Parallel Processing in the Saxon XSLT Processor

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

One of the supposed benefits of using declarative languages (like XSLT) is the potential for parallel execution, taking advantage of the multi-core processors that are now available in commodity hardware.

This paper describes recent developments in one popular XSLT processor, Saxon, which start to exploit this potential. It outlines the challenges in implementing parallel execution, and reports on the benefits that have been observed.

1. Introduction

In recent years, increased hardware speeds have been achieved largely by packing more processors onto a chip. To take full advantage of the processor capacity, therefore, it is necessary to take advantage of parallelism. Languages that can exploit parallel processing, ideally "under the hood" without involvement of the programmer, therefore have great appeal.

The potential for parallel execution has always been one of the justifications for making XSLT a purely declarative language. Declarative languages, without mutable variables, give the compiler much more freedom to rearrange the order of execution, including the ability to perform tasks in parallel. To take a simple example, XSLT's `<xsl:for-each>` "loop" is not actually a loop in the sense of traditional procedural programming languages; rather it is a mapping operator: it applies a function (the body of the `<xsl:for-each>` instruction) to an input sequence (the result of evaluating the expression in the *select* attribute). As demonstrated by the popularity of the map-reduce paradigm, a functional mapping operation is an ideal candidate for parallelisation. While most XSLT processors today will process the selected items one by one, in order, there has always been the freedom to process them in a different order, or in parallel. The difficulty, of course, is to decide when this is an appropriate strategy.

The connection between declarative languages and parallel processing is not new. See for example [11], or [1], or [7]. For example, Chakravarty writes:

> *Declarative programming languages have long been seen as good candidates for programming parallel computers. Their clean semantics makes them suitable for optimizing program transformations, on the source level and during compilation. However, researchers have found it difficult to present efficient parallel execution models for declarative languages. Generally speaking, it appears to be impossible to automatically identify the exact parallelism that leads to a reduction in execution time, and the irregular access to dynamic data structures can result in considerable overhead on distributed memory machines.*

The idea of exploiting parallelism has been mooted since the earliest days of XSLT: here for example is Eric Ray writing on the xsl-list forum on 1 Oct 2000:

> *The subtree processing model of XSLT seems to make it a good application for parallel processing (i.e. using multiple CPUs to process different subtrees simultaneously). Since many people have remarked on the inherent slowness of XSLT processors, I wonder if anyone has succeeded in creating an XSLT processor that successfully divides the work among different processors, resulting in a gain of processing speed. Has anyone tried this? Some of the implementations are written in Java, such as XT. Do they use multi-threading and can they take advantage of multiple CPUs?*

Research attempts at parallel execution of XSLT transformations have been reported. See for example [10], or [8] et al. A characteristic of these systems is that the entire architecture of the processor is designed from the ground up to support parallelisation. While this can yield useful research results, there is a danger that from an engineering perspective, other objectives get sacrificed.

In the commercial domain, there are high-end XSLT processors from IBM and Intel, marketed as hardware-assisted XSLT accelerators, which may well make use of parallel processing internally, but if so, no details are available in the public domain. Altova's marketing literature for RaptorXML intriguingly claims "the engine takes advantage of today's ubiquitous multi-CPU computers to deliver lightning fast processing of XML and XBRL data"; but it is hard to find any technical details on how it does so.

This paper describes the incremental approach to parallelism adopted in the Saxon product (see [9]). Saxon is a widely used XSLT and XQuery processor available in both open-source and commercial editions. Unlike some of the research vehicles described in the literature, Saxon is not designed from the ground up for parallel processing, and there has been no attempt to make radical changes to its architecture to take advantage of multi-core computers. But where opportunities for parallel processing have presented themselves, they have been grasped, and for some workloads they deliver substantial benefits. This paper describes how parallel processing is used in Saxon today, and explains some of the benefits that can be achieved, and the challenges that need to be overcome.

2. Running Multiple Transformations in Parallel

The most trivial form of parallelism, which has probably been offered by all XSLT processors since day one, is the ability to apply the same stylesheet independently to several source documents at the same time. Although this capability is quite easy to achieve, and is probably taken for granted by most users, it is worth saying a few words about it, firstly because it delivers substantial benefits, and also because it creates a few challenges.

Clearly some workloads will benefit immensely from this approach. XSLT is often used server-side on high-throughput web publishing platforms to render XML documents on demand into HTML for viewing on the browser. Typically such platforms have a small number of XML document types, with a large number of instances of those types, and the same XSLT stylesheets are used with high frequency. In such an environment, there is considerable benefit in compiling the stylesheet to efficient code (actual machine code or something higher level), and in executing that code in parallel threads to meet the throughput and response time targets of the online user community.

