially populated configuration in which only mailboxes M_1 and M_3 are used (by threads T_1 and T_3, respectively).

If a thread creates a task tree and then submits it to the task scheduler for execution, and tasks in the tree have affinities to mailboxes, then these tasks are inserted into the creator thread's local deque as well as into the mailbox corresponding to the task's affinity value. Threads in the task scheduler attempt to obtain tasks to execute. First, a thread tries to receive a task from the mailbox the thread is connected to (Rule 4 in Figure 5). If the mailbox is empty, the thread falls back trying to remove a task from the shared queue (Rule 5) or to randomly stealing a task (Rule 6).

Figure 6(a) shows a task tree with three levels; tasks in the tree have affinities specified for the task scheduler configuration shown in Figure 6(b). The example corresponds to a possible partitioning of an iteration space of 1000 iterations (as done, e.g., by the `parallel_for` pattern). If the task tree is repeatedly executed with the same set of affinities and with the same task scheduler configuration, the same subset of the iteration space is sent to the same mailbox. Thus, each thread processes the same subset of the iteration space. As a result, the computation has good cache locality. In standard TBB the `parallel_for` algorithm template is based on this principle: If used with an `affinity_partitioner` object, the `parallel_for` template stores task affinities into the partitioner object and reuses them on future executions.

In standard TBB affinities are only a *hint* on the preferred place of a task's execution, that is, the task scheduler is allowed to *ignore* task affinities to better balance the load. More specifically, tasks are not executed by the thread specified by affinities for three main reasons: (1) *steals*, (2) *revokes*, and (3) *changes in the task scheduler configuration*. First, affinitized tasks are also inserted into the local deque of the thread that creates them, thus they can be *stolen* before they are received at the mailbox they have affinity to. Second, if the thread that created tasks executes Rule 3, it can *revoke* tasks from mailboxes and execute them locally itself. Third, the task scheduler configuration *changes at runtime*; Figure 6(c) shows the task scheduler partially populated with only two threads. For this scheduler configuration the affinities of the task tree do not make sense (because there is no thread connected to mailboxes M_2 and M_4), thus all tasks of the tree (including tasks with affinity to threads T_2 or T_4) will be executed by either thread T_1 or thread T_3.

Programmers cannot foresee dynamically changing runtime conditions, thus TBB does not encourage programmers to specify affinities for tasks. Instead, TBB keeps task affinities internal to the library's implementation. The only case where TBB uses affinities (only internally) is the `parallel_for` template used in combination with an `affinity_partitioner` object. The `parallel_for` algorithm template automatically and internally adapts the affinities of task trees to match the effective place of execution. E.g., let us assume a task tree generated by a `parallel_for` partitioned with an `affinity_partitioner`. Let us furthermore assume that, when unfolding the task tree, the partitioner sets the affinity value of a task `A` in the tree to value 1 (indicating that `A` is preferably executed by the thread connected to mailbox M_1). If task `A` is executed by a thread connected to a different mailbox M_i (because of any of the previously mentioned three reasons), the runtime overwrites the partitioner's record about the task's affinity; after the update the task has its affinity value set to i and this value is used when the task is re-executed. This strategy is beneficial assuming that the affinities specified by the partitioner match the configuration of the task scheduler for some time in the future. However, updating task affinities is not acceptable in NUMA systems: *In NUMA systems task affinities must stay constant because each task must execute at the processor where its data is located.*

4. Reconciling TBB and NUMA

Implementing support for NUMA-aware programming involves all layers of TBB's architecture, and, in some cases, it requires tight coupling between the layers. The discussion in this section follows the layers bottom to top. This section focuses on aspects specific to TBB-NUMA, contrasting it to standard TBB where interesting.

4.1 Threads

Previous parallel frameworks (e.g., Intel OpenMP) allow the user to pin thread to cores, i.e., each thread is allowed to execute only at one *specific core*. TBB-NUMA automatically pins each thread to a *specific processor*. If a thread is pinned to a processor, the thread is allowed to execute at *any core of this specific processor*, but not at cores of any other processor. Threads are pinned to processors when they are created by the RML (worker threads) or when they register with the RML (master threads). Threads are distributed round-robin across the processors of a system (the first thread registered/created is pinned to Processor 1, the second to Processor 2, and so on); we assume all processors are identical with regard to number and capabilities of cores, thus the RML guarantees that there is an approximately equal number of threads pinned to each processor at any given point of time. TBB-NUMA is aware of the memory system's layout and threads are pinned to processors without user intervention.

The OS scheduler has fewer constraints with per-processor pinning than with per-core pinning, thus it can possibly balance load better if there are external (non-TBB) threads running on the system. If threads are not pinned, the TBB-NUMA runtime cannot give any guarantees to the layers above the threading layer, hence per-processor pinning is the minimal constraint that must be imposed on the OS scheduler to support NUMA-awareness.

4.2 Resource Management Layer

Similar to the RML in standard TBB, the TBB-NUMA RML distributes threads between all registered task schedulers. In addition to the standard TBB, the TBB-NUMA RML is aware of which processor each registered thread is pinned to and it distributes threads so that in each registered task scheduler there is an approximately equal number of threads from each processor. Let us assume an example program with two task schedulers running on a 2-processor 8-core system; there are 8 threads registered with the RML. In this case the RML assigns four threads to each task scheduler, with two threads pinned to Processor 1 and with two threads pinned to Processor 2. Distributing threads this way guarantees that each task

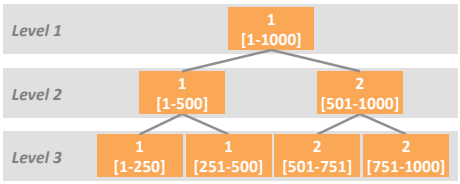
4′ The task removed from mailbox M_i, where the current thread is pinned to Processor i.

