Journal of Universal Computer Science, vol. 21, no. 6 (2015), 816-841
submitted: 29/11/14, accepted: 20/2/15, appeared: 1/6/15 © J.UCS

Towards Efficient Locality Aware Parallel Data Stream Processing

Zbyněk Falt
(Charles University in Prague, Czech Republic
falt@ksi.mff.cuni.cz)

Martin Kruliš
(Charles University in Prague, Czech Republic
krulis@ksi.mff.cuni.cz)

David Bednárek
(Charles University in Prague, Czech Republic
bednarek@ksi.mff.cuni.cz)

Jakub Yaghob
(Charles University in Prague, Czech Republic
yaghob@ksi.mff.cuni.cz)

Filip Zavoral
(Charles University in Prague, Czech Republic
zavoral@ksi.mff.cuni.cz)

Abstract: Parallel data processing and parallel streaming systems become quite popular. They are employed in various domains such as real-time signal processing, OLAP database systems, or high performance data extraction. One of the key components of these systems is the task scheduler which plans and executes tasks spawned by the application on available CPU cores. The multiprocessor systems and CPU architecture of the day become quite complex, which makes the task scheduling a challenging problem. In this paper, we propose a novel task scheduling strategy for parallel data stream systems, that reflects many technical issues of the current hardware. In addition, we have implemented a NUMA aware memory allocator that improves data locality in NUMA systems. The proposed task scheduler combined with the new memory allocator achieve up to 3× speed up on a NUMA system and up to 10% speed up on an older SMP system with respect to the unoptimized versions of the scheduler and allocator. Many of the ideas implemented in our parallel framework may be adopted for task scheduling in other domains that focus on different priorities or employ additional constraints.

Key Words: Parallel, multicore CPU, NUMA, cache aware, task scheduling, memory allocator

Category: H.2.2, H.2.6, H.3

1 Introduction

Parallel processing is becoming increasingly important in high performance systems, since the hardware architectures have embraced concurrent execution to increase their computational power. Unfortunately, parallel programming is much more difficult and error prone, since the programmers are used to think and express their intentions in sequential manner. Many different paradigms and concepts have been devised to simplify the design of concurrent processing.

In this paper, we focus on systems decomposed into processing stages connected by streams of data. This kind of decomposition naturally appears in data stream processing where the data are generated in real-time and real-time response is also required. Nevertheless, systems without real-time demands like database systems (OLAP in particular) may be decomposed similarly. Furthermore, many parallel frameworks offer design patterns based on pipelines or streams.

An application that employs stream-based decomposition is usually expressed as an oriented graph (in database systems usually denoted as *execution plan*), where the vertices are processing stages (also called operators, filters, or kernels) that process the data and the edges prescribe how the data are passed on between these stages. From the parallel computing point of view, the main advantage is that the stages may be executed concurrently even if each stage contains strictly serial code. Furthermore, individual operators may often be rewritten into subnetworks of operators acting on partitioned data. This approach moves the parallelization effort from the programming level to the plan-design level, which is usually more intelligible and manageable. On the other hand, the programmer loses the immediate control over the execution strategy and the performance of the parallel execution is determined mainly by the quality of the key runtime components, such as the task scheduler or the memory allocator.

One of the systems that implements this idea is Bobox [Bednárek et al., 2009]. Bobox was mainly designed to process OLAP queries on structured and semi-structured data effectively [Bednárek et al., 2009]. It currently supports SPARQL query language [Prud'Hommeaux et al., 2006] and partially XQuery language [Boag et al., 2002] and TriQuery language [Bednárek and Dokulil, 2010]. One of the most challenging problems of this system is to effectively and efficiently plan and execute the work of the operators on the available CPU cores.

In this paper, we propose a novel locality aware task scheduling strategy (called LAS) for data streaming systems. This strategy incorporates important hardware factors such as cache hierarchies and nonuniform memory architectures (NUMA). We have implemented this strategy in the Bobox task scheduler and achieved significant speedup on modern host systems. In order to design fully locality aware system, our scheduler is accompanied by NUMA aware memory allocator which manages virtually all data that are processed by Bobox. Al-

though our performance analysis was conducted using Bobox, the scheduler and the memory allocator can be easily adopted for other streaming systems as well.

The paper is organized as follows. Section 2 revise the most important facts regarding state-of-the-art CPU architectures and NUMA systems. The related work is revised in Section 3. Our locality aware scheduler is described in Section 4 and the memory allocator is described in Section 5. Section 6 presents the experimental results that evaluate the benefits of our innovations. Section 7 concludes the paper.

2 CPU Fundamentals and Task Scheduling

In this section, we revise fundamental facts regarding the architectures of modern multicore CPUs and NUMA systems. We also put these facts in the perspective of task scheduling and memory management employed in parallel data processing systems.

2.1 CPU Architecture

The CPU architectures became rather complex in the past few decades. We will focus solely on the properties which directly affect the parallel execution of tasks that cooperate via shared memory. A generic schema of a modern multicore CPU is presented in Figure 1.

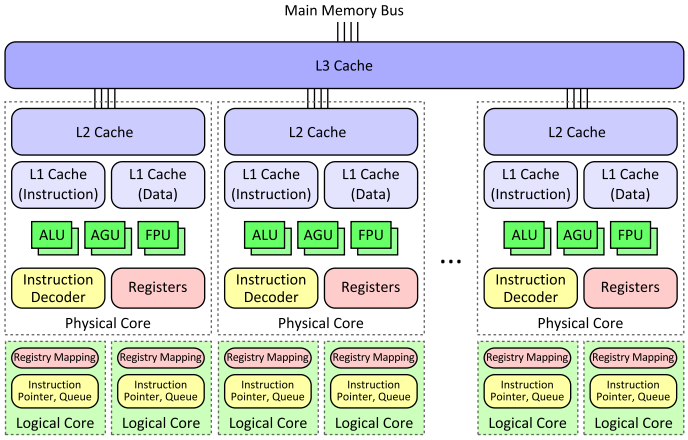

Figure 1: A generic schema of multicore CPU

A mainstream CPU comprises several physical cores which are equivalent and quite independent. These cores usually share only the memory controller and sometimes certain levels of cache. The physical cores are often divided into two

or more logical cores by means of Hyper-threading (Intel) or Dual-core modules (AMD) technology. The logical cores share most of the units of the physical core, but they can make a better use of these units as they are rarely utilized all simultaneously by one sequence of instructions. In the remainder of the paper, we will use the term CPU core to denote logical core – i.e., the computational frontend of the CPU which processes one stream of instructions.

Current CPU cores can process data in much higher speeds than what can current DRAM chips accommodate. Therefore, a large portion of the processor chip is dedicated to the memory cache. Most of the present architectures employ three levels of cache (designated L1, L2, and L3), where L1 is the smallest and closest to computational units and L3 is the largest and slowest. In sequential processing, an algorithm can benefit from the processor caches when it exhibits a locality of reference – i.e., when it repeatedly access the same data or data that are close by in the memory space.

In parallel processing, the effect of the cache is twofold. On one hand, the cache becomes even more important as the combined instruction processing speed of the entire CPU is basically the sum of speeds of individual cores. On the other hand, the caches are usually shared by some of the cores (logical cores usually share the L1 cache of their physical core and physical cores often share the L3 cache of the whole CPU). As a result, the overall performance of an algorithm is not only affected by it data access pattern (e.g., whether it follows the locality of reference), but also by scheduling details such as whether two threads reading the same data run on CPU cores that do or do not share some level of cache.

