Task Scheduling in Data Stream Processing*

Zbyněk Falt and Jakub Yaghob

Department of Software Engineering, Charles University,
Malostranské nám. 25, 118 00 Prague, Czech republic
{falt, yaghob}@ksi.mff.cuni.cz
http://www.ksi.mff.cuni.cz/

Abstract. One possible technique of data processing is its transformation into a data stream and the execution of particular operations on the data tuples. These operations can be usually processed concurrently especially when the plan of operations is branched. Therefore, this way of data processing is suitable for evaluation in parallel environment. On the other hand, the ordering of the execution of tasks associated with the operations is closely related to the utilization of the hardware components. For that reason, the task scheduler is very important in these systems. In this paper, we introduce our implementation of a task scheduler which deals well with various hardware factors such as caches and NUMA[1] factor.

1 Introduction

As the amount of data which are intended for processing becomes larger, new approaches for their processing are researched. Since the performance of the systems has been recently increased mainly through the addition of computational units, the parallel processing of data is a natural evolution in this research area. New algorithms, new approaches, and new technologies, which utilize this direction of progress, are developed.

On the other hand, the development of parallel data processing is more difficult than single threaded development. The developer must solve issues such as the decomposition of the problem to independent parts which may be processed in parallel, thread management, their synchronization and communication.

Of course, there exist many frameworks which try to make these things easier. One of the approaches, that makes especially the processing of data queries easier, is the transformation of input data into data streams and the execution of particular operations on that streams. The execution plan looks like directed graph where nodes are the operations and edges determine dataflow among the operations. The biggest advantage of this approach is that developer of this plan does not have to take parallelization into account since this is done automatically

* This paper was supported by the grant SVV-2011-263312 and by the grant agency of Charles University (GAUK), project no. 277911.

[1] Non-Uniform Memory Access

during the evaluation. It is possible, because operations on different parts of the streams or whole branches of the plan may be processed concurrently.

One of the systems that implements this idea is Bobox [6]. The main aim of the Bobox project is to process semantic and semistructured data effectively [7]. Currently, support for SPARQL [15] query language is under development and partial support for XQuery [8] and Tri Query [5] language is already developed.

The contribution of this paper is a presentation of the scalable scheduling techniques for systems which process data streams. These techniques deal well with different hardware factors such as size of caches and NUMA factor. Therefore they enable to evaluate data queries faster on modern systems. Although they are analyzed and tested on the Bobox system, they can be easily adopted by other similar systems.

The remainder of this paper is as follows. In the Section 3.1, there is the description of execution plan evaluation and tasks creation. In the Section 4, we measure and analyze the influence of various hardware factors on the efficiency of the system. The Section 5, contains detailed description of the implementation of our scheduler and in the Section 6, we present and analyze an experimental comparison between our old simple and new implementation of the scheduler. The Section 7 summarizes the results and introduces our future plans.

2 Related work

Parallel processing of data stream is a very actual research topic and many systems similar to the Bobox are being developed, for example Borealis [1], STREAM [2], Nile [12] or NiagarsST [14]. However, the aim of all the systems is mainly the processing of an infinite data streams or real-time data streams. This is one of the biggest differences between them and the Bobox, because the main scope of the Bobox is efficient processing of the offline finite data streams.

Of course, these differences determine different requirements on the scheduler and the scheduling strategies. The common requirements for all systems are throughput and efficient utilization of available resources. Additionally, in the processing of infinite or real-time data streams, memory usage and response time are also very important and some papers address this problems [13], [3] or [17].

In the Bobox-like systems the factors of response time do not have to be taken into account, as they are not meant to be the frameworks for processing real-time data. The finiteness of the input data ensures that the memory usage will not grow beyond control.

The other major difference is that the Bobox supposes there is no parallelism in the operator evaluation. This means that in contrary to other systems each operator is allowed to run only on at most one thread. This restriction makes operators implementation easier. On the other hand, the only way of achieving concurrency in the operator implementation is their decomposition to several atomic operations. For example, sorting can be decomposed to several single threaded sorting operators, which might be evaluated in parallel. Their results can be then merged together by the net of merge operators.

Because one operation can be composed of many partial operations, the scheduler must be optimized according to this fact. The main reason is that the communication among these suboperations is more intensive than among the operators. To deal with it, our scheduler optimizes the work with the CPU cache and tries to keep this communication in the caches as much as possible.