5′ The task removed from the task scheduler's shared queue i, where the current thread is pinned to Processor i.

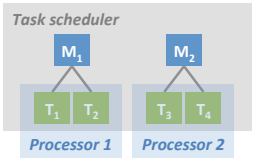
5″ The task removed from the task scheduler's shared queue 0 (queue w/o affinity for any processor).

5‴ The task removed from the task scheduler's shared queue k, where the current thread is pinned to Processor i and $i \neq k$.

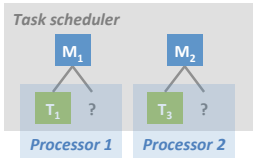
Figure 7. Rules substituted by TBB-NUMA to fetch next task (relative to standard TBB).



(a) Task tree with task-to-processor affinities (top) and partitioning (bottom).



(b) TS fully populated.



(c) TS partially populated.

Figure 8. Mailboxing (TBB-NUMA).

scheduler has access to all memory system resources (i.e., last-level caches, memory controllers, and cross-chip interconnects) and unfortunate assignments like that in Figure 2 are avoided.

4.3 TBB-NUMA Task Scheduler

Unlike in standard TBB, in TBB-NUMA the programmer can specify task affinities explicitly. Task affinities are *hints* in TBB-NUMA as well, but, unlike in standard TBB, affinities are *sticky* in TBB-NUMA. That is, the TBB-NUMA runtime is not allowed to modify a task's affinity when the task is executed on a different processor (i.e., a processor not originally intended by the programmer). To help the TBB-NUMA task scheduler still honor affinities (and balance load at the same time), the TBB-NUMA runtime implements a set of optimizations in addition to standard TBB. We first define the semantics of task affinities in TBB-NUMA, then we describe the optimizations to handle *scheduler configuration changes*, *steals*, and *revokes* (the reasons mentioned in Section 3.6 due to which the scheduler can ignore affinities).

4.4 Task-to-Processor Affinities

In TBB-NUMA tasks have affinity to a processor (instead of a mailbox as in standard TBB). That is, a task with an affinity value equal to i is not meant to be executed by the *single thread* connected to mailbox M_i as in TBB but by *any thread* running at Processor i. Thus, TBB-NUMA replaces Rule 4 of standard TBB (Figure 5) by Rule 4' (Figure 7). Because affinity values have a clear meaning backed by the TBB-NUMA runtime system, the programmer is allowed to use them (either directly via the task scheduler interface or indirectly by reusing parallel algorithm templates). To support per-processor task affinities, the number of mailboxes of a TBB-NUMA task scheduler is equal to the number of processors of the

```
1  void set_idle(affinity id, bool flag) {
2   mailbox[id].counter += flag ? 1 : -1;
3  }
4  bool is_idle(affinity id) {
5   bool threads_expected =
6        num_threads_active[id]
7            < num_threads_allotted[id];
8   return mailbox[id].counter > 0
9            || threads_expected;
10 }
```

Figure 9. Indicating idleness (TBB-NUMA).

machine (i.e., on a P-processor system there are P mailboxes). A task with an affinity value of i is inserted into mailbox M_i and, as the RML allows only threads pinned to Processor i to use this mailbox, the task is slated to be executed at the appropriate processor. Figure 8(b) shows the layout of a task scheduler populated with 4 threads (on a 2-processor system); two threads are pinned to each processor.

Figure 8(a) shows a NUMA-aware affinitization of a task tree (also for a 2-processor system). In the example the first half of the iteration space (iterations `[1-500]`) is mapped to Processor 1, the second half (iterations `[501-1000]`) is mapped to Processor 2. If data accessed by iterations `[1-500]` (`[501-1000]`) is allocated at Processor 1 (Processor 2), the computation has good data locality and thus good performance with TBB-NUMA.

4.4.1 Handling Configuration Changes (Reason 1)

TBB-NUMA handles the problem of changing task scheduler configurations by hardware-aware resource management. The TBB-NUMA RML allocates threads to task schedulers so that each scheduler has an approximately equal number of threads pinned to each processor. As a result, in every task scheduler the number of threads using each per-processor mailbox is approximately the same. Thus, every task scheduler has approximately the same share of each processor's computational and memory system resources. Figure 8(c) shows a task scheduler populated with two threads (two threads less than in Figure 8(b)). Each mailbox is served by one thread pinned to each processor and the affinitized task tree will execute with good data locality, just as when the task scheduler is fully populated (Figure 8(b)).

In some scenarios (when the number of thread schedulers registered with the RML is close to or is larger than the total number of threads registered with the RML) threads cannot be allocated to schedulers so that each mailbox is served by an equal number of threads. But as long as the number of registered schedulers is low (which is frequently the case in practice), it is possible to evenly distribute threads between task schedulers.

4.4.2 Handling Steals (Reason 2)

An affinitized task is present in two places: in the local deque of the thread that created it and in the mailbox it is sent to. Affinitization is successful if the task is removed from the mailbox by the thread connected to the mailbox. Affinitization is unsuccessful if the task is stolen by a thread that has no work to do and it has fallen back to random stealing (Rule 6) (the stealing thread has fallen through Rules 1–5 and obtains work according to Rule 6).

Standard TBB prevents a stealing thread from obtaining an affinitized task if there is a good chance that the task is going to be removed from the destination mailbox soon. Before stealing an affinitized task, each thread checks if the destination mailbox of the task is *idle*. If the mailbox is idle, the stealing thread *bypasses* the mailbox and tries to obtain a task from some other thread.