Saxon has always been optimized for this kind of workload. Perhaps excessively so: figures published by [5] show that while Saxon's run-time performance ranks with the best, this is sometimes at the expense of relatively poor compile time performance, which can be attributed to the amount of time spent optimizing. This is a good strategy for this server-based workload, but a poor one for single-shot adhoc processing where the cost of compiling a large family of stylesheets, such as those used for the DITA or DocBook vocabularies, may dwarf the cost of a typical transformation.

Although parallel execution of independent transformations may appear trivial, it is not without its complications. Two problems in particular have been recurrent over the years: contention, and reliability.

2.1. Contention

A very intensive operation in performing an XSLT transformation is the matching of element and attribute names appearing in the source document with names used in the stylesheet. Comparison of strings is slow, especially when they include lengthy namespace URIs. To eliminate this cost, Saxon has for many years implemented a *NamePool* which maps strings to integer codes, so during execution, the string comparisons are replaced by much faster integer comparisons. Of course, the same mapping must be used when a stylesheet is compiled and when a source XML document is parsed into a tree representation. But because a single compiled stylesheet can be used to transform multiple XML documents, this means that all the XML documents must use the same integer codes, and this means that the

NamePool used to allocate these codes is a shared resource, and as such it can suffer contention. This has been known, in some cases, to cause a significant bottleneck.

A number of techniques have been used to reduce this problem. In earlier releases, all QNames were represented by integer codes, including for example the names of variables and functions, and names in the result document. Today these codes are used only in source documents to which XPath processing is applied. In addition, the module that builds source documents uses caching techniques to minimise contention: synchronized access to the *NamePool* has therefore been greatly reduced in the common case where the vocabulary of names reaches steady state quickly. Nevertheless, it remains a potential bottleneck. There is scope to reduce contention further by partitioning, for example by creating one *NamePool* per namespace, but this can only be a partial solution. A more radical approach has been considered, in which there is a mapping table (one per transformation) from integer codes used in the stylesheet to (different) integer codes used in the source document. This indirection could noticeably reduce transformation speed, but if it increases the scope for parallelism, it could be worthwhile. This lesson illustrates the need to find engineering compromises between different objectives and a variety of workloads; the danger with a research project that focuses on parallelisation to the exclusion of all else is that it fails to achieve a balance.

Contention of course becomes even more of a problem once parallel processing is used within a single transformation. In fact, it becomes the limiting factor on what can be achieved.

Another point worth mentioning here is that the need to avoid contention tends to impose a design where stylesheet compilation and optimisation is completed before execution starts. This way, the data structures representing the compiled stylesheet, whatever form they take, are read-only and therefore contention-free at execution time. However, as we have seen, there are workloads where compiling everything before execution starts is far from optimal. In the massive stylesheets that come with DocBook or DITA, most of the template rules define processing for elements that rarely or never occur in a typical source document; effort spent compiling and optimising these template rules is wasted if they are never used. A just-in-time compilation approach in such cases has many attractions; but it also runs the risk of increasing contention when used in a shared workload.

2.2. Reliability

Even with the very limited form of multi-threading described in this section, there has been a steady trickle of bugs over the years. These bugs are rare, but potentially devastating. They often take years to discover (because the occurrence is probabilistic), and when they do occur they are hard to diagnose. It is often impossible to reproduce the problem "in the lab", that is, anywhere other than the site running a production workload. One such bug a year is too many. We have been very fortu-

nate that the users who discovered these bugs have had the technical competence and commercial patience to take the lead in collecting the data needed to solve them.

Preventing such bugs arising is not easy (see for example [12]) Quote: "Creating software that can be run by multiple threads concurrently is a daunting task—dwarfed only by the act of testing that code". Saxon is implemented in Java, and switching to a different language is not a realistic option, given the existence of around 250K lines of code. Even if it were an option, it's not clear that a different language would really help. Java offers the basic primitives needed to coordinate multiple threads; the problem is that it offers very little in the way of tooling to ensure that a complex program is thread-safe. The basic discipline to ensure that multiple executions of the same stylesheet can run concurrently is very simply stated: code that runs at execution time must not modify the expression tree. One can envisage tools (assisted by annotations in the code) that check such an assertion statically, but we are not aware of any. Saxon includes about 400 classes that interact directly with the expression tree, and if we rely on programmer discipline alone, mistakes will occasionally happen.

