Charles University in Prague
Faculty of Mathematics and Physics

Master Thesis

Martin Kruliš

Algorithms for Parallel Searching in XML Datasets

Department of Software Engineering
Malostranské nám. 25
Prague, Czech Republic

Supervisor: RNDr. Jakub Yaghob, Ph.D.
Branch I2: Software Systems

Acknowledgements

First of all, I would like to thank my supervisor, Jakub Yaghob, for introducing me to the world of parallel computation and for his time and guidance which was most helpful. I am very grateful to all my friends with whom I have consulted various issues related to this topic.

I also thank my family for supporting me during my studies, to my girlfriend Háňa for her patience and encouragement, and to my roommate Vašek who supplied me perseveringly with innovative ways of procrastination while I was working.

Finally, I would like to express my gratitude to Martin Mareš who has generously lent me his TeX macros and building scripts and to everyone who read this thesis and helped me with corrections, especially Janča, Milča and Katie.

Martin Kruliš
Prague, July 2009

Table of Contents

Anotace

Název práce: Paralelní vyhledávání nad XML daty

Autor: Martin Kruliš

Katedra: Katedra Softwarového Inženýrství

Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.

e-mail vedoucího: Jakub.Yaghob@mff.cuni.cz

Abstrakt:

Práce se bude zabývat problematikou indexace XML dat a efektivního vyhledávání s indexem. Hlavním těžištěm bude snaha navrhnout datové struktury a algoritmy, které umožní úlohu vyhledávání maximálně paralelizovat a využít tak potenciálu vícejádrových procesorů. Součástí práce bude i pilotní implementace navržených algoritmů.

Dotazování bude probíhat pomocí vlastního jazyka (ne nutně v textové podobě), což umožní používat různé vyhledávací jazyky jako frontendy. Jako ukázka použití bude implementován vzorový frontend umožňující vyhledávání pomocí podmnožiny jazyka XPath.

Klíčová slova: XML, vyhledávání, XPath, dotaz, algoritmus, paralelní, vícevláknové, vyhodnocování

Annotation

Title: Algorithms for Parallel Searching in XML Datasets

Author: Martin Kruliš

Departement: Department of Software Engineering

Supervisor: RNDr. Jakub Yaghob, Ph.D.

Supervisor's e-mail address: Jakub.Yaghob@mff.cuni.cz

Abstract:

This thesis will address the problems of indexing XML datasets and finding effective searching methods for indexed data. Defining data structures and algorithms that take highly parallel approach to the searching problem is considered to be main objective, therefore the implementation may benefit from the power of multicore CPUs. Prototype of such implementation will be presented with the thesis.

Internal query model will be developed as a common layer. Multiple front-ends representing different query languages will be able to operate on the top of this model, thus the engine will not be dependent on any particular language. Sample front-end for a subset of XPath will be implemented along with the prototype.

Keywords: XML, search, XPath, query, algorithm, parallel, concurrent, evaluation

1. Introduction

In 1965 Gordon E. Moore, Intel co-founder, first observed the intriguing phenomenon that the number of transistors, which can be placed on integrated circuit, grows exponentially with time. In fact, this number doubles approximately every two years. It is reasonable to assume that many attributes of microprocessors, such as frequency or cache capacity, are directly proportional to the amount of integrated transistors. Therefore it was believed that speed of processors will also double about every two years. Unfortunately, at the dawn of 21th century this assumption was proven wrong by Intel.

It was discovered that energy consumption and thermal production of a microprocessor is quadratically proportional to its nominal frequency. The frequency escalation trend hit a barrier of about 3Ghz, beyond which the processors based on current architectures consume too much energy and produce too much heat. In order to solve this crisis, processor developers have abandoned the frequency pursuit and focused on optimizing the architecture and creating multicore microchips.

1.1. Thinking Parallel

At present time, dual-core and quad-core processors are quite common in PC configurations. Graphic cards are becoming more sophisticated and the computational power of their tens or hundreds of stream processors can be used for general purpose computing. We have to ask ourselves, what we shall do with all these cores. As the hardware outran the software again, we have to learn new ways of developing applications, designing algorithms, and even thinking about problems. The efficiency and time complexity have become less important and what really matters is how applications scale with more processors.

Writing algorithms, which can properly distribute work so that it can be processed concurrently, is quite a challenge. Parallelism brings many issues that do not concern us in single-threaded applications and which must be addressed properly, especially

- thread management,
- synchronization,
- and memory allocation.

Furthermore, we must not forget that there are problems which may not be parallelized at all, since they are serial in nature. For example reading data from a pipe or computing expression $y = x_1 \otimes x_2 \otimes x_3 \otimes \dots$ where $\otimes$ is nonassociative operation. The most difficult part of parallel programming is to find out which parts of the algorithm are not serial and to exploit the parallelism inside them.

1.2. Looking for a Needle in XML Haystack

This work will focus on problems of searching data in XML [XML] document and explore possible ways of evaluating a search query concurrently. Even though the search engine presented with this thesis is independent on a query language, it has been inspired mostly by XPath [CD+99], the state of the art in XML. Subset of XPath was also implemented as a front-end to our engine (for more details see appendix A).

There are many ways of evaluating XPath queries. Traditional recursive approach may be the simplest for implementation, however, it is usually the slowest one since it leads to exponential time complexity for some queries. Fortunately, there are more efficient ways of processing XPath [GKP05]. These algorithms introduce two general ideas. First, we should determine which parts of context information are used by each subquery and if there are subqueries that do not require the complete context information, some optimizations may be performed. Second, we must prevent unnecessary repetitive evaluation of subqueries in order to maintain polynomial time complexity.

1.3. Objectives

The problem of evaluating queries over XML datasets is rather vast and well beyond the scope of this thesis. There are many details in the specification which affect the problem significantly: Whether we are searching in one XML document or whole set of documents, whether we execute one query at a time or multiple queries at once, whether we have XML data loaded into memory or we read them sequentially from a data stream.

In this work we will attend to simple XPath evaluation of a single query at a time in one XML document. We expect that the document is already loaded into the main memory in a DOM-like [DOM] tree with simple name-based index for XML elements. We will try to design algorithms which would scale optimally with number of processors and implement the most promising ones in our prototype. Finally, we will test them on real and synthetic data and compare them with some known XPath implementations.

In chapter 2, we will analyse problems related to multi-threaded enviroment and possible approaches to scalable XPath evaluation with proper XML representation. Algorithms and data structures used in our prototype are revised in chapter 3 followed by implementation techniques and details in chapter 4. Chapter 5 is dedicated to practical results of our search engine and their interpretation. Finally, in chapter 6, we will summarize contributions and outline possible improvements and further research objectives.

2. Analysis

2.1. Concurrency Issues

Parallelism introduces many issues which must be treated properly in order to design scalable application while keeping the overhead and programming complexity in acceptable limits.

2.1.1. Threading Model

First and the most essential thing we need to consider is how to exploit multiple processor cores. Many operating systems and platforms offer various tools for parallel computation and we have to choose the most suitable ones. However, the technical details of scheduling, context switching and other low-level parts of parallelism will not be discussed in this thesis. Henceforth, we expect that our platform provides us with classical threads and some mechanisms which allow us to create, dispose and synchronize them.

When a workload is divided among multiple threads, we have to decide whether the threads will be created and disposed dynamically, or whether we create fixed amount of threads at startup and keep them until the application terminates. First approach seems more convenient since it does not consume more system resources than it really needs. Fixed number of threads lack the flexibility of dynamic creation. On the other hand, creating and disposing of threads is bound with nontrivial overhead and it is much faster waking sleeping threads then creating them.

Practical results [Rei07] suggest that it is more convenient to choose fixed amount of threads. Question remains how many threads we should allocate. Since we would like the application to scale properly with number of processors and we are also minimizing the overhead, the obvious choice would be to allocate one thread for each logical processor available. This way we will be able to load the hardware entirely while the threads will not struggle for time slices.

When we use a thread pool of fixed size, we must also design strategy how to keep these threads occupied. First of all, we need to decide whether the job planing will be preemptive or nonpreemptive. Preemptive approach would give us full control over the job assignment. Unfortunately, the price for full control is context-switching overhead and vast code complexity. The alternative is to use nonpreemptive planing which will not be difficult to implement. However, when a task is assigned to a thread, it must finish before any other work can be dispatched to that thread. For the sake of simplicity and speed, we shall not examine the preemptive model further and from now on, we will consider only simple nonpreemptive planing.

Finally, we have to decide how to divide workload among the threads. Let us say we have a job of size N, which can be divided into fragments down to size 1, and four threads in the pool. Naturally, the naive solution would be to divide the job into tasks of size $N/4$ and assign each task to one of the threads. This approach will most certainly minimize the overhead and will perform excellently if it takes approximately the same time to process each task. If it turns out that one of the tasks will compute much longer than the others, three threads will be waiting idly until the fourth one finishes.

Better solution is to create many tasks regardless the size of the pool. When a thread becomes idle, it simply takes another task to process. This way the threads keep themselves occupied and since the tasks are very small, none of them will wait too long when the last task is being finished. Unfortunately, the more tasks we have the more time is consumed by their administration and dispatching. If they are too small, the overhead will take over useful work and the whole concept will become quite ineffective, so we have to find the acceptable minimal size of the tasks.

2.1.2. Synchronization

Another serious issue of parallel programming is the unpredictability of execution order. When two threads are executed concurrently, we cannot tell how the instructions will blend.

The deterministic concept of programming requires that the same data on the input will lead to the same results on the output no matter how many threads we use or in what order their instructions are executed. Local state of each thread is isolated from the others so the only real problem are operations accessing shared memory.

Obviously, concurrent reading does not cause any trouble since it is completely idempotent. Concurrent writing or combination of reading and writing operations is usually problematic. For better illustration let us examine the traditional example of code incrementing global variable x executed in two threads simultaneously.

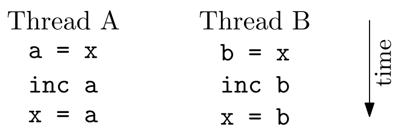
```
Thread A    Thread B
 a = x       b = x
 inc a       inc b
 x = a       x = b
```

Figure 2-1: Race condition example

Let us say the original value of x was 42 so after two increments it should be equal to 44. However, if the instructions are executed as shown above, the final value stored in x will be 43. Problem demonstrated in this example is also known as race condition.

Race condition is a flaw in the code which causes that the result of computation is dependent on the order in which the instructions are processed. The term originates from the idea of two threads “racing” each other. Segments of code which cause race conditions are called *critical sections*. In order to compute results correctly we must ensure *mutual exclusion* of critical sections which operate over the same variables, thus executing these sections atomically.

Synchronization Primitives and Techniques

There are several ways of synchronizing access to memory by excluding concurrent execution of critical sections. The most common methods are

- implicit synchronization,
- atomic operations,
- and locking on mutexes.

Implicit synchronization is the best method of mutual exclusion, but it does not fit every situation. The general idea is to divide the data into disjoint parts and let each thread process a different part. For example, let us have an array A of integers, which we want to increment, and two threads. If we divide A into two halves and let one thread process first half and the other the second half, no other synchronization is required since both threads are operating on different data. Example with incrementing integers might be trivial, however, it is not always easy to find a way how to divide data in order to achieve implicit synchronization.

If the nature of the problem prevents us from synchronizing data implicitly, we have to ensure mutual exclusion by other means – the *synchronization primitives*. For very short critical sections which contain only simple operation (such as incrementation in previous example), the exclusion may be realized by *atomic operation*. Atomic operation is usually hardware specific instruction or combination of instructions which are guaranteed to execute atomically. These instructions are usually quite slower than common instructions as they require some hardware synchronization.

Mutexes

When a critical section cannot be covered by single atomic operation, we use a mutex. *Mutex*[1] is a synchronization primitive with simple semantics. When a thread wants to access memory guarded by mutex, it must lock the mutex before entering critical section. At the end of the section, the lock is released. Mutex ensures that only one thread at a time may hold the lock. Other threads trying to lock the mutex are suspended, waiting for the lock to be released.

There are two types of mutexes. *Spin-lock* mutexes are based on active waiting. Basically, it is a loop which tests whether the lock has been released and terminating only when the thread acquires the lock. Spin-locks are usually implemented using atomic operations, thus completely in userspace.

Classical mutexes use services of operating system which suspend the waiting thread and wake it again once the lock has been released. Spin-locks are suitable for short waiting and they have lower latency. However, waiting on a spin-lock consumes precious processor time.

Atomic operations and classical mutexes may be combined to create more efficient synchronization primitive – *a futex*. Futex (fast userspace mutex) has the same semantics as mutex, but it is almost entirely implemented by atomic operations in userspace. No system call is invoked unless threads collide on the lock and one of them needs to be suspended. Therefore, futex reduces the necessity of using expensive system calls for locking.

[1] Word mutex was created by combination of words *mutual* and *exclusion*.

2.1.3. Memory Allocation

Memory allocation and deallocation are performed quite often by an application. The memory allocator presents serious hotspot, which may cause severe performance problems in multithreaded enviroment. It is obvious that access to memory allocator must be synchronized, otherwise its internal data structures may be corrupted. If concurrent tasks call the allocator too often, the threads will spend a lot of time waiting on synchronization primitive protecting the allocator and creating a convoy effect. Hence, code will be executed almost sequentially and the application will be even slower than single-threaded version.

We can reduce this problem by allocating as much memory as possible before concurrent tasks are executed and by preferring local stack of the thread over the global heap for data structures allocated concurrently. Unfortunately, memory preallocation cannot be used in many situations considering we usually do not know in advance how much memory will be required. Futhermore, systems based on non-uniform memory architecture [NUM] will perform better if each thread allocates its own memory. Memory allocated directly by the thread will be closer to corresponding processor core, thus faster to access.

Scalable Allocator

As the pre-allocation does not seem promising, we will design a scalable memory allocator which uses a different approach. General idea (described in [Rei07]) is to create separated heap for each thread in the pool. When a thread allocates memory, it uses its own heap, thus it does not block nor wait for other threads.

Scalable allocator has a minor problem with memory deallocation, to be more specific, when a thread deallocates memory that has been allocated by another thread. In this case, two possible solutions can be found. Either we allow threads to access foreign heaps or we just mark memory for deallocation and let the owning thread deal with it later.

First option requires that we synchronize access to every heap. Therefore, even if a thread is accessing its own heap, it must lock corresponding mutex to ensure that no other thread will modify it meanwhile. It seems that we have got to all this trouble with multiple heaps for nothing. However, if we expect that disposing of foreign memory is rare, the collision on the mutex protecting a heap would be also very rare, so the performance should improve significantly.

The second solution expects that heap may be accessed only by its owning thread so no mutual exclusion is required. If thread needs to dispose of foreign memory, it does not access the heap but rather add the disposed address into the *'to-dispose'* list of owning thread instead. When a thread disposes (or allocates) its own memory, it also disposes everything on its *'to-dispose'* list. The list itself may be implemented easily by atomic operations.

Both solutions for disposing memory are mode effective than simple allocator with locking. The former requires locking for accessing the heap, but it is also quite simple. The latter is more complicated, but it might perform slightly better under stress conditions.

2.1.4. False Sharing and Cache Optimizations

When optimizing any algorithm for speed, current microprocessor architectures must be taken under advisement, which brings us to another memory-related issue. Main memory is not accessed directly by the processor, but there is a cache (in fact two or three levels of cache) between them. Usually, each processor core has its own cache. A synchronization protocol[2] is in place in order to maintain cache and memory coherency. This protocol may cause serious performance loss if the memory is accessed without respect to caches.

Data are copied between cache and memory in blocks of fixed size called *cache lines*. Cache line is tens of bytes long (usually 64 B) and it cannot be fragmented. If two variables happen to be on the same cache line and each is accessed by a different thread executed on a different core, the cache coherency protocol will get busy. When a variable is accessed, the protocol tries to load the line in dispute into cache of the processor core from which it was accessed. So if both variables are accessed simultaneously, the line will be transferred from one core to another and back again which will dramatically decrease performance.

The described phenomenon is called *false sharing* since both involved threads are not sharing any real data, but due to unfortunate circumstances they are struggling over single cache line.

The best way to prevent this effect is to align and pad every variable which could be a victim of false sharing, so it will be alone on a cache line. On the other hand, it is not a good idea to pad every variable in the application, because such memory dissipation may not reflect positively on performance as well.

We must also be careful when using implicit synchronization. When data are divided for separate threads, we should not split them in the middle of cache line. This may be ensured by simple rule: We will always divide data into blocks of size 2^k where $k \geq 6$, so each block is formed by one or more cache lines.

2.2. XPath Evaluation

XML queries can be evaluated in many different ways. Many of them are quite effective in single-threaded applications, but only a few present enough opportunities for parallelism. We will examine some of these strategies from the perspective of concurrent evaluation.

XPath language is quite extensive. It contains arithmetic and logic operations, relational and equality comparisons, various functions and, most importantly, location paths. Location paths are subqueries that retrieve nodes from XML DOM [DOM] structure, therefore they present the most difficult and the most intriguing component of the language.

2.2.1. XPath Semantics Revision

Before we explore different approaches to the problem, we shall revise the most important definitions in XPath specification [CD+99].

[2] For example MESI protocol used in IA32 architecture.

Evaluation Context

Every XPath query and its subexpression is evaluated with respect to a *context*. A context consists of

- a context *node*,
- two positive integers *position* and *size*,
- variable bindings,
- a function library,
- and a set of namespace declarations.

Since many of these items are not relevant to our work nor they change during evaluation, we reduce the context information to context node, position and size. If we use a node from a node-set as an evaluation context for XPath subexpression without specifying context position and size explicitly, we shall define them implicitly as follows: the context position is the index of context node within original node-set and context size is equal to number of nodes in the set. Context position is also affected by order of nodes in the original node-set. Nodes are arranged in document order or in reverse document order, depending on how they were retrieved (see [CD$^+$99] or appendix A for details).

Location Paths

Location path '$step_1/step_2/\ldots$' is evaluated as follows. The $step_1$ is evaluated using initial evaluation context for the entire path and yields node-set S_1. The $step_2$ is resolved then, taking every node in S_1 as evaluation context, thus yielding set of node-sets $\{S_{2,i} | i \in S_1\}$. Before third step is processed, the nodes are united in single node-set $S_2 = \cup S_{2,i}$, so the recursion is not exploited any further. The set S_2 is used as context set for the third step and so on. Result yielded by the last step is also a result of the whole location path.

Each location step consists of *initial part* (`axis::filter`) followed optionally by a list of predicates. When initial part is resolved, it yields *initial node-set* which is then filtered by predicates. The initial node-set is formed by all nodes that are in relation (defined by the axis) with the context node and match specified filter[3]. A predicate is evaluated separately for every node in the initial node-set as context. If predicate returns *false* for some nodes, these nodes are removed from the initial node-set. Nodes that remain in the initial set after filtering are yielded as the result of the step.

2.2.2. Recursive Approach

Naive implementation of location step evaluation described in XPath language specification leads to *a recursive algorithm*. The algorithm itself does not need to be implemented recursively, however, we will use the term *recursive* for its description of the approach is quite apposite.

Potential problem with recursive evaluation is hidden in predicates. If the initial part of some step yields similar node-set for different contexts, its predicates are evaluated repeatedly for the same nodes, thus doing needless work. We will demonstrate it on practical example.

[3] We will use only name filter which selects nodes of given name.

Exponential Phantom Menace

Let us have a XML document with a root element which has many children named '`a`' (we shall denote them a_i where $i \in \{1 \ldots N_a\}$). Some of the a_i nodes may also have a child named '`b`'. We would like to evaluate simple location path query:

```
/descendant::a/following::a[b]
```

First step ('`descendant::a`') will retrieve all elements named '`a`' from the document, thus the second step is evaluated for every node a_i. For a_1 the '`following`' axis yields nodes $a_2, a_3, \ldots, a_{N_a}$, so the predicate `[b]` will be evaluated $N_a - 1$ times. Unfortunately, for a_2 the axis yields $a_3, \ldots, a_{N_a}$ and the `[b]` will be evaluated again for these nodes ($N_a - 2$ times). Obviously, the predicate will test needlessly too many nodes, so the whole process has $\mathcal{O}({N_a}^2)$ time complexity.

We might argue that the presented query is poorly designed, therefore it deserves to be evaluated slowly. Nevertheless, we must bear in mind that recursive approach may for some queries lead to exponential time complexity $\mathcal{O}(N^k)$ where N is the size of the document and k represents the depth of the query.