The strategy implemented in the paper [10] also optimizes the utilization of the CPU caches. However, in contrast to our work, it tries to optimize the accesses to caches during operator evaluation. We focused on the utilization of the caches to speed up the communication among operators.

Problems of task scheduling are also a very important part of frameworks or environment which are not related to data streams processing, as shown for instance in TBB [16] or OpenMP [9], [11]. As these papers take into account cache related problems and solve them via increasing the data locality, they are not concerned with factors such as NUMA factor.

3 Evaluation of an execution plan

As mentioned briefly in Section 1, the execution plan looks like a directed graph. The nodes of this graph represent operators and the edges determine the dataflow in the plan. Operators can have an arbitrary number of input edges through which they receive data tuples, and an arbitrary number of output edges along which they send the resulting tuples. Since tuples are typically small and some overhead is related to their transfer, they are grouped together into packets. This grouping helps to reduce the communication overhead. The other important thing is, that the operators are allowed to communicate with each other only through these packets. Thus, they should not share any variables or other resources.

We denote executions plans as requests in the rest of the paper, because each plan is typically related to some data query request.

3.1 Tasks creation and their types

The fact that each operator can run on at most one thread determines the set of states of each operator. These states are:

- *Nothing* – The operator has nothing to do.
- *Scheduled* – The operator has been scheduled to perform its operation.
- *Running* – The operator is processing input data or performing another operation. After that it will be switched to the *Nothing* state.
- *Running and scheduled* – The operator has been scheduled during the *Running* state. When the operator finishes its operation it will be switched to the state *Scheduled*. The scheduling requests are serialized in this way and therefore the operator cannot be run twice at a time even if it is scheduled multiple times.

When a packet is received by the operator, it is switched to state *Scheduled* or *Running scheduled*. Every time the operator is switched to the state *Scheduled*, new task is created. The objective of the scheduler is to select tasks and run them on the threads which it has chosen.

The receipt of the packet is not the only event when the task is created. The other situation is, for example, when the operator which generates new data sends data packet, but it still has many other packets remaining. In this case, the operator schedule itself again after the packet is sent and the next time it is run, it sends the next packet etc.

It is obvious that the tasks associated with packet reception require a bit different handling. The fact that the packet was received probably means that the content of the packet is hot in cache. Thus, the earlier the associated task will be performed, the faster this performance probably will be. Other types of tasks are not directly associated with an packet, therefore it does not matter when or even where these tasks will be run. The first task type is called immediate and the second task type is called deferred.

4 Factors influencing efficiency

This section contains a list of the most important hardware factors which the scheduler should take care of. The importance of these factors is experimentally verified on the Bobox system.

4.1 Cache

One of the goals we wanted to reach was an effective utilization of caches. One way of doing it is to try to keep the content of packets in the cache. Therefore, the packet transfer does not cause many cache misses. If we consider only this factor, we conclude that the smaller packets the better the performance is since they fit better into the cache. On the other side, there is some overhead with this transfer such as synchronization of data structures, scheduler decision algorithms etc. If we consider only this overhead, then bigger packets decrease this overhead and increase the efficiency.

Our task was to find a reasonable compromise between these two extremes. We measured the relation between the size of packets and the efficiency of the system. We executed the same request with different packet sizes. The results are shown in Figure 1 and confirm the hypothesis. With smaller packets the overhead grows up and with larger packets there is an overhead with cache misses.

According to the plot, the ideal size of packets is slightly lower than the size of L2 cache, which is 256KB (see Section 6 for more detailed hardware specification). It can be easily explained since not only the packets are stored in the cache, but there are also data structures of scheduler, memory allocator or operators. These data structures are heavily accessed during the evaluation, so keeping them also in the cache improves efficiency. Therefore the packet size must be lowered by the size of these structures in order that they all fit into the cache. Unfortunately, the size of these structures is hard to estimate.