(Having said this, we could do better with soak testing. We should probably have a test where we run each of the 10,000 stylesheets in the W3C test framework concurrently in a dozen threads for 24 hours or so, and check that each thread produces correct output. As it is, our concurrency testing is a woefully small part of our total test programme.)

Again: if reliability is imperfect with the relatively trivial parallelism described in this section of the paper, then we need to be extremely cautious about introducing more ambitious parallelism, because reduced reliability is not a price we are prepared to pay for any performance benefits.

3. Multi-threading and Streaming

While memory sizes appear astronomic compared with a few years ago, the size of data files that people want to transform grows at a similar rate, and there will always be a handful of users who need to transform files that are too big to fit in physical memory. Streamed XSLT processing, which avoids the need to build a tree representation of the source document in memory, has therefore been an increasing area of focus in recent years. It is the main focus of XSLT 3.0 ([13]), and is a major area for implementation work in Saxon ([4]).

Multi-threading and streaming are not orthogonal. Indeed, many of the opportunities for multi-threading become more difficult when processing has to be streamed: it is then no longer possible, in the terms used by Eric Ray cited above, to process different subtrees in parallel, because this relies on buffering data in memory. (But for a counter-argument to this assertion, see the paper by Jakub Malý at this conference: [6]). However, all is not lost.

The first use of multi-threading in Saxon was in fact to implement a form of streaming. This provided a mode of processing in which the source document was split into a sequence of subtrees, and each subtree was transformed independently (this is still a simple and useful processing model that is often good enough to solve the streaming requirement). The reason for using multi-threading was primarily to solve a push-pull conflict in the processing pipeline: see [3] and [4]. We refer to a software component as operating in pull mode when it performs a sequence of read operations to obtain its input, and as operating in push mode when it is invoked repeatedly by a supplier of data to process data as it becomes available. A conflict arises when two components in a pipeline both want to be in control: in this case, an XML parser which wants to push data to the XSLT/XPath processor, and an XPath processor which wants to pull data from the XML parser. One solution is to run the first component (the XML parser) to completion, putting all the data in memory, before starting execution of the second component (the XSLT processor). This is the traditional architecture of today's XSLT processors. An alternative solution, adopted in Saxon, is to use two threads for the two processes, passing data from one to the other via a synchronized queue.

But although this approach breaks the document into subtrees that are processed independently, in the current implementation they are processed sequentially rather than in parallel. There are only two threads, one parsing and building the subtrees, the other processing them one at a time as they become available. It would be difficult to split the parsing thread into multiple threads, because it reads the input data sequentially. Splitting the processing thread would be easier, though it would still need coordination to ensure that results are written to the final result tree in the right order.

This approach has fallen into disuse in more recent releases, though it is still used in some cases, for example in the streamed implementation of the new XSLT 3.0 `<xsl:merge>` instruction. The reason is that it is no longer required: the push-pull conflict has been eliminated by rewriting the XPath engine to operate in push mode, accepting input in the form of events triggered by the XML parser. Any performance benefits obtained by running two threads rather than one were an incidental part of the design (the main objective being to reduce memory requirements). It would of course be possible to continue running the parser and XSLT processor in separate threads even when there is no push-pull conflict forcing this, but we would need to make careful measurements to ensure that this actually delivered benefits.

4. Multi-threading in Saxon Today

In the current Saxon release (9.6) there are four main ways multi-threading is used, and they will be described in this section. In all cases, multi-threading is a feature offered only in the Enterprise Edition of the product.

4.1. The collection() function

The *collection()* function reads a set of input files. The W3C specification is deliberately vague about what constitutes a collection, because it needs to accommodate a variety of different database architectures. Although the facility was designed to allow searching a collection of documents held in an XML database, it is also very useful for transforming a collection of raw documents held in filestore (for example, I have a stylesheet that transforms the thousands of documents making up the W3C XQuery test suite into a set of tests suitable for testing XSLT).

By default, Saxon-EE implements the *collection()* function in multiple threads. A pool of threads is allocated (we choose a number based on the number of CPUs available, for want of any better indicator), and the parsing of the source documents making up the collection is distributed among these threads. The XPath expression that invoked the *collection()* function receives the parsed documents in the order in which parsing is completed.

This is a very straighforward use of multi-threading for a task that is easily distributed. There is very little scope for contention (the only shared resources being the *NamePool*, discussed above, and the queue on which each parsing thread places the parsed document on completion). Because XML parsing cost can often dominate transformation cost, the benefit is high, and the risks in terms of contention and reliability are low.