On the other hand, recursive approach works fine for common queries, it is quite easy to implement, and it does not consume more than $\mathcal{O}(k \cdot N)$ additional memory for temporary results.

Parallelization

Recursive approach does not offer any opportunity how to parallelize evaluation of whole query, however, we may still parallelize each location step. The most promising is evaluation of predicates. Each predicate is evaluated separately, therefore we may create a task for every node in the initial set and execute them concurrently. The initial part of the step might be parallelized as well and we will discuss this matter later in 2.3. Finally, the union of multiple node-sets (at the end of location step) can be performed concurrently with proper data structures.

2.2.3. Iterative Methods

Iterative methods try completely different approach to processing XPath. Their main objective is to eliminate the greatest problem of recursive algorithm – redundant work which leads to exponential time complexity. We will revise methods presented in [GKP05] and evaluate them from the perspective of parallelism.

Bottom-up Evaluation

Bottom-up algorithm evaluates each atomic subquery for every possible context and records computed values into *context-value tables*. Leaf atomic expressions are evaluated first, then expressions that require intermediate values from other subexpressions are computed by joining data in value tables. When table for the whole query is completed, the result is simply found on the proper line. We shall demonstrate it on example from [GKP05]. Let us have following location path query.

```
descendant::a/following-sibling::*[position() != last()]
```

By processing the query we obtain parse tree with six subexpressions:

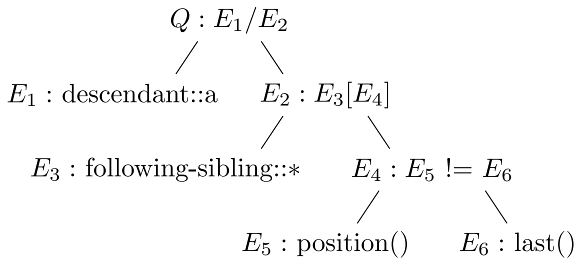

Figure 2-2: Parse tree

Each expression E_i has its own table $T(E_i)$. Tables for the leaf expressions E_1, E_3, E_5 and E_6 may be resolved immediately. Tables for Q, E_2 and E_4 are computed afterwards by joining tables of underlying expressions.

This algorithm was designed especially to prove that iterative approach leads to polynomial time complexity. If we denote N number of nodes in processed XML document, every table has $\mathcal{O}(N^3)$ rows at most, while in the worst case each row contains node-set with N nodes, hence each table requires $\mathcal{O}(N^4)$ space.

The most complex joining operation is in expression Q (location step E_1/E_2). To compute each row in $T(Q)$ we need to lookup node-set in $T(E_1)$ and use every node in it as a context for $T(E_2)$. There are up to N nodes in this node-set so we need to merge $\mathcal{O}(N)$ lines from $T(E_2)$ each containing at most N nodes. As we said, this must be done for every row in $T(Q)$, thus final time complexity of this join is $\mathcal{O}(N^5)$.

We have shown only the most important indices why iterative methods should have polynomial time complexity. Formal proof can be found in [GKP05].

Top-down Evaluation

Even though bottom-up algorithm runs in polynomial time, it is quite inefficient since many rows in context-value tables are computed needlessly. Top-down method shares the main idea with bottom-up algorithm and removes some unnecessary work.

The evaluation proceeds from the top of the parse tree, starting with the root expression. Before each subexpression is processed, set of all required contexts is prepared. Then, the subexpression is evaluated solely for those contexts, thus only necessary intermediate values are computed. When all descendants of an expression are evaluated, its result is joined from intermediate values the same way as in the bottom-up algorithm.

Described method may resemble the recursive approach. The difference is that we do not evaluate subexpressions repeatedly using one contexts at a time, but rather evaluate them "at once" for whole vector of contexts. This way the redundant work is avoided and exponential boom prevented. According to [GKP05] the time complexity of top-down evaluation algorithm is $\mathcal{O}(N^4 \cdot q^2)$ where q is the number of atomic expressions in Q.

Min-context Algorithm

The final stage of evolution in iterative methods presents the minimal-context algorithm. It brings many improvements into the top-down evaluation, most importantly the context restriction.

Some of the atomic expressions do not depend on the whole context information. For example *position*() function requires only the position value from the context, completely ignoring context node and size, location step requires only context node, and first location step of absolute location path does not require any context information at all. Knowing this, we may optimize the amount of intermediate values computed by top-down algorithm.

Even though min-context optimization may improve evaluation speed, the asymptotic time complexity remains $\mathcal{O}(N^4 \cdot q^2)$ for the worst case. Space complexity is improved from $\mathcal{O}(N^3 \cdot q^2)$ (for top-down evaluation) to $\mathcal{O}(N^2 \cdot q^2)$ (see [GKP05] for more details).

Summary

Iterative approach offers quite pleasant polynomial time complexity. The cost of this convenience is more difficult implementation and quadratic space complexity which renders this methods useless for large documents. On the other hand, iterative methods present many opportunities for parallelism.

When atomic expression is evaluated for given set of context, each result could be computed by independent task, thus by independent thread. Written data do not collide so implicit synchronization can be used. Joining intermediate values into final result by merging node-sets could also be parallelized with proper data structures.

At topmost level, the bottom-up evaluation is most promising. All leaf expressions can be evaluated concurrently and other expressions could be parallelized too when dependency issues are solved. In top-down evaluation the concurrent processing of subqueries is more complicated. Location steps E_x/E_y and predicates $E_x[E_y]$ cannot be parallelized at all, because in both cases intermediate values of the first subexpression are required for establishing context set for evaluation of the second one.

Iterative methods seem to be better than recursive approach from perspective of parallelism. Unfortunately, their space complexity $\mathcal{O}(N^2 \cdot q^2)$ may cause problems with large documents. To be specific, DOM structure with 10^5 nodes or more cannot be processed by iterative algorithms on present hardware.

2.2.4. Plumbing a Pipeline

Methods presented so far were originally developed to process XPath in single-threaded applications. Now we will take more direct approach to parallelism and try to create algorithm using the well-known design pattern – a pipeline.

Pipeline Concept

Pipeline is common design pattern that was inspired by manufacturing assembly line. It is formed by acyclic oriented graph where vertices are stages and edges represent the direction in which data flows among the stages. Each stage processes data in some way and passes them on. Also, there is at least one stage which generates the data and at least one stage that collects the results.

Stages may work on the data concurrently without explicit synchronization. Each stage is processing different data block which will not be passed further before the stage completes its job. Furthermore, some of the stages may process multiple data blocks at a time if the task performed on one data block does not depend on the other blocks from same collection.

There are many ways how to implement pipeline. However, technical details about mapping stages to threads or synchronizing dataflow are beyond the scope of this work. The pipeline problematic is extensively covered in [Rei07].

Compiling Query into Pipeline

Questions brought to our attention are whether and how could a pipeline be used to compute XPath queries. The general idea is not to use one universal pipeline for every XPath expression but to compile a specialized one that will only compute results for a given query.

First, we decompose the query to atomic expressions as shown in 2.2.3. Almost every expression is turned into stage and relations among expressions are expressed by stage connections. Location steps E_x/E_y are created only by connecting the last stage of pipeline segment computing E_x to the first stage of E_y segment. Complex expressions (such as predicates, binary expressions, etc.) can be handled in two ways:

- hierarchial approach (creating nested pipelines)
- decomposition of problematic operations into two or more stages (encapsulating pipeline segments of nested expressions)

We will demonstrate both approaches on a compilation of following query:

```
descendant::a/following::b[c/@ref = @id]
```

Hierarchial approach creates single stage for composed expressions in the top-level pipeline and all subqueries are plugged into nested pipelines. Thus the expression looks like atomic task from outside while it is evaluated by another pipeline inside. If the nested pipeline would be too simple to actually parallelize any work, any other previous method (i.e. recursive or iterative) may be used locally.

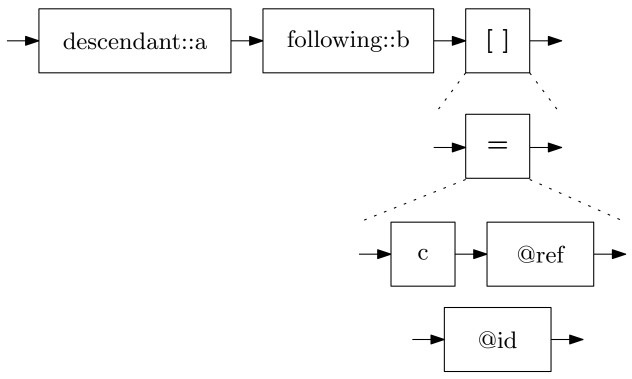

Figure 2-3: Nested pipelines

Nesting concept increases overhead while the pipeline effect is diminished. The more stages we have interconnected the more tasks may be computed simultaneously. We could solve this by integrating all pipelines into the main pipeline. In order to do that, we need to split some of the stages into two – opening and ending. Opening part tags the data and prepares them to be processed by subexpression segments of the pipeline. Ending part removes tags and summarise intermediate values into the result of composed expression.

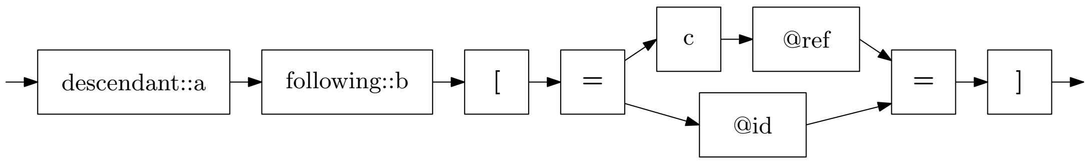

Figure 2-4: Pipeline with encapsulating stages

We could combine both approaches and there might also be other methods how to construct pipeline from given query. However, for the sake of space, we will not explore them more thoroughly.

Summary

Despite the fact we have not demonstrated the pipeline construction more formally nor we have solved every technical detail, the pipeline concept seems to have a lot of potential. Stage can start working on partial result of previous stage before the previous one yields all data blocks, thus more work can be done concurrently.

On the other hand, some of the stages might require all data blocks before they can proceed with their own work. For example the *last*() function used inside a predicate needs to know the size of filtered node-set, thus it must wait for all data blocks yielded by preceding stage.

Another issue which must be addressed is the one of redundant computations. We have observed in recursive approach that repetitive evaluation of predicates for the same contexts leads to exponential time complexity. Iterative methods solved this problem by evaluating each subexpression exactly once for whole vector of contexts. The evaluated expression has full information about requested results, thus it could organize its work and no intermediate values are computed more than once.

In the pipeline, stage starts computing intermediate values before it knows all the contexts, therefore it might be unavoidable to compute some of the intermediate values repeatedly. Fortunately, every stage is usually evaluating block of contexts instead of just one, thus the problem is not as severe as in recursive approach. We can also integrate cache in every stage, so when an intermediate value is returned, it is cached and never computed again.

Asymptotic space complexity will most certainly not exceed the complexity of bottom-up algorithm even if we use a cache table for each stage, but the real memory consumption will be probably higher.

The pipeline concept was presented just for comparison with traditional methods. Even though this topic is worth further exploration, we shall focus on more conservative and traditional algorithms.

2.3. Indexing DOM Structure

When XML document is loaded into memory, it is represented by Document Object Model [DOM]. DOM is in fact a tree and its nodes represent elements, attributes and other XML data. Each element node can reach only its parent, siblings, children and attributes directly (in constant time). DOM does not provide us with more complex operations. So we have to implement them ourselves by traversing the tree.

Since retrieving nodes based on their axis-defined relations is essential to location step evaluation, it may be beneficial to create an index over DOM structure which speeds up these operations. There are many types of indices so we cannot possibly cover them all. We will examine only the traditional and a few specialized types.

2.3.1. Left-right-depth Index

Left-right-depth index (we shall denote it also as *LRD*) is often used in many XML algorithms to determine the relation of two nodes in constant time. Variations on this index can be found for example in [Gru02], [KPS04] or [GKP05]. The index is based on tagging nodes of DOM structure with three integral numbers – left-value, right-value and node depth. We may choose to tag only element nodes or every node in the tree.

Left-values are assigned to nodes incrementally by preorder traversal of the DOM tree and they reflect position of the node in the document. If we are tagging attribute nodes as well, we should tag them before children elements of current node. *Right-value* is equal to largest left-value of all descendants (and attributes). Nodes without any descendants or attributes have their left-value equal to their right-value. *Node depth* is simply defined as depth in the DOM tree, where document node has depth 0, root element 1, its children 2 and so on.

Identifying Relations

As promised, described index allows us to determine a relation between two nodes in constant time. Let us have nodes u and v waiting for comparison.

- If $\mathit{left}(u) < \mathit{left}(v) \wedge \mathit{right}(u) \geq \mathit{right}(v)$, node u is ancestor of v (and v is descendant of u).
- If $\mathit{right}(u) < \mathit{left}(v)$, node u is preceding v (and v is following u).

By using the depth value we are able to test even more complicated relations. For example whether v is grandchild of u: $\mathit{left}(u) < \mathit{left}(v) \wedge \mathit{right}(u) \geq \mathit{right}(v) \wedge \mathit{depth}(u) = \mathit{depth}(v) - 2$. There are many applications of LRD and we will exploit them later, mostly in combination with other indices.

Two-dimensional Representation

We can use left-value and right-value to place nodes into two-dimensional grid as suggested in [Gru02]. Every node v is represented by one point $(\mathit{left}(v), \mathit{right}(v))$ of two-dimensional space U^2 where U is set of all identifiers (left-values) in the document.

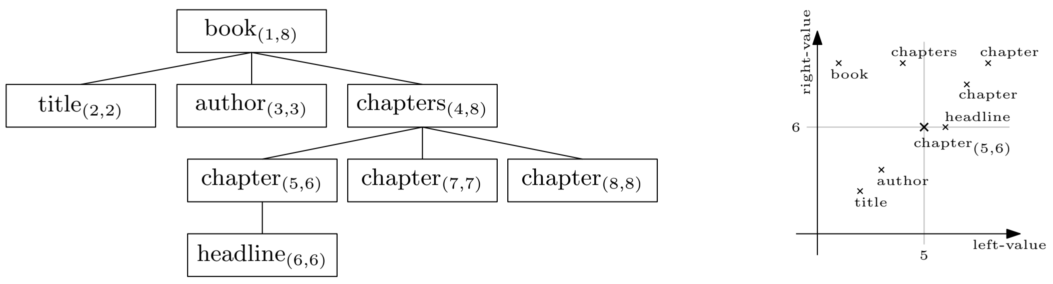

Figure 2-5: Two-dimensional node representation

We can make a simple observation about XPath axes. If we unite all major axes (*ancestor*, *descendant*, *following* and *preceding*) and the context node, we will get the whole document. Furthermore, each two sets in this union have empty intersection. Hence, major axes create perfect decomposition of the document.

$$\begin{aligned} &\mathit{ancestor}(v) \cup \mathit{descendant}(v) \cup \mathit{following}(v) \cup \mathit{preceding}(v) \cup v = \\ &= \{u | u \in \mathit{document}(v) \wedge \mathit{type}(u) = \mathit{element}\} \end{aligned}$$

If we take context node c and divide our grid by horizontal and vertical line so that both lines intersect the point representing c, each of four created quadrants corresponds to one of the major axes. The right-bottom quadrant represents *descendant* axis as it contains all nodes with left-values greater than $\mathit{left}(c)$ and right-values lesser than $\mathit{right}(c)$. Analogically, the upper-left nodes are ancestors of the context node.

The bottom-left quadrant contains all nodes with their left-values lesser than $\mathit{left}(c)$ and right-values lesser than $\mathit{right}(c)$. Since left-right value ranges cannot partially overlap, the right-values must be also lesser than $\mathit{left}(c)$, so it obviously complies with the very definition of preceding axis described earlier. Analogically, we determine that the last remaining (upper-right) quadrant corresponds to *following* axis.

Now we need an effective method how to accelerate the major axes using this index. Obvious choice is to use R-trees or similar multi-dimensional trees for representing points in two-dimensional space. However, these trees are quite complex for implementation and present little potential for parallelisation, thus we shall not explore them further.

2.3.2. Element Index

The most common location step filter is the name filter. Step with such filter adds only nodes with specific name into initial node-set. Therefore, it might be useful to create element index based on names.

The index is formed by hash table where element names are keys. Each name holds a list of references to the DOM structure pointing to all elements with this name.

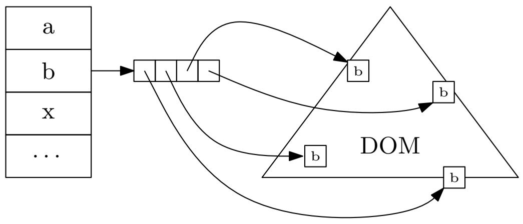

Figure 2-6: Element index

Question remains how the links in the reference list should be ordered. The most obvious solution would be to order them by the position of nodes to which they point to. As we will demonstrate later, this ordering is very useful.

We should also define how the index speeds up the evaluation of location paths. In order to do that, we will combine element index with LRD described in 2.3.1. Let us take a look on each axis and how it can benefit from these indices.

Implementing XPath Axes

Implementation of *descendant* axis is quite simple. The reference list is ordered by position of the nodes in the document, thus by their left-values. Furthermore, we know that every descendant of element e has its left-value in range $(\mathit{left}(e), \mathit{right}(e)\rangle$. Such nodes are definitely stored in continuous subrange of the reference list. Since the list is ordered, a binary search algorithm can be applied to find both beginning and end of the subrange.

Following axis uses the same approach. If we have an element e, all following nodes have their left-values greater than $\mathit{right}(e)$. Thus we just find first reference on the list which fulfils this condition and then we take all nodes from its position to the end of the list.

Preceding axis is little more complicated than the previous two. Let us say we are looking for all predecessors of context node c. In order to determine whether node n precedes c, we need to compare $\mathit{right}(n)$ with $\mathit{left}(c)$. Unfortunately, the reference list is not ordered by right-values.

Since the right-value is always greater or equal left-value (of the same node), we will use all nodes with left-value lesser than $\mathit{left}(c)$ as *candidates*. These candidates contain two types of nodes – predecessors of c and ancestors of c. We can test each candidate for conditions described earlier at LRD index and filter out the predecessors.

Of course, we might create second element index where references will be ordered by right-values and implement evaluation of *preceding* axis analogically to *following* axis. This is classical time-space dilemma and final decision will depend on how much memory we can use and whether *preceding* axis is often used in our XPath queries.

Ancestor axis cannot be effectively accelerated by this index. On the other hand, we expect that XML documents are shallow, therefore the best way to find all ancestors of context node is to follow parental links in DOM structure and filter nodes one by one.

Children and Sibling Axes

Techniques described for *descendant*, *following* and *preceding* axes could be extended also to *child*, *following-sibling* and *preceding-sibling* relations. In order to do that, an element index must be also built for every node with children. Such index contains of course only children elements of associated node.

2.3.3. Special Indices

Finally, we shall examine two rather special DOM indices. Each one is designed for different purpose, thus having different advantages and disadvantages.

Index for Descendant Paths

Child and descendant axes are the most exploited ones in XPath queries. Therefore, it might be useful to build a special index just for them. We shall revise a method described in [CSF$^+$01] for indexing descendant paths.

Let us have location path `/a/b/c`. Common algorithms evaluate the path step by step. Root element '`a`' will be probed for all children '`b`' and they will be used to retrieve all their '`c`' children. The result is computed in three steps and many intermediate values (e.g. set containing '`b`' nodes) are computed and then dismissed. It is also possible that intermediate values are much larger than the final result. If the root element '`a`' contains many '`b`' children, but only few of them have direct descendant '`c`', we will spend a lot of time creating and processing intermediate set with '`b`' nodes needlessly. It would be better if we could retrieve final node-set directly without computing intermediate values.

We will create an index structure which will help us to retrieve node-sets by their descendant paths. Henceforth, we denote P_n a descendant path leading from root to node n. This path is uniquely defined by the node n, because there is exactly one path between any two nodes in connected tree. Each node n in the DOM is inserted into the index using sequence of all element names on P_n as its key. Nodes with paths P_i and P_j which have the same sequence of element names are rendered under the same key, thus in one node-set.

Implementing effective index structure for this application is not quite easy. One of the possibilities is to use treelike structure called *Index Fabric* which is based on Patricia Trie. This structure is thoroughly described in [CSF$^+$01].