A mailbox is marked as idle in two cases: (1) when a thread falls through dequeuing from its local deque (Rule 3), but has not yet peeked at its mailbox yet (Rule 4), or (2) the thread connected

to the mailbox has left the task scheduler. In the first case bypassing is well-justified because the task will be received in a short time by the thread connected to the mailbox. The second case needs more explanation. A thread leaves the task scheduler when there is no work available for it. Alternatively, the RML can revoke the thread from the current task scheduler and then assign it to some other task scheduler. If the thread is associated with the current task scheduler again, it will empty its mailbox and the task will be executed at the intended location. But if the thread is permanently assigned to some other task scheduler, the task will be revoked (executed locally) by the thread spawning it and the task's affinity will be updated (which conforms with the TBB principle of non-constant task affinities).

In TBB-NUMA task affinities are immutable (by design), thus the idling mechanism of standard TBB must be revised. If a task is executed repeatedly, due to the constant affinities, a task with affinity value i will be submitted to the same mailbox M_i over and over again. If there are no threads connected to mailbox M_i, other threads in the scheduler will bypass mailbox M_i (assuming the idling mechanism of standard TBB). Bypasses reject work thus they can result in a high performance penalty. The idling mechanism of standard TBB must be updated also because TBB-NUMA allows multiple threads attached to a single mailbox.

TBB-NUMA uses an idling mechanism that is tightly coupled with resource management. The idling mechanism of TBB-NUMA (implemented by the `set_idle()` function in Figure 9) is based on incrementing/decrementing a counter. A thread increments the counter before receiving from its mailbox and decrements it after it has received a task. Unlike with standard TBB, with TBB-NUMA a thread does not indicate idleness when it leaves the task scheduler. To allow stealing those tasks that are not likely to be picked up at their destination mailbox and thus provide good load balance, the `is_idle()` function inspects both the counter and the number of threads allocated/active in the destination's mailbox, which avoids unnecessary bypasses.

4.4.3 Handling Revokes (Reason 3)

Affinitized tasks are inserted into the local deque of the thread that creates them. An affinitized task is *revoked* if the creator thread retrieves it (Rule 3) before it can be received at the destination processor. Unlike Rule 6, Rule 3 does not bypass affinitized tasks (to guarantee that each task is eventually executed before the program terminates). TBB-NUMA attempts to avoid revokes in two ways: by controlling task submission order via reflection (in case of wide task trees) and by detaching subtrees (for shallow trees).

Controlling task submission order In wide task trees each task (except leaves) has at least two children tasks. When unfolding wide trees, it is beneficial to submit tasks affinitized for the current processor last (the processor where the thread unfolding the tree executes). As Rule 3 retrieves tasks in LIFO order, tasks enqueued earlier have a chance to be picked up for execution at their destination thread before the creator thread revokes them.

E.g., when unfolding level 2 of the task tree shown in Figure 8(a), the `parallel_for` template spawns the right subtree and continues executing the left subtree if the current processor is Processor 1; otherwise it spawns the left subtree and continues executing the right subtree (assuming a 2-processor system). To control task submission order the creator thread must determine its current processor. The library supports this kind of reflection through the `task_scheduler_init::get_current_cpu()` call. This call is used internally by the `parallel_for` template. Code using the task scheduler directly can also rely on this reflection-based capability to control submission order.

Detaching subtrees Figure 10 shows a shallow task tree. E.g., the pipeline algorithm template of TBB generates shallow subtrees:

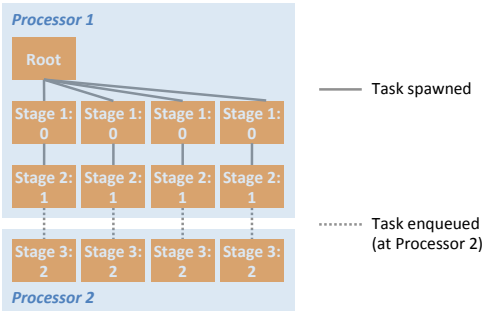


Figure 10. Shallow tree: 2-stage pipeline with affinities.

The pipeline template generates a distinct subtree for each input element processed by the pipeline; each task in a subtree corresponds to a different pipeline stage. The task tree shown in Figure 10 corresponds to a 3-stage pipeline computation.

In a shallow task tree each task (except the root task) has only one child task that is executed next by the task scheduler (according to Rule 3). If the memory accesses of a pipeline computation are dominated by accesses to input elements, shallow task trees can be beneficial for L1/L2 cache locality, because tree shallowness guarantees that each input element is processed by the same thread (thus the input element is in the cache used by this thread). In some cases, however, a child task predominantly accesses data other than the input element it processes. Moreover, in some cases the child task's accesses do not hit in the L1/L2 cache and are served by last-level caches (or even by DRAM). In these cases it can be beneficial to schedule the child tasks at threads executing at well-defined processors to achieve good last-level cache/DRAM data locality.

E.g., in the pipeline computation in Figure 10, Stage 1 of the pipeline is not associated with any processor (its task-to-processor affinity is 0), but Stage 2 accesses data associated with Processor 1 and Stage 3 accesses data associated with Processor 2 (Stage 2 and Stage 3 have a task-to-processor affinity value of 1 and 2, respectively). Spawning affinitized tasks and sending them to the mailbox of the appropriate thread does not help in this case because the affinitized task will be revoked (Rule 3 has priority over Rule 4). To allow a child task to execute at the processor it is associated with, the child task must be *detached* from its parent task, that is, it must be sent to the destination processor without inserting it into the local queue.

Standard TBB facilitates detaching tasks through the `enqueue()` call. *Enqueued tasks* are inserted into a queue shared by all threads in a task scheduler, threads receive enqueued tasks according to Rule 5. In standard TBB enqueued tasks are not allowed to have affinities to threads. TBB-NUMA extends standard TBB by allowing enqueued tasks to have affinities as well: the TBB-NUMA task scheduler has $P + 1$ shared queues Q_i (assuming a P-processor system), tasks with affinity for Processor i are enqueued at Q_i, $1 \leq i \leq P$, tasks with no task-to-processor affinity defined are enqueued at Q_0, thus Rule 5 of TBB is replaced by a set of rules in TBB-NUMA (Rule 5′-Rule 5‴ in Figure 7).