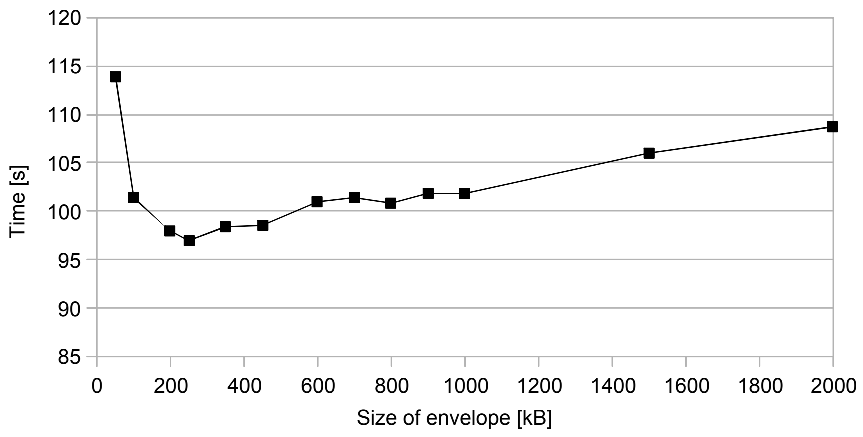

Fig. 1. The influence of the packet size on the system performance

4.2 NUMA factor

The number of processors in a SMP system cannot grow infinitely. As the number of processors in the system is increasing, shared resources are turning into bottlenecks. The most critical shared resource is the main memory. Therefore NUMA systems are being developed as a solution to this problem.

Basically, the NUMA system consists of several nodes. Each node acts as SMP system, i.e. it has its own local main memory. These nodes are connected together and the whole system shares the physical address space. Thus, one node can access local memory of another node, but this access is slower than an access to its local memory. This slow down is known as NUMA factor. The NUMA factor tells us how many times slower is access to non-local memory in the comparison to access to local memory.

It is obvious that processor should access preferably its local memory. In the opposite case, the computation is slowed down by the NUMA factor. Of course, this problem is related more to memory allocation than to scheduling, but NUMA non-aware scheduler can cause performance penalties, too. For example when the scheduler move a computation, which has allocated some objects in its local memory, from one node to another.

To measure the influence of the NUMA factor on the Bobox system, we did the following test. Firstly we performed a request on one NUMA node and all allocations return its local memory. Secondly, we performed the same request on the same NUMA node, but all the memory was allocated on another node. The results (see 2) proves that the influence of NUMA factor cannot be ignored.

For a comparison, we performed synthetic test which measures the NUMA factor of our testing system. The test measured time needed for reversion of the order of items in large array of integers. The test were run for all combinations of nodes where the algorithm was performed and nodes where the memory was

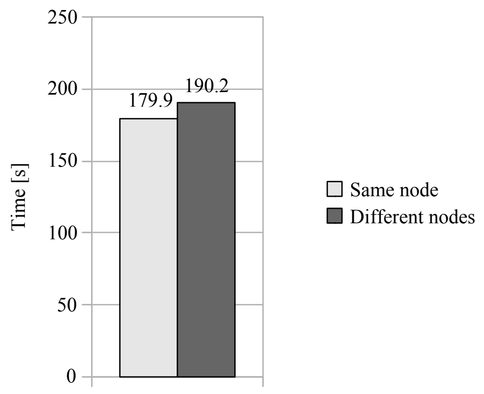

Fig. 2. The influence of the NUMA factor on the system performance

allocated. The results are shown in the Table 1, where all numbers are recalculated relatively to the lowest times (which are naturally on the diagonal). Nodes are numbered from 0 to 3. The factor is quite low in the system, so the difference between both methods in the Figure 2 is not so significant, but the higher the factor is, the bigger difference will be.

	0	1	2	3
0	1.0	1.4	1.5	1.4
1	1.5	1.0	1.4	1.4
2	1.4	1.5	1.0	1.5
3	1.5	1.5	1.5	1.0

Table 1. Matrix of NUMA factors in the system

5 Scheduler

5.1 Data structures and management of tasks

Data structures of the scheduler reflect the hardware structure of the host system and the fact that system may be evaluating many requests at a time. Therefore each NUMA node contains the list of all requests it is currently evaluating, each request contains the list of all deferred tasks and each thread contains the list of immediate tasks which were created by this thread. When the host system is not NUMA system but ordinary SMP, it is still considered a NUMA system with only one node.

The algorithm of what to do when a new task is created is quite simple, because the real work is done by the second part of the algorithm, which is described below. A new immediate task is inserted into the list of immediate tasks of the thread which created the task. A new deferred task is inserted into the list of deferred tasks of the request to which the task belongs.