Described techniques encode and accelerate only *raw paths* (absolute descendant paths). Whole concept may be improved by encoding *refined paths* instead (see [CSF$^+$01]). This improvement allows us to accelerate also descendant paths with wildcards or to optimize frequently occurring patterns.

Multi-dimensional Index for XPath Axes

Idea of addressing nodes by their root-to-node paths can be exploited even further as suggested in [KPS04]. In following revision, we expect that DOM structure has been already indexed by LRD index.

A *path* to node n is sequence of identifiers $\mathit{left}(\mathit{root})$, $\mathit{left}(u_1)$, $\mathit{left}(u_2)$, $\ldots$, $\mathit{left}(u_{d(n)-1})$, $\mathit{left}(n)$ where $d(n)$ is depth of the node n (thus length of the path) and u_i are nodes between root and n lying on the path. We can omit the root since it is present in every path and represent every node as a vector

$$(\mathit{left}(u_1), \mathit{left}(u_2), \ldots, \mathit{left}(u_{d(n)-1}), \mathit{left}(n), 0, \ldots, 0)_D$$

of length D where $D = \max(d(n)), n \in \mathit{document}$ (vectors for nodes with $d(n) < D$ are padded with zeroes). This representation assigns each node single point in U^D space where U is set of all left-values in the document.

Furthermore, we assign numeric identifier $\mathit{Id}(u)$ to each element name (by incremental indexing). Path to every node n has associated *labeled path* $\mathit{Id}(u_1)$, $\mathit{Id}(u_2)$, $\ldots$, $\mathit{Id}(u_{d(n)-1}$, $\mathit{Id}(n))$, which can be transformed into vector the same way the real path is. Each real path has exactly one corresponding labeled path, but labeled path may stand for multiple real paths.

Having space of real paths and associated labeled paths, we can implement location step axis using simple point and range queries. XPath axis evaluation has three steps:

- Translate each node name to identifier assigned by *Id* relation.
- Retrieve all labeled paths using point and range queries.
- Join retrieved labeled paths with real paths to obtain results.

Range and point queries are best solved using multi-dimensional trees such as UB-trees, R-trees or R*-trees. Even though these structures may be used in our case as well, the solution was originally designed for database systems which can process multi-dimensional data natively.

Summary

Suggested special indices may improve the performance especially on large documents, however, they are not that much interesting from parallelisation point of view. They are also optimized for XML data that does not fit into main memory, which is of little use for us. Finally, both indices require nontrivial data structures that are bound with vast coding complexity, thus we will not use these indices in our implementation.

2.4. Redundant Computations

The more threads we have the more difficult is to keep them occupied. When thread becomes idle and there is no regular task which can be dispatched, we may still provide the thread with some work even though we do not know yet whether the computed results will be used. Following example demonstrates the problem and implied dilemma.

2.4.1. Minimal Evaluation Rule vs. Parallelism

Let us have a logical query $Op_1 \wedge Op_2$ where $Op_{1,2}$ are operands which both yield a Boolean value. A minimal evaluation rule is applied in most languages. The Op_1 is evaluated first, and if it resolves as *false*, Op_2 will not be evaluated at all since the result of whole expression is already known. If the first operand yields *true*, we must evaluate second operand to determine the result. In query languages we may assume that evaluating an operand is an idempotent operation with no side-effects, therefore second operand may be always evaluated without changing semantics of the expression.

This rises fair question whether we should execute both operands simultaneously to speed up the evaluation process. If both operands are evaluated in approximately the same time and we have a spare thread, the result will be always computed by the time Op_1 is evaluated. This way we did not worsen the time complexity in case Op_1 yields *false* and we have reduced it to the half in case Op_1 yields *true*.

Unfortunately, there is also another side to this idea. It is quite rare to have two operands which are evaluated for approximately the same time. If the first operand resolves much faster and yields *false*, we will still have to wait for the second operand to finish since we do not have any means how to terminate tasks prematurely in nonpreemptive scheduling system. Furthermore, the second task may slow down evaluation of the first one thanks to hardware issues (cache coherency, logic units sharing, etc.). Last but not the least problem is the overhead bound to starting two tasks simultaneously and waiting for them to finish. If both operands are resolved quickly, the overhead may exceed useful work, thus leaving parallel version even slower than serial evaluation.

We can partially bypass at least the first of the problems by polling technique. The procedure which evaluates second operand shall check periodically whether the first one has already finished and terminate its execution if Op_1 has yielded *false*. However, polling has also certain overhead, thus it may slow down evaluation even further, and there is little we can do about the other issues.

There are too many unpredictable details, hence this dilemma cannot be decided by theoretical approach. It is also possible that some problems may benefit from redundant computations while others will not. We will test this concept on parallel evaluation of predicates by implementing both redundant and sequential approach and measuring time consumed by evaluation in chapter 5.5.

3. Algorithms and Data Structures

First chapter (3.1) is dedicated to revision of standard parallel design patterns described in Threading Building Blocks [Rei07] and implemented in TBB libraries [tbb]. Understanding of these patterns is required since they are used quite often in our work.

Following chapters describe all algorithms, data structures, techniques and methods used in our implementation of XPath. We will also expose their advantages and disadvantages in comparison with alternatives.

3.1. Implementing Parallelism

First of all, the issues of parallelism need to be solved. Developing a complex framework for concurrent computations is a task well beyond the scope of this thesis. We shall use solution implemented by Intel in the Threading Building Blocks libraries [tbb]. All algorithms and data structures described in this section may also be found in TBB book [Rei07].

3.1.1. Job Decomposition and Dispatching

Job decomposition depends especially on chosen threading model. As suggested in 2.1.1, the best solution is to create one thread for each logical processor core, so we can fully exploit the hardware potential while avoiding oversubscription and reducing the overhead to minimum. Remaining issue is how the workload should be divided among the threads in order to keep them all occupied.

Tasks

Work shall be divided into small fragments called *tasks*. Each task is in fact a functor object which caries reference to code that shall be executed and to data being processed by the code. Tasks are indivisible from logical point of view, however, they may spawn other tasks.

Usually, when work is decomposed into several tasks, it is necessary to synchronize their termination. In other words, we should have a mechanism which detects that all tasks working on the same job have finished, so another job may be started or resumed. TBB [Rei07] presents two task patterns for these situations.

Blocking style presents classical recurrence pattern. When task spawns some children, it yields its thread and waits on the stack until all its children terminate.

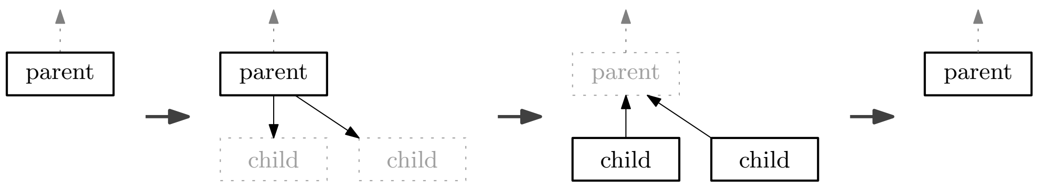

Figure 3-1: Blocking concept

Blocking pattern has one disadvantage. The parent task is waiting on the stack until its children terminate. This may lead to unnecessary memory consumption or, in extreme situations, even to stack overflow. On the other hand, the intermediate state of the parent task may be held simply on the stack without necessity to copy it somewhere else.

If blocking concept is not desired, *continuation-passing* style may be used. In this case, parent task does not spawn only children, but a *continuation* task as well. All references from children are set to the continuation instead of the parent. Furthermore, uplink reference to grandparent task is transferred from parent task to the continuation. Parent task does not wait for children but it finishes its own work and terminates. Since it has no references nor any references point to it, the parent is then disposed.

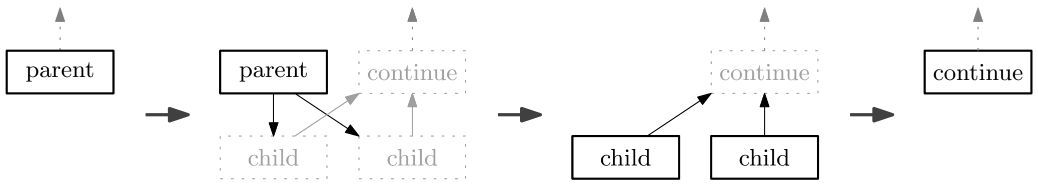

Figure 3-2: Continuation-passing concept

This way the parent may also do some useful work before terminating and it does not block any memory on the stack. However, any intermediate state required for continuation must be transferred to newly created task.

Scheduling

As we have seen in previous two recurrence patterns, the tasks are organized in directed graph. Each task forms a vertex with one outgoing edge, pointing to its parent. A task has also a *refcount* indicator which keeps number of references pointing at it and a *depth* value which resembles depth in the graph. The depth is usually equal to the depth of parent task plus one. Root task does not have any pointer to parent and its depth is 0.

Each thread has its own *ready pool* where tasks ready to run are kept. The pool is organized as an array of lists where the array is indexed by task's depth, thus it keeps tasks with the same depth in one list.

When last reference pointing to a task is removed (so its refcount reaches 0) the task is marked as ready for execution and added into ready pool of a thread which created the task. Ready tasks are picked one by one and executed nonpreemptively by corresponding threads.

Main priority of the scheduler is minimizing both memory demands and cross-thread communication. To achieve this, tasks are executed in depth-first order. When thread is picking another task for execution, first item in the deepest non-empty list is selected. This approach has two great advantages:

- Optimizes usage of caches (the deepest task are the most recently created, thus they are hottest in the cache)
- Minimizes memory usage (the task trees are usually broad and shallow)

Finally, we will address the issue of load balancing. When task is marked as ready to execute, it is added into ready pool of its owning thread. Tasks are not “donated” into other pools. Therefore, situation may happen that one thread is quite busy while another relaxes. To avoid this, a *task stealing* technique is implemented.

When thread does not have any tasks in its pool, it tries to steal one from another thread. Victim is chosen as the first shallowest task of a random pool. This way a breadth-first unfolding is achieved since shallowest tasks will most likely spawn many descendants. If no victim is found, the thread is suspended for a short period of time and then the task stealing is tried again.

Described recurrence patterns are not the only ways how the tasks may be handled, and we have not presented every detail of the scheduler. However, no more complex patterns are needed for our implementation nor we require better understanding of TBB internals. All these issues are vastly elaborated in [Rei07].

3.1.2. Standard Loop Parallelization

Although the task scheduler offers solid basis for almost any parallel algorithm, more conservative approach may sometimes be more beneficial. It seems that many common problems can be solved by more strict patterns. Therefore, TBB implements three templates for parallelization of standard loops.

Parallel For

The most simple pattern is *parallel for* which performs concurrent execution of standard ‘for’ loop. We shall demonstrate its function on an example. Let us have an array[1] A with n items and we need to invoke function *foo*() on each item in A. Normally, we would write simple two-line code:

```
1. For $i \leftarrow 0$ to $n - 1$:
2.     $foo(A[i])$
```

If the *foo*() is thread-safe and operates only with its argument, we may invoke any number of *foo*s on different items concurrently. Trivial solution would be to create a task for each item in A. However, if the items are small or execution of *foo* takes only a few instructions, it would be better to create task for small groups of items. If the items in group are aligned on cache lines and the whole task spans over 10^5 instructions at least, both false sharing and overhead problem will be minimized. Appropriate number of items processed by one task is also called *grain size* and we shall denote it g.

```
Algorithm: ParallelFor
1. Determine appropriate grain size $g$
2. Create $\lceil n/g \rceil$ tasks with ranges $\langle 0, g-1\rangle, \langle g, 2g-1\rangle, \ldots$
3. Spawn tasks and wait for them to terminate
```

While each task implements simple for-loop:

```
1. For $i \leftarrow rangeStart$ to $rangeEnd$:
2.     $foo(A[i])$
```

[1] We shall always index arrays from 0 to $size - 1$.

Previous algorithm demonstrates how parallel for is implemented using blocking style recurrence pattern. Continuation-passing style may be used as well, however, it might be difficult to transfer intermediate state from parent to continuation task.

3.1.3. Parallel Reduce

The parallel for loop may be used only for the most trivial tasks. Different patterns are required in more difficult situations. Sometimes, even as simple algorithm as finding minimum of an array requires special treatment.

Minimum

Let us assume we would like to find minimum from array A. Serial code would be quite simple:

```
Input: Array $A[0 \dots n-1]$ (assume $n > 1$)
Output: Minimum $m$
  1. $m \leftarrow A[0]$
  2. For $i \leftarrow 1$ to $n-1$:
  3.     If $m > A[i]$: $m \leftarrow A[i]$
```

If we rewrite serial code using parallel for directly, each task shall perform the following code. Let us assume we have global variable m shared among the tasks, which is initialized by the main thread before all tasks are spawned.

```
Input: Array $A$
Output: Minimum stored in global variable $m$
  1. For $i \leftarrow rangeStart$ to $rangeEnd$:
  2.     Lock $m$
  3.     If $m > A[i]$: $m \leftarrow A[i]$
  4.     Unlock $m$
```

It is obvious that access to m must be synchronized in order to avoid race condition, however, synchronization will force m to serious hotspot and parallel version will run even slower than original serial code due to locking overhead. The solution may be slightly improved:

```
  1. $m' \leftarrow A[rangeStart]$
  2. For $i \leftarrow rangeStart + 1$ to $rangeEnd$:
  3.     If $m' > A[i]$: $m' \leftarrow A[i]$
  4. Lock $m$
  5. If $m > m'$: $m \leftarrow m'$
  6. Unlock $m$
```

This will reduce the amount of locking, allowing tasks to run at least part of their code concurrently, but it still forces them to wait on a lock in the end. It is clear now that we need more sophisticated pattern than parallel for to solve our problem efficiently. We will try the divide and conquer approach.

Divide and Conquer Solution

The minimum can be computed also by recursive decomposition. We split A into two ranges of approximately the same size and find minima m_1, m_2 for both of them. Them we determine minimum of A as $m = \min(m_1, m_2)$. The recursion is stopped on ranges containing only one item since this item is also minimum of the range.

```
Algorithm: RecursiveMin
Input: Array $A$, subrange $\langle l, r\rangle$ of the $A$
Output: Minimum $m$
  1. If $l = r$:
  2.        $m \leftarrow A[l]$, return $m$ and terminate
  3. $c \leftarrow \lfloor (l + r)/2 \rfloor$   (center of the range)
  4. $m_1 \leftarrow \mathit{RecursiveMin}(A, \langle l, c\rangle)$
  5. $m_2 \leftarrow \mathit{RecursiveMin}(A, \langle c + 1, r\rangle)$
  6. $m \leftarrow \min(m_1, m_2)$ and return $m$
```

Previous algorithm can be parallelized much better since m_1 and m_2 may be computed concurrently. However, in order to keep tasks effective, we should stop recursive unfolding sooner and use standard iterative approach for small ranges. The code of a task with blocking style pattern looks like this:

```
Algorithm: ParallelMin
Input: Array $A$, subrange $\langle l, r\rangle$ of the $A$, grain size $g$
Output: Minimum $m$
  1. If $r - l < g$:   (range is too small to be decomposed further)
  2.        Compute $m$ by simple loop and terminate
  3. $c \leftarrow \lfloor (l + r)/2 \rfloor$   (center of the range)
  4. Spawn task $\mathit{ParallelMin}(A, \langle l, c\rangle, g)$ computing $m_1$
  5. Spawn task $\mathit{ParallelMin}(A, \langle c + 1, r\rangle, g)$ computing $m_2$
  6. Wait for both tasks to terminate
  7. $m \leftarrow \min(m_1, m_2)$ and return $m$
```

Synchronization of access to variable m was completely replaced by synchronization of tasks which is provided by the scheduler and unavoidable even in parallel for template. Also some merges of immediate m values may be done concurrently, thus faster.

Parallel Reduce Template

All algorithms based on divide and conquer paradigm can be divided into three parts: problem decomposition, recursive solution of subproblems, and merging intermediate values from subproblems into the result. As we have shown in previous example, subproblems may be solved concurrently.

Implementations of different divide and conquer algorithms have many similarities. In order to avoid redundancies in written code and to streamline programming, we will design a template called *parallel reduce*. This template accepts one task object with following routines which are corresponding to phases of divide and conquer concept:

- *split* operation (decomposition)
- main routine (where actual work is done when decomposition is stopped)
- *join* operation (composition)

Split operation divides task into two and the work is evenly distributed between original and newly created task. If the task is small enough that further decomposition is not desired, the main routine is used to compute intermediate result of the task. When previously splitted tasks are completed, join operation merges their results.

The greatest benefit of presented template is that programmer has to write only three simple routines, leaving everything else on the parallel reduce. Here is generic code executed by the task.

Algorithm: T.evaluate()

```
1. If $T.size > g$:
2.     $T' \leftarrow T.split()$   (divide work between $T$ and newly created $T'$)
3.     Spawn new task $T'$
4.     $T.evaluate()$   (evaluate modified $T$ recursively)
5.     Wait for $T'$
6.     $T.join(T')$
7. Else:
8.     Invoke the main routine of $T$
```

Presented solution seems similar to the trivial implementation with parallel for, but it has two major advantages. First, no locking is required since all synchronization is achieved implicitly or by task dependencies. Second, thanks to the task stealing mechanism, some of the split and join operations are executed concurrently.

We have presented only one trivial application of parallel reduce, but it may be used for more complex algorithms like Quick sort, multiplication of long numbers etc.

3.1.4. Parallel Scan

Last algorithm template, the *parallel scan*, is designated for special category of problems – computation of parallel prefix. Let $\oplus$ be associative operation with left-identity element $id_\oplus$. Furthermore, let us have arrays X and Y, both of size n. We would like to compute values of Y by following formula:

- $Y[0] = id_\oplus \oplus X[0]$
- $Y[i] = Y[i-1] \oplus X[i]$ (for $i > 0$)

In serial programming, a simple loop can handle this operation:

```
1. $t \leftarrow id_\oplus$   (accumulator variable)
2. For $i \leftarrow 0$ to $n-1$:
3.     $t \leftarrow t \oplus X[i]$
4.     $Y[i] \leftarrow t$
```

First examination of this code does not reveal any possible ways how this problem may be parallelized. Every computed value directly depends on the previous one, thus it cannot be computed concurrently. However, the scan may be parallelized if we suffer some redundancy.

Main Idea

First, we shall divide interval $\langle 0, n-1 \rangle$ into four ranges of approximately equal size. The algorithm will proceed in two phases. In the first phase, we precompute a $\oplus$ sum of X for each range and denote s_i the sum of i-th range. Then we can use s_1 as starting value for second range, $s_1 \oplus s_2$ starting value for third range and $s_1 \oplus s_2 \oplus s_3$ for the last range. So in the second phase, we may compute Y values for each range concurrently using four threads.

Obviously, the s_1 sum is also last value of the first range in Y array, therefore we can compute Y values for first range in preprocessing phase and pick s_1 from Y without doing the same job twice. Furthermore, value s_4 is not used anywhere, thus we need not to compute it. The progression of parallel scan is depicted on following diagram. We have denoted first phase as *pre-scan* and the second phase *final scan*.

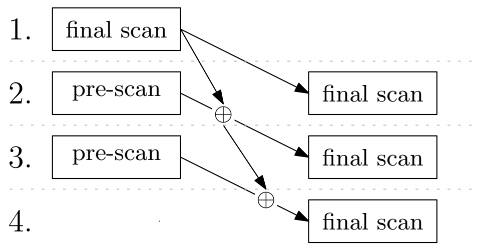

Figure 3-3: Computation of parallel scan

Each task does one quarter of original work and there are six tasks. Thus the parallel scan does 1.5× the original work. However, if both phases are parallelized using at least three threads, it can be computed in about half the time (plus threading overhead) of the serial implementation. Of course, the speedup will not be that good on two threads, but it still outperforms serial code.

Implementation and Practical Issues

In final implementation, we shall not divide workload only to four parts, but we use approach similar to parallel reduce. We pick suitable grain size g and divide the arrays into ranges of that size. With more tasks the total work redundancy will tend to 2.0×, but the problem will scale optimally with more threads.

We should also ensure that redundant work is minimized if there are no available threads to execute parallel scan tasks. For example, when only one thread is available, it would be prudent not to do any redundant work so the solution will not be much slower than traditional serial code.

This may be achieved by appropriate order of task execution. Let us assume we have only one thread evaluating parallel scan. First executed task must compute final scan of the first range. If other threads were available, pre-scan of second range would be computed concurrently. However, when the only thread completes first range, the final scan of second range may be computed instead of pre-scan. This way the single-threaded evaluation of parallel scan will compute only final scans, so it will be almost as fast as serial solution[2].