2.2 Multiprocessor Systems

Multiprocessor (multi-socket) configurations combine several CPUs into one system. CPUs which do not have integrated memory controllers are usually connected in a similar way as the physical cores in the CPU die. This configuration is called symmetric multiprocessing (SMP). Multiprocessor systems, where the CPUs have the memory controllers integrated, are usually organized in nonuniform memory architecture (NUMA). In case of NUMA, each CPU manages part of the system memory, whilst the memory is shared among all processors in the system. The architecture is so called nonuniform, since the latency and the bandwidth of the host memory is not uniform over the whole address range. When a process accesses some data, the latency of such operation depends on the distance between the core where the process runs and the physical memory module where the data reside.

NUMA systems often suffer from first-touch memory allocation strategy and NUMA-unaware scheduling algorithms [Bailey, 2007, Lameter et al., 2013]. The first-touch physical memory allocation assigns physical memory frames to virtual

memory pages based on which NUMA node accesses the pages first. Unfortunately, this approach is based on an assumption that the computing thread that first accessed the memory will dwell on the same NUMA node and the allocated memory will be used primarily by that thread. However, the scheduler of the operating system may eventually decide to move the thread to another processor core on a different NUMA node to maintain the balance of the workload. In such case, the process will access all its allocated memory remotely by issuing internode transactions, which require data routing and which use interprocessor communication buses that have lower throughput than DRAM buses.

The described problem expresses itself more severely in case of multithreaded applications. A multithreaded application usually allocates memory in one thread (e.g., the main thread) and access the memory from multiple threads that cooperate on the same problem. If the application creates as many threads as there are logical CPU cores available in the system (which is typical behavior of most parallel frameworks [Jung, 2012, Chandra, 2001]), the threads will occupy NUMA nodes evenly, thus many of them will need to access the memory of different nodes, even if the scheduler maintains strict thread-to-core affinity.

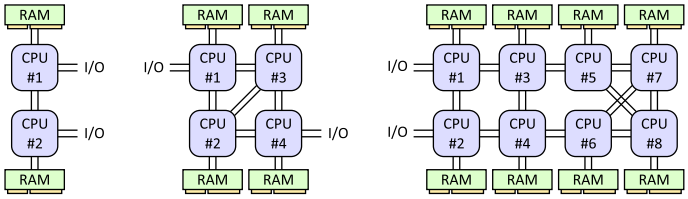

Figure 2: Examples of 2-way, 4-way, and 8-way NUMA configurations

Examples of NUMA configurations are presented in Figure 2. The topology of the processors in a NUMA system forms another level of hierarchy, which is similar to the hierarchy of the internal CPU cores based on the cache sharing. The latency of the memory is determined by the length of the shortest path between the node that access the data and the node, where the data are stored. CPU cores that share the same memory node are closest from the NUMA perspective, cores from neighboring NUMA nodes are considered to be one step further, and so on.

This hierarchy plays significant role in task planning. Related or cooperating tasks should be scheduled on cores that are near, since they will likely benefit from having the data in the same memory node or even in the same cache. Completely independent tasks should be scheduled on different NUMA nodes, so they can keep their intermediate data in different memory nodes and utilize the whole capacity of separate caches.

2.3 Task Scheduling

In order to achieve parallelism on modern CPUs, the work needs to be divided into portions that can be processed concurrently. Traditional division into threads is too coarse and tedious, hence most of the parallel systems deal with tasks. The *task* comprises both the input data and the procedure that process the data. The results of a task can either replace the input data (in-place modifications) or they can be stored in newly allocated buffer. Tasks are scheduled and processed by available CPU cores. It has been established [Reinders, 2007] that the tasks can be effectively employed in the implementation of more complex parallel patterns such as parallel loops, reduction, pipeline, or data stream processing.

In this work, we focus solely on systems where the tasks are generated dynamically by other tasks or by external events (e.g., user requests). Such systems must employ dynamic scheduling, which can cope with the ever changing situation. The dynamic scheduler manages the tasks which are ready to run and decides when to assign them to the CPU cores as they become available.

Furthermore, task schedulers often employ some form of restrictions for implicit synchronization like task dependencies. When a task is spawned, it may not be ready to execute immediately. In such case, the task scheduler needs to manage *waiting* tasks along with the *ready* tasks. When a waiting task conditions for execution are met, the scheduler change its state to *ready* and eventually assigns it to an available CPU core. However, we are focusing on highly data-driven parallel systems, which do not use the waiting tasks, but the tasks are spawned, when they are ready to be executed. Henceforth, we use the term *task spawning* for introducing ready tasks to the scheduler.

3 Related Work

3.1 Task Scheduling

As modern CPU architectures became quite complex, optimizing the performance of applications through elaborate task scheduling strategies is a challenging task and a very hot topic in current research. The fact that finding an optimal scheduling plan is a NP-hard problem causes that all scheduling strategies attempts to find a suboptimal solution using heuristics and approximation techniques [Sinnen, 2007].

In the streaming systems, there are several aspects of task scheduling optimization, such as memory usage [Babcock et al., 2004], efficient cache utilization [Cieslewicz et al., 2009], response time, throughput, or their mutual combinations [Jiang and Chakravarthy, 2004, Safaei and Haghjoo, 2010].

In this work, we relaxed many aspects and just tried to maximize data locality in order to increase the performance of the system. This allowed us to adopt techniques used in non-streaming systems. Our previous work [Bednárek et al., 2013] was the first step towards this goal. We showed that the data flow awareness (i.e., using immediate and deferred tasks) in streaming systems increase the data locality; however, the scheduling algorithm lacked the support of NUMA and nontrivial SMP systems.

Some of these issues were solved in several works, e.g., in popular parallel frameworks such as OpenMP [Duran et al., 2008, Broquedis et al., 2009] or Intel Threading Building Blocks [Kukanov and Voss, 2007, Jung, 2012]. The classification of tasks to immediate and deferred type improves the chance that the threads are working with data that are hot in the cache. However, it has been also established that the bottleneck of the system is the task stealing when the tasks are stolen from randomly chosen threads. Hence, we have addressed this issue in our work.

The task stealing optimization is researched thoroughly in the work of Chen et. al. [Chen et al., 2011, Chen et al., 2012]. In fact, the algorithm CATS/CAB from this work is similar to our LAS algorithm described in Section 4; however, there are several important differences between these two algorithms. LAS partitions physical processors more precisely according to the structure of shared caches, whilst CATS/CAB creates always one group per physical processor (socket). Furthermore, LAS algorithm for task stealing within a group also considers the cache hierarchy, which is beneficial when the cores in one group share more than the last level of cache. Additionally, the LAS sets affinity of threads together for the whole group. This has two advantages: First, we can freely add and remove threads to the thread pool which enables support of I/O operations [Kruliš et al., 2013]. Second, this strategy copes better with Hyper-Threading Technology, since it does not restrict the operating system from its own load balancing strategy [Pirasteh et al., 2015]. Finally, we optimize the situation when the system processes multiple independent requests, such as individual database queries.