To illustrate how enqueuing handles revokes, let us consider the 3-stage pipeline example again (Figure 10). Let us assume that the root task runs at Processor 1 (as shown in the figure). Stage 1 has no task-to-processor affinity, thus the root task unfolds Stage 1 tasks using spawns. Let us assume that Stage 1 tasks are then also executed at Processor 1. The next stage (Stage 2) has affinity for Processor 1, but as all Stage 1 tasks are already running at Processor 1, Stage 2 tasks do not have to be detached (thus they are spawned). When, however, the task tree is unfolded further (i.e., Stage 3 tasks are created), these tasks have affinity for Processor 2

```
int size = lastrow - firstrow + 1;;
data_distribution dd(size);
affinity_partitioner ap;
bool UseNUMA;

// Performance-critical computation
class Loop3 {
public:
 void operator() (blocked_range<int>& r) {
  double sum;
  for (int j = r.begin(); j < r.end(); j++) {
   sum = 0.0;
   for (int k = rowstr[j]; k < rowstr[j + 1]; k++)
    sum = sum + a[k] * p[colidx[k]];
   w[j] = sum;
  }
 }
};

void conj_grad() {
 // ...
 for (cgit = 1; cgit <= cgitmax; cgit++) {
  // ...
  parallel_for(blocked_range<int>(1, size + 1),
         Loop3(),
         UseNUMA ? dd : ap);
  // ...
 }
}

int main() {
 if (UseNUMA) {
  dd.enforce(a);
  dd.enforce(colidx);
 }
 for (int it = 1; it <= NITER; it++) {
  conj_grad();
  // ...
 }
}
```

Figure 11. Example of using TBB-NUMA: `cg`.

(a processor different from the current processor), thus Stage 3 tasks are not spawned but enqueued (with affinity for Processor 2). Threads at Processor 2 will then dequeue these tasks (Rule 5′) and each stage is executed where the programmer originally intended. Threads at Processor 1 (the threads that originally unfolded the upper levels of the task tree) in the meantime unfold new subtrees to process any remaining input elements.

The decision whether to spawn or to enqueue tasks when unfolding a task tree depends on the processor the current thread is pinned to. The TBB-NUMA pipeline template uses reflection to determine the current thread's processor. E.g., if the root task of the example in Figure 10 executes at Processor 2 instead of Processor 1, enqueuing is used already when unfolding Stage 2. Finally, similar to the affinity of mailboxed tasks, the affinity of enqueued tasks is a *hint* on the preferred place of execution, that is, if a thread cannot get a task from the shared queue associated with its processor (i.e., Rule 5′ fails), the thread tries all other queues in the task scheduler (i.e., it falls back to Rules 5″ and 5‴).

4.5 Programming with TBB-NUMA

TBB-NUMA extends TBB, that is, the programmer can define which rules the task scheduler uses, the rules of standard TBB or rules specific to TBB-NUMA. If TBB-NUMA is enabled, the `parallel_for` algorithm template can be used with an additional parameter, a data distribution object that specifies the distribution of data for the iteration space processed by the loop. Figure 4.5 shows an example program, `cg`, in which the `parallel_for` template is used with a data distribution object (see Section 5.1 for more details about benchmark programs). TBB-NUMA includes a set of predefined data distributions (e.g., the block-cyclic distribution shown in Figure 8(a) for a 2-processor system). If needed, the programmer can define custom data distributions: The `parallel_for` template interfaces with data distributions through a single method; this method is used to determine the affinity of a subrange of the iteration space when the task tree corresponding to the iteration space is unfolded. The `pipeline` template allows specifying the per-processor affinity of pipeline stages (see example usage in case of the `ferret` benchmark in Figure 4.5). Finally, with TBB-NUMA the semantics of task affinities is clearly defined (task-to-processor affinity), thus the programmer can use task affinities directly with the task scheduler interface (see example in Figure 4.5).

```
data_distribution dd(image_database.size());
bool UseNUMA;

class QueryIndex : public filter {
 affinity_id _affinity;
public:
 QueryIndex(affinity_id affinity)
  : _affinity(affinity) {};

 void* operator(void* vitem) {
  if (_affinity != NO_AFFINITY)
   // query entire image database
   image_database.query();
  else
   // query partition of image
   image_database.query_partition(_affinity);
 }
};

int main() {
 if (UseNUMA)
  dd.enforce(image_database);
 // ...
 if (!UseNUMA)
  pipeline.add_filter(new QueryIndex(NO_AFFINITY));
 else
  for (int i = 1; i <= get_num_cpus(); i++)
   pipeline.add_filter(new QueryIndex(i));
 // ...
 pipeline.run();
}
```

Figure 12. Example of using TBB-NUMA: `ferret`.

In addition to specifying hints on the schedule of computations, TBB-NUMA defines helper functions to enforce data distributions on memory regions as well. Data distributions are enforced through memory migrations (e.g., through the `move_pages()` system call in Linux). Both data distributions and computation schedules depend on the actual hardware configuration. TBB-NUMA determines the number of processors at runtime and passes on this information to user programs. As a result, programs can be parametrized for a generic NUMA system and are thus portable.

5. Evaluation

The evaluation presented in this section attempts to answer three questions: (1) do optimizations improve data locality and performance (Section 5.2), (2) are optimizations composable (Section 5.3), and (3) are optimizations portable (Section 5.4).