[2] Unfortunately, some scheduler overhead is unavoidable, unless we implement

Tasks

Internal tasks used for parallel scan share the same paradigm with parallel reduce. Programmer must provide split, join and the main routine, and the parallel scan template takes care of the rest. The main routine differs from parallel reduce problem, because it may be invoked twice (for pre-scan and for final scan), so it has a flag that determines in which phase it has been executed.

```
Algorithm: ParallelScan (main routine)
Input: Array $X$, starting value $s$
Output: Sum $s$, filled array $Y$ (after final scan)
  1. For $i \leftarrow rangeStart$ to $rangeEnd$:
  2.     $s \leftarrow s \oplus X[i]$
  3.     If this is final scan:
  4.         $Y[i] \leftarrow s$
```

The starting value s is equal to $Id_{\oplus}$ for the pre-scan and to $Y[rangeStart - 1]$ for final scan. At the end of pre-scan, actual $\oplus$ sum is stored in s. The joining routine just performs $\oplus$ operation on computed sums of joined tasks. For more details about parallel scan see [Rei07].

3.1.5. Data Structures

Ordinary data structures are usually not thread-safe and operations performed by multiple tasks concurrently must be synchronized. Naive implementation which uses single mutex to lock whole data structure every time it is accessed creates usually a hotspot. If the algorithm works with the data often, tasks are forced into serial execution, thus the solution becomes slower than simple nonparallel code.

We shall use better techniques to improve scalability of our data structures:

- immutable data,
- implicit synchronization,
- and specialized solutions using atomic operations.

Immutable data structures are used in cases when the structure is constructed and no modification are required afterwards. Such structure is then read-only, so concurrent access does not pose any threat. This technique is used for DOM structure and its index.

Implicit synchronization is the most prefered way of avoiding race condition on mutable structures. In our case, it is most often used for processing node-sets (which are represented by a vector). The whole set is divided into disjoint subsets (ranges) and each one is treated by different task.

For some cases when implicit synchronization cannot be used, we design special data structures that use lightweight atomic operations to ensure data coherency avoiding heavy mutex locking. Both structures are used for building node-sets concurrently.

both serial and parallel version of the scan.

Concurrent Vector

Analogically to STL `vector` [STL], this data structure represents automatically growing array of items, in our case nodes (pointers to DOM tree). The *concurrent vector* has following features:

- Items may be accessed randomly by given index[3] in $\mathcal{O}(\log\log N)$ time (which is in our case as good as $\mathcal{O}(1)$). Access to these items is not synchronized.
- Multiple tasks may grow the array concurrently. When task grows the vector, it also obtains offset of the appended path, so it may be filled by items using implicit synchronization.
- Growing vector does not invalidate old indices and iterators (never relocates memory), so other tasks may access previously allocated ranges concurrently.

The concurrent vector is used for building node-sets when each node is inserted into constructed set exactly once, so we do not need to take care about duplicities.

Concurrent Hash-map

If the uniqueness of inserted nodes cannot be ensured (e.g. when multiple node-sets are merged together), we have to use more sophisticated structure – a hash table. *Concurrent hash-map* is similar to SGI's STL extension `hash_map` [STL], which is in fact a set of key-value pairs where each key is unique.

Concurrent version uses simple read-write locking mechanism over the items which is implemented by atomic variables. When task needs to read or write an item, it must acquire an accessor object for that item. There are two types of accessors (constant and regular) which correspond to the read and write types of locks.

For the purpose of node-set building, we use concurrent hash-map with left-values as keys and pointers to the DOM tree as values. We could also use pointers as keys and ignore the values, however, unique integers are much more suitable for hashing than memory addresses.

3.2. XML Representation and Indexing

Our approach to query processing expects that XML [XML] document is completely loaded into main memory with index to accelerate XPath axes. Following section will describe used data structures.

3.2.1. Document Object Model

When XML document is loaded into memory, it must be properly represented. WWW Consortium designed DOM specification [DOM] which describes object model for XML data. This model is in fact a tree where nodes represent XML elements, attributes, text data, etc. Since DOM is rather complex, we use similar tree structure which is more simple but not fully DOM compliant.

[3] Index of the fist item is zero.

Nodes

We recognize only the following types of nodes. Other types described in DOM (like namespaces, processing instructions, etc.) are omitted since they are not interesting from our point of view.

- *Text node* is simple node which contains one plain text string. It has no name nor children.
- *Attribute* represents name-value pair associated with an element. Attributes also cannot have children.
- *Document root* is the top most node in the DOM tree. It does not have any special attributes nor value and it has exactly one child element.
- *Element* is the most important node in DOM. It has name, list of attributes and children nodes. References to attribute nodes are stored in an array ordered by their position in the document. In order to retrieve attributes by their names in constant time, simple hash table index which maps names to positions in the list is constructed for each element[4]. Children of the element are stored in simple array, ordered by their position in the document. Each child may be either a text node or a nested element.

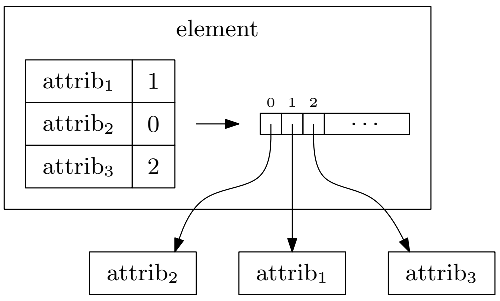

Figure 3-4: Representation of references to attributes in the element node

Name Index

Element and attribute nodes have names. Name is a string value of arbitrary length, therefore comparing two strings may take up to $\mathcal{O}(l)$ time where l is length of the shorter string. Since node names are also used in XPath queries, these comparisons are quite frequent and should be optimized.

In order to do that, we create a translation table for each document. This table assigns each name a numeric identifier (starting from 1) and provides means to translate names to identifiers and vice versa. Identifier-to-name translation is simple, considering the names are stored in an array and every identifier is in fact a position of corresponding name in this array. Reverse index is realized by balanced binary search trees (implemented in STL `map` [STL]).

It would be better to use hash table or trie for name-to-identifier translation. However, when name is translated, its identifier is used ever since, thus the translation is performed only once when query is parsed. Furthermore, elements of real

[4] Let us just recall that element may not have two attributes with the same name.

XML documents only have few distinct node names so it does not really matter whether translation takes $\Theta(\log k)$ (where k is number of names) or $\Theta(1)$. The binary tree was chosen as we have encountered several problems with using `hash_map` template with strings on some platforms.

Henceforth, when we discuss node names or name comparison, we in fact mean their numeric identifiers. We also expect that these identifiers may be copied and compared in constant time.

3.2.2. Node-sets

Node-sets are represented by dynamic arrays (a STL `vector` [STL], to be precise) of pointers to the DOM structure. Each pointer must be unique in the set and they are ordered by left-values of nodes to which they point. When nodes are mentioned in following text, we speak about pointers to immutable DOM tree.

Node-set does not provide any means of synchronization since it is often used as immutable object. When modified, it expects that only one task works with it at a time. Multiple threads may still work with the items if the size of the array remains constant. Concurrent access to items is synchronized implicitly.

This way the node-set does not carry the burden of locking. On the other hand we must define other means of building node-set by simultaneously executed tasks.

Building a Node-set

There are three issues that make parallelisation of node-set building a complex task:

- resizing the set,
- uniqueness of the items,
- and proper ordering (by left-values).

When nodes are written to the set concurrently, the array must not be resized. Modification of set size can be performed only by one thread at a time, and synchronization implied to this problem would affect the performance seriously. Thus, if we do not know the size of constructed set in advance, we have to use special data structures described in 3.1.5. Otherwise, we just allocate sufficient space for the set and use implicit synchronization for written nodes.

If the uniqueness of nodes is not ensured by the algorithm natively, we use a concurrent hash-map to build node-set. First, all nodes are added to the map by multiple tasks and the map takes care of duplicities. Then the node-set is resized according to number of items in the map and all nodes are copied to the set using parallel scan (3.1.4) with following main routine:

```
Input: rangeStart and rangeEnd iterators to the hash-map, offset
Output: node-set $S$, updated offset
1. For iterator $\leftarrow$ rangeStart to rangeEnd:
2.     If this is final scan:
3.         $S[offset] \leftarrow iterator.value$
4.     $offset \leftarrow offset + 1$
```

Concurrent hash-map does not provide random access to items, but it offers splittable iterator for parallel reduce and scan. Therefore, parallel scan is used to compute offsets where the nodes will be placed in pre-scan and in the final phase, nodes are really copied.

When the uniqueness of nodes is guaranteed by other means, but the size of created node-set is unknown in advance, we may use concurrent vector. The situation is analogical to concurrent hash-map. Nodes are assembled in the concurrent vector and then copied into a node-set. The difference is that vector supports random access, thus the nodes can be copied simply by parallel for.

Finally, we have to address the problem of ordering. When nodes are copied from concurrent hash-map or vector, they are not arranged in any particular order. We have to sort them, to ensure node-set coherency. Parallel implementation presented in TBB [Rei07] as parallel sort is used.

Removing Nodes and Compacting

Removing an item from ordered array has unpleasant time complexity $\mathcal{O}(N)$ and it is hardly parallelizable. Node-sets are often filtered (e.g. by predicates), so we have to optimize this operation. We shall proceed in two phases. First, all nodes to be removed are marked (their pointers are set to `NULL`). Marking does not shift nodes nor change size of the set, therefore it is perfectly parallelizable. When first phase is completed, the set is compacted and every `NULL` pointer removed.

Since we would like to parallelize the compacting routine as well, we rather copy remaining nodes to new array using parallel scan. First pass computes offsets to compacted array where each range is copied and the second pass finally copies the nodes.

```
Input: node-set $X$ with $NULL$ values, $offset$
Output: compacted node-set $Y$, updated $offset$
  1. For $i \leftarrow rangeStart$ to $rangeEnd$:
  2.     If $X[i] \neq NULL$:
  3.         If this is final scan:
  4.             $Y[offset] \leftarrow X[i]$
  5.         $offset \leftarrow offset + 1$
```

3.2.3. Index

The DOM structure may be indexed many ways, as we have shown in chapter 2.3. Some of presented indices are quite complex and more suitable for database management systems. We have chosen only those which are simple enough to be implemented in standalone programs and which have significant impact on parallel XPath evaluation.

First of all, we use LRD index (presented in 2.3.1) which has many applications. This index is embedded directly in DOM structure and is built on the fly when document is loaded. Each DOM node (including attributes and text nodes) has its left, right and depth value. Left-values are assigned incrementally starting from root (document) node which has left-value zero.

For accelerating location path axes, we also use the element index (described in 2.3.2) for the entire document and for nodes with more than a few children[5].

[5] The threshold was set to 16 children after some experiments.

Document index is implemented by hash table (STL `hash_map` [STL]) with name identifiers as keys and and node-sets as values. Children index also uses a hash table which maps XML nodes to their indices while every children index has the same representation as document index.

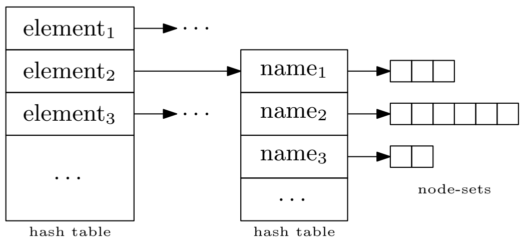

Figure 3-5: Implementation of children index

Both document and children indices are constructed right after document is loaded into DOM structure. Document index is created by preorder traversal of the DOM tree, appending each element to proper node-set. The same principle is used when children index is built, except children of each element are stored in different indices.

3.3. Location Paths

The location paths are the most intriguing component of XPath language. Different algorithms for their evaluation were discussed in 2.2 and now we describe the concept which was used in our implementation.

3.3.1. Evaluation Concept

We have presented three general techniques (in 2.2) that may be used in XPath evaluation:

- the recursive approach,
- iterative methods
- and the pipeline concept.

The pipeline concept might be promising and it is worth further research. However, we have chosen more conservative way based on recursive approach and iterative methods.

Recursive approach is quite simple for implementation and requires only linear additional space. Unfortunately, it leads to exponential time complexity for some queries. Iterative methods solve the problem of exponential phantom menace, but they also require up to $\mathcal{O}(N^2)$ additional space, thus cannot be practically used for large documents.

Since we consider processing large documents (over 10^5 nodes) a priority, we require an algorithm that manages to evaluate queries in $\mathcal{O}(k \cdot N)$ space where k is size of the query. Therefore, our implementation is based on recursive approach with two improvements:

- *Minimal context rules* are applied for each subquery. When a subquery does not require entire context information for its evaluation, we may optimize it. In some cases, we may even use iterative methods while linear space complexity is preserved. This way we create faster solution which can be applied even for large documents.
- If the amount of available memory is limited, exponential time complexity can not be prevented for some queries. However, we might reduce it significantly using *caches for subqueries*.

Recursive Algorithm

Suggested optimizations will be described in following sections. First, we formalize the recursive algorithm without them. Each step of location path is in fact a projection $C \to S$ where both C, S are node-sets ($\subseteq DOM$)). Location path is evaluated sequentially. First location step has only the context node of the entire path in context set C. Node-set yielded by first step is used as context set for the second one, second step yields context set for third one and so on. Last step returns result of the whole path.

Step s consists of axis, filter and list of predicates $\mathcal{P}_s = \{p_1, p_2, \ldots, p_{\pi_s}\}$, where number of predicates for node s is denoted π_s. Axis and filter are evaluated for each node n from C (as described in 3.4) and yield set of node-sets S_n. Each node-set S_n is filtered by predicates $\mathcal{P}$ and then united in result set S.

```
Algorithm: EvaluateStep
Input: Node-set $C$, axis, filter, list of predicates $\mathcal{P}$
Output: Node-set $S$
  1. Prepare empty node-set $S$
  2. Parallel for each node $n$ in $C$:
  3.       Use axis and filter to get node-set $S_n$ from $n$
  4.       Filter $S_n$ by predicates $\mathcal{P}$
  5.       $S \leftarrow S \cup S_n$
  6. Return $S$
```

In our implementation, multiple tasks from parallel for will unite nodes with set S, therefore access to S must be synchronized. If we use concurrent hash-map to build S, both synchronization and uniqueness of nodes in the set will be ensured.

Predicates

Predicates used for filtering nodes are arbitrary subqueries and their results are always converted to Boolean. If predicate yields a number, it is compared with context position ($predicate = position()$) and Boolean value yielded by this comparison is used. Otherwise, the predicate expression is just enclosed by $boolean()$ function which ensures the correct type of the result.

When a node-set is filtered by a predicate, the predicate is executed recursively for every node in the set using the node, its position and size of the set as context values. If the predicate yields *true*, tested node is included into the result. First predicate from $\mathcal{P}$ uses initial node-set S_n^0 as its input and returns S_n^1. Then S_n^1 is used as input for p_2 which yields S_n^2 and so on. The last predicate returns the final result of the entire filtering process.

Algorithm: FilterPredicates
Input: Initial node-set S^0, list of predicates $\mathcal{P}$ (of size π)
Output: Filtered node-set S^π

```
1. For $i \leftarrow 1$ to $\pi$:
2.     Prepare empty set $S^i$
3.     Parallel for $j \leftarrow 0$ to $size(S^{(i-1)}) - 1$:
4.         $r \leftarrow p_i(S^{(i-1)}[j], j+1, size(S^{(i-1)}))$   (evaluate $p_i$)
5.         If $r$ is true:
6.             $S^i \leftarrow S^i \cup S^{(i-1)}[j]$   (add tested node into next set)
7. Return $S^\pi$
```

While nodes are inserted into each set S^i concurrently, access to S^i must be synchronized. In the implementation we use implicit synchronization. Before S^i is filled, space for each node from $S^{(i-1)}$ is reserved and marked as empty. When node $S^{(i-1)}[j]$ is inserted to S^i, it is stored at position j, so it will not conflict with other nodes. After filling phase is completed, S^i is compacted (as described in 3.2.2).

3.3.2. Vector Operations

When a subquery is evaluated, it requires a context information which consists of context node, position and size. Many subqueries (location steps, predicates, ...) are evaluated multiple times for different contexts. In fact, when recursive approach is used, these subqueries are often evaluated repeatedly for the same contexts, but we shall deal with this problem later, in 3.3.5. In following section, we will focus solely on optimizing some queries using techniques from iterative algorithms.

Let us have an XPath expression E, which is evaluated for two contexts with the same context node (n, p_1, s_2) and (n, p_2, s_2). What if E does not require context position nor size for evaluation? Both contexts will lead to the same result as they have the same context node. Unfortunately, when subqueries are evaluated using one context at a time, this redundant evaluations cannot be easily prevented.

Top-down algorithm described in [GKP05] suggests that subexpressions should be evaluated not for one context at a time, but for whole vector of contexts. Parent expression prepares all contexts and then gives them to a subquery at once, so the subquery may better plan its work and avoid repetitive evaluation. The problem is that vector of contexts may take up to $\mathcal{O}(N^3)$ considering that there are N available nodes in DOM structure and context position and size are numbers so that $1 \leq p \leq s \leq N$. Furthermore, each context value may yield another node-set with size $\mathcal{O}(N)$, so the total space complexity is $\mathcal{O}(N^4)$.

The vector operations described by top-down algorithm could be still used in some special cases though. If evaluated expression utilizes only a part of the context information, for instance only a context node, a simple node-set with size of $\mathcal{O}(N)$ can be used instead of vector of (n, p, s) tuples. Still, each context node may spawn another set with $\mathcal{O}(N)$ nodes, thus total space requirements are $\mathcal{O}(N^2)$ which is still far from linear complexity.

In location step evaluation (as described in 3.3.1) we need not remember, which context node yielded which node-set considering all sets are united in the end. Location step requires set of context nodes and yields only one node-set, thus it requires only linear space for both input and output. Henceforth, when we refer to *vector operations*, we always mean a projection $S_1 \rightarrow S_2$ where both S_1, S_2 are node-sets ($\subseteq DOM$) and algorithms implementing a vector operation will be denoted with $_{\langle\rangle}$ suffix.

Location Steps as Vector Operations

Location steps are in fact vector operations as we have just shown. Unfortunately, the XPath specification defines, we should evaluate initial part of the step (axis and filter) independently for each context node, so each initial node-set is filtered by predicates separately. This seems quite inefficient at first considering one node is filtered multiple times by the predicates. We have to bear in mind that each node may have different position or the initial node-set could have different size and these context information might be vital for evaluation.

On the other hand, when none of the predicates requires context position or size, only the context nodes are important. We can evaluate initial part of location step as vector operation which unites all node-sets before they are processed by predicates and the united set is filtered instead.

This approach allows us to optimize axis evaluation as we will demonstrate in 3.4.3. Furthermore, number of nodes filtered by predicates is reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. Even though we have optimized only location steps with no predicates that would require context position or size, it is still significant improvement. Many location steps do not have predicates at all, or use a nested location step as predicate expression (which requires only context node for its evaluation). Now we shall describe how to determine which parts of context information are required by a predicate.

3.3.3. Minimal Context Rules

In previous section, we demanded to know which parts of context are required by an expression for evaluation. In order to do so, we establish list of *context flags* to label each subexpression in query tree. We use three types of flags, each corresponding to one part of the context:

- flag n for context node,
- p for position,
- and s for size.

Every subexpression E is tagged with set of flags denoted $cf(E) \subseteq \{n, p, s\}$. When a flag is present in the set $cf(E)$, corresponding part of context is required for evaluation of E. Formally, we define cf as following.

expression E	context flags $cf(E)$
arithmetic operators $(+, -, *, \text{div}, \text{mod})$	$cf(operand_1) \cup cf(operand_2)$
comparisons $(<, \leq, >, \geq, =, !=)$	$cf(operand_1) \cup cf(operand_2)$
logical operators (and, or)	$cf(operand_1) \cup cf(operand_2)$

expression E	context flags $cf(E)$
absolute location path	$\{\}$
relative location path	$\{n\}$
location step	$\{n\}$
[*predicate*]	$cf(predicate)$
list of predicates $\mathcal{P}$	$\bigcup cf(p), p \in \mathcal{P}$
union ($\vert$)	$cf(operand_1) \cup cf(operand_2)$
position()	$\{p\}$
last()	$\{s\}$
functions without arguments	$\{\}$
other functions	$\bigcup cf(a), a \in arguments$
numeric and string literals	$\{\}$

Table 3-1: Context flags for different types of expressions

Context Classes

Even though there are eight (2^3) types of expressions depending on which of n, p, s flags they have, we will streamline the situation by recognizing only three *context classes*:

- *Context-free* class represents expressions with empty *cf* set. These expressions are either constant (like literals) or contain absolute location paths.
- Expressions in *context-node* class requires only context node, thus their $cf(E) = \{n\}$. Typical representants are relative location paths and location steps.
- Finally, in *full-context* class we place every remaining expression.