3.2 Scalable Memory Allocation

Current memory allocators for multiprocessor systems should provide the following features if they are to excel in the terms of efficiency and performance:

- **Speed**: Allocator should exhibit good performance both in single threaded and in multi-threaded environment.
- **Scalability**: The performance of the allocator should scale (linearly) with the number of processors.

- **False sharing avoidance**: The allocator should not introduce false sharing of cache lines.
- **Low fragmentation**: The allocator should minimize its memory overhead. In other words, the allocator should optimize the ratio between the amount of memory claimed from the operating system and the amount of memory provided to the application.

Hoard [Berger et al., 2000] is one of the first and one of the most famous memory allocators for shared-memory systems. Hoard addresses all the requirements postulated above. It uses private per-processor private heap and one global heap. The memory of the operating system is claimed in chunks of the same size (which is a multiple of the system page size) called *superblocks*. Each superblock is an array of regular blocks and it maintains a list of its blocks that are still unallocated. The list is operated as a LIFO stack to improve cache reusability.

Memory allocations are served from the private heap. The allocator tries to accomodate a request by fetching a block from the most occupied superblock, which still has sufficient free space. The global heap is used only as a cache for unused superblocks. When a private heap could accomodate a request by its internal superblocks, it tries to claim a superblock from the global heap. If the global heap is empty, a new superblock is allocated from the operating system. When a private heap releases a superblock, it is returned back to the global heap.

Many scalable allocators based on techniques of Hoard has been developed, such as Google Thread-Caching Malloc [Google, 2014], ptmalloc [Jung, 2011], TLSF allocator [Masmano et al., 2004], or Miser [Tannenbaum, 2014]. Although these allocators try to optimize some particular scenarios, an experimental study [Ferreira et al., 2011] indicates that these optimizations do not significantly outperform Hoard in general. The Bobox allocator presented in our work is also based on the general ideas of the Hoard allocator. However, our objective is not to improve Hoard, but rather to utilize proven memory scheduling algorithm, which can be adopted for NUMA architectures and used along with our task scheduler.

The creators of the MAMA system [Kahan and Konecny, 2006] demonstrated that the superblock approach has unacceptable memory overhead when the number of cores reaches the order of thousands (like in Cray's MTA). The MAMA system successfully reduced these demands and achieved linear space complexity and logarithmic time complexity with respect to the number of cores.

There are also different types of memory allocators that target different environments. For instance, McRT-Malloc [Hudson et al., 2006] targets transactional memory, Myrmics [Lyberis et al., 2013] optimizes Global Address Space runtime system, whereas X-malloc [Huang et al., 2010] aims for GPU/SIMD en-

vironment. Although these allocators are not applicable in our system, some of their ideas influenced the design of the Bobox scalable allocator.

4 Locality Aware Task Scheduler

In general, a task scheduler shall manage a list of ready tasks and assign them to available computational units in a way that promises the best approximation of the scheduling goal - the best utilization of resources, the greatest throughput, or the shortest latency. In real-life systems, the scheduler can never possess accurate prediction of the future behavior of the tasks; instead, it must derive its decisions from the attributes assigned to the tasks by their creators. In other words, the scheduler depends on the ability of the application to specify the relevant properties of the tasks being spawned. Thus, the target class of parallel applications must be defined and studied before the scheduler is designed.

In our work, we focused on systems which simultaneously process different *requests*, each request being composed of variable number of tasks. Different requests are assumed to share little or no data. Running the requests simultaneously may improve the utilization of computing resources, because the requests are independent. On the other hand, concurrent execution will degrade caching efficiency as the requests compete for the limited cache space. Consequently, finding a tradeoff between simultaneous and sequential execution of requests becomes an important part of the scheduling strategy.

Furthermore, we assume that the main scheduling objective is the maximization of the overall throughput (i.e., achieving the highest number of requests finished per unit of time possible) while maintaining fairness (i.e., prioritizing the work of the requests according to the order in which they arrived). We also assume that our system is the sole user of a multiprocessor machine; thus, the number of available computational resources is stable.

Besides organizing tasks into requests, we also assume that the application is able to categorize the tasks being spawned into the following two types:

- *Immediate tasks* will be executed as soon as possible and preferably close to the task that spawned them. Therefore, they may utilize some data which are still hot in the cache (e.g., results of the spawning task). Immediate tasks can be used to express cooperative work or work continuation patterns.

- *Deferred tasks* will be executed as soon as there are free computing resources available. Therefore, data shared between the spawning and the spawned tasks are not likely to survive in the cache. Deferred tasks usually express detached portions of work, so they can be more easily stolen by an available (yet possibly distant) CPU core (even on a different NUMA node).

The choice between immediate and deferred type shall be based on the amount of data shared between the spawning and the spawned tasks compared to the total size of data accessed by the spawned task:

- If majority of the data accessed by the new task is shared with the old task, the execution of the new task will be strongly affected by the presence of the shared data in cache and, consequently, the new task shall be designated as immediate.
- If the shared data form only a small part of the working set of the new task, the new tasks will not significantly benefit from their presence in cache; thus, postponing the new task by designating it as deferred will not significantly hurt the performance.
- Tasks that represent initial part of a larger portion of compact work (i.e., tasks that are expected to spawn many other tasks) should be spawned as deferred, so they are more likely to be adopted by a more distant core.
- Initial tasks of new requests are spawned as deferred.

This scheduler architecture was initially designed for the use in parallel database systems where each request (query) is implemented by a network of operators (called execution plan) [Bednárek et al., 2013]. In these systems, quantization of the data into blocks together with the network structure defines the decomposition of requests into tasks. Hence, the distinction between immediate and deferred tasks is a natural consequence of the task dependencies in the system. Besides database systems, our concept may be applied in many task-based environments like TBB [Jung, 2012] with their continuation and child tasks.

Our locality aware task scheduler (LAS) works within the environment defined above and it aims to improve the overall throughput by the following design principles:

- Tasks that are likely to share data should be executed close by (both in the terms of time and cache/NUMA distance).
- Tasks created on the same CPU core or on the same NUMA node will benefit from caches more likely than tasks created on distant nodes.
- Tasks of different request will not benefit from caches.

4.1 Thread Pool

The cost of creating or destroying a threads is usually quite high in most operating systems. Parallel frameworks employ thread pool design pattern to avoid this cost. A pool of threads is created when the framework is initialized and the

threads are kept suspended on a synchronization primitive such as semaphore or mutex until they are needed.

The initialization process of our task scheduler scans the host system and detects the connection topology and properties of the CPUs. CPU cores which share at least one level of cache are bundled together in logical *core groups* and a separate *thread pool* is created for each group. Let us emphasize that a core group is always contained within one NUMA node and since the cores of one chip usually share the L3 cache, exactly one thread pool per NUMA node is created in most situations[1]. The thread pool holds one thread for each CPU core in the corresponding group and the threads have their affinity set to this core group. Hence, a thread from a pool will always be executed on one of the cores from its group, but the exact core from the group is selected by the operating system. The thread can easily determine its associated CPU core using appropriate operating system functions.