5.1 Experimental Setup

Three NUMA systems are used to run experiments (see Table 1). We did a detailed performance evaluation on how remote memory accesses affect the performance of programs from the NAS and PARSEC benchmark suites on these systems. For the paper, we select five programs for which remote memory accesses cause significant performance degradation. We include in the selection programs that use different forms of parallelism: two programs

```
1  data_distribution dd(grid.size());
2  bool UseNUMA;
3
4  template<class WorkerType>
5  class GridLauncher: public task {
6  public:
7   task* execute() {
8    task_list list;
9    affinity_id affinity;
10   for (int i = 0; i < grid.num_partitions(); i++) {
11    WorkerLauncher<WorkerType> &c =
12     *new(task::allocate_child())
13          WorkerLauncher<WorkerType>(i);
14    if (UseNUMA)
15     c.set_affinity(grid.affinity_for_partition(i));
16    list.push_back(c);
17   }
18   spawn(list);
19  }
20 };
21
22 void AdvanceFrame() {
23  GridLauncher<ClearParticlesWorker>& cp =
24   *new (task::allocate_root())
25   GridLauncher<ClearParticlesWorker>();
26   task::spawn_root_and_wait(cp);
27
28  // create grid for other worker types
29 }
30
31 int main() {
32  if (UseNUMA)
33   dd.enforce(grid);
34  for(int i = 0; i < frameNum; i++)
35   AdvanceFrame();
36 }
```

Figure 13. Example of using TBB-NUMA: `fluidanimate`.

are loop-parallel, two programs use the task scheduler interface directly, one program is based on pipeline parallelism (see Table 2). For some benchmarks, small modifications are needed to make benchmarks amenable for NUMA optimizations. These modifications are described in detail in [20]. For the `ferret` benchmark we use the TBB-port described in [29].

Table 1. Hardware configuration.

	Intel E7-4830	Intel E5520	AMD 6212
Microarchitecture	Westmere	Nehalem	Bulldozer
# of processors/cores	4/32	2/8	4/16
Main memory	4x16 GB	2x24 GB	4x32 GB
Cross-chip interconnect	QPI	QPI	HT
Last-level cache	4x24 MB	2x8 MB	4x8 MB

Table 2. Benchmark programs from PARSEC (P) and NAS (N).

Program	Input	Suite	Par. algorithm template used
`cg`	input size C	N	`parallel_for`
`mg`	input size C	N	`parallel_for`
`streamcluster`	10M input points	P	none
`fluidanimate`	500K particles/500 frames	P	none
`ferret`	database (700M images)/ 3500 input images	P	`pipeline`

5.2 Data Locality Optimizations

Performance in NUMA systems depends on two aspects: the data distribution policy used and the policy used to schedule computations. We evaluate performance for a series of different execution scenarios; an execution scenario is defined by the pair (data distribution policy, computation schedule policy) used. Two scenarios consecutively listed in the evaluation differ in only one aspect, that is, they differ either in the data distribution policy used or the computation schedule policy used, but not both. Optimizations for loop-parallel programs are evaluated in five scenarios:

(noap, FT) Default version of the program that does not use task affinitization and uses the **f**irst-**t**ouch page placement policy (default in many OSs incl. Linux).

(noap, INTL) Default version of the program with the **int**er**l**eaved page placement policy in place. The interleaved page placement policy distributes pages across processors in a round-robin fashion. Interleaved page placement is recommended for source-level optimizations by [18] and is used in automatic systems as well [8]. Interleaved page placement improves performance by reducing contention on memory interfaces, but it does not reduce the number of remote memory accesses. In many systems (incl. those in Table 1) interleaved placement is equivalent to disabling NUMA in the BIOS.

(ap, INTL) The loops of the program are affinitized with the TBB-standard `affinity_partitioner` [30]; pages are placed interleaved, as in the previous configuration. The `affinity_partitioner` is designed to improve cache performance. This configuration shows the benefits of using this partitioner.

(NACS, INTL) The previous configurations can be obtained with standard TBB, this configuration is achievable only with TBB-NUMA. This configuration uses **N**UMA-**A**ware **C**omputation **S**cheduling (NACS), that is, the task scheduler is given hints about the distribution of data in memory. Normally, NACS effects both caching and DRAM data locality. However, to assess how NACS effects *caching only*, data distribution is not enforced in this configuration. Instead, interleaved placement is used, thus this configuration differs only in one parameter (the schedule of computations) from the previous configuration.

(NACS, NADD) **N**UMA-**A**ware **D**ata **D**istribution is enforced (in addition to NACS in the previous configuration). The results in this configuration show the benefits due to both cache and DRAM data locality.

The evaluation in this section uses the 4-processor 32-core Westmere system (see Section 5.4 for evaluation on the other systems). Figure 14(a) and 14(b) show the relative execution time of `cg` and `mg` in all five configurations. Execution time is relative to the `(noap, FT)` configuration, which has a relative execution time of 1. Lower relative execution time means better performance.

Relative to the best-performing configuration achievable with standard TBB, `(ap, INTL)`, `cg` improves 18% (relative execution time of 0.34 with `(NACS,NADD)` vs. relative execution time of 0.4 with `(ap, INTL)`. The computation time of `mg` improves around 12%, but its overall performance is slightly worse than the best configuration achievable with standard TBB because the cost of data migration (distributing data) cancels the improvement in computation time.

To show that performance optimizations improve both cache- and DRAM locality, we measure the number of uncore transfers a program generates in each of the five examined configurations. There are four types of uncore accesses: local cache/DRAM accesses and remote cache/DRAM accesses. Figures 15(a) and 15(b) show the uncore traffic breakdown of `cg` and `mg`, respectively. In the `(NACS, NADD)` configuration almost all remote memory accesses are eliminated (relative to both the baseline `(noap, FT)` and the `(ap, INTL)` configuration).