We call the context-free class the *weakest*, because empty context flag set is always a subset of any other class. Analogically, the full-context class will be the *strongest*. Expression with non-empty $cf \subseteq \{p, s\}$ is also considered to be *stronger* than context-node expression for the context position and size is always generated together with nodes, hence node flag is in fact implicitly included.

Vector Operations Again

Now we can redefine our rule for vector optimization of location steps using context classes. When a location step has list of predicates $\mathcal{P}$ that do not belong to full-context class, we can use vector optimization for axis evaluation and to reduce number of nodes filtered by predicates. Otherwise, if at least one predicate is full-context type, the original *EvaluateStep* algorithm must be used.

3.3.4. Evaluation of Context-free Subqueries

The most problematic subqueries are non-literals from the context-free class. Expressions which do not need a context for evaluation yield always the same result, thus we would like to avoid their repetitive evaluation. We will illustrate the problem on following example.

Let us have simple document that represents information about system users and groups. A group is always owned by one user and we would like to list all groups that are owned by very smart people:

```
/group[@ownedBy = /user[iq >= 150]/@id]
```

When parsed, the query will form following syntactic tree.

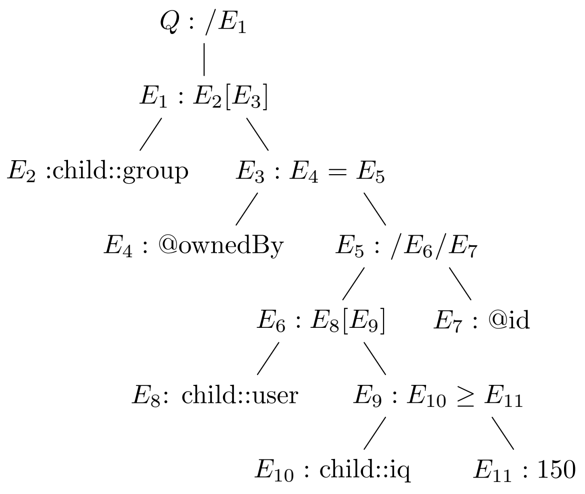

Figure 3-6: Parse tree of sample query

The predicate which filters groups is quite complex and it must be evaluated for every potential group in the document. Obviously, the right side of comparison (E_5) always yields the same result as the absolute location path is context-free. Without any optimizations, the set of smart users will be retrieved for each group even though it could have been done only once.

We can insert a cache mechanism to avoid repetitive evaluation of context-free subexpressions. The expression is evaluated when its value is required for the first time and the value is cached. This approach works satisfactorily in serial enviroment. However, if the predicate is evaluated simultaneously for multiple contexts, access to cache must be synchronized.

Synchronization unavoidably causes one of following problems. Either one task holds a lock for the cache while it computes required value and other tasks must wait, or multiple tasks evaluate the context-free expression needlessly. Both situations reduces parallelism significantly for multiple threads are blocked or performing redundant work.

Eager Evaluation

Key to better performance is to evaluate context-free expressions before parallel execution is initiated. This approach may evaluate some expressions even if their results are not required in the end. On the other hand, situations when a subexpression is not evaluated at all are assumed to be very rare and the eager approach has positive impact on parallelism.

Before a predicate is evaluated, its parse subtree is traversed. Each context-free subexpression is evaluated and replaced by literal value:

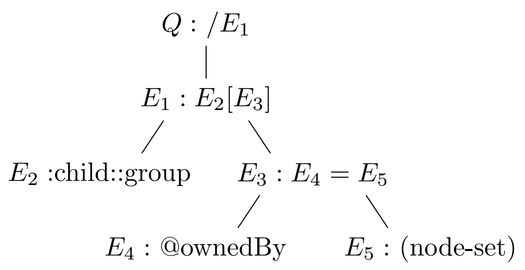

Figure 3-7: Modified parse tree (E_5 is replaced by a literal)

The right operand of comparison from previous example has been resolved and replaced by constant node-set. Then the predicate may be evaluated concurrently for each group without processing the absolute location path multiple times.

3.3.5. Caches

We have dealt with context-free subexpressions and introduced vector operations for location steps without full-context predicates, but we still have not completely solved the problem of redundant evaluation. For better demonstration, let us have a location path which retrieves '`b`' elements:

```
//a/following::b[(count(c) >= 3) and (last() < position()*2)]
```

The predicate contains *position*() and *last*() functions as well as location path, thus it has *cf* value $\{n, p, s\}$ and falls into full-context class. Obviously, we cannot use vector optimization to evaluate second location step, so we have to fall back to the orignal algorithm.

The first problem is that the predicate may happen to be evaluated multiple times for the same context. Two different '`a`' nodes might have the same set of following '`b`' elements, hence these sets are filtered twice needlessly.

Second problem is that right operand of the logical 'and' operation is a full-context expression while the left operand comes only into context-node class. When the predicate is evaluated for multiple different contexts with the same context node, left operand will be also evaluated multiple times even though it yields always the same result.

Implementation

We solve both problems by caches. A *cache* is simple unary expression which does not modify semantics, but eliminates repetitive evaluation of its operand. We say that the operand subexpression is *covered* by the cache.

The cache is inserted right above cached expression in the query tree and it has the same result type as its operand. Therefore, we have four different caches depending on their result types (boolean, number, string or node-set). Cache has also the same context class as its operand. Since we do not cache context-free expressions (as the eager context-free evaluation is used as described in 3.3.4), there are two variants of caches – a full-context and a context-node class.

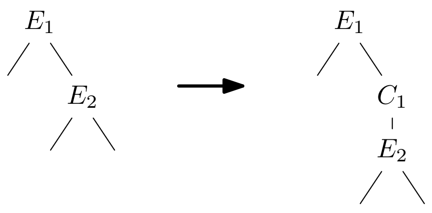

Figure 3-8: Covering expression E_2 by a cache

Inside the cache, values are stored in concurrent hash-map. Context-node cache uses left-value of a context node as hash key. The full-context cache hashes tuples $(\mathit{left}(n), p, s)$. When requested value is not in cache, it is evaluated by an operand and inserted into the map. Corresponding key is locked meanwhile and if accessed by another task, this task is blocked until the value is computed. Unlike in context-free class, we expect that these collisions are rare, hence the tasks will not be blocked often.

Rules for Inserting and Disposing Caches

For the sake of clarity, we shall denote predicates and their subqueries to be *nested* expressions while other subqueries are be *top-level* expressions. Only nested expressions are affected by problems described earlier.

Top-level expressions (except for location steps) are always evaluated just once. Nested expressions are usually evaluated multiple times for different contexts, so we will focus solely on them. When a top-level location step is evaluated, all its predicates (and their subqueries) must be covered by caches. We initialize the cache objects just before and dispose them right after the evaluation. Following rules are applied when caches are initialized.

- Literal values must not be covered by a cache directly.
- When an expression is covered by a cache, all its subexpressions from the same or stronger class are also considered to be covered. This rule is transitive and does not apply for predicates (a predicate cannot be covered transitively). All expressions from weaker classes must be covered by another cache of their own.
- If location step has a full-context predicate, all predicates of that step must be covered by caches. Otherwise, no predicate of that step needs to be covered, but these rules are applied recursively on nested predicates.

Furthermore, the predicates are always covered by Boolean caches considering they are implicitly enclosed by *boolean*() function.

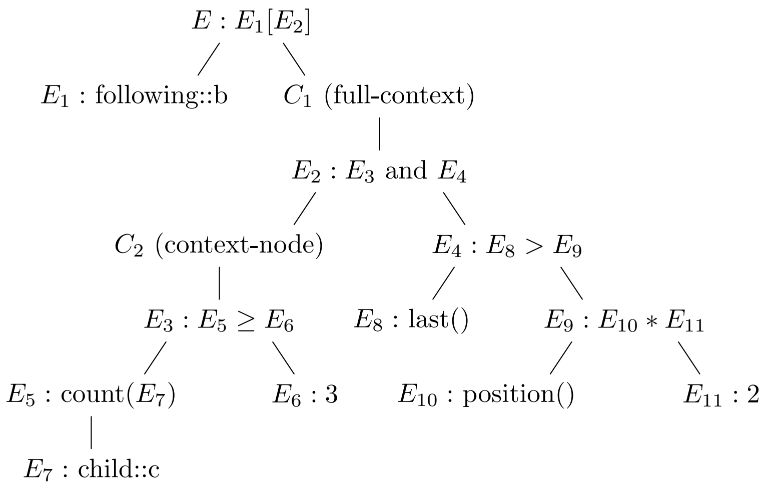

Figure 3-9: Application of cache rules on a sample query

In our sample, E_2 is covered by a cache since it is full-context predicate. Also E_3 is covered for it comes into weaker context class than its parent, thus it is not covered transitively.

Space Requirements

Obviously, the full-context cache with node-set result type is the largest one. If completely filled, it takes up to $\mathcal{O}(N^4)$ space, hence it has the same asymptotic complexity as iterative approach. In fact, the cache is little worse in comparison with top-down algorithm since hash-map has greater overhead than simple vector.

On the other hand, caches are much more flexible. Small documents may use up to $\mathcal{O}(N^4)$ space and avoid redundant evaluations completely. If there is not enough memory, we may put restrictions on cache size and make it forget records. This way we cannot avoid computing some intermediate values repetitively, but the cache decreases number of such events significantly. Therefore, we can evaluate also very large documents – it just takes more time. This adaptative space-time tradeoff cannot be easily achieved with iterative methods considering they always require at least $\mathcal{O}(N^2)$ memory.

3.3.6. Predicates

We have already introduced the general algorithm for filtering node-sets by predicates in 3.3.1. Now we shall revise it once again with the optimizations described in previous sections.

Before a location step is evaluated, the context-free subexpressions in predicates are resolved (and replaced by literals) and constant predicates are handled. This preprocessing is done only for top-level location steps.

Algorithm: FilterPredicatesPreprocessing
Input: list of predicates $\mathcal{P}$ (of size π)
Output: modified $\mathcal{P}$

```
 1. $empty \leftarrow false$
 2. Parallel for $i \leftarrow 1$ to $\pi$:
 3.     Evaluate context-free subexpressions of predicate $p_i$
 4.     If $p_i$ is boolean literal *true*:
 5.         Remove $p_i$ from $\mathcal{P}$
 6.     Else If $p_i$ is boolean literal *false*:
 7.         $empty \leftarrow true$
 8. If *empty* is *true*:   *(at least one literal is constantly false)*
 9.     Mark the location path as empty node-set literal and terminate
10. Else: Proceed with predicates filtering
```

If the preprocessing finds at least one predicate that is constantly *false*, the whole location path is marked as empty since it always returns node-set with no nodes. Otherwise, the preprocessing terminates successfully and we proceed with location step evaluation.

If there is no full-context predicate in $\mathcal{P}$, vector operations may be used to optimize evaluation (as suggested in 3.3.2). The initial part of the step is evaluated and all node-sets merged before they are filtered by predicates. Then, we may use optimized algorithm for filtering which does not create intermediate node-sets, but rather computes final result from the initial set.

```
Algorithm: FilterPredicatesIntern$_{\langle\rangle}$
Input: Initial node-set $S^0$, list of predicates $\mathcal{P}$ (of size $\pi$)
Output: Filtered node-set $S^1$
 1. Parallel for $i \leftarrow 0$ to $size(S^0)$:
 2.     $j \leftarrow 1, res \leftarrow true$
 3.     While $j \leq \pi$ and *res* is *true*:
 4.         $res \leftarrow p_j(S^0[i])$   *(evaluate $p_j$ using only context node)*
 5.         $j \leftarrow j + 1$
 6.     If *res* is *true*:
 7.         Add $S^0[i]$ to $S^1$
 8. Return $S^1$
```

When $\mathcal{P}$ comes into full-context class, we have to filter each initial node-set separately and merge them after filtering. If the location step is top-level, all predicates and their subexpressions must be covered by caches as described in 3.3.5. Caches are disposed after the location step is resolved since it will not be evaluated again.

Before node-sets are filtered, the predicates are grouped into *segments*. We denote σ_i the i-th segment. A segment is continuous subrange of $\mathcal{P}$ constructed by these two rules:

- A full-context predicate is always the first predicate in the segment. Therefore, two following full-context predicates ought to be separated into two segments.
- A full-context predicate followed by a predicate from weaker class or two following context-node predicates must be in the same segment.

This way we have ensured that only the first node in segment requires context position and size. Therefore, we need to create intermediate node-sets only for each segment, not for each predicate. Predicates in one segment are then processed using the same technique as *FilterPredicatesIntern*$_{\langle\rangle}$ routine.

```
Algorithm: FilterPredicatesIntern
Input: Initial node-set $S^0$, segments $\sigma_i$ of predicates $\mathcal{P}$ (of size $\varsigma$)
Output: Filtered node-set $S^\varsigma$
 1. For $i \leftarrow 1$ to $\varsigma$:
 2.     Prepare empty set $S^i$
 3.     Parallel for $j \leftarrow 0$ to $size(S^{(i-1)}) - 1$:
 4.         $res \leftarrow true$
 5.         For each predicate $p$ from $\sigma_i$ and while $res$ is $true$:
 6.             $res \leftarrow p(S^{(i-1)}[j], j+1, size(S^{(i-1)}))$
 7.         If $res$ is $true$:
 8.             $S^i \leftarrow S^i \cup S^{(i-1)}[j]$   (add tested node into next set)
 9. Return $S^\varsigma$
```

Parallel Evaluation

Finally, we have to consider a dilemma of redundant computations suggested in chapter 2.4. When multiple predicates are evaluated for single node (in vectorial version or in one segment of classical filtering function), the results can be computed concurrently. On the other hand, if any of the predicates yields *false*, following predicates should not be evaluated since the tested node cannot be added into the result set anyway.

All the operations are idempotent and have no side-effects, thus parallel evaluation will not change the semantics. However, we cannot tell whether this approach is better than serial evaluation unless we test it. The standard implementation will be modified as follows.

```
 1. $res \leftarrow true$
 2. Parallel for each predicate $p$ from the segment:
 3.     If $res$ is $false$: Terminate loop   (avoid some redundant work)
 4.     If $p(context)$ yields $false$:
 5.         $res \leftarrow false$
 6. ...
```

Access to *res* must be synchronized. However, the variable is set to *true* before the parallel for is started and each task may only change it to *false* (never back to *true*). Assuming that result can be only *true* or *false*, no race condition may happen when the value is written, but we still have to ensure, that every task reads actual value of *res* from memory.

We will execute and compare both standard and parallel version of predicate evaluation in chapter 5.5.

3.4. XPath Axes

The general idea of the XPath axes implementation using element index was already presented in 2.3.2. We shall now formalize it and design parallel algorithms. Henceforth, we denote *Index* the element index and *ChildIndex* the children index. An *Index*[*name*] refers to a node-set containing all elements with defined *name* and *ChildIndex*[*n*][*name*] is node-set containing all children of node n with given *name*.

Each of the following algorithms requires context node (denoted *ctxNode*) and filter *name* as input, and it yields node-set S as output. It is assumed that every algorithm has also read-only access to the document and children index.

The *name* could be either regular node name or a wildcard '$*$' which matches any name. Name is represented by an identifier described in 3.2.1. The wildcard has a special identifier that is never used for any node. Since we do not want to complicate retrieving algorithms, we also include the wildcard into the document index. The *Index*[$*$] refers to node-set which contains all elements of the document. We also define *ChildIndex*[n][$*$] as list of all element-children of node n.

Searching the Node-sets

Before we start with describing axes implementation, we shall present a subroutine which is used in following algorithms quite often. It is standard binary search which find first element in the set with given or greater left-value.

Algorithm: BinSearch
Input: Node-set S, left-value l
Output: Index i of first node in S with its left-value greater or equal to l

```
1. $i \leftarrow 0$, $j \leftarrow size(S)$   (start with $\langle 0, size(S))$ range)
2. While $i < j$:
3.     $m \leftarrow (i+j)/2$   ($m$ is the middle of $\langle i, j)$)
4.     If $left(S[m]) \geq l$:
5.         $j \leftarrow m$
6.     Else:
7.         $i \leftarrow m+1$
8. Return $i$
```

The binary search algorithm works in $\Theta(\log size(S))$ time.

3.4.1. Algorithms for Major Axes

Descendants

Descendants and successors of the context node are stored in one continuous subrange of the index node-set. Descendants have their left-values greater than *left*(*ctxNode*) and lesser or equal to *right*(*ctxNode*) so we just find border nodes by binary search algorithm and copy everything between them.

Algorithm: GetDescendants

```
1. If $Index[name]$ does not exist: Return empty $S$ and terminate
2. $l \leftarrow BinSearch(Index[name], left(ctxNode)+1)$
3. $r \leftarrow BinSearch(Index[name], right(ctxNode)+1)$
4. Prepare node-set $S$ with reserved size $(r-l)$
```

```
5. Parallel for $i \leftarrow 0$ to $(r - l - 1)$:
6.     $S[i] \leftarrow Index[name][i + l]$   (copy nodes from $\langle l, r)$ range)
7. Return $S$
```

Algorithm uses binary search routine and copies the results, thus it works in $\Theta(\log N_{name} + r)$ time where N_{name} is number of nodes with given name in the document and r is size of the result.

There is also a simple variation for this axis called *descendant-or-self* which includes the self node into the result if it matches given *name*. We can modify presented algorithm just by shifting index for the first *BinSearch* call (using $left(ctxNode)$ instead of $left(ctxNode) + 1$).

Following Nodes

Following nodes are retrieved the same way as descendants, except they have their left-values in $(right(ctxNode), \infty)$ interval.

Algorithm: GetFollowing

```
1. If $Index[name]$ does not exist: Return empty $S$ and terminate
2. $l \leftarrow BinSearch(Index[name], right(ctxNode) + 1)$
3. $r \leftarrow size(Index[name])$
4. Prepare node-set $S$ with reserved size $(r - l)$
5. Parallel for $i \leftarrow 0$ to $(r - l - 1)$:
6.     $S[i] \leftarrow Index[name][i + l]$
7. Return $S$
```

Algorithm for following nodes is almost identical to *GetDescendants*, thus it has the same time complexity.

Preceding Nodes

Preceding axis is slightly more complicated than already presented *following* and *descendant* axes. The problem is that predecessors of context node are not in one compact range when nodes are ordered by their left-values in the set. However, we know that nodes with their left-value lesser than $left(ctxNode)$ are in fact union of all predecessors and ancestors. If we filter out the ancestors, only preceding nodes remain.

Algorithm: GetPreceding

```
1. If $Index[name]$ does not exist: Return empty $S$ and terminate
2. $r \leftarrow BinSearch(Index[name], left(ctxNode))$
3. If $r = 0$: Yield an empty node-set and terminate
4. Prepare node-set $S$ with reserved size $r$
5. Invoke parallel scan on $\langle 0, r - 1\rangle$ interval of $Index[name]$
6. Update actual size of $S$ to *offset* value returned by parallel scan
7. Return $S$
```

We cannot use simple parallel for, because we do not know positions of nodes in final set in advance. Therefore, we use parallel scan which first computes the offsets and then copies the nodes. The scan has following main routine:

```
Algorithm: GetPrecedingScan
Input: Index node-set $I$, preallocated node-set $S$, $ctxNode$, $offset$
Output: Filled node-set $S$, updated $offset$
1. For $i \leftarrow rangeStart$ to $rangeEnd$:
2.     If $right(I[i]) \leq left(ctxNode)$:    (not ancestor of context node)
3.         If this is final scan:
4.             $S[offset] \leftarrow I[i]$
5.         $offset \leftarrow offset + 1$
```

If we expect that the XML document has depth $\mathcal{O}(\log N)$ (where N is the number of nodes), the maximal number of ancestors for one node is also $\mathcal{O}(\log N)$. Hence, the time complexity would be $\mathcal{O}(\log N_{name} + r + \log N)$ which is in fact $\mathcal{O}(\log N + r)$.