Each core in every group manages its own *queue of immediate tasks*. When an immediate task is spawned, it is inserted to the immediate queue of the core, where the spawning thread runs. Each core group manages one shared *queue of deferred tasks*. Analogically, when a deferred task is spawned, it is inserted to the deferred queue of the core group, where the spawning thread belongs to. The deferred queue is in fact a more complex data structure than a simple FIFO stack, since it maintains the task of each request separately. It is implemented as a priority queue of double-ended queues, where the priority queue is organized by *request ID*s and the inner queues hold tasks of individual requests. Since request IDs are assigned sequentially, the structure provides quick access and extraction of the youngest and the oldest tasks from the oldest and second oldest request.

4.2 Task Scheduling Strategy

The main paradigm employed in the LAS scheduling strategy is to emphasize data locality awareness. When a thread becomes available (completes a task), it invokes the scheduling algorithm which finds a new task for the thread. The scheduler attempts to find a task which is related (i.e., a task that shares a significant portion of data) to the previous work processed on the same CPU core where the thread resides. This approach maximizes the possibility of cache reuse and NUMA node locality.

Unfortunately, the required relation between tasks is not usually specified by the programmer, nor it can be easily inferred from the source code. There are only two important facts regarding a spawned task that we can utilize for scheduling: On which core the task was spawned and whether it was spawned

[1] However, there are some situations when CPU cores are organized in a way that some of them do not share any cache. In such rare cases, there are multiple thread pools per NUMA node created.

as immediate task or deferred task. These facts determine the task queue where a newly spawned task is inserted (as described in Section 4.1), so the scheduling algorithm is based on determining an appropriate task queue.

In order to better formalize the data locality in the perspective of task scheduling, we define a *cache distance* and a *NUMA distance* of each two cores in the system. The cache distance is equal to the lowest level of cache the two cores share. For instance, if the cores share L2 and L3 cache, their cache distance is 2. Additionally, if the cores do not share any cache, their cache distance is infinity. The NUMA distance is equal to the shortest path between the NUMA nodes, where the cores are situated. Cores from the same node have NUMA distance 0, cores from neighboring nodes have distance 1, and so on.

The task scheduling strategy is formalized as a set of conditional rules. The scheduling algorithm tries to verify the conditions of each rule in the prescribed order and use the first applicable rule. When such a rule is found, it is applied and it yields a task that is immediately executed by the available thread.

1. The immediate queue of the current core is tested. If it is not empty, the youngest task from the queue is taken. The definition of immediate tasks suggests that there are some data shared between the tasks in the queue and the tasks that spawned them. The youngest task was spawned last (possibly by the previous task), thus it has the best chance that the shared data are still hot in the cache.

2. The other cores of the same group are scanned in their increasing cache distance[2]. If a nonempty immediate queue is found, its oldest task is taken. This way the scheduler may find a task that still has some relevant data in the cache, but it also does not interrupt the most recent work of any other core since it steals the oldest task in the queue.

3. If the deferred queue of the corresponding core group is not empty, the youngest deferred task of the oldest request is taken from this queue. This rule ensures that all threads of one core group work on the same (the oldest) request if possible.

4. Other core groups are scanned (in increasing NUMA distance) until a non-empty deferred queue is found. If such queue exists, the oldest deferred task of the *second* oldest request is taken. If the queue contains tasks of only one request, its oldest task is taken instead. This strategy assumes that a core group is heavily engaged in the processing of the oldest request and it would not be wise to disrupt this work when another request is available. On the other hand, this rule does not prevent different NUMA nodes from

[2] Cores with the same cache distance may be scanned in any order.

cooperating on a single request in case there are not enough requests in the system to occupy all the nodes.

5. Immediate queues of cores from other groups which are located on the same NUMA node[3] are scanned. If nonempty queue is found, its oldest task is taken. The immediate queues are scanned in implementation defined yet deterministic order and the thread always remembers the last nonempty queue found. When this rule is applied again, the scan is resumed where it previously ended, so the core groups are tested evenly in a round robin manner.

If all steps fail (i.e., there is no available task to execute), the thread is suspended, so it will not consume the system resources. The suspended threads may be resumed when a new task is spawned. The resuming algorithm is described in the following section.

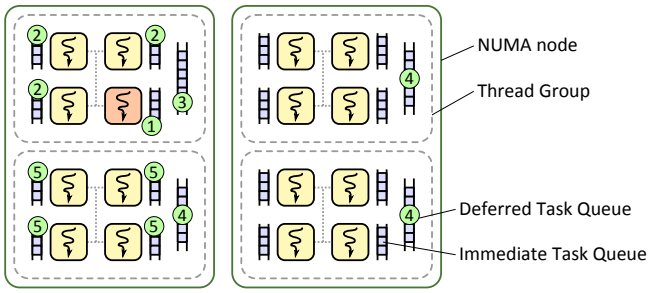

Figure 3: Schema of task queues and their relations to task scheduling

The whole algorithm and the thread group hierarchy is depicted in Figure 3. Let us summarize the main implications of our scheduling strategy. First of all, the strategy attempts to keep related tasks (i.e., tasks of one request) on a single NUMA node if possible. The first three rules does not even consider stealing a task from another NUMA node and the fourth rule attempts to steal a task of a request which is probably not being processed yet by another node. The immediate planning rules also respect the cache sharing hierarchy to maximize the chance that some data in the cache are utilized by multiple tasks.

[3] No such core group may exist when exactly one group is assigned to each NUMA node.

4.3 Resuming Suspended Threads

When a thread does not have another task to process, it suspends itself on a synchronization primitive[4]. Each core group has one such synchronization primitive and the suspended threads are added to its waiting queue.

When a new task is spawned, the spawning thread attempts to wake one of the suspended threads. First, it tries to wake a thread in the same thread pool (the same core group). If no suspended thread is found in the group, it scans all other groups and attempts to wake a thread there. The groups are scanned in increasing NUMA distance from the original group and the search finishes once a group with a suspended thread is found or after all groups are scanned.

The thread-resuming process is handled slightly differently in case of immediate tasks and in case of deferred tasks. When an immediate task is spawned, the search for a pool with a suspended thread ends at the boundary of the NUMA node. There is no point in waking threads on other nodes since the scheduling rules prevent the immediate tasks to venture beyond its NUMA node. When a deferred task is spawned, core groups across all NUMA nodes are tested.

5 Memory Allocation

Our locality aware scheduler is accompanied by a NUMA-aware allocator. The allocator is inspired by the state of the art scalable memory management systems [Berger et al., 2000] and adopted for parallel data processing on NUMA nodes. In our applications, we have optimized only the allocation of memory blocks between 8 and 512 kilobytes. These sizes are typical for memory buffers that transfer the data between individual tasks (e.g., the envelopes in Bobox system). Smaller blocks were allocated by standard `libc` allocator. Larger blocks were allocated directly by virtual memory management API of the operating system. Let us emphasize that the allocator can be used also for both smaller and larger memory blocks shall we need it.

The allocator is divided into two major components – the *superblock allocator* and the *local allocator*. The superblock allocator operates with large blocks of memory using the virtual memory management API of the operating system. The local allocator provides the standard `malloc`/`free` functions (or the `new`/`delete` operators respectively) for the application code whilst it utilizes the functions of superblock allocator. The structure of the whole allocator is depicted in Figure 4.

[4] Current implementation uses standard semaphore and atomic operations that handle related metadata.