When a worker thread finishes its current task, it starts to find another task which it will execute. The algorithm which selects this task is a bit more sophisticated and its goal is to maximize the usage of the principle of locality, which increases the efficiency of the system. Our strategy is to find the first task in this order:

1. The newest task in the list of immediate tasks that belongs to the thread. Since this task is immediate, it is associated with some data packet which was recently created. The content of the packet was probably in the cache after the creation. The fact that the task is the newest increases the probability that no other packet moves the packet out of the cache.
2. The oldest task in the list of deferred tasks that belongs to request, that was evaluated by the thread for the last time. Each request has probably some data loaded in the cache and switching to another request would cause more cache misses than the continuation with the same request. There are two reasons, why we decided to choose the oldest task from the list.
   The first is, that if we chose the newest task, then the execution plan would be evaluated in the DFS[2] manner instead of BFS[3]. But BFS manner tends to generate more concurrent tasks than the DFS, so the level of parallelism is higher.
   The other reason is, that each operator has its own input buffers for incoming packets and deferred tasks are typically used for deferred packet creation (see Section 3). Thus, the later this task will be executed, the more probable is, that the input buffer of the successive operator will be empty. This increases probability, that the newly created packet will not be rejected by the operator and also probability, that the oldest packets in the input buffer (i.e. packets which the successive operator will process) are hot in cache.
3. The oldest existing task of the oldest request which the current node is evaluating. Therefore old requests are processed primarily. Each request allocates many resources during its evaluation – memory, files etc. These resources are usually allocated successively during the request evaluation. It is obvious that if all request were processed with the same priority, then the total amount of allocated resources would be increasing faster than if one request is processed preferably and others are suppressed. Additionally, when the oldest request is finished, its allocated resources will be freed and will be available for other requests. Therefore, this strategy tries to minimize the number of allocated resources during requests evaluation.
4. The oldest task in the list of immediate tasks of another thread from the same node. This stealing of the task violates the principle of locality, on the other hand it is better to utilize idle thread than let it sleep.
5. If no such task exists, then the seeking thread is suspended.

[2] Depth-first search
[3] Breadth-first search

This architecture causes, that one request can be evaluated only on one node at a time. This is a suitable behavior for NUMA systems because during the evaluation threads work only with its local memory. On the other hand, this might be a problem for request with high degree of parallelism. In that case, only a part of the system performance is used for its evaluation. Another problem is that system might become unbalanced, which means, that some nodes are overloaded, while others are idle.

The second problem is partially solved by the algorithm, which chooses a node that has the most free resources available when a new request is created and passes the request to that node. The algorithm assumes free resources to be the free memory and the number of idle threads. This solution is partial because when the request is created, it is unknown how many resources it will be spending. Therefore the initial decision may be wrong in the global context. This is solved by a dynamic load balancing described below.

The first mentioned problem is solved by the load balancing as well.

5.2 Other algorithms

Load balancing When there is at least one idle logical processor in the system, a special load balancing thread is running. This thread evaluates every 100ms load of the system. When there is at least one overloaded and at least one partially idle node, then load balancing algorithm tries to solve this imbalance. The algorithm selects the newest request of the overloaded node and moves it to the idle node. The idea is that the newest request has probably the smallest amount of allocated memory, therefore this transfer should be the most painless.

If there is no request to move (i.e. node is overloaded and is processing only one request), then the request becomes shared with the idle node. At this moment, the request starts to be evaluated by both nodes. Of course, it is possible that one queue is shared among more than two nodes. This happens when the request has a high level of parallelism and is able to overload more nodes.

Sleeping threads A very important attribute of task scheduler is whether it can put to sleep the threads which have nothing to do. An active waiting for a next task is really inefficient. There are three main reasons. The first one is that an active waiting keeps the CPU units busy. For cores with Hyper-Threading technology, where these units are shared between two logical processors, this behavior is very undesirable since the waiting threads slow down the other threads.

The second reason is, that an active waiting means a repeated checking whether there is a new task. Because of the synchronization of the shared structures this checking loads the bus, memory or cache coherency protocol. Thus, this approach decreases performance even of non-waiting threads.

And the last one is, that almost all modern systems have some kind of power saving technology which switches idle processors to low-power state. Therefore, thread sleeping reduces the power consumption of the host system.

However, the thread sleeping brings one problem we have to cope with. When the scheduler allows thread sleeping, there must be a mechanism for deadlock prevention. This is related to the situation when there are more tasks than running threads and though idle threads exist. This situation must be avoided.

Our solution is quite simple. Each NUMA node keeps the number of the tasks for which it is responsible, i.e. a node must ensure that all these tasks will be processed. The thread must not be suspended when the number of tasks is greater than the number of running tasks. Therefore all tasks for which the node is responsible will be processed before all threads will be put to sleep. When any task or request is created and there is an idle thread in the corresponding node, then this thread is woken up.