Ancestors

As suggested earlier, we expect that XML documents are shallow. Therefore, we do not accelerate *ancestor* axis by any index, but use simple DOM traversing instead.

```
Algorithm: GetAncestors
1. Node pointer $p \leftarrow ctxNode$
2. Prepare empty node-set $S$
3. While $p$ has a parent:    (p is not document root)
4.     $p \leftarrow parent(p)$
5.     If name of $p$ matches $name$:
6.         Append $p$ to $S$
7. Reverse $S$    (ensure proper ordering of the set)
8. Return $S$
```

If the expected depth of the document is $\mathcal{O}(\log N)$, the algorithm works with $\mathcal{O}(\log N)$ time complexity.

Like for the descendants, *ancestor* axis has a variant called *ancestor-or-self* which also includes the self node into the result (if it matches given *name*). This axis may be implemented by minor modification of *GetAncestors* algorithm. We just shift the $p \leftarrow parent(p)$ line (4th step) at the end of the 'while' loop. The modification will work since root element has valid parent node (document root), thus it will be also processed.

3.4.2. Children and Siblings

Retrieving children and sibling nodes depends on whether the contex node (or its parent in case of siblings) has children index. Nodes with too few children do not have any index, so the results must be picked directly form the DOM tree.

```
Algorithm: GetChildren
1. If index $ChildIndex[ctxNode]$ exists:
2.     If $ChildIndex[ctxNode][name]$ does not exist:
3.         Return empty $S$ and terminate
4.     Copy entire $ChildIndex[ctxNode][name]$ to $S$ using parallel for
```

```
 5. Else:
 6.     Prepare empty set $S$
 7.     For each child $c$ of *ctxNode*:    *(we process children sequentially)*
 8.         If name of $c$ matches *name*:
 9.             Append $c$ to $S$
10. Return $S$
```

Preceding and following siblings are acquired in similar way as children. The only difference is that parent of the contex node must be reached first. Subset of its children with their left-values lesser (or greater) than left-value of context node represents preceding (or following) siblings. Unlike preceding nodes, preceding siblings may not contain any ancestors, thus we need not filter them out.

3.4.3. Vector Operations for Major Axes

We have devised an optimization of location steps based on vectorial approach (in 3.3.2). This optimization requires that the initial part of the location step is evaluated as vector operation.

Each vector operation described below requires context node-set C as input (instead of single context node) and, like traditional axis algorithms, it yields node-set S as an output.

It is obvious we can implement vector operations simply by invoking their scalar versions and merge the results. If the concurrent hash-map is used for merging, each intermediate node-set can be computed and added by separate task and these tasks can be executed simultaneously.

Algorithm: Operation$_{\langle\rangle}$

```
1. Prepare concurrent hash-map $H$
2. Parallel for each node $n$ from $C$:
3.     $S_n \leftarrow Operation(n, name)$
4.     Add node-set $S_n$ into $H$
5. Export nodes from $H$ to $S$ and sort $S$
6. Return $S$
```

However, this algorithm does not benefit from knowing all context nodes in advance. Some operations may be optimized to reduce number of computed intermediate values which should improve the performance. Henceforth, we will refer to this approach as to *naive algorithm*.

Descendants

Naive algorithm is not very efficient for the *descendant* axis. Let us say that two nodes n_1, n_2 from initial set are in ancestor-descendant relation, more precisely n_2 is descendant of n_1. Then a node-set yielded by $GetDescendants(n_2)$ is in fact a subset of $GetDescendants(n_1)$. Knowing this, we need not to compute descendants of n_2 nor to add them to the results again.

Context node-set is, as any other set, ordered by left-values of nodes, therefore all potential descendants of node n must be after n in the set. A simple rule can be used to avoid adding nodes into the result needlessly: When descendants of node n are added into the result, we skip all nodes from initial set with their left-value lesser or equal to $right(n)$. This way we ensure node uniqueness in the result, therefore we can use concurrent vector instead of a hash-map.

```
Algorithm: GetDescendants$_{\langle\rangle}$
  1. Prepare concurrent vector $V$
  2. Invoke parallel scan on context node-set, which fills nodes into $V$
  3. Export nodes from $V$ to $S$ and sort $S$
  4. Return $S$
```

We use parallel scan for processing the context node-set. First phase calculates initial right-value for each segment. All nodes within bounds of this value are skipped. The second phase adds descendants of remaining nodes into concurrent vector.

```
Input: Context node-set $C$, max. right-value $maxRight$ from prev. range
Output: Added nodes in concurrent vector $V$, updated $maxRight$
  1. For $i \leftarrow rangeStart$ to $rangeEnd$:
  2.     If $left(C[i]) > maxRight$:
  3.         $maxRight \leftarrow right(C[i])$
  4.         If this is final scan:
  5.             Add $GetDescendants(C[i], name)$ into $V$
```

Joining routine for the parallel scan returns maximum of given *maxRight* values.

Presented algorithm iterates over the initial node-set and retrieves descendants for each node (in the worst case). Hence the time complexity is $\mathcal{O}(size(C) \cdot \log N_{name} + r)$. If the initial node-set is rather large but the document has only very few elements that match given *name*, we have to iterate needlessly over many nodes with no suitable descendants.

We may try different approach to the problem. Instead of iterating over context node-set, we rather test every candidate from $Index[name]$. A candidate should be added into the result if it has at least one ancestor present in the context set. So we just traverse DOM structure and test each candidate's ancestor whether it is in C using binary search algorithm.

```
Algorithm: GetDescendants$_{2\langle\rangle}$
  1. Prepare $S$ with the same size as $Index[name]$, filled with $NULL$s
  2. Parallel for $i \leftarrow 0$ to $size(Index[name]) - 1$:
  3.     $n \leftarrow Index[name][i]$
  4.     While $n$ has a parent and $S[i]$ is still $NULL$:
  5.         $n \leftarrow parent(n)$
  6.         If $C[BinSearch(C, left(n))] = n$:   (Does $C$ contain node $n$?)
  7.             $S[i] \leftarrow Index[name][i]$
  8. Compact $S$   (remove remaining $NULL$ values)
  9. Return $S$
```

The GetDescendants$_{2\langle\rangle}$iterates over index node-set and test ancestor of each node (expected depth is $\mathcal{O}(\log N)$) by looking for it in the context set (binary search algorithm takes $\mathcal{O}(\log(size(C)))$ steps). The final time complexity is $\mathcal{O}(N_{name} \cdot$

$\log(\mathit{size}(C)) \cdot \log N)$, so it should outperform the original algorithm if the N_{name} is very small.

Unfortunately, practical results show that the first algorithm performs better. On dual-core system with approximately 25000 nodes in context node-set and 16 nodes in the index set, the latter algorithm takes about twice the time of the former solution.

We belive that the reason for this behaviour is different style in which both algorithms access the memory. First one exhibits the principle of locality when it reads initial node-set sequentially, while the second one jumps from place to place as it traverses DOM and uses binary search algorithm on large array. Therefore, the former approach is more optimized for caches and memory access which makes it faster even though it does slightly more work. We will only use the first algorithm for performance testing in chapter 5.

Following and Preceding

The *following* axis can be optimized greatly as vector operation. We just need to find node from context set with the smallest right-value and then use it as context node for scalar operation *GetFollowing*.

The node with smallest right-value is either first node in the context set or one of its descendants which is also present in C. Therefore, we just check right-values of nodes starting from beginning of the context node-set. Once we find a node that is not descendant of its actual predecessor in C, the search is over and we have our context node.

```
Algorithm: GetFollowing$_{\langle\rangle}$
  1. $i \leftarrow 0$
  2. While $i < \mathit{size}(C) - 1$ and $C[i+1]$ is descendant of $C[i]$:
  3.     $i \leftarrow i + 1$
  4. Return $S$ yielded by $\mathit{GetFollowing}(C[i], \mathit{name})$
```

If we expect that the height of DOM tree is $\mathcal{O}(\log N)$, the search for proper context node will take also at most logarithmic time. Remaining part of the algorithm has the same time complexity as the scalar version of this operation.

Analogically, the *preceding* axis takes initial node with the greatest left-value and use it as context node for scalar GetPreceding. This is even more simple than it was for *following* axis considering such node is the last node in the context set.

```
Algorithm: GetPreceding$_{\langle\rangle}$
  1. Return $S$ yielded by $\mathit{GetPreceding}(C[\mathit{size}(C) - 1], \mathit{name})$
```

Ancestors

Ancestor axis uses naive algorithm with little optimization. Ancestors of every context node are processed concurrently and added into a hash-map. When ancestor node is being inserted while it is already present in the map, remaining ancestors of the context node are not tested nor added since another task must have already processed them.

Algorithm: GetAncestors$_{\langle\rangle}$

```
1. Prepare concurrent hash-map $H$
2. Parallel for $i \leftarrow 0$ to $size(C) - 1$:
3.     $n \leftarrow C[i]$
4.     While $n$ has a parent and $parent(n) \notin H$:
5.         $n \leftarrow parent(n)$
6.         If the name of $n$ matches $name$:
7.             Add $n$ into $H$
8. Export nodes from $H$ to $S$ and sort $S$
9. Return $S$
```

Described optimization works only if there are some ancestors that match given *name*. If there are very little nodes inserted in hash-map, many ancestors will be processed multiple times. We can avoid this by using another hash-map where tested ancestors are inserted if they do not match the *name*. Every processed ancestor will be present exactly in one of these hash-maps, hence we can avoid repetitive testing of nodes by checking both maps for presence of an ancestor. However, practical results show that overhead caused by operations on second hash-map will take over the practical benefit of this improvement.

3.4.4. Vector Operations for Direct Axes

Children and Siblings

Child axis guarantees that different nodes have disjoint children. Therefore, we may use naive algorithm with concurrent vector instead of hash-map. Since we do not know the total number of all children retrieved, we cannot optimize this problem further.

Preceding and following sibling axes use the same approach as the *ancestor* axis. Following siblings of single context node are always inserted in document order. When a sibling is being added while it is already in the hash-map, all its following siblings must have been already processed by another task, so we need not to insert them again.

Algorithm: GetFollowingSiblings$_{\langle\rangle}$

```
1. Prepare concurrent hash-map $H$
2. Parallel for each node $n$ from $C$:
3.     $S_n \leftarrow GetFollowingSiblings(n, name)$
4.     $i \leftarrow 0$
5.     While $i < size(S_n)$ and $S_n[i] \notin H$:
6.         Add $S_n[i]$ into $H$
7.         $i \leftarrow i + 1$
8. Export nodes from $H$ to $S$ and sort $S$
9. Return $S$
```

Analogically, we add preceding siblings in reverse document order (from last to first) and use the same paradigm as for following siblings.

```
Algorithm: GetPrecedingSiblings$_{\langle\rangle}$
  1. Prepare concurrent hash-map $H$
  2. Parallel for each node $n$ from $C$:
  3.     $S_n \leftarrow \mathit{GetPrecedingSiblings}(n, \mathit{name})$
  4.     $i \leftarrow \mathit{size}(S_n) - 1$
  5.     While $i \geq 0$ and $S_n[i] \notin H$:
  6.         Add $S_n[i]$ into $H$
  7.         $i \leftarrow i - 1$
  8. Export nodes from $H$ to $S$ and sort $S$
  9. Return $S$
```

Sibling axes might be optimized a little more by precomputing leftmost (or rightmost) context node in every children list. However, it is inconclusive whether this approach would have positive impact on performance. Furthermore, sibling axes are not the most often used axes from XPath, thus we do not improve them any further.

Parents

Scalar method for retrieving parents is standard DOM operation as each node holds reference to its parent. When *parent* axis is processed as vector operation, multiple initial nodes may have the same parent, thus we have to ensure uniqueness of result nodes. Filtering initial node-set for siblings would be rather complicated though. Therefore, we use naive algorithm and insert all parents of initial nodes into a hash-map.

```
Algorithm: GetParent$_{\langle\rangle}$
  1. Prepare concurrent hash-map $H$
  2. Parallel for each node $n$ from $C$:
  3.     Add $\mathit{parent}(n)$ into $H$
  4. Export nodes from $H$ to $S$ and sort $S$
  5. Return $S$
```

Attributes

Unlike parents, attributes attached to different nodes are unique, so we can use naive algorithm with concurrent vector instead of hash-map.

```
Algorithm: GetAttributes$_{\langle\rangle}$
  1. Prepare concurrent vector $V$
  2. Parallel for each node $n$ from $C$:
  3.     If $\mathit{name}$ is '$*$':    (wildcard matches every attribute)
  4.         Parallel for each attribute $a$ of node $n$:
  5.             Add $a$ into $V$
  6.     Else:   (a node can have only one attribute of given name)
  7.         If $n$ has attribute '$\mathit{name}$': Add this attribute into $V$
  8. Export nodes from $V$ to $S$ and sort $S$
  9. Return $S$
```

We also have to distinguish, whether we are looking for one attribute or we would like to retrieve them all (when name is a wildcard). When attribute name is specified, we can extract it directly using DOM operations (described in 3.2.1). Otherwise we just copy the whole list of attributes to the result.

3.5. Remaining Issues of XPath Queries

Even though location paths are the most intriguing part of XPath, the language specification contains also other expression types (see appendix A). Many of them are not interesting from our point of view since they cannot be parallelized efficiently. However, there are two types of expressions where concurrency may be exploited.

3.5.1. Union of Node-sets

Union is binary operator '|' which accepts any expressions returning node-set as operands. The operator unites nodes returned by its operands into single set. Usually, binary operators use left associative rule, thus expression $(op_1 \mid op_2 \mid op_3)$ is in fact $((op_1 \mid op_2) \mid op_3)$. For better performance, we aggregate multiple following unions into one n-ary expression.

Union operands may be evaluated concurrently using the parallel for template. The results are gathered by concurrent hash-map which ensures node uniqueness and synchronizes writing operations.

```
Algorithm: Union
Input: Operands $op_1, \ldots, op_k$, evaluation context $c$
Output: Node-set $S$
  1. Prepare concurrent hash-map $H$
  2. Parallel for $i \leftarrow 1$ to $k$:
  3.        $S_i \leftarrow op_i(c)$
  4.        Add $S_i$ into $H$
  5. Export nodes from $H$ to $S$ and sort $S$
  6. Return $S$
```

First-node Optimization

When a node-set is converted into string or number, string value of the first node in the set is taken (and converted into number if necessary). If we know that the result of the union will be converted, we may optimize the algorithm to yield only the first node of the result instead of constructing the whole set. We use parallel reduce to find the first node.

The main routine of the reduce just evaluates corresponding operand and yields its first node. If operand returns an empty set, a *NULL* value is yielded. Joining routine only returns node with smaller left-value (we additionally define $\mathit{left}(\mathit{NULL}) = \infty$ for this comparison). If both nodes are not present, the join also yields *NULL*.

3.5.2. Comparisons

XPath has six binary operators ($<, \leq, >, \geq, =, \neq$)[6] which compare their operands and return boolean value. This value signals whether the operands are in relation defined by the operator or not. Comparisons are quite complex since their semantics depends on result type of compared operands. Details of this operations are thoroughly described in appendix A as well as in [CD⁺99].

Comparisons of two scalar (string, number or Boolean) values are not interesting for parallelisation. We shall focus solely on operations where at least one operand is a node-set.

Comparing Scalar with Node-set

When node-set is compared with a string or a number, every node from the set is taken and its string value is retrieved. This string value is then compared with the scalar. If at least one of these comparisons is true, the whole result is true.

We can compare nodes with the scalar concurrently using the parallel for template. However, when the result becomes *true*, we should not compare any other nodes and terminate immediately. Access to the result value will be synchronized using atomic operations.

```
Algorithm: Compare (Scalar – Node-set)
Input: Node-set $S$, scalar $s$ and comparison operator $\bowtie$
Output: Boolean value $res$
  1. $res \leftarrow false$
  2. Parallel for each node $n$ in $S$:
  3.     $i \leftarrow rangeStart$
  4.     While $i \leq rangeEnd$ and $res \neq true$:
  5.         If $(string(n) \bowtie s)$ is $true$:
  6.             $res \leftarrow true$
  7.         $i \leftarrow i + 1$
  8. Return $res$
```

Since simultaneously running tasks cannot be stopped exactly at the same moment when *res* is set to *true* some redundant work will be done. However, practical results show that the parallel approach performs better even though it does some unnecessary comparisons.

Comparing Two Node-sets

According to XPath specification, a comparison of two node-sets $S_1 \bowtie S_2$ is true if and only if a pair of nodes (n_1, n_2) exists, so that $n_1 \in S_1, n_2 \in S_2$ and $n_1 \bowtie n_2$ is true. This concept works for any type of comparison, but each type may be optimized in different way.

Every following algorithm takes two node-sets S_1, S_2 for input. If one of the input sets is empty, the result is obviously *false*. Henceforth, we assume that both node-sets are not empty.

We examine the equality operator first. Two nodes are equal if their string values are equal, therefore we need to compute string value for every node in both

[6] We shall use $\bowtie$ symbol as representant of an arbitrary comparison operator.

sets. Furthermore, we would like to avoid comparing every possible pair of nodes which would lead to time complexity $\Theta(size(S_1) \cdot size(S_2))$.

The algorithm has two phases. In the first phase we insert string value of every node from the first set into a string concurrent hash-map. Then we test string value of every node in the second set whether it is present in the hash-map. If two matching strings are found, comparison result is true. Both phases are easily parallelizable and have the time complexity $\Theta(size(S_1) + size(S_2))$.

Algorithm: Compare$_=$ (Node-set – Node-set)

```
1. $res \leftarrow false$
2. Prepare an empty concurrent hash-map of strings $H$
3. Parallel for each node $n$ in $S_1$:   (render $S_1$ string values into $H$)
4.       Add $string(n)$ into $H$
5. Parallel for each node $n$ in $S_2$:
6.       If $res$ is $true$: Terminate loop
7.       If $string(n) \in H$:   (pair of equal nodes found)
8.             $res \leftarrow true$
9. Return $res$
```

Inequality comparison can be optimized even further. The first phase of this algorithm proceeds the same way as equality comparison. When hash-map contains at least two different string values and second node-set is not empty, the result is obviously true. No node from second set can have its string value equal to two different strings from the map, hence a pair of unequal nodes must exist. If the hash-map contains exactly one string value, we will compare it as scalar with the second node-set with *Compare* algorithm described earlier. Both phases are linear, hence the comparison requires $\Theta(size(S_1) + size(S_2))$ time.

Algorithm: Compare$_{\neq}$ (Node-set – Node-set)

```
1. Prepare an empty hash-map of strings $H$
2. Parallel for each node $n$ in $S_1$:   (render $S_1$ string values into $H$)
3.       Add $string(n)$ into $H$
4. If $size(H) > 1$:   (at least two different values $\Rightarrow$ unequal pair exists)
5.       Return $true$
6. Else:   ($S_1$ is not empty, hence $H$ must have exactly 1 item)
7.       Return $Compare(S_2, H[0], \neq)$   (use scalar – node-set comparison)
```

Relational operators $(<, \leq, >, \geq)$ always compare numbers, so every string value of each node is converted to number before compared. If such conversion is not possible, numeric value of the node will be *NaN* (not-a-number). Let us recall that any comparison with *NaN* is always *false*.

We will also use the two-phase technique for the following algorithm. However, unlike in string comparisons, we need not to build a hash-map with all numeric values. We just compute minimal and maximal numeric value of the first set and compare every value from second set with them. If at least one value compared with minimum or maximum returns true, the result of node-set comparison will be also true. We shall use parallel reduce for retrieving minimum and maximum (as described in 3.1.3).

```
Algorithm: Compare$_{<,>}$ (Node-set – Node-set)
Input: Node-sets $S_1, S_2$, relational operator $\bowtie$
1. $res \leftarrow false$
2. Use parallel reduce to compute $min$ and $max$ of $S_1$
3. If $min$ and $max$ are $NULL$: Return $res$ and terminate
4. Parallel for each node $n$ in $S_2$:
5.     If $res$ is $true$: Terminate loop
6.     $x \leftarrow number(string(n))$
7.     If $(min \bowtie x)$ or $(max \bowtie x)$:
8.         $res \leftarrow true$
9. Return $res$
```

Parallel reduce works in linear time, therefore the time complexity of the relational comparison of two node-sets is $\Theta(size(S_1) + size(S_2))$.

4. Implementation

In this chapter we will take a brief look at important implementation techniques and architectural decisions.