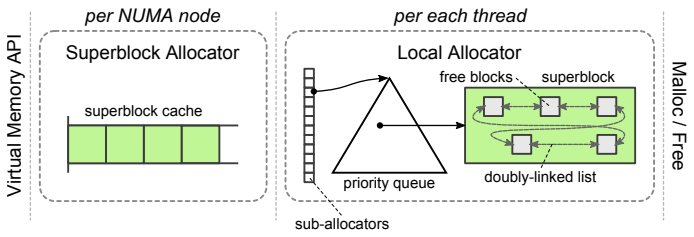

Figure 4: Structure of our NUMA-aware allocator

5.1 Local Memory Allocator

The local memory allocator is replicated, so that every thread has its own memory pool. Since the data structures are always accessed only by the thread that owns them, the local memory allocation does not require any locking mechanism nor thread synchronization.

The local allocator comprises several sub-allocators. Each sub-allocator S_i operates with memory blocks of fixed size $|S_i|$. When a new memory block of size N is requested, appropriate sub-allocator S_i is selected, so that $|S_i| \geq N$ and $|S_i| < |S_j|$ $(\forall j : |S_j| \geq N)$. The memory is then allocated by the sub-allocator S_i, hence it is possible that the allocated block is slightly larger than requested.

In our implementation, we have selected the sizes of allocated memory blocks $|S_i|$ in a way, that the first sub-allocator S_0 uses blocks of $|S_0| = 8$kB and every other sub-allocator S_i $(i = 1, \ldots)$ uses blocks $|S_i| = 1.07 \cdot |S_{i-1}|$ and each size S_i is rounded up to nearest multiply of cache line size to avoid false sharing problem. This setup utilizes 63 sub-allocators to cover the range between 8 and 512 kilobytes and the allocation overhead of the fix-sized blocks is kept under 7%. We can select different multiplicative constants than 1.07 to either reduce the overhead further or to reduce the number of sub-allocators; however, in our case, there were only a few sub-allocators actively used by the system.

Each sub-allocator manages a list of its own superblocks and a priority queue of superblocks that are not fully occupied (i.e., available for allocation). The priority queue is organized by the number of available blocks within each superblock. When a new block is being allocated by the sub-allocator, the most occupied superblock is selected from the priority queue. The free blocks in every superblock are managed by doubly linked list, whilst their pointers and related metadata are managed within the free blocks themselves.

As mentioned before, the sub-allocators does not employ any thread synchronization for the memory allocation, since every thread allocates from its own memory pool. The memory deallocation is a slightly more complicated matter, since a thread may need to deallocate memory block that belongs to (i.e., was originally allocated by) another thread. In such case, some form of synchroniza-

tion is inevitable. To minimize the possibility of synchronization collision, we have employed the following deallocation algorithm.

The memory is always deallocated only by the thread that owns it. If a thread needs to deallocate foreign memory, it returns it to the corresponding *memory recollection bin* of the owning thread. Every thread has a recollection bin for every other thread, hence only the synchronization between the deallocating and the owning thread has to be solved. The recollection bin is a linked list of pointers, which is protected by a simple spin-lock. A thread scans over its recollection bins every once a while (usually when it finishes a task) and disposes all memory blocks that have been inserted there. Since the memory recollection is not performed excessively often and since there is a separate lock for each pair of threads, synchronization collisions on this data structure are very rare.

Finally, the allocator implements simple memory reclaim strategy. The deallocated memory block is inserted to the doubly linked list of free blocks of its corresponding superblock in a LIFO manner. Therefore, when a new block is allocated from that superblock, the last deallocated block is returned first. This way, the allocator increases the chance that the reclaimed memory has still some parts in the CPU caches.

5.2 Superblock Allocator

The superblock allocator is used for allocating larger blocks directly from the virtual memory management of the operating system. The superblock allocator is replicated per NUMA node and when a memory superblock is allocated, it is also touched by the allocating thread. Thanks to the first touch policy [Bailey, 2007, Lameter et al., 2013] for physical memory allocation employed by current operating systems, the allocated virtual memory pages are mapped preferably to physical memory frames that reside at the NUMA node that performed the allocation.

The superblock allocator is shared by all the local allocators of threads that are located on the same NUMA node. Both allocation and deallocation algorithm uses one global lock. However, since the local allocator accommodates most allocation requests internally, collisions on this lock are extremely rare.

Superblock deallocation uses caching and memory reclaim strategy to reduce the overhead of memory zeroing algorithm. When a superblock is released by a local allocator, it is stored in a LIFO stack. Superblock allocation is primarily accommodated from this stack if possible. The LIFO ordering is used to support the memory reclaim strategy employed by the local allocators and to reduce TLB cache misses.

In our implementation, we have used superblocks of 10MB size. A different size may be selected to modify the properties of the allocator. Larger blocks will reduce the overhead of superblock allocation even further, while smaller

blocks will reduce the overall memory footprint since the allocation of the virtual memory will be more fine grained. In any case, the superblock size should be aligned to the memory page size (e.g., 4kB in case of IA32 architecture) and we have empirically observed that the minimal superblock size should be in the order of megabytes.

6 Experiments

We have extensively tested our LAS implementation to verify its impact on the processing time of parallel data processing systems. In this work, we present the most important experimental results. They demonstrate the benefits of locality aware scheduling and NUMA aware memory allocation on a real OLAP workload created by processing complex queries in an in-memory SPARQL engine [Falt et al., 2013].

6.1 Experimental Methodology

The experiments demonstrate the improvement of query processing time caused by better usage of computing units and improved memory locality when the LAS scheduler is used. All the performed tests were measured by the system real-time clock; all tests were performed at least 5 times and the presented results were computed as an average of the measured times. All measurements of each test deviated no more than 1% from the computed average.

The LAS scheduler was compared with a baseline scheduler which is based on the ideas implemented in Intel Threading Building Blocks library [Jung, 2012] and also in our previous work [Bednárek et al., 2013]. We denote this baseline scheduler as *not locality-aware scheduler* (NLS) and it works as follows: Each thread keeps its local queue of immediate task and all threads share one queue of deferred tasks. An available thread attempts to find a task to executes in the following order:

1. The latest immediate task from the local queue of the thread.
2. The oldest deferred task from the shared queue.
3. The oldest immediate task from the local queue of another thread.

Our NUMA aware memory allocator was employed for all the scheduling experiments and we also tested it separately. For comparison, we have selected the standard `malloc`/`free` allocator implemented in the `libc` library, since it is widely used and thus it provides a good baseline. The tests that compare the memory allocators use the same experimental setup and the LAS strategy for task scheduling.

6.1.1 Data and Queries

For the testing of our SPARQL engine, we have selected the SP^{2}Bench benchmark [Schmidt et al., 2009], which is a well established benchmark for RDF processing. The benchmark also includes a data generator tool, which we have used to prepare a dataset comprising 5 million RDF triples. The set of 5 million triples has been used for most tests; however, some tests required smaller sets for various practical reasons. These cases are explicitly pointed out in the text.

The SP^{2}Bench benchmark defines a set of testing queries which can generate different and quite complex query execution plans. Our implementation of the SPARQL engine is able to generate parallel execution plans for these queries without any significant serial bottlenecks. In other words, these queries are capable of utilizing all worker threads when they are evaluated. Furthermore, each query produces different type of workload (i.e., different spawning pattern of tasks), so we can show the behavior of the locality aware scheduler under different circumstances.