The implementation must also prevent the situation when some request is shared but some nodes are idle because they are not responsible for any task of this request. We have decided for a simple but effective solution – the responsibilities for new deferred tasks of the request are distributed in the round-robin way among all sharing nodes.

The implementation of thread sleeping itself can also influence the performance of the system. The straightforward solution is to created semaphore for each node. The value of the semaphore would be equal to the number of tasks for which is the node responsible. New task would increase the value and each thread would decrease the value before it would try to find another task.

Unfortunately each change of the semaphore value yields to a system call, which can be quite slow. Our implementation tries to avoid this. Each thread has its own binary semaphore. Moreover the number of running threads and the number of tasks are counted in an atomic variable. Before a thread is suspended, the number of running thread is increased and the semaphore of the thread is acquired. When the number of running threads is lower than the number of tasks, then the number of running threads is increased and one semaphore of some selected thread is released.

This implementation ensures, that when the node is fully loaded, i.e. all threads are running, then no operation is called on semaphores. Furthermore there is no global semaphore which could become a bottleneck.

NUMA support NUMA system support is already integrated in the architecture of the scheduler. Each node has its own task queue and the scheduler tries to keep all the requests only on one node. Only in the case when one node is overloaded it tries to share computations among more than one node. This strategy prevents it from redundant movement of requests among the nodes.

The next advantage follows from the fact that only the queues of deferred tasks are shared. Deferred tasks are not related so tightly to data flow as the immediate tasks. If an packet is created (and its memory allocated) and sent to other operator, the task associated with this packet will be processed on the same node (and the thread will work with its local memory). The resulting packet of the operation will be processed on the same node as well for the same reason.

On the other hand, access to non-local memory cannot be fully avoided, because operators such as for example merge or join can have more inputs which are processed together.

CPU cache support It follows from the measurement in the Section 4 that performance of the system depends on the size of the packets and the size of the CPU caches. The optimal size according to the results is the same as size of L2 cache lowered about the size of data structures of scheduler, allocator etc.

The problem is that this size is hard to estimate, since it depends on many factors, such as behavior of the operators or the structure of the input data. We have decided to solve this problem in a very simple way. As the optimal size of an packet, we consider one half of the L2 cache size. For this size, all experiments gave the best results. But we plan to solve this problem better in future (see Section 7).

6 Results

To measure the benefits of the new scheduler, we used system with 4 Intel Xeon E7540 running at 2GHz CPUs. Each processor has 6 cores with the Hyper-Threading support. Altogether the whole system has 48 logical processors. Each core has its own 32kB data and 32kB instruction L1 cache and 256kB L2 cache. All cores in the one processor share the 18MB L3 cache. The system has 4 NUMA nodes; each node has 32GB operating memory. The operating system is the Red Hat Enterprise Linux.

We used XQuery [4] compiler to generate sufficiently big execution plan which consists of around 170 operators. As input data we used 10MB XML file. The results do not include the time needed to read the input data from hard drive. Each time in the chart is calculated as the median of 5 measurement of the same test to eliminate inaccuracy. To measure scalability, we were increasing the number of concurrent queries exponentially. The label $N \times M$ in the figure means that we ran N times M queries concurrently.

For comparison, we used very simple scheduler, which does not distinguish between immediate and deferred tasks. Each thread keeps its own list of tasks and processes this list in the FIFO order. If there is no task in its own list, it tries to steal a task from other threads in round-robin order. For threads suspending, one global semaphore is used. The value of the semaphore is kept the same as the number of tasks. The size of packets is constant without respecting the cache size.

The result is shown in the Figure 3. Our new implementation of the scheduler improves the performance about 8 – 12% according to the load of the system. The experiments show that the described ideas are correct and help improve the efficiency of the data processing.