Figures 14(c)–14(e) show the performance of the remaining three, non–loop-based programs. Only loop-based programs can use the `affinity_partitioner`, thus the `(ap, FT)` and `(ap, INTL)` configurations are invalid for non–loop-based programs. As

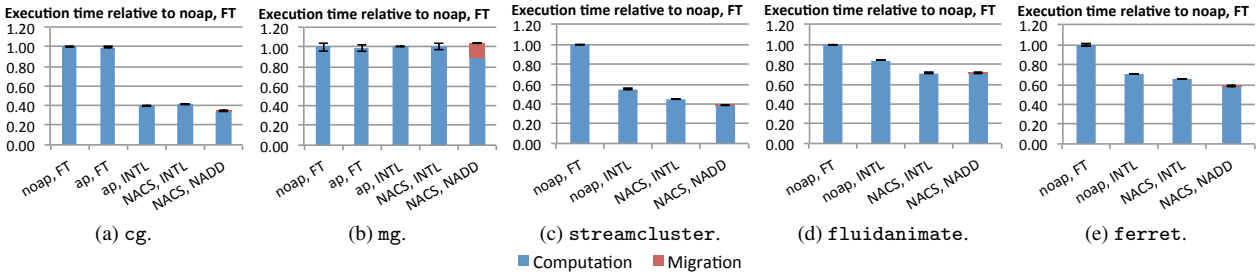


Figure 14. Performance (end-to-end execution time) **w/o contention** (Westmere).

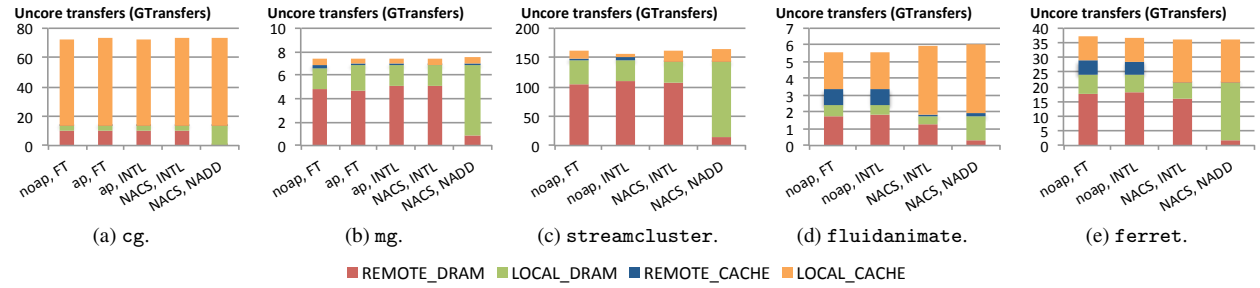


Figure 15. Uncore raffic **w/o contention** (Westmere).

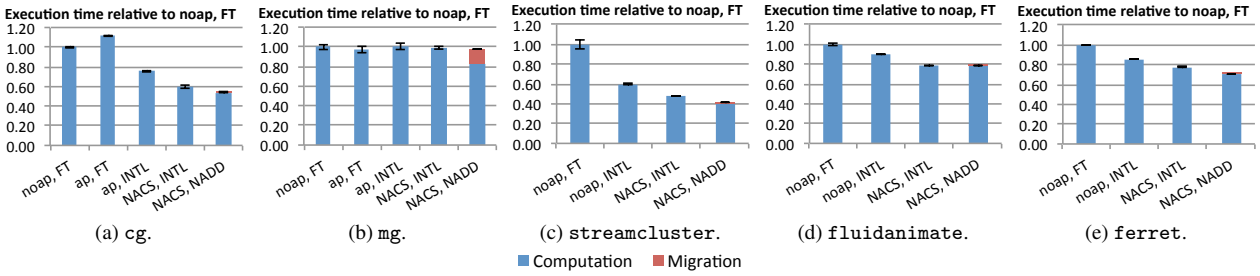


Figure 16. Performance (end-to-end execution time) **w/ contention** (Westmere).

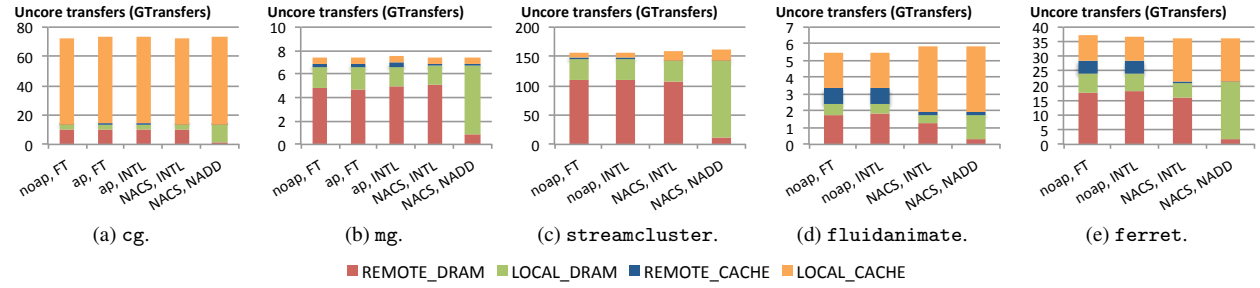


Figure 17. Uncore traffic **w/ contention** (Westmere).

