Loosely Coupling Java Algorithms and XML Parsers: a Performance-Oriented Study

Giuseppe Psaila
Università degli Studi di Bergamo
Facoltà di Ingegneria
Viale Marconi 5, I-24044 Dalmine (BG), Italy,
psaila@unibg.it

Abstract

The adoption of XML to represent any kind of data and documents, even complex and huge, is becoming a matter of fact. However, interfacing algorithms and applications with XML Parsers requires to adapt algorithms and applications: event-based SAX Parsers need algorithms that react to events generated by the parser.

But parsing/loading XML documents provides poor performance (if compared to reading flat files): therefore, several researches are trying to address this problem by improving the parsing phase, e.g., by adopting condensed or binary representations of XML documents.

This paper deals with the other side of the coin, i.e., the problem of coupling algorithms with XML Parsers, in a way that does not require to change the active (polling-based) nature of many algorithms and provides acceptable performance during execution; this problem becomes even more important when we consider Java algorithms, that usually are less efficient than C or C++ algorithms.

This paper presents a study about the problem of loosely coupling Java algorithms with XML Parsers. The coupling is loose because the algorithm should be unaware of the particular interface provided by parsers. We consider several coupling techniques, and we compare them by analyzing their performance. The evaluation leads us to identify the coupling techniques that perform better, depending on the specific algorithm's needs and application scenario.

1. Introduction

The adoption of XML to represent any kind of data and documents, even complex and huge, is becoming a matter of fact. However, interfacing algorithms and applications with XML Parsers is not a trivial task: in fact, event-based SAX Parsers require algorithms and applications to react to events generated by the parser. Nevertheless, building main memory DOM representations of XML documents does not require applications to be reactive, but it is not suitable to represent huge documents and it is a very inefficient solution.

Recently, some researchers understood that poor performance in parsing/loading XML documents might constitute a serious obstacle to effectively adopt XML-based solutions in many applications, such as web services and e-business systems; therefore, several researches are trying to address this problem by improving the parsing phase, e.g., by adopting condensed or binary representations of XML documents.

We think that this is one side of the coin; the other side is constituted by the problem of coupling algorithms with XML parsers, in a way that does not require to change the active (polling-based) nature of many algorithms. This problem becomes particularly important when we consider Java algorithms, since memory management provided by the Java Virtual Machine is normally significantly less efficient than it can be expected with other languages, such as C++. This problem typically arises with data mining and knowledge discovery algorithms, but it is not limited to this application context. The reader might object that computationally expensive algorithms are not suitable for Java implementation; however, a lot of algorithms and applications are nowadays implemented in Java (see, for instance the Weka suite [6], a very large pool of algorithms for machine learning, entirely implemented in Java and not specifically tied to XML). Thus, the problem of efficiently coupling Java algorithms with XML Parsers is relevant, especially when algorithms have to work on huge XML documents.

This paper presents a study about the problem of coupling Java (and polling-based, active and non-reactive) algorithms with XML parsers. We consider loosely coupling techniques, because we base our work on the requirement that algorithms should not be re-implemented or, simply,

should not be aware of the specific source format; consequently, algorithms should be unaware of the particular interface provided by parsers. We conduct our study by considering several techniques for loosely coupling; we discuss them from the implementation point of view and we compare them by analyzing their performance. The evaluation leads us to identify the coupling techniques that perform better, depending on the specific algorithm's needs and application scenario.

The paper is organized as follows. Section 2 presents significant related work. Section 3 presents some typical problems and practical situations which motivated our work, as well as the case study (namely, the DBLP XML Records [3]) we adopted to make performance evaluations. Section 4 presents the proposed technical solutions, whose efficiency is evaluated in Section 5; in particular, Section 5.3 discusses the application of the evaluated coupling techniques to practical application contexts. Finally, Section 6 draws the conclusions.

2. Related Work

XML is now widely adopted within (networked) applications. Consider, for instance, web services: they are based on the SOAP protocol [5], which uses XML for message headers and content. However, practical use of XML in web services showed poor performance [13]: a SOAP implementation is up to 10 times slower than a corresponding pure Java RMI implementation; it seems that the reason of this inefficiency reside on the complexity of XML parsing.

The problem arises in the context of databases as well [21]. In effect, database vendors are extending their DBMS engines with new functionality that directly support XML; however, it has been recognized that these extensions may significantly reduce the performance shown by a DBMS supporting these new features.

Thus, the problem of coupling algorithms, and in particular algorithms encoded in Java, and XML Parsers is a critical issue, that can be afforded and intended in different way, both in terms of efficiency and effectiveness. Let us consider the efficiency point of view.

Recently, researchers started working on the problem of improving parsing time for XML documents. Of course, the best thing to do is to improve the efficiency of stand alone parsers, but this is a difficult task, since inefficiency of general purpose parsers comes up from their generality. Anyway, researchers are working on this topic and some result is arising: for example [23] exploits efficient data structures that speed-up XML parsing when psrsed XML documents belong to a limited family of documents (such as in the context of web services).

Another approach researchers are investigating is called *Schema-Specific Parsing* [14]: based on a given XML Schema specification, a parser is generated that recognizes only XML documents compliant with the source XML Schema specification. This approach is good in contexts where the XML document family to process is known in advance.

A third approach is the development of hardware accelerators [18], specifically developed for very fast XML document parsing. This solution is suitable for embedding XML processing within devices, but not for general purpose computers.