There are decisions to be made about the order of results. Although W3C does not mandate that collection results are delivered in any particular order, users may have an expectation about the order. Another complication is that an expression like `collection()/doc` is mandated to deliver results in document order, which is somewhat arbitrary, but it cannot be assumed that this is simply the order in which the results become available. (Smart users will write `collection()!doc` to avoid any risk of triggering a sort; but not all users are this smart, and some will deliberately prefer a construct that works in XPath 2.0 as well as 3.0.)

In Saxon 9.6, the multi-threading of the *collection()* function was implemented in the default *CollectionURIResolver* class, which is tasked with taking a URI as input and delivering a sequence of documents as output. There are two drawbacks to this design. Firstly, multi-threading doesn't work if the user substitutes their own *CollectionURIResolver*, which is a perfectly reasonable thing to do. Secondly, the approach is incompatible with streaming. If we want each of the documents in the collection to be processed using streaming, then having a *CollectionURIResolver* that pre-builds each document in memory scuppers this. The design has therefore been changed for Saxon 9.7.[1] XSLT 3.0 introduced a new function *uri-collection()* to handle this case. In the new design, the *CollectionURIResolver* returns (synchronously) a sequence of URIs, and the stylesheet can then process the collection either by constructing

[1] Anything this paper says about future releases is subject to change without notice.

in-memory documents (using the *collection()* function) or, for example, by streaming: the code might be written:

```
<xsl:for-each select="uri-collection('my-dir')"> <xsl:stream href="."> <xsl:apply-templates mode="streaming"/> </xsl:stream> </xsl:for-each>
```

This change puts the responsibility for multi-threading onto the *collection()* function or the `<xsl:for-each>` instruction respectively.

The performance benefits of multi-threading the *collection()* function can be illustrated by a simple experiment. The query

```
count(collection('shakespeare')//LINE)
```

took 160ms to count all the `LINE` elements across the corpus of Shakespeare's plays without multi-threading, reducing to 80ms with multi-threading enabled. In this test, 8 threads were used.

4.2. Multiple result documents

When Saxon-EE encounters an `<xsl:result-document>` instruction, it starts a new thread to process it. The original thread continues processing with the next instruction after the `<xsl:result-document>`. When a transformation produces multiple result documents, they are therefore produced in parallel.

This use of multi-threading is considerably more complex, because we now have different instructions in the stylesheet executing simultaneously. It is simplified, however, by the fact that the output of each thread is written (typically serialized to disk) independently of the other threads, so there is no need to combine the outputs of different threads on completion. Nevertheless, there can be interactions between threads. These mainly arise because of the use of lazy evaluation. The different threads can access the same local and global variables, which would be fine if variables really were immutable, but internally, Saxon evaluates variables lazily (and progressively), so access to variables needs to be synchronized. This applies only to variables declared outside the scope of the `<xsl:result-document>` instruction; for variables inside its scope, each thread has its own copy. Any apparent cost that might arise from repeated evaluation of the same variable is eliminated by Saxon's compile-time optimization rewrites, which use loop-lifting to extract expressions from loops if their value is not dependent on the looping variables.

Another complication which might not be immediately obvious is the use of the XSLT 3.0 *try/catch* mechanism to recover from dynamic errors that occur during the execution of the `<xsl:result-document>` instruction. This is the only way that the spawned thread can affect anything that happens in the original thread. Before an `<xsl:try>` instruction completes, it must check that all threads spawned within its scope have completed successfully, and if necessary, it must wait for them to complete. In fact dynamic errors also need to be considered even in the absence of *try/catch* instructions, because the top-level invocation of the transformation via an API call such as *transform()* needs to throw an exception if any dynamic error has

occurred in the transformation. Although we could make the concurrency visible at the application level, we choose not to: the *transform()* method does not return until all threads have completed, and if any thread raises a dynamic error, the call to *transform()* throws an exception.

Executing multiple instructions simultaneously has various other implications which are perhaps mundane, but worth mentioning because getting the detail right can be a lot of effort. One such detail, for example, is the need to ensure that messages produced by different threads (using `<xsl:message>`) are not intermingled in a log file, at least to the extent that the output from each invocation of `<xsl:message>` retains its integrity.