7 Conclusion

In this paper, we proposed the architecture of task scheduler for the systems, which process data streams in parallel environment. We also implemented this scheduler and measured its behavior in the Bobox system. The new scheduler is about 8 – 12 percent more efficient and is NUMA aware. Moreover the techniques,

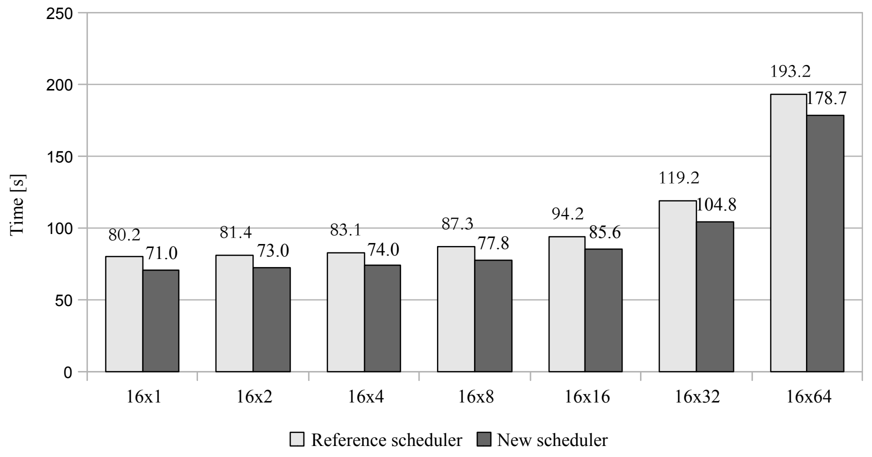

Fig. 3. The effectivity of the new implementation of the scheduler. $N \times M$ means that we ran N times M queries concurrently.

we described, can be used not only in our system, but it is possible to adopt them easily by other similar systems, which contain some kind of task scheduling mechanism.

On the other hand, many challenges still remain. One of them is the analysis of runtime characteristics of the request and dynamic changes of some parameters according to the characteristics to achieve better results. This is related mainly to the size of packets, which is hard to estimate statically without the knowledge of exact behavior of operators.

We not only plan to do optimizations for the size of cache but also for its hierarchy. This means that we want to take cache sharing into account as well. This brings some issues because it is hard to say which threads share caches and which do not since the operating system can change the affinity of the threads automatically. The straightforward solution of manual setting the affinities of threads directly to logical processor does not work well, primarily when processors have the Hyper-Threading technology. The reason is that the thread scheduler of the operating system cannot perform any load balancing among logical processors, which might yield to performance degradation.

References

1. D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA*. Citeseer, 2005.

2. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: the stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, page 665. ACM, 2003.
3. B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The International Journal on Very Large Data Bases*, 13(4):333–353, 2004.
4. D. Bednárek. *Bulk Evaluation of User-Defined Functions in XQuery.* PhD thesis, Faculty of Mathematics and Physics, Czech Republic, 2009.
5. D. Bednárek and J. Dokulil. Tri Query: Modifying XQuery for RDF and Relational Data. In *2010 Workshops on Database and Expert Systems Applications*, pages 342–346. IEEE, 2010.
6. D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral. The Bobox Project - A Parallel Native Repository for Semi-structured Data and the Semantic Web. *TAT 2009 - IX. Informačné technológie - aplikácie a teória*, pages 44–59, 2009.
7. David Bednarek, Jiri Dokulil, Jakub Yaghob, and Filip Zavoral. Using methods of parallel semi-structured data processing for semantic web. *Advances in Semantic Processing, International Conference on*, 0:44–49, 2009.
8. S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language. *W3C working draft*, 15, 2002.
9. R. Chandra. *Parallel programming in OpenMP.* Morgan Kaufmann, 2001.
10. J. Cieslewicz, W. Mee, and K.A. Ross. 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, 2009.
11. A. Duran, J. Corbalán, and E. Ayguadé. 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, 2008.
12. M.A. Hammad, M.F. Mokbel, M.H. Ali, W.G. Aref, A.C. Catlin, A.K. Elmagarmid, M. Eltabakh, M.G. Elfeky, T.M. Ghanem, R. Gwadera, et al. Nile: A query processing engine for data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, page 851. IEEE, 2004.
13. Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. *Key Technologies for Data Management*, pages 16–30, 2004.
14. D. Maier et al. NiagaraST. `http://datalab.cs.pdx.edu/niagara/`, 2010. [Online; accessed 21-December-2010].
15. E. Prud'Hommeaux, A. Seaborne, et al. SPARQL query language for RDF. *W3C working draft*, 4, 2006.
16. J. Reinders. *Intel threading building blocks.* O'Reilly, 2007.
17. Ali A. Safaei and Mostafa S. Haghjoo. Parallel processing of continuous queries over data streams. *Distrib. Parallel Databases*, 28:93–118, December 2010.