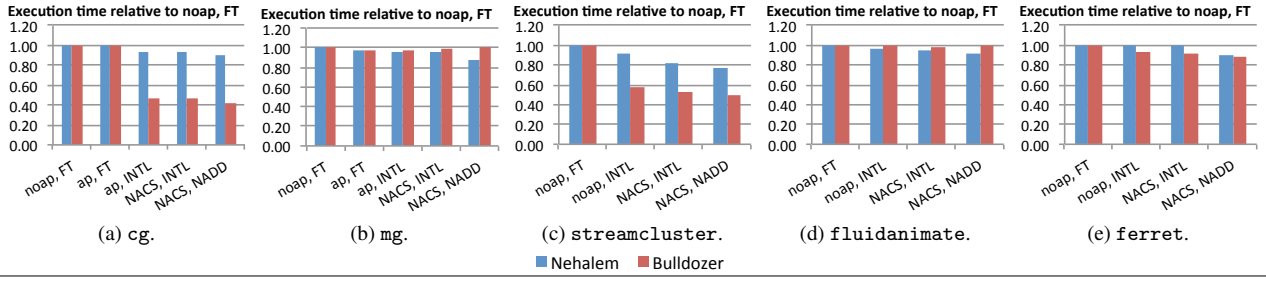


Figure 18. Performance (end-to-end execution time) **w/o contention** (Nehalem and Bulldozer).

a result, non-loop-based programs are evaluated in four instead of five configurations: the invalid configurations are replaced by the (`noap, INTL`) configuration (version of the program with no affinities specified, interleaved page placement policy). The principle that two subsequently listed configurations change only in a single parameter still holds after this change. Similar to loop-based programs, the two last configuration scenarios shown in the figures can be realized only with TBB-NUMA. For the non–loop-based programs NUMA-aware memory system optimizations result in performance improvements between 16–44% over the best possible configuration achievable with standard TBB (e.g., in case of `streamcluster` relative execution time of 0.38 with (`NACS, NADD`) vs. relative execution time of 0.55 with (`noap, INTL`), the best configuration that can be realized with standard TBB). Figures 15(c)–15(e) show the breakdown of uncore traffic for all three programs in all configurations. Memory system optimizations reduce the number of remote accesses for these programs, too.

We use the benchmark programs with large data sets. The results in Figure 15 show: (1) For some of the programs cache locality does not matter much as data sets do not fit even into the large L3 caches of the system (i.e., almost all uncore transfers are from local/remote DRAM); (2) for some programs TBB-NUMA increases L3 cache locality (i.e., the fraction of L3 transfers); (3) using the `affinity_partitioner` from standard TBB (where available) does not increase L1/L2 cache locality (equivalent to reducing the amount of uncore transfers). For smaller data sets, the `affinity_partitioner` improves performance by increasing L1/L2 cache locality, as reported by [21], but for large data sets, TBB-NUMA results in better performance than TBB.

5.3 Composability

This section evaluates how the properties of memory system optimizations are preserved when only a part of the hardware is available for executing an optimized computation (a scenario that occurs when an optimized computation is combined with other computations to form a larger application). Each benchmark is executed concurrently with a contender computation. The contender computation is parallelized and demands all threads from the RML (just like the benchmark program it is co-run with). As a result, the contender computation and the benchmark program contend for RML threads and the RML divides threads between the benchmark program and the contender program. This setup is similar to the scenario shown in Figure 4(b), with the difference that only two task schedulers (TS) but no OpenMP runtime (OMP) use the RML.

The contender computation is floating-point intensive and its working set fits into the private L1/L2 caches of the cores. As a result, uncore transfers measured are predominantly caused by the benchmark program (and not by the contender computation).

Figures 16(a)–16(e), show the performance of all benchmark programs in all relevant configurations, Figures 17(a)–17(e) show the breakdown of uncore traffic corresponding to each program/-configuration. Performance is reported as execution time relative to the (`noap, FT`) configuration with enabled contender. For each program/configuration performance results and the breakdown of uncore traffic is similar to the corresponding case with no contention. We record minor differences in relative performance numbers and uncore traffic because with contention there is a different amount of per thread cache capacity available to programs than without contention. In conclusion, the TBB-NUMA runtime preserves the properties of memory system optimizations even if only a part of the hardware is available.

The experiments described in this section characterize the performance of optimized computations when there is contention for the RML (due to both the benchmark program and the contender computation requesting threads from the RML at the same time). Co-executing computations can contend for memory system resources (e.g., for cache capacity and main memory bandwidth) as well [9], but in the experiments we describe they do not, as the floating-point intensive contender computation generates an insignificant amount of memory system traffic. Runtime systems are in a good position to mitigate contention for memory system resources, as shown by Dey et al. [10]. This paper, however, focuses only on preserving the properties of NUMA-optimized code when there is contention on the threadpool.

An interesting aspect is that using the `affinity_partitioner` with `cg` causes a slowdown under contention (Figure 16(a)). The RML is shared with the contender computation and threads frequently "migrate" between the two computations (i.e., threads previously assigned to the task scheduler running `cg` are frequently reassigned to the task scheduler running the contender computation and vice versa). When a thread leaves a task scheduler (because the RML assigned the thread to an different task scheduler), the mailbox of the thread (i.e., the mailbox in the task scheduler the thread was previously connected to) is marked as idle. Affinitized tasks present in a mailbox that is marked idle are not removed by stealing threads (i.e., treads looking for work; see Section 4.4.2 for further details). Instead, the tasks are kept in the mailbox until the thread that created them can process them.

In summary, if threads frequently migrate between task schedulers, stealing threads often reject work (i.e., reject to execute tasks) by bypassing mailboxes in which work (i.e., affinitized tasks) is present. As a result, processor cores are often idle, which results in a slowdown in the case of the `cg` benchmark. The TBB-NUMA task scheduler is coupled with resource allocation that reduces the number of (unnecessary) bypasses.

5.4 Portability

To show that memory system optimizations are portable, we run the same set of programs on two additional systems. As memory system optimizations are implemented for a generic NUMA system, the programs are executed on these systems without modification. Performance results are shown in Figures 18(a)–18(e); the variation of the measurement readings is negligible. On the Nehalem, optimizations result in 3–18% performance improvement over the best configuration that can be realized with standard TBB. On the Bulldozer we measure 6–18% improvement (and no improvement (`fluidanimate`) or a 3% slowdown (`mg`)).