Another detail is the evaluation of the *last()* function: if one result document is produced for each element in some input sequence, then it is quite possible that the *last()* function will be called within the scope of the `<xsl:result-document>` instruction. When *last()* is evaluated against a particular sequence, Saxon has a number of strategies; if the sequence is the result of a path expression, then the path expression will be evaluated twice, once to compute the value of *last()* (which is then retained for future use), and once to retrieve the actual elements. So the various threads handling different items in the input sequence need to co-ordinate with each other to ensure that the cached value of *last()* is shared between them.

Attempting to measure the effect of this optimization, it appears that the effect depends on how much computation is actually done within the `<xsl:result-document>` instruction. In transformations that are merely splitting the input into multiple outputs (where the body of `<xsl:result-document>` is nothing more than an `<xsl:copy-of>` instruction) it appears to make very little difference to the total elapsed time. This appears to be because the transformation time is limited by the I/O activity of reading the input, and creating and writing the serialized output files. In other cases, where more intensive transformation work is involved, we will often see a doubling of overall execution speed.

4.3. Multi-threaded <xsl:merge>

The new `<xsl:merge>` instruction in XSLT 3.0 allows pre-sorted input files to be merged, using streaming to avoid building a tree representation of the files in memory. An example application would be merging the transaction logs from multiple sales outlets into a single transaction log, ordered by time-stamp.

Saxon's implementation of `<xsl:merge>` uses one thread for each input file. This allows Saxon to use SAX (push-based) parsing, as well as spreading the load over multiple processors. There's no intrinsic reason why several StAX (pull-based) parsers couldn't be instantiated in a single thread, one per input file, in which case Saxon could pull data from each one as required without the use of multiple threads; but using multiple push parsers is convenient both because of the performance benefits of spreading the workload, and also because of the engineering benefits of

using the same approach to parsing source documents that is used in other parts of the product.

The design of <xsl:merge> is such that each input source delivers a sequence of *snapshots* — subtrees of the source document. Each snapshot is built by the parser (as a small in-memory tree) and is then placed on a shared queue. The <xsl:merge> process examines these queues (one per source document) and selects the next one for processing based on the values of the merge keys.

At this stage we have not made any performance measurements for <xsl:merge>, either with or without streaming or parallel processing. It's probably a feature of minority interest: the capability is important if you need it, but not everyone does. So it hasn't been at the top of our list for optimization.

4.4. Multi-threaded <xsl:for-each> and <xsl:apply-templates>

The main XSLT instructions used to process a sequence of nodes from the input tree are <xsl:for-each> and <xsl:apply-templates>. In both cases Saxon allows the user to request multi-threading by means of a vendor extension attribute, *saxon:threads="N"*. Unlike the other facilities described in this section, there is no multi-threading "out of the box" in this case; it is available only on request.

This facility essentially allows map-reduce applications to be written in XSLT, with parallelism under the control of the user rather than the compiler. This is not necessarily a disadvantage; users may be able to achieve better results than a system optimizer.

This design is a cautious one. We know that a feature like this will be ignored by 95% of users. To some extent this is our aim, because we know the feature is dangerous. There's a danger of bugs, but more particularly, there's a danger of misuse. We have no idea how many threads to allocate to such an instruction, so we leave it to the user to decide; but we know that most users have no idea either. The more adventurous will hopefully find a good design by trial and error, knowing how to measure the effect on their particular workload. There will probably be a few who see the feature, guess a number, and never test their assumptions, but such users deserve what they get. Hopefully we will slowly get experience and feedback of what works well and what doesn't, and perhaps rules of thumb will emerge that are sufficiently sound for us to automate the process. Perhaps use of a fixed value is the wrong approach anyway; perhaps <xsl:for-each> should allocate N-M threads where N is the some maximum for the transformation as a whole, and M is the number of threads currently active. Only experiment, with a variety of realistic benchmarks, will provide the answer.

Unlike the use of multiple threads for the *collection()* function (multiple input files) and the <xsl:result-document> instruction (multiple output files), its use on <xsl:for-each> and <xsl:apply-templates> instructions creates a serious risk that performance is degraded by the cost of interaction between the threads. These in-

structions are defined by the language semantics to deliver their results in a particular order, and this means that the results of each thread must be saved in memory and reassembled in the correct order before the instruction completes. This cost can be significant, bearing in mind that XSLT instructions will normally stream their results directly to the serializer, without building temporary trees in memory. Nevertheless, one of our users has reported a reduction in the elapsed time of a heavy transformation by a factor of three by using 8 threads in an `<xsl:for-each>` instruction.

5. Futures

I have described the ways in which Saxon uses multi-threading today. What of the future?