We have selected the queries Q2, Q4, Q5a, Q6, Q7 Q8, Q9, and Q11 from the benchmark. These queries sufficiently represent different scenarios for the task scheduler. The remaining queries of the benchmark have either similar execution plan to one of the selected queries or they have limited workload, so their evaluation is too quick to be measured by the real-time clock.

All the queries are evaluated in two possible configurations:

- We execute a single instance of the selected query. This experiment demonstrates the situation when there are not enough requests in the system and the cores must cooperate on tasks that are heavily interconnected and share a lot of data.
- The selected query is executed multiple times in parallel (16 times in all our measurements). This experiment demonstrate the situation when multiple independent requests are being processed and the scheduler must choose a good tradeoff between the latency of individual requests and the overall throughput to utilize system resources (especially the CPU caches) optimally.

6.1.2 Hardware Setup

We have performed all test on two hardware configurations. These configurations demonstrate the benefits of locality awareness both on a symmetric multiprocessor and in a NUMA environment.

- The first server is equipped with two Intel Xeon E5310 processors clocked at 1.6Ghz. This particular processor model comprises 4 physical cores and

two shared 4MB L2 caches. The first two cores share the first L2 cache and the second two cores share the other. Each core has its own L1 cache (32kB + 32kB for data and instructions respectively). Furthermore, the server has 8GB of RAM connected to both processors via the front-side bus (FSB). This configuration represents a nontrivial SMP system and our scheduling strategy creates 4 thread pools for this configuration (one for each pair of cores that share L2 cache).

- The second server is based on cache-coherent 4-node NUMA architecture. It is equipped with four Intel Xeon E7-4820 processors clocked at 2.0Ghz. Each processor comprises 8 physical cores with Hyper-Threading Technology – i.e., each processor has 16 logical cores and the server has 64 logical cores in total. The physical cores have their own L1 (32kB + 32kB) and L2 (256kB) caches. All cores of a processor share an L3 cache (18MB). The server is equipped with 128GB of RAM where each NUMA node holds 32GB. This configuration demonstrates a typical NUMA system and our scheduling strategy creates 4 thread pools (one for each NUMA node).

Both servers were running Red Hat Enterprise Linux version 6 and the Intel C compiler version 14 was used for compiling all of our source code.

6.2 Scheduling on SMP System

We have made minor adjustments to the experimental setup in order to accommodate for the hardware properties of our SMP system – especially the lower amount of installed operating memory. The query Q6 was performed on a dataset of 1 million triples in case of single-query test and 250 thousands triples in case of multi-query test. Furthermore, queries Q4 and Q8 were performed on 1 million triples in case of multi-query test. These modifications were necessary to avoid memory swapping mechanism which is undesirable as we are considering solely in-memory parallel data processing.

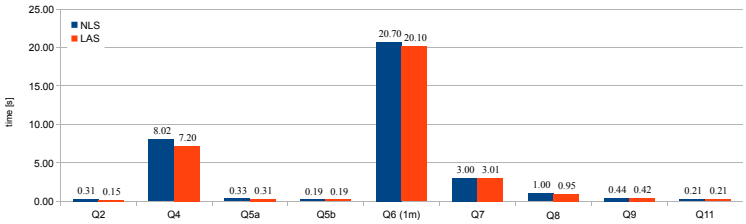

Figure 5: Results of single-query tests on SMP

The results of single-query tests are presented in Figure 5. As expected, the NLS scheduling algorithm performs quite well on SMP system. Queries Q4 and

Q6 exhibit noticeable improvement from the LAS algorithm and Q2 is almost 2× faster. This query consist of a single long pipeline; therefore, it is very sensitive to the locality aware planing. Other queries contain similar pipelines as well, but these pipelines are divided into multiple independent parts, since they are interleaved with sorting operators, which disrupt the data flow in a specific way. However, the LAS strategy always performed at least as efficiently as NLS strategy.

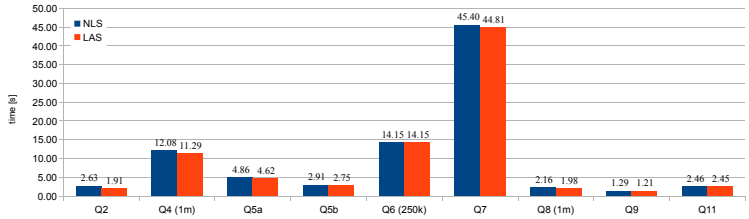

Figure 6: Results of multi-query tests on SMP

The results of multi-query tests are presented in Figure 6. This second experiment emphasize the LAS qualities even further, since it outperforms NLS strategy in almost every case. The main reason is that the LAS better separates individual requests, so the requests do not have to compete for space in CPU caches as much as in NLS case.

6.3 Scheduling on NUMA System

The tests on the NUMA system are designed to determine the influence of the NUMA factor, which could play a significant role on modern systems. The main problem is that accessing memory of another node renders both involved nodes rather busy. Furthermore, the intensive internode communication utilizes heavily the data buses between the processor sockets.

Once again, we have to reduce the size of the dataset for the query Q6 to 1 million triples in order to avoid memory swapping. On the other hand, the NUMA system has much more operating memory than our SMP system, thus we do not need to impose any other restrictions (not even for the multi-query tests). Let us also emphasize that we have used a logarithmic scale on the time axis for the NUMA tests, so we can present the results of time consuming test along with tests that can be evaluated rather fast.

Both sets of experiments performed on the NUMA system are presented in Figure 7 and in Figure 8. The LAS algorithm attempts to keep each request on a single NUMA node as much as possible. In case of single-query tests, the scheduler tries to keep different branches of the execution plans on different

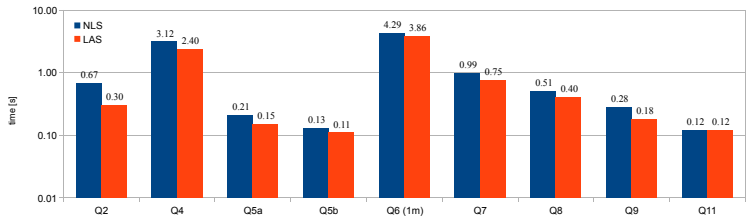

Figure 7: Results of single-query tests on NUMA (logarithmic scale)

NUMA nodes to minimize the data interference between the nodes. The NLS does not distinguish between the NUMA nodes, hence the relationship between a thread and the memory being accessed by that thread is almost arbitrary.

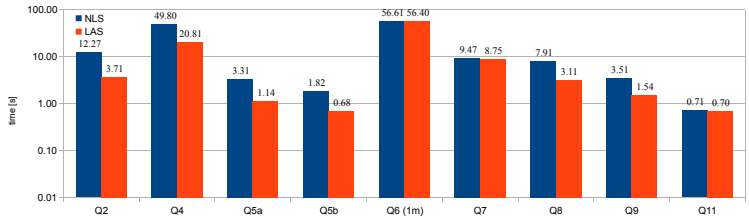

Figure 8: Results of multi-query tests on NUMA (logarithmic scale)

The multi-query tests (Figure 8) emphasize the benefits of NUMA aware scheduling along with NUMA aware memory allocation. Several queries exceeded 2× speedup over NLS strategy and query Q2 achieved the best observed speedup 3.3×.