6. Related Work

TBB-NUMA uses the `concurrent_queue` of standard TBB to implement per-processor mailboxes in the task scheduler. Recent work proposes NUMA-aware queuing and locking techniques [11, 13, 23], and wait-free queues have also been developed [17]. Although the TBB `concurrent_queue` is highly optimized, it is neither NUMA-aware, nor wait-free, thus TBB-NUMA could profit from the previously mentioned techniques by using them to enqueue/dequeue tasks more efficiently. The goal of TBB-NUMA (and the focus of this paper) is, however, to optimize the memory system performance of the tasks executed by the work-stealing system and not the queuing itself, therefore we leave the investigation of using NUMA-aware queues with TBB-NUMA to future work.

Previous work on NUMA memory system optimizations [6, 8, 18, 34] relies on three mechanisms: profile-based data migration, interleaved page placement, and data replication. In TBB-NUMA data distributions are set up by programmer-controlled data migration that achieves good data locality without profiling overhead. In Section 5 we compare the performance of TBB-NUMA-based optimizations to interleaved page placement. Data replication has several disadvantages, the most important of them is that replicas must be kept consistent, which causes overhead. To limit the overhead,

state-of-the-art systems disable page replication after a small number of detected writes [8]. Most benchmarks used for evaluation in this paper frequently read-write performance-critical memory regions, thus we do not include data migration in the evaluation.

Several runtime systems (e.g., Lithe [26], Microsoft's ConcRT, and Poli-C [4]), support composable parallel software, but none of these systems is designed to preserve the data locality of NUMA-optimized code. The Callisto resource management layer [16] reduces scheduler-related interference between multiple, independent parallel runtime systems co-executed on a single machine, but it does not consider interference on the memory system level (incl. NUMA-related aspects). There are several approaches to improve the locality of work stealing [7, 14], but [7] focuses only on improving cache utilization and not on reducing the number of remote memory accesses; [14] supports data locality optimizations, but balances load individually within the scope of each processor (and not between all processors of a system, as TBB-NUMA does). Space-bounded schedulers are known [31] to give better data locality than work-stealing schedulers (with the cost of higher scheduling overheads). TBB-NUMA keeps scheduling overhead low while favoring locality.

7. Conclusions

TBB-NUMA supports *portable* and *composable* software for NUMA systems by defining the semantics of thread affinity. TBB-NUMA is based on Intel TBB to demonstrate the practicality of this approach (and to allow a programmer to decide when to use NUMA-specific functionalities). TBB-NUMA provides a unified interface to the runtime system and allows memory-system-aware resource management.

There exist several tools to provide information about NUMA performance bottlenecks, but programmers so far lack a unified way to control the execution of parallel programs on NUMA systems. TBB-NUMA allows the programmer to pass directives (based on insights and/or performance monitoring information) about computation *and* data placement to the runtime system. With NUMA systems increasing in size we expect the gap between local and remote memory accesses to increase as well, thus we expect data locality optimizations to be even more important in the future.

Acknowledgments

We thank Michael Stumm, Frank Müller, Yves Geissbühler, Albert Noll, and the anonymous referees for their helpful comments and acknowledge computing resources provided by SNF grant 206021_133835.

References

[1] *OpenMP Application Programming Interface, Version 3.1*, July 2011.

[2] *Intel(R) Threading Building Blocks Reference Manual*, 2012.

[3] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA'00*.

[4] Z. Anderson. Efficiently combining parallel software using fine-grained, language-level, hierarchical resource management policies. In *OOPSLA'12*.

[5] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore processors. In *USENIX ATC'11*.

[6] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *SOSP'89*.

[7] Q. Chen, M. Guo, and Z. Huang. CATS: cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *ICS'12*.

[8] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth. Traffic management: A holistic approach to memory placmeent on NUMA systems. In *ASPLOS'13*.

[9] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *ISPASS'11*.

[10] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. ReSense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Trans. Archit. Code Optim.*, 2013.

[11] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA locks. In *SPAA'11*.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI'98*.

[13] E. Gidron, I. Keidar, D. Perelman, and Y. Perez. SALSA: scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. In *SPAA'12*.

[14] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *IPDPS'10*.

[15] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *MICRO'09*.

[16] T. Harris, M. Maas, and V. J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *EuroSys'14*.

[17] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *PPoPP'11*.

[18] R. Lachaize, B. Lepers, and V. Quéma. MemProf: a memory profiler for NUMA multicore systems. In *USENIX ATC'12*.

[19] X. Liu and J. Mellor-Crummey. A tool to analyze the performance of multithreaded programs on NUMA architectures. In *PPoPP'14*.

[20] Z. Majo and T. R. Gross. (Mis)Understanding the NUMA memory system performance of multithreaded workloads. In *IISWC'13*.

[21] J. Marathe, V. Thakkar, and F. Mueller. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *J. Par. Distrib. Comput.*, 2010.

[22] C. McCurdy and J. S. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *ISPASS'10*.

[23] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *PPoPP'13*.

[24] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. A case for user-level dynamic page migration. In *ICS'00*.

[25] T. Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *OOPSLA'09*.

[26] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. In *PLDI'10*.

[27] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *EuroSys'10*.

[28] C. Reddy and U. Bondhugula. Effective automatic computation placement and dataallocation for parallelization of regular programs. In *ICS'14*.

[29] E. C. Reed, N. Chen, and R. E. Johnson. Expressing pipeline parallelism using TBB constructs: A case study on what works and what doesn't. In *SPLASH'11 Workshops*.

[30] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *IPDPS'08*.

[31] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola. Experimental analysis of space-bounded schedulers. In *SPAA'14*.

[32] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys'07*.

[33] M. M. Tikir and J. K. Hollingsworth. Hardware monitors for dynamic page migration. *J. Par. Distrib. Comput.*, 2008.

[34] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *ASPLOS'96*.

[35] 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*.