4.1. Languages and Libraries

The choice of programming language is always the most essential for every implementation. There are many alternatives, therefore we have to specify our needs and priorities first.

Evaluation speed is the most crucial issue in our case, so we cannot use interpreted languages. We will also require certain level of control over some lower operations especially concerning memory access and allocation. Memory safe languages with native implementation of memory disposal (like Java or $C^\sharp$) are out of the question since the compacting process of garbage collector would affect our results. Also, we would like to avoid exotic languages so that anyone can study or use it. Finally, the available bindings for used libraries must be taken under advisement.

Considering presented issues, the most acceptable choices are C or C++. We will use C++ so that we can benefit from the advantages of object oriented programming as well as from the standard template library STL [STL] which is embedded in language specification.

In order to maintain coding complexity in acceptable scope, we require libraries for parsing XML documents and managing concurrent execution of code (threads, synchronization primitives, etc), so we do not need to implement them ourselves.

We have chosen libxml2 [lib] library for XML document parsing. Even though it is written in C, the SAX parser presents trivial API which can be easily used also in C++ application. Other libraries (like Xerces [Xer]) were dismissed as they were too complex for our needs.

There are two libraries for parallel programming that comply with our requirements. The OpenMP [Ope] is in fact a compiler extension which presents pragma directives and some functions for concurrent execution of code. The alternative was Threading Building Blocks [tbb], a stand-alone multiplatform library with prepared programming templates and data structures. Both choices have their advantages and disadvantages. We have chosen the Threading Building Blocks considering they do not require any support from compiler and they offer more complex tools for parallelism.

4.2. Query Representation

Queries are represented by simple object model. This model is independent on query language even though it is greatly inspired by XPath. Every atomic expression has its corresponding class. Classes with similar properties have common base class.

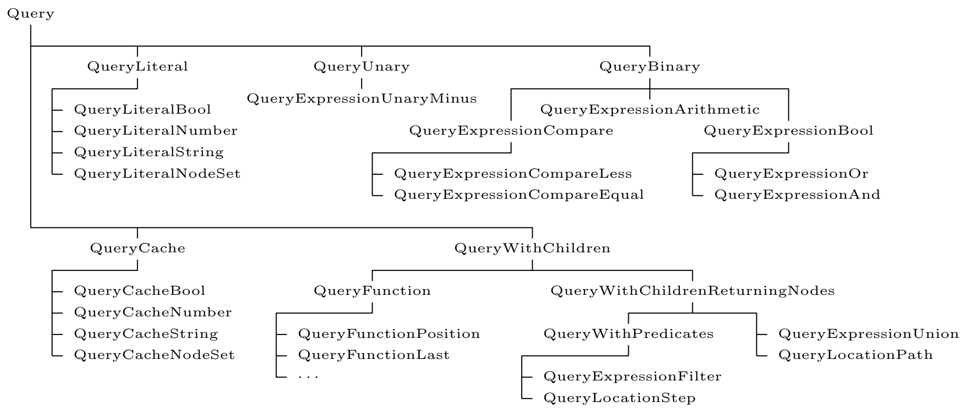

Figure 4-1: Class diagram of query object model

Arithmetic, logic and comparison operators are implemented as binary expressions. Even though more complex operations could be also decomposed into binary tree, we rather aggregate them. Expressions with more than two operands are descendants of `QueryWithChildren` class. This way all predicates, location steps in a path or operands in union expressions can be evaluated more efficiently.

Every expression has fixed result type which may be boolean, number, string or a node-set. However, sometimes the result is converted to another type either implicitly (defined by semantics of parent expression) or explicitly by conversion functions. It would be prudent to inform the query that the result will be converted, so the the query can optimize its work.

In order to achieve this, each expression object has four different methods of evaluation, each yielding different data type. When an expression requires a value of specific type from its operand, it calls corresponding method on the subexpression, so the subexpression is aware of the conversion and it can optimize its work. This technique is used for example in first-node optimization of union expression (3.5.1).

4.3. Parsing XPath and Query Optimizations

XPath is simple context-free language which can be processed by LALR(1) parser[1]. We could have used one of existing parser generators such as Bison [Bis] to create a deterministic pushdown automaton. However, the XPath language is simple enough to be implemented directly.

The parser consists of lexical and syntactic analyser. Lexical analyser parses query string into tokens. These tokens are then processed by syntactic analyser which constructs the object model. Instead of pushdown automaton, we use set

[1] Lookahead left-to-right parser which looks one token ahead.

of methods, each corresponding to one nonterminal symbol (see appendix A), and indirect recursion.

4.3.1. Optimizations

When query is parsed from its text representation into object model, several optimizations are performed to increase the efficiency of evaluation algorithms. The optimization process consists of following tasks:

- Constant subexpressions are computed and replaced by proper type of literal object.
- Predicates that are always true are removed.
- Location steps or filter expressions with predicate which is always false are replaced by empty node-set literal.
- Two following unary minuses or two nested *not*() functions are removed since they have no effect.
- Any function for explicit conversion (*boolean*(), *number*() or *string*()) is removed if its argument already has proper data type.
- Location steps '`self::*`' without predicates are removed.
- If location step '`descendant-or-self::*`' without predicates is followed by descendant or child location step which does not have any predicate in full-context class, the '`descendant-or-self::*`' step is removed and child axis is transformed into descendant. For example '`a//b`' is transformed into '`a/descendant::b`'.

Last two optimizations are the most important as they improve performance for queries written in abbreviated syntax such as '`./x`' or '`//y`'. Without these optimizations, both presented samples would be evaluated as two location steps even though they can be reduced to single step.

5. Experimental Results

In order to verify our claims about scalability and determine how much the designed improvements are beneficial, our solution was submitted to various tests. We have also compared it with two known XPath implementations – libxml [lib] and Xalan [Xal] using standardized XPathMark [XPa] test.

5.1. Preparation

Before testing results are presented, we will revise our methodology and hardware used.

5.1.1. Methodology

Performed tests focus solely on execution speed. We will measure time required to evaluate a query using system real-time clock. Other operations such as loading XML data or building element index will not concern us. Real-time clock will better reflect the practical characteristics of the implementation and cover both application and kernel time (thus include context switching, thread synchronization, etc.).

Unfortunately, results clocked by the selected method are predisposed to be highly influenced by other processes running on the same system and hardware interruptions. We have designed methodology which should reduce this influence error as much as possible.

Each test consists of *query list*. When the tested XML document is loaded into DOM structure and indexed, each query on the list is evaluated 10×, yielding measured times t_i ($i \in \{1, \ldots, 10\}$). First, a *raw average* $\bar{t}$ is computed as

$$\bar{t} = \frac{1}{10} \sum_{i=1}^{10} t_i$$

Raw average is used to filter out times which have been biased too much. A new set T is constructed containing only those time values which are smaller than $\bar{t}$ with 25% correction.

$$T = \{t_i | i \in 1, \ldots, 10 \wedge t_i \leq \bar{t} \cdot 1.25\}$$

The T is obviously not empty considering there must be at least one t_i which is lesser or equal to $\bar{t}$. Elapsed time is then determined as average of values in the filtered set

$$t = \frac{1}{|T|} \sum_{\tau \in T} \tau$$

It is still possible that all ten measured values are distorted due to some long lasting activity running on the system at the same time as the tests. Therefore, each test was repeated 3× in different daytime. These three results were closely compared and if one of the values was obviously tainted, the test was repeated. Final time values presented in result tables below were computed as average from all three results.

The data were not subjected to extensive statistical analysis as the main objective of this chapter was only to show simple conclusions which can be determined just from presented values. All measured values are also published on attached DVD, so that anyone can analyse them further.

5.1.2. Hardware Specifications

All test were performed on Dell M905 server with four quad-core AMD Opterons 8356 (i.e. 16 cores) clocked at 2.3 GHz. Server was equipped with 64 GB of RAM (running on 667 MHz) organized in 4-node NUMA [NUM]. A RedHat Enterprise Linux (version 5.3) was used as operating system.

5.2. Scalability

First set of tests is designed to determine scalability of our solution. In other words, we will measure how much the application benefits from multiple processor cores.

5.2.1. Testing Data

XML Documents

We use synthetic XML documents with following structure, for scalability tests. Each document has root element '`root`'. Element nesting is created completely randomly and restricted by rules in the table below.

Element	Children	Attributes
`root`	`a`, `b`, `c`	
`a`	`b`, `c`, `d`, `e`	`id`(1.0), `info`(0.3)
`b`	`c`, `d`, `e`, `f`	`id`(0.5)
`c`	`b`, `d`, `e`, `g`, `h`	`info`(0.7)
`d`	`a`, `d`, `e`, `f`, `g`, `h`	`x`(0.5), `y`(0.6), `z`(0.1)
`e`	`e`, `f`, `g`	`ref`(0.1)
`f`	`g`, `h`	`ref`(0.3), `x`(0.3)
`g`	`h`	`ref`(0.9), `y`(0.1)
`h`	text only (or empty)	`z`(0.1)

Table 5-1: Structure of synthetic documents

Some of the elements have attributes. Number in parenthesis after each attribute identifier defines frequency of appearance in the document (e.g. attribute '`z`' is attached to one tenth of all '`h`' elements). Attribute '`id`' contains unique identifier value and '`ref`' contains reference to existing id value. All other attributes have integers as values.

Documents in the following tests have been denoted D_i, where i resembles number of elements in thousands (document D_{25} contains 25'000 elements). Documents D_{10} and D_{25} have maximal depth 8, D_{50} has depth limit 9, and depth of D_{100} is 10.

Queries

Queries have been chosen carefully to cover various aspects of XPath location steps with different axes, predicates filtering and node-set comparisons. There are no simple queries in our selection since they would be resolved very quickly even on large documents and time intervals shorter than 1 ms have too big a measuring error, thus we cannot use them to determine scalability.

id	XPath query
Q_1	`//a//b//following::h[2]`
Q_2	`//c[.//h[following::a[ancestor::*[not(self::a)]]][3]]`
Q_3	`//g[@ref=following::e/@ref or @ref=preceding::f/@ref]`
Q_4	`//*[@id=..//@ref]`
Q_5	`//h[following::d]/parent::g/following-sibling::f`

Table 5-2: Queries for testing scalability

5.2.2. Measured Times

Following tables show measured times for each query using 1, 2, 4, 8 and 16 threads. All values are presented in milliseconds. Each table also shows the total time for all five queries and total speedup relative to one thread.

Threads:	1	2	4	8	12	16
Q_1	4.853	4.223	2.91	1.932	1.806	1.886
Q_2	52.11	105.4	71.62	46.67	38.2	33.5
Q_3	144.6	84.44	45.72	24.17	17.34	13.01
Q_4	1364	741.4	382.7	212.4	147.9	126.8
Q_5	5.445	3.442	2.156	1.393	1.296	1.18
Total:	1572	940.9	509.1	294.6	218.5	192.4
Speedup:	1×	1.67×	3.09×	5.34×	7.19×	8.17×

Table 5-3: Evaluation times in ms for XML document D_{10}

Threads:	1	2	4	8	12	16
Q_1	17.84	11.79	6.543	5.442	4.86	4.612
Q_2	326.3	541.8	421.1	269.3	226.5	200.9
Q_3	1558	740.3	447.3	258.8	192.9	153.9
Q_4	14153	6341	3955	2043	1564	1406
Q_5	26.4	16.49	9.56	5.753	5.442	4.495
Total:	16083	7653	4844	2590	2006	1786
Speedup:	1×	2.1×	3.32×	6.21	8.02×	9.01×

Table 5-4: Evaluation times in ms for XML document D_{25}

Paradox of 2.1× speedup on two cores is most likely caused by slightly tainted results of Q_4 on one thread.

Threads:	1	2	4	8	12	16
Q_1	85.01	45.89	21.95	12.57	11.96	10.53
Q_2	1050	1736	1338	827	720	601.3
Q_3	9062	4764	2728	1433	1116	834.7
Q_4	49434	24267	13746	6651	5542	4210
Q_5	152.8	64.08	36.04	20.84	18.98	16.24
Total:	59785	30879	17874	8952	7421	5689
Speedup:	1×	1.94×	3.34×	6.68×	8.06×	10.51×

Table 5-5: Evaluation times in ms for XML document D_{50}

Threads:	1	2	4	8	12	16
Q_1	254.4	141	69.7	37.33	32.19	25.95
Q_2	3586	5783	4417	2881	2249	2050
Q_3	54286	30874	19143	10157	6752	5803
Q_4	157903	91896	55235	30163	20704	19145
Q_5	455.3	211.8	109.3	68.53	60.46	44.65
Total:	216486	128908	78978	43315	29810	27085
Speedup:	1×	1.68×	2.74×	5×	7.26×	7.99×

Table 5-6: Evaluation times in ms for XML document D_{100}

First three tables indicate that larger documents are better parallelized by more threads. The best speedup (11.74× on 16 cores) was achieved by Q_4 on D_{50} and the total average over 8× speedup is well above expected values.

Unexpectedly, the largest document do not scale as well as smaller ones. Probably, the serial parts of the algorithms and synchronization overhead grow significantly with the size of data set. It is also possible, that results measured on larger documents are more distorted by interactions with other processes running on the system.

Query Q_2 has exhibited strange behaviour for all documents. Time measured for one thread is always smaller than times on two and four threads and even with sixteen threads the speedup does not exceed 2×. This phenomenon is most likely caused by synchronization on query caches and we examine this hypothesis in the following chapter.

5.3. The Impact of Query Caches

We have solved the problem of repetitive execution of predicates by installing query caches for intermediate values. These caches may prevent needless work, but they unavoidably slow down the evaluation process. We shall test their impact on performance using following queries on document D_{50} from previous tests.

id	XPath query
Q_1	`//a/following::b[following-sibling::b[*]][.//c]`
Q_2	`//a/following::b[.//c][2]`
Q_3	`//a/following::b[.//c[.//e[.//f[.//g]]]][2]`
Q_4	`//h/following::g[@ref][2]`

Table 5-7: Queries for testing impact of caches

The queries were designed to test both recursive and vectorial approach to evaluation and behaviour of caches on nested predicates. Henceforth, we denote $\mathcal{C}_i$ implementation without caches running on i threads and $\mathcal{C}_i^+$ implementation using caches on i threads.

	$\mathcal{C}_1$	$\mathcal{C}_1^+$	$\mathcal{C}_4$	$\mathcal{C}_4^+$	$\mathcal{C}_8$	$\mathcal{C}_8^+$	$\mathcal{C}_{16}$	$\mathcal{C}_{16}^+$
Q_1	823.6	222	232.4	262.5	111.7	173.1	64.94	122.3
Q_2	1922	970.2	583.5	1145	326.6	735.6	187.5	521.6
Q_3	5435	957.7	1336	1158	693.4	741.7	415.4	535.4
Q_4	10808	4450	3437	5334	2112	3455	1232	2529
Total:	18990	6601	5593	7904	3252	5113	1916	3724
Speedup:	1×	2.88×	1×	0.71×	1×	0.64×	1×	0.51×

Table 5-8: Comparison of cached and not-cached evaluation (times in ms)

Caches have significant impact on performance if only one thread is used. On more threads, the overhead of synchronization mechanisms in concurrent hash-map used in caches takes over and our solution become slower than simple version without them.

It would be prudent to try different approach for synchronizing access to cached values. We can also develop a heuristic function to determine how much will cache help a particular query and cover only those which are most likely to benefit from it. Last possibility will be to avoid caches completely if the size of intermediate values and available memory allows us to use iterative methods for XPath evaluation.

5.4. Practical Tests

In order to compare our implementation with other known XPath libraries, we have prepared set of practical tests. These tests use document and queries that are carefully designed to resemble real data and typical searching problems.

5.4.1. Testing Data

Document used for testing was generated by xmlgen, a XML document generator developed under XMark [XMa] project. This document simulates an auction website (a real e-commerce application) and contains over 3 million elements.

Queries evaluated on the document are taken from XPathMark performance tests [XPa]. These queries are especially designed to determine speed of tested XPath implementation. Queries that were not compatible with our subset of XPath were omitted. The document, its DTD and list of applied queries may be found on the attached DVD.

5.4.2. Measured Times

Result times measured for our implementation on 1, 4, 8 and 16 threads are shown in following table along with results of tests performed using the same methodology on libxml (version 2.7.3) [lib] and Xalan-c (version 1.10) [Xal] XPath implementations.

	libxml	Xalan	$\mathcal{C}_1^+$	$\mathcal{C}_4^+$	$\mathcal{C}_8^+$	$\mathcal{C}_{16}^+$
Q_1	544.1	67.89	29.88	13.17	9.267	8.923
Q_2	651.3	1341	3.118	2.809	2.126	2.397
Q_3	704.7	378.9	3.468	2.733	2.223	2.382
Q_4	829.7	50.46	31.74	8.8	5.171	3.807
Q_5	1656	110.5	13.63	3.921	2.701	2.448
Q_6	3592	86.92	42.05	12.07	6.875	5.275
Q_7	20963	788.5	49.12	15.93	9.941	8.79
Q_8	7896	196.2	41.8	12.5	7.366	6.031
Q_9	3324	300.9	26.46	11.92	8.28	6.777
Q_{10}	∞	∞	20870	5302	3892	2756
Q_{11}	33903	2750	120.1	53.92	46.97	39.69
Q_{12}	19757	4277	71.46	25.49	21.96	17.63
Q_{13}	19799	4253	66.54	24.55	19.19	16.87
Q_{14}	∞	∞	791.2	268.3	154.3	106.6
Q_{15}	∞	∞	7566	1896	1075	606.6
Q_{16}	16096	1442	48.31	14.81	8.75	7.347
Q_{17}	942.2	134	82.94	23.66	12.46	7.272
Q_{18}	∞	∞	41681	10992	7688	5436
Q_{19}	749.3	84.14	63.84	20.36	17.68	20.11
Q_{20}	232.4	135.6	179.3	136.3	165.3	149.8
Q_{21}	2632	181.8	34.54	10.28	6.175	4.73
Q_{22}	8209	54290	2.337	2.772	1.813	2.173

	libxml	Xalan	$\mathcal{C}_1^+$	$\mathcal{C}_4^+$	$\mathcal{C}_8^+$	$\mathcal{C}_{16}^+$
Q_{23}	565.5	49.04	17.05	7.41	5.177	6.86
Q_{24}	20944	2747	131.2	54.23	42.77	35.59
Q_{25}	3518	439.9	379.6	328.8	375.4	364.7
Q_{26}	∞	∞	8629	2207	1261	724.7
Total:	∞	∞	80977	21456	14859	10366
Total†:	167508	74105	1438	786.4	777.6	719.6

Table 5-9: Times in ms for XPathMark tests

The ∞ value stands for tests that were terminated manually after more than 2000 seconds (well over half an hour). Total time denoted with † represents sum of all measured times without problematic queries Q_{10}, Q_{14}, Q_{15}, Q_{18} and Q_{26}.

The results clearly state that our implementation is much faster on large documents and complex queries than well known XPath libraries. Furthermore, the libxml and Xalan does not have the *following* and *preceding* axes optimized, hence queries which use them are evaluated very slowly.

5.5. Concurrent Evaluation of Predicates

In chapter 3.3.6, we have suggested that predicates could be in some situations evaluated concurrently which might exploit parallelism even further. Unfortunately, we cannot ignore the problem of redundant work and synchronization overhead. We will compare both approaches on testing data to determine the effect of parallel evaluation of predicates.

For this set of tests, we will use the synthetic document D_{50} described in 5.2.1. First two queries have simple descendant location paths as their predicates, so it is expected that the predicates will be resolved rather quickly. Last test has more complex predicates, therefore it might benefit from parallel evaluation even more.

id	XPath query
Q_1	`//a/following::b[.//c][.//e][2]`
Q_2	`//a/following::b[.//c][.//e][.//f][.//g][2]`
Q_3	`//a[.//@ref=..//@id][count(.//following::h[3])>10][@info>.//h]`

Table 5-10: Queries for testing parallel evaluation of predicates

All tests were preformed on 16 cores in order to fully express the parallel nature of the algorithms. Since caches affect predicate evaluation significantly, we have performed the tests both with and without them. In the following table, the $\mathcal{C}_{16}$ denotes standard implementation executed with 16 threads while $\mathcal{P}_{16}$ stands for implementation with parallel predicate evaluation. The $^+$ sign marks cached versions as it did in chapter 5.3.