All performed experiments proved that the locality awareness could have a significant impact on the performance of the parallel streaming system running on either SMP or NUMA systems. Furthermore, the LAS strategy does not impose any additional overheads, so it can achieve the same results as NLS even in situations that cannot benefit from locality aware scheduling.

6.4 Memory Allocation

As mentioned before, we have compared our NUMA aware allocator (which we denote *Bobox allocator* for these experiments) with standard `malloc`/`free` allocator of the C runtime library. In this experiments, we use the same scenario as for the scheduling experiments, but we have performed only the multi-query tests. Multi-query tests consume significantly more memory, thus provide a greater challenge for the allocator. Furthermore, they have better workload de-

composition over the NUMA nodes, thus emphasizes the importance of locality aware allocation.

We have reduced the dataset size for query Q6 in case of the NUMA system and for queries Q4, Q6, and Q8 in case of the SMP system in the same way and for the same reasons as in previous experiments. Let us also emphasize that the time axis uses logarithmic scale.

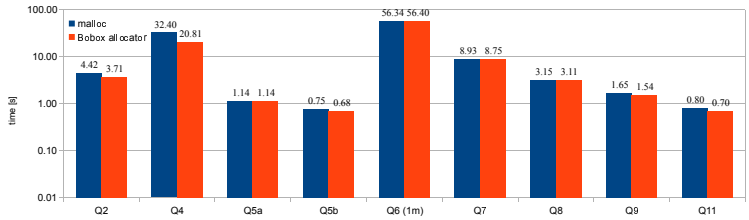

Figure 9: Results of memory allocator tests on NUMA (logarithmic scale)

The times measured on NUMA system are shown in Figure 9. Query Q4 benefits from the new allocator the most since it exhibits approximately 35% improvement over standard `malloc`. This query is exceptionally sensitive to the data locality as we have already established earlier (Figure 8). Additionally, query Q4 produces rather large result set (approximately 18 millions tuples); therefore, it makes a good use of the super block reusing strategy.

Another significant improvement was observed for query Q2 (about 15%). This query is sensitive to data locality as well, but it does not consume as much memory. Therefore, the speedup is caused mainly by the NUMA aware allocation (combined with the NUMA aware scheduling).

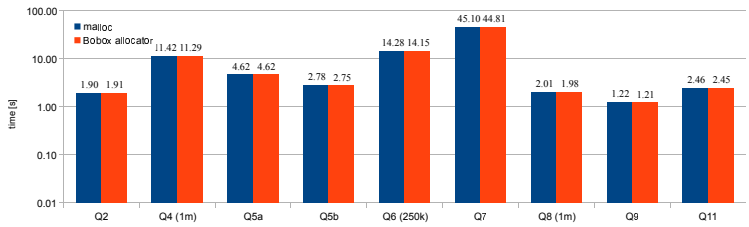

Figure 10: Results of memory allocator tests on SMP (logarithmic scale).

The results of the SMP system are summarized in Figure 10. No query has demonstrated significant speedup; however, minor improvements can be observed for almost every query. It is our opinion that these improvements are the result

of the LIFO memory recycling strategy that reduces the number of cache misses and TLB misses.

The results indicate that the Bobox allocator have indeed improved the processing time of locality sensitive execution plans. Furthermore, the proposed allocator has never exhibited worse performance than the baseline allocator, so it can be used as a replacement in all situations.

6.4.1 Windows Memory Allocator

In addition to the experiments presented in previous sections, we also performed the memory allocator experiments in the Windows operating system, since it uses different runtime libraries, different memory allocation algorithm, and different thread scheduling strategy. For this experiment, we used a desktop computer with Intel Core i7 2600 clocked at 3.4GHz. The processor comprises 4 cores, each with its own L1 cache (32kB+32kB) and L2 cache (256kB). All cores shares single 8MB L3 cache. The processor uses Hyper-Threading technology; therefore, it has 8 logical cores. The system is equipped with 16GB main memory and it runs 64-bit Windows 7 SP1 as an operating system. We used Microsoft Visual Studio 2013 Professional with Update 1 in Release x64 configuration to compile our code.

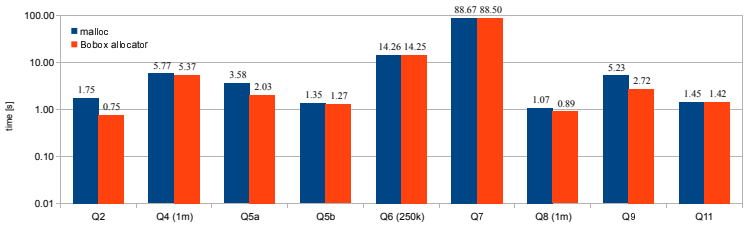

Figure 11: Results of memory allocator tests on desktop PC with Windows 7 (logarithmic scale)

The results of the Windows experiments are presented in Figure 11. The queries Q6, Q7, and Q11 show little difference between the allocators, since the most of their processing time is actually spent by intensive computations and they retain most of the allocated memory during the whole execution. Other queries benefit significantly from the new allocator. For example, queries Q5a and Q9 exhibit approximately 2× speedup and query Q2 has speedup over 2.3×. The reason for these results is twofold. The Bobox allocator reuses the superblocks intensively which reduces system calls and page zeroing time. Furthermore, the current version of Visual Studio 2013 runtime libraries [Microsoft, 2014a] serial-

izes the access of multiple threads to the memory heap [Microsoft, 2014b], thus creating an application-wide bottleneck.

7 Conclusions

In this paper, we presented a novel task scheduling strategy that takes advantages of current CPU architectures and both SMP and NUMA multiprocessor systems. Our scheduler can effectively improve the data locality and thus the cache-hit ratio when employed on parallel data stream processing systems. We have implemented a prototype of the scheduler and integrated it into the Bobox framework, which allows the evaluation of execution plans created manually or translated from various query languages. When applied on a SPARQL benchmark that process RDF data, the system achieved up to 10% speed up on double-processor SMP system and up to 3× speed up on four processor NUMA system for selected queries with respect to previous version of the scheduler.

In the future work, we are planning to extend our scheduler to other domains of task processing. We would like to improve generic frameworks that also use tasks to achieve parallelism, but which process different types of datasets (not only streaming data). On that front, we have already started integrating LAS into Intel Threading Building Blocks library and the preliminary results are very promising. Furthermore, we would like to extend the scheduler (and the whole Bobox framework) to support work offloading to parallel accelerators such as GPUs and Xeon Phi devices, where the term data locality gets yet another dimension since the transfers between the host system and the parallel device are rather costly.

Acknowledgment

This work was supported by the Czech Science Foundation (GACR), projects P103-13-08195S and P103-14-14292P, and by Specific Research project SVV-2014-260100.

References

[Babcock et al., 2004] Babcock, B., Babu, S., Datar, M., Motwani, R., and Thomas, D. (2004). Operator scheduling in data stream systems. *The International Journal on Very Large Data Bases*, 13(4):333–353.

[Bailey, 2007] Bailey, A. (2007). The Windows NUMA API-What It Is and Why You Care. `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/the-windows-numa-api-what-it-is-and-why-you-care/`.