We need to consider developments in two categories: internal use of multi-threading to support operations such as the *collection()* function or the `<xsl:merge>` instruction, and language features that allow users to take advantage of multi-threading, along the lines of the existing multi-threaded `<xsl:for-each>`.

In the first category, a natural candidate is a multi-threaded sort: both for explicit `<xsl:sort>` elements, and for the implicit sorting that occurs when a sequence of nodes needs to be delivered in document order. Saxon uses an implementation of QuickSort, and this lends itself well to parallel implementation. Another candidate might me a multi-threaded implementation of `<xsl:for-each-group>`.

In the second category, we will be guided by user experience with the facilities we already provide. It may well be that the need now is not for more multi-threading features, but for instrumentation to help users establish whether their multi-threading strategies are proving effective. Until we get more feedback on how the features work in practice, I don't see us introducing more automatic multi-threading; I can't see Saxon deciding to use multi-threading for `<xsl:for-each>` or `<xsl:apply-templates>` without an explicit user request.

Another interesting instruction with multi-threading possibilities is the new `<xsl:fork>` instruction in XSLT 3.0. This was developed for use with streaming, and allows several actions to operate on a single pass of a streamed input document. The current Saxon implementation is not multi-threaded (input events are passed to each of the actions in turn, and the actions are performed sequentially. But a multi-threaded implementation would be very natural.

6. Conclusions

In this paper I have described the facilities in the current Saxon release (specifically, Saxon-EE 9.6) to allow multi-threaded stylesheet execution. A few users are already getting substantial benefits from the use of these features, but they are not widely known about or understood. Hopefully this paper will help to increase awareness.

What is needed now is for users to report their experiences, to experiment and report their results, and for the product to improve in response to this feedback.

References

[1] Manuel M. T. Chakravarty. On the Massively Parallel Execution of Declarative Programs Ph. D. Dissertation, Technische Universität Berlin, 1997. http://www.cse.unsw.edu.au/~chak/papers/diss.ps.gz

[2] Kay, Michael. Anatomy of an XSLT Processor. Published online by IBM DeveloperWorks https://www.ibm.com/developerworks/library/x-xslt2/

[3] Kay, Michael. You Pull, I'll Push: On the Polarity of Pipelines. Presented at Balisage: The Markup Conference 2009, Montréal, Canada, August 11 - 14, 2009. In Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3 (2009). doi:10.4242/BalisageVol3.Kay01. http://www.balisage.net/Proceedings/vol3/html/Kay01/BalisageVol3-Kay01.html

[4] Kay, Michael. Streamability in Saxon. XML Prague 2014. http://archive.xmlprague.cz/2014/files/xmlprague-2014-proceedings.pdf

[5] Kay, Michael and Lockett, Debbie. Benchmarking XSLT Performance. XML London 2014. http://www.saxonica.com/papers/xmllondon-2014mhk.pdf

[6] Malý, Jakub. Parallel XSLT Processing of Large Documents. XML Prague 2015. http://archive.xmlprague.cz/2015/files/xmlprague-2015-proceedings.pdf

[7] Rishiyur S. Nikhil (Bluespec) and Arvind (MIT). Making the transition from sequential to implicit parallel programming: Part 5 Online newsletter, UBM Electronics, Sept 2007. http://www.embedded.com/design/mcus-processors-and-socs/4007173/Making-the-transition-from-sequential-to-implicit-parallel-programming-Part-5

[8] A Scalable XSLT Processing Framework based on MapReduce Journal of Computers, Vol 8, No 9 (2013), 2175-2181, Sep 2013 10.4304/jcp.8.9.2175-2181 http://www.ojs.academypublisher.com/index.php/jcp/article/view/jcp080921752181

[9] Saxonica: XSLT and XQuery Processing http://www.saxonica.com/

[10] Tianyou Li ; Qi Zhang ; Jia Yang ; Yuanhao Sun Parallel XML Transformations on Multi-Core Processors IEEE International Conference on e-Business Engineering, 2007. ICEBE 2007.

[11] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew Partridge, and Simon L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell Proceedings of Programming Languages Design and Implementation,

Philadelphia, USA, 1996. https://www.macs.hw.ac.uk/~dsg/gph/papers/ps/gum-ifl95.ps

[12] Watts, Nick Getting Started: Testing Concurrent Java Code Online Blog, July 2011. http://thewonggei.com/2011/07/18/getting-started-testing-concurrent-java-code/

[13] XSL Transformations (XSLT) Version 3.0. W3C Working Draft, 2 October 2014. Ed. Michael Kay. http://www.w3.org/TR/xslt-30