	$\mathcal{C}_{16}$	$\mathcal{P}_{16}$	$\mathcal{C}_{16}^{+}$	$\mathcal{P}_{16}^{+}$
Q_1	242.9	370.5	647.2	682.1
Q_2	241.1	642.5	730	925.2
Q_3	2103	1994	2108	1995
Total:	2587	3007	3485	3602

Table 5-11: Serial and parallel evaluation of predicates

Parallel evaluation of predicates does not improve the performance as we can see on measured results. First two queries are much slower (with or without caches) when predicates are executed concurrently. Last query shows slight improvement, however the 5% speedup is almost insignificant.

Poor performance of parallel predicate evaluation is most likely caused by overhead of the scheduler and redundant work. It seems more convenient to test predicates serially and let parallelism express itself in each predicate subquery. It is also possible we might get better results if we design a heuristic method which decides how the predicates will be executed. Complex predicates which cannot utilize multiple threads on their own can be parallelized while simple predicates or predicates that are highly scalable will be executed serially.

6. Conclusions

In the presented work, we have thoroughly explored methods for processing XPath queries and indexing XML data with strong emphasis on scalability and parallel implementation. The main objective was to design and implement algorithms and data structures for parallel XPath evaluation. Various different approaches and techniques were illustrated in chapter 2 and chosen solutions were later formalized in chapter 3.

Presented algorithms were implemented in our prototype application and tested on various XML documents and many XPath queries. The results indicate that our solution scales very well up to 16 processor cores and it is reasonable to believe that it can utilize even more threads when the size of the XML document and complexity of processed query is sufficient.

Presented solution has outperformed other known implementations even when it was restricted to one core. Using multiple cores, it forges its lead as libxml and Xalan are single threaded.

Finally, we have attended the dilemma of redundant computations, in our case the parallel evaluation of predicates. Even though this concept seemed quite promising at the beginning, practical tests have shown that in common situations these techniques do not improve performance much and sometimes even slow down the evaluation.

There are many questions still open for further research, however we have successfully achieved our objectives and proved that XML query evaluation is highly parallelizable problem.

6.1. Possible Improvements

Our prototype implements only a subset of XPath, however it would not be difficult to integrate remaining details of the language into our model. Also our XML representation is not completely DOM compliant yet, however, it carries the basic concepts, hence it can be easily completed to full specification.

In chapter 5.3, we have discovered that query caches slow down evaluation significantly. This problem can be solved by using iterative methods instead of caches. These methods must be also combined with memory-saving techniques since they require $\mathcal{O}(N^2)$ of additional space. As alternative, the caches might be maintained if heuristic function is developed in order to estimate how much a cache accelerates evaluation of particular query and cover only those expressions which will benefit from it.

Our implementation can exploit parallelism only on documents with sufficient size (thousands of nodes at least). Additional algorithm based on bottom-up method should be implemented to optimize evaluation of complex queries on small documents.

Bibliography

[Bis] Bison – GNU parser generator. *http://www.gnu.org/software/bison/*.

[CD$^+$99] J. Clark, S. DeRose, et al. XML Path Language (XPath) Version 1.0. *W*3C Recommendation, 16:1999, 1999.

[CSF$^+$01] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. pages 341–350. Morgan Kaufmann, 2001.

[DOM] XML Document Object Model. *http://www.w3.org/DOM/*.

[GKP05] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *A*CM Trans. Database Syst., 30(2):444–491. ACM, New York, NY, USA, 2005.

[Gru02] T. Grust. Accelerating XPath location steps. pages 109–120. ACM, New York, NY, USA, 2002.

[iee] IEEE 754: Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985.

[KPS04] M. Kratky, J. Pokorny, and V. Snasel. Implementation of XPath axes in the multi-dimensional approach to indexing XML data. *C*urrent Trends in Database Technology-Edbt 2004 Workshops: EDBT 2004 Workshops, PhD, DataX, PIM, P2P&DB, and Clustweb, Heraklion, Crete, Greece, March 14-18, 2004: Revised Selected Papers, page 219. Springer, 2004.

[lib] Libxml2 – The XML Library for GNOME. *http://xmlsoft.org/*.

[NUM] Non-Uniform Memory Architecture (or Non-Uniform Memory Access). *http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access*.

[Ope] OpenMP – API for parallel programming. *http://openmp.org/*.

[Rei07] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[STL] C++ Standard Template Library. *http://www.sgi.com/tech/stl/*.

[tbb] Intel Threading Building Blocks – an open source library for parallel programming. *http://www.threadingbuildingblocks.org/*.

[Xal] Xalan for C++ – XSLT and XPath library developed by Appache. *http://xml.apache.org/xalan-c/*.

[Xer] Xerces for C++ – XML parsing library (SAX and DOM parsers) developed by Appache. *http://xml.apache.org/xerces-c/*.

[XMa] XMark – benchmark for various XML technologies. *http://www.xml-benchmark.org/*.

[XML] Extensible Markup Language. *http://www.w3.org/XML/*.

[XPa] XPathMark – benchmark for XPath 1.0. *http://sole.dimi.uniud.it/ massimo.franceschet/xpathmark/*.

A. XPath Specification

Front-end of the searching engine uses subset of XPath 1.0 [CD+99] language. Some of the XPath features were not implemented since main objective of the thesis lies elsewhere. When speaking about the subset, we will use the term *xpath* for it is not yet fully grown XPath. It is also expected that reader has basic experience with XPath, therefore some details are not described very thoroughly.

The main differences between full XPath and xpath are:

- Underlying DOM model (and xpath therefore as well) ignores comments, processing instructions, and XML namespaces.
- No variables may be used in xpath.
- Function library is somewhat limited. Only few core functions were implemented.
- Location steps does not recognize node type identifiers (such as `node()` or `text()`). Only node name identifiers and '`*`' wildcard may be used after axis specifier.

For better clarity, this appendix will describe lexical and syntactic structure of xpath language. String tokens enclosed by `''` and identifiers written in uppercase denote lexical elements and identifiers in CamelCase are used for nonterminal elements. We will use the same grammar definition syntax which was used in XPath specification [CD+99].

A.1. Data Types and Conversions

When xpath expression is evaluated it yields a result which has one of the four basic data types:

- Boolean (*true* or *false*)
- number (a floating-point number)
- string (sequence of characters[1])
- node-set (collection of nodes without duplicates)

Data conversions between basic types are allowed. They are either implicit (defined by the language semantics) or explicit (by conversion functions). Each data type may be converted into any other type except that nothing can be converted to a node-set.

Core function library defines conversion functions `boolean()`, `number()` and `string()`. Each one converts its only argument into data type of the same name. Implicit conversions use the same rules as these functions. For more details see section A.5.

[1] We expect the string is encoded in plain ASCII or compatible charset.

A.2. Evaluation Context

Every xpath expression is evaluated within given *context*. The context is provided by caller and consists of:

- the *context node* (reference to DOM)
- positive integer representing *size* of the context
- positive integer representing *position* of the node in context (*position* $\leq$ *size*)

Context may change for some parts of the expression (e.g. the predicates). Details about when the context is changed are described in language semantics (A.4).

A.3. Lexical Structure

The lexical structure is very similar to structure of XPath. Tokenization process uses two basic rules:

- When tokenizing, the longest possible token is always returned.
- Whitespace may be used between two tokens even if it is not explicitly allowed by the grammar. Therefore whitespace is not passed from tokenizer to syntactic analyzer.

One expression token is defined as:

```
ExprToken ::= ‘(’ | ‘)’ | ‘[’ | ‘]’
            | ‘.’ | ‘..’ | ‘@’ | ‘,’ | ‘::’
            | NUMBER
            | STRING
            | NODE
            | FUNCTION
            | AXIS
            | Operator
            | WILDCARD
```

For better readability, we will not use any lexical identifiers for the simplest tokens and we will rather write their text form in quotes.

Numbers in xpath are written in standard decadic format. They may even contain decimal part separated by ‘`.`’. However, the number does not contain a sing. Instead of the sign, number may be preceded by unary operator ‘`-`’. Processing unary operators is responsibility of syntactic analysis. Formally the number is defined as:

```
NUMBER ::= [0-9]+ ( ‘.’ [0-9]+ )? | ‘.’ [0-9]+
```

String literal token is any sequence of characters enclosed by `’’` or `""`. Literal must not contain character `’` or `"` inside (depends on type of quotes used). Formally the string token is defined as:

```
STRING ::= ‘’’ [^’]* ‘’’ | ‘"’ [^"]* ‘"’
```

Elements (`NODE`, `FUNCTION` and `AXIS`) are all represented as *identifiers*. Identifier token starts with letter or underscore and may be followed by any number of letters, underscores, hyphens or numbers. The formal definition is:

```
Identifier ::= [_a-zA-Z][-_a-zA-Z0-9]*
```

A set of rules is defined to determine which element is standing for parsed identifier:

- If the following token is '`(`', the identifier must be recognized as `FUNCTION`.
- If the following token is '`::`', the identifier must be recognized as `AXIS`.
- Otherwise the identifier is recognized as `NODE`.

The identifier may also stand for an operator. Details about `Operator` element are described later.

If the identifier is resolved as `FUNCTION`, the function name must match one of the implemented functions in core library. Otherwise an error is reported. If the identifier is resolved as `AXIS`, it must match one of the defined axis identifiers. The xpath implements all axes defined in XPath except for '`namespace`'.

```
AXIS ::= 'ancestor'
         | 'ancestor-or-self'
         | 'attribute'
         | 'child'
         | 'descendant'
         | 'descendant-or-self'
         | 'following'
         | 'following-sibling'
         | 'parent'
         | 'preceding'
         | 'preceding-sibling'
         | 'self'
```

Axis names are quite self explanatory. The only terms that might be little misleading are '`following`' and '`preceding`' (and their variants with '`-sibling`' postfix). For context node c the node n is "following" iff n is after c in document order and n is not a descendant of c nor an attribute node. The following-sibling nodes are following nodes which have the same parent as the context node. Preceding and preceding-sibling nodes are defined analogously.

Finally, only operator and wildcard tokens remain to be described. An operator is formally defined as:

```
Operator ::= '/' | '//' | '|' | '+' | '-'
             | '=' | '!=' | '<' | '>' | '<=' | '>='
             | MULTIPLY
             | OperatorName

OperatorName ::= 'div' | 'mod' | 'and' | 'or'
```

The `MULTIPLY` operator is represented by the '`*`' character. Unfortunately an asterisk is also used as a `WILDCARD` for node name test. Therefore we must define rules how to determine the type of '`*`' when such token is parsed. Similar problem emerges with the `OperatorName` tokens. They are represented by identifiers, so we need to tell them apart from function, axis, and node identifiers.

XPath specification states that if the preceding token is not ‘@’, ‘::’, ‘(’, ‘[’, or an `Operator`, the ‘*’ must be recognized as `MULTIPLY` operator and any identifier must be recognized as `OperatorName`. Otherwise the asterisk is parsed as a `WILDCARD` and type of the identifier is determined by rules described above (for function, axis and node identifiers).

A.4. Syntactic Structure and Language Semantics

The xpath syntax is described by simple context-free grammar. Starting non-terminal symbol `Expr` stands for one xpath expression.

```
Expr ::= OrExpr
```

Logical operators have the highest priority. They convert result of each operand into boolean and then perform logical operation so the result is also boolean. See description of function `boolean()` for details about conversions.

```
OrExpr ::= AndExpr
         | OrExpr ‘or’ AndExpr

AndExpr ::= EqualityExpr
          | AndExpr ‘and’ EqualityExpr
```

Comparison operators can compare any data types defined in xpath and result of such operation is always boolean. Furthermore, the equality testing operators have higher priority than relational operators.

```
EqualityExpr ::= RelationalExpr
               | EqualityExpr ‘=’ RelationalExpr
               | EqualityExpr ‘!=’ RelationalExpr

RelationalExpr ::= AdditiveExpr
                 | RelationalExpr ‘<’ AdditiveExpr
                 | RelationalExpr ‘>’ AdditiveExpr
                 | RelationalExpr ‘<=’ AdditiveExpr
                 | RelationalExpr ‘>=’ AdditiveExpr
```

Since the operators can compare any data type with any other, special rules must be defined how to compare different types.

If neither operand returns node-set and the operator is ‘=’ or ‘!=’, then the less common operand is converted to the type of the other operand. We define relations between standard types as follows: A number is less common than a boolean and a string is less common than a number. If we compare number and string for example, the string is converted into number. Comparison of same types is intuitive therefore not described here.

If neither operand returns node-set and the operator is ‘<’, ‘>’, ‘<=’ or ‘>=’, then both operands are converted into a number (using `number()` function). Numbers are compared according to IEEE 754 [iee] rules.

Comparisons that involve node-sets are defined using terms of comparisons that do not involve node-sets. This is done uniformly for both equality and relational operators. Following rules expect, that at least one of the operands yields node-set as its result.

- If both objects to be compared are node-sets, then the comparison will be true if and only if there is a node in first set and a node in second set so that comparison of their string-values is true.
- If the other operand is string, then the comparison is true if and only if there is a node in the node-set such that the comparison of its string-value and the other string is true.
- If the other operand is number, then the comparison is true if and only if there is a node in the node-set such that the comparison of its string-value converted into number (using `number()` function) and the other number is true.
- If the other operand is boolean, then the node-set is converted into boolean (using `boolean()` function) and compared with the other boolean.

Both logical and relational operators are left associative. For example expression $3 > 2 > 1$ (which should be true from mathematical point of view) is equivalent to $(3 > 2) > 1$. First comparison $(3 > 2)$ turns out to be *true*. Therefore second comparison becomes *true* > 1 (*true* is converted into 1.0) and it evaluates to *false*.

Arithmetic operators convert result of each operand into number and the result arithmetic expression is also number. See description of function `number()` for details about conversions.

```
AdditiveExpr ::= MultiplicativeExpr
               | AdditiveExpr ‘+’ MultiplicativeExpr
               | AdditiveExpr ‘-’ MultiplicativeExpr
```

```
MultiplicativeExpr ::= UnaryExpr
                     | MultiplicativeExpr MULTIPLY UnaryExpr
                     | MultiplicativeExpr ‘div’ UnaryExpr
                     | MultiplicativeExpr ‘mod’ UnaryExpr
```

The minus sign may be used also as unary operator.

```
UnaryExpr ::= UnionExpr
            | ‘-’ UnaryExpr
```

Union expression requires all operands to return node-sets. The result is also a node-set created by performing standard union operation on the sets.

```
UnionExpr ::= PathExpr
            | UnionExpr ‘|’ PathExpr
```

At this point all operators have been described. Therefore the only remaining syntactic structures are literals, function calls, location paths, and filter expressions.

```
PathExpr ::= LocationPath
           | FilterExpr
           | FilterExpr ‘/’ RelativeLocationPath
           | FilterExpr ‘//’ RelativeLocationPath
```

If the filter expression has some predicates or if it starts a location path, it must return a node-set.

```
FilterExpr ::= PrimaryExpr
             | FilterExpr Predicate
```

```
PrimaryExpr ::= ‘(’ Expr ‘)’
              | STRING
              | NUMBER
              | FunctionCall
```

Function call definition consists of function identifier and list of arguments enclosed in parenthesis and separated by commas. Parenthesis are present even if the function does not have any arguments. Each argument is an xpath expression.

Amount and type of arguments and the result type is completely dependent on the function. See A.5 for list of implemented functions.

```
FunctionCall ::= FUNCTION ‘(’ ( Argument ( ‘,’ Argument )* )? ‘)’
```

```
Argument ::= Expr
```

Location paths are the only expression that generates node-set results. We distinguish two types of paths:

```
LocationPath ::= RelativeLocationPath
               | AbsoluteLocationPath
```

Absolute location path does not start in contex node, but in the root node of XML document in which the contex node is present. The abbreviation ‘`//`’ stands for ‘`/descendant-or-self::*/`’ location step, therefore addressing any descendants.

```
AbsoluteLocationPath ::= ‘/’ RelativeLocationPath?
                       | ‘//’ RelativeLocationPath
```

Relative location path starts in context node and consists of one or more location steps. Relative location path may also start by a filter expression. In such case result of the filter expression is taken as initial node-set for first location step.

```
RelativeLocationPath ::= Step
                       | RelativeLocationPath ‘/’ Step
                       | RelativeLocationPath ‘//’ Step
```

One location step is, in fact, a projection from node-set to node-set. It is evaluated as follows. At first, all nodes that are reachable from initial node-set are acquired. Whether a node is reachable from another node is defined by the axis of the step. Acquired nodes are filtered by the node test (see below). Finally the result is filtered by predicates (in order in which the predicates are specified).

The abbreviation ‘`.`’ stands for ‘`self::*`’ and ‘`..`’ is equal to ‘`parent::*`’. Resemblance of ‘`.`’ and ‘`..`’ symbols to directory paths is not just coincidental for their semantics are quite the same.

```
Step ::= AxisSpecifier? NodeTest Predicate*
       | ‘.’
       | ‘..’
```

Axis specifier defines the direction in which the searching will proceed. Abbreviation ‘`@`’ stands for ‘`attribute`’ axis. If no axis is specified, the ‘`child`’ axis is taken as default.

```
AxisSpecifier ::= AXIS ‘::’
                | ‘@’
```

Node test is used for filtering nodes only by their name. It may either contain a node identifier (and then only nodes with that name will be returned) or a `WILDCARD` token (which stands for any name).

```
NodeTest ::= WILDCARD
           | NODE
```

Finally, we have to describe predicates. A predicate is xpath expression enclosed in brackets.

```
Predicate ::= '[' Expr ']'
```

When set of nodes is being filtered the predicate is executed for each node in the set. Context node for the predicate is the tested node, size of the context is equal to node-set magnitude and position is index of context node within the set (in document order). If the filtered node-set is result of location step which uses reverse axis (preceding or ancestor), the nodes are indexed in reverse document order.

Only nodes for which the predicate has evaluated *true* (after conversion to Boolean) are included into the final result. If multiple predicates are used for filtering, each predicate generates new node-set (which must be subset of initial node-set). Generated node-set is used as initial node-set for next predicate which also generates new set and so on. Node-set generated by last predicate is considered to be final result of the location step.

Another abbreviation is used in predicate syntax. When predicate expression evaluates to number, it is not converted into boolean. Instead it is compared with the position value of current context. For example: `child::elem[42]` is an abbreviation for `child::elem[position()=42]`.

A.5. Core Function Library

Only functions specified in this section were implemented in xpath. Unlike full XPath, no other functions may be specified in evaluation context.

Function: *boolean* `boolean(`*object*`)`

Converts its argument into a boolean using following rules:

- A number is converted into *true* if and only if it is neither positive or negative zero nor *NaN*.
- A string is converted into *true* if and only if it is not empty.
- A node-set is also converted into *true* only if it has at least one node.

Function: *number* `number(`*object*`)`

Converts its argument into a number using following rules:

- A boolean is converted into 1.0 if it is *true* and into 0.0 if it is *false*.
- A string that consist of one number in decadic notation with optional decimal part and optional minus sign (the leading and trailing whitespace is ignored) is converted into a double precision number. Any other string is converted into *NaN*.
- A node-set is first converted to a string using `string()` function and the string value is converted into number as it was already described above.

Function: *string* `string(`*object*`)`

Converts its argument into a string using following rules:

- Boolean *true* is converted into a '`true`' string and *false* is converted to '`false`'.
- A number is converted as follows:
  - *NaN* is converted to '`NaN`'
  - positive infinity is converted to '`Infinity`'
  - negative infinity is converted to '`-Infinity`'
  - positive or negative zero is converted to '`0`'
  - integer is returned in decadic form without decimal part
  - otherwise the number is presented in decimal format (with the decimal dot)
- A node-set is converted into a string by returning the string-value of the first node from the set (in document order). If the set is empty an empty string is returned.

Function: *number* `position()`

Returns the position value of actual context.

Function: *number* `last()`

Returns the size of actual context (position of the last node in the context).

Function: *number* `count(`*node-set*`)`

Returns total number of nodes in given node-set.

B. Attachments

A DVD with following contents is attached to the thesis.

- Source codes of our implementation with necessary makefiles for Linux and project files for Microsoft Visual Studio 2008.
- Comparison implementations using libxml and Xalan-c libraries for XPath processing.
- Used libraries (TBB, libxml2 and Xalan-c)
- XML generator used for generating synthetic documents described in 5.2.1.
- XML documents and queries used for testing.
- All measured result times for every test.
- Reference documentation of code generated by Doxygen from source code.
- This thesis in PDF and PostScript format.