[Bednárek and Dokulil, 2010] Bednárek, D. and Dokulil, J. (2010). TriQuery: Modifying XQuery for RDF and Relational Data. In *2010 Workshops on Database and Expert Systems Applications*, pages 342–346. IEEE.

[Bednárek et al., 2009] Bednárek, D., Dokulil, J., Yaghob, J., and Zavoral, F. (2009). The Bobox Project - A Parallel Native Repository for Semi-structured Data and the Semantic Web. *ITAT - IX. Informačné technológie - aplikácie a teória*, pages 44–59.

[Bednárek et al., 2009] Bednárek, D., Dokulil, J., Yaghob, J., and Zavoral, F. (2009). Using methods of parallel semi-structured data processing for semantic web. *Advances in Semantic Processing, International Conference on*, pages 44–49.

[Bednárek et al., 2013] Bednárek, D., Dokulil, J., Yaghob, J., and Zavoral, F. (2013). Data-flow awareness in parallel data processing. In *Intelligent Distributed Computing VI*, pages 149–154. Springer.

[Berger et al., 2000] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. (2000). Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128.

[Boag et al., 2002] Boag, S., Chamberlin, D., Fernández, M., Florescu, D., Robie, J., Siméon, J., and Stefanescu, M. (2002). XQuery 1.0: An XML query language. *W3C working draft*, 15.

[Broquedis et al., 2009] Broquedis, F., Furmento, N., Goglin, B., Namyst, R., and Wacrenier, P.-A. (2009). Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 79–92. Springer.

[Chandra, 2001] Chandra, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann.

[Chen et al., 2012] Chen, Q., Guo, M., and Huang, Z. (2012). CATS: Cache Aware Task-stealing Based on Online Profiling in Multi-socket Multi-core Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 163–172, New York, NY, USA. ACM.

[Chen et al., 2011] Chen, Q., Huang, Z., Guo, M., and Zhou, J. (2011). Cab: Cache aware bi-tier task-stealing in multi-socket multi-core architecture. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 722–732. IEEE.

[Cieslewicz et al., 2009] Cieslewicz, J., Mee, W., and Ross, K. (2009). Cache-conscious buffering for database operators with state. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 43–51. ACM.

[Duran et al., 2008] Duran, A., Corbalán, J., and Ayguadé, E. (2008). Evaluation of OpenMP task scheduling strategies. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, pages 100–110. Springer-Verlag.

[Falt et al., 2013] Falt, Z., Čermak, M., and Zavoral, F. (2013). Highly Scalable Sort-Merge Join Algorithm for RDF Querying. In *Proceedings of the 2nd International Conference on Data Management Technologies and Applications*.

[Ferreira et al., 2011] Ferreira, T. B., Matias, R., Macedo, A., and Araujo, L. B. (2011). An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 92–98. IEEE.

[Google, 2014] Google (2014). Tcmalloc: Thread-caching malloc. `http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html`.

[Huang et al., 2010] Huang, X., Rodrigues, C. I., Jones, S., Buck, I., and Hwu, W.-m. (2010). XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-coreMachines. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology*, pages 1134–1139. IEEE.

[Hudson et al., 2006] Hudson, R. L., Saha, B., Adl-Tabatabai, A.-R., and Hertzberg, B. C. (2006). Mcrt-malloc: a scalable transactional memory allocator. In *Proceedings of the 5th international symposium on Memory management*, pages 74–83. ACM.

[Jung, 2011] Jung, J. J. (2011). Service Chain-based Business Alliance Formation in Service-oriented Architecture. *Expert Systems with Applications*, 38(3):2206–2211.

[Jung, 2012] Jung, J. J. (2012). Evolutionary Approach for Semantic-based Query Sampling in Large-scale Information Sources. *Information Sciences*, 182(1):30–39.

[Jiang and Chakravarthy, 2004] Jiang, Q. and Chakravarthy, S. (2004). Scheduling strategies for processing continuous queries over streams. *Key Technologies for Data Management*, pages 16–30.

[Jung, 2011] Jung, J. J. (2011). Service Chain-based Business Alliance Formation in Service-oriented Architecture. *Expert Systems with Applications*, 38(3):2206–2211.

[Kahan and Konecny, 2006] Kahan, S. and Konecny, P. (2006). "mama!": A memory allocator for multithreaded architectures. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 178–186, New York, NY, USA. ACM.

[Kruliš et al., 2013] Kruliš, M., Falt, Z., Bednárek, D., and Yaghob, J. (2013). Task scheduling in hybrid CPU-GPU systems. *Informačné Technológie-Aplikácie a Teória*.

[Kukanov and Voss, 2007] Kukanov, A. and Voss, M. (2007). The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4):309–322.

[Lameter et al., 2013] Lameter, C., Hsu, B., Sosnick-Pérez, M., Bacon, D., Rabbah, R., Shukla, S., Danowitz, A., Kelley, K., Mao, J., Stevenson, J. P., et al. (2013). Numa (non-uniform memory access): An overview. *Queue*, 11(7):40.

[Lyberis et al., 2013] Lyberis, S., Pratikakis, P., Nikolopoulos, D. S., Schulz, M., Gamblin, T., and de Supinski, B. R. (2013). The myrmics memory allocator: hierarchical, message-passing allocation for global address spaces. *ACM SIGPLAN Notices*, 47(11):15–24.

[Masmano et al., 2004] Masmano, M., Ripoll, I., Crespo, A., and Real, J. (2004). Tlsf: A new dynamic memory allocator for real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 79–88. IEEE.

[Microsoft, 2014a] Microsoft (2014a). C Run-Time Library Reference. `http://msdn.microsoft.com/en-us/library/59ey50w6.aspx` [Online; accessed 2014-05-08].

[Microsoft, 2014b] Microsoft (2014b). HeapAlloc function (Windows). `http://msdn.microsoft.com/en-us/library/windows/desktop/aa366597(v=vs.85).aspx` [Online; accessed 2014-05-08].

[Pirasteh et al., 2015] Pirasteh, P., Hwang, D., Jung, J. J. (2015). Exploiting Matrix Factorization to Asymmetric User Similarities in Recommendation Systems. *Knowledge-Based Systems*, 83:51–57.

[Prud'Hommeaux et al., 2006] Prud'Hommeaux, E., Seaborne, A., et al. (2006). SPARQL query language for RDF. *W3C working draft*, 4.

[Reinders, 2007] Reinders, J. (2007). *Intel Threading building blocks*. O'Reilly.

[Safaei and Haghjoo, 2010] Safaei, A. A. and Haghjoo, M. S. (2010). Parallel processing of continuous queries over data streams. *Distrib. Parallel Databases*, 28:93–118.

[Schmidt et al., 2009] Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. (2009). SP^{2}Bench: a SPARQL performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE.

[Sinnen, 2007] Sinnen, O. (2007). *Task scheduling for parallel systems*, volume 60. John Wiley & Sons.

[Tannenbaum, 2014] Tannenbaum, B. (2014). Miser – A Dynamically Loadable Memory Allocator for Multi-Threaded Applications. `https://software.intel.com/en-us/articles/miser-a-dynamically-loadable-memory-allocator-for-multi-threaded-applications`.