A different way to reduce XML parsing time is to change XML: this is the idea behind the at least 19 proposals for *binary* representation of XML documents [17], known as *Binary XML*: the idea is that a binary representation is processed faster than a textual one. A specific W3C Working Group, named XBC XML Binary Characterization Working Group, is now working on the problem.

As far as the effectiveness point of view is concerned, this has not been widely investigated yet. Anyway, the interesting work [16] argues that a tight integration between Java programs and XML document management might be reached if Java were extended with features which support XML document processing in a native way. This means that XML Schema specifications are seen as class definitions; thus, an XML document defined over the specified XML Schema specification is a complex object that instantiates that class. This fact allows to perform XPath queries that can be checked against the XML Schema specification at compile time, as well as it is possible to check the correctness of newly generated XML documents at run time and in a totally automatic way. This solution is very interesting, because it can significantly help Java programmers to improve correctness of Java XML-based applications.

In effect, our work is in the middle between effectiveness and efficiency. It concerns effectiveness, in that it might provide a solution to couple algorithms and XML parsers with limited impact over the original algorithm implementation. It concerns efficiency, since we studied solutions with limited impact on the overall performance of algorithm implementations when coupled with XML parsers. At the best of our knowledge, no similar work has been done at the moment.

3. Problem and Case Study

Researchers and developers are introducing XML for a large number of computationally intensive applications. Some of them are Data Mining and Knowledge Discovery tools, Data Warehousing tools, ETL (Extract-Transform-Load) tools and Meta-data exchange formats such as the Common Warehouse Metamodel CWM [1]. In particular, a CWM document is based on more than 100 different ele-

ment types, and when it describes complex data warehouses it can contain thousands of XML element occurrences.

If we consider the context of Data Mining, several initiatives are defining XML formats for representing source data and extracted patterns, such as PMML [4] and CRISP-DM [2]; in particular, PMML may represent both source data and extracted patterns in the same document, which may be huge in practical cases. Furthermore, some research works are defining XML-based frameworks to represent and actually manage the overall knowledge discovery process (called *Inductive Databases* [11]); the basic idea of these frameworks is that data mining tools might be easily added to the framework, by adapting their interfaces and data loaders; if we consider complex data mining activities such as association rule extraction, they usually require to scan several times huge data set, that are represented as XML documents in these frameworks; obviously, tool developer wishes to adapt existing algorithms without changing the algorithms, but simply coupling them with suitable data loaders.

For instance, a typical class of data mining tool is constituted by *classifiers*. An incredible variety of classifiers has been devised in the last decade and more, but only few are optimized to deal with huge data sets. One of the first ones designed to scale over large data sets is described in [19]; however, although optimized and scalable, it requires to perform several scans over the data, as most classifiers do (see [24]). Consequently, the integration of classification algorithms with XML parsers should take into account that parsing XML documents might be a real bottleneck. Nevertheless, these algorithms are complex and loosely coupling them with XML parsers is desirable, even in the case they only perform one single scan over the data set [24].

Observe that even when data mining tools are specifically devised to extract patterns from within XML documents, they can significantly suffer when huge documents are analyzed. For example, in [12, 15] tools for extracting association rules from within XML documents and their implementations are described. However, the implementations build a DOM representation of the document and then perform several passes over the DOM tree. This solution is inadequate for huge documents (see Section 3.1), since it requires an impressive amount of main memory; these tools might be significantly improved by loosely coupling their core with a SAX parser.

Based on the previous considerations, that hold not only for the considered application cases, we can finally specify the requirements for the problem we address in this paper.

1. Algorithms are complex and coupling them with XML parsers should have limited impact, or better should not have impact, on their main code.

2. DOM or any other kind of main memory representation cannot be used. A stream-like solution is necessary.

3. For multi-scan algorithms, avoid multiple parsing of the source XML document.

Only loosely coupling algorithms and XML Parsers can meet the previous requirements,. Then, the problem we address in this paper is the following: *study different techniques for loosely coupling algorithms and XML Parsers that meet the previous requirements, in order to find those with the best performance*.

Dimension	*Elements*
298225014	6964761

Table 1. Characteristics of the DBLP-XML Records Document

3.1. The DBLP XML Records

In order to test the proposed alternative for loosely coupling XML parsers, we adopted a well know huge XML document as test case, i.e. the DBLP XML Records document available on the Database and Logic Programming Bibliography Web site [3].

The characteristics of this file are impressive: 284 Mbyte and about 7 Million XML Elements, with references to more than 675000 articles. It is simply impossible to deal with this document by loading a DOM representation; in fact, it has been widely proved that a DOM tree needs 2 to 5 times main memory than the size of the XML document it represents [21]. In order to enforce that, we run a simple Java application that only had to build the DOM representation of the DBLP XML Records: it was necessary to set the Java Virtual Machine heap size to 1.5 GByte.

Figure 1 reports a fragment from the DBLP DTD. The reader familiar with DTD notation can notice that this document is mainly a collection of bibliographic descriptions. Bibliographic items are catalogued as journal articles (element `article`), conference articles (element `inproceedings`), proceedings, books, Ph.D. and master theses; occurrences of these elements are all in the content of the document root.

Each bibliographic item is in turn structured. We report the definition of `inproceedings` as a representative for all categories. In its content, we can find elements describing authors, editors, title, etc.. While elements like `author` are very simple (they only contain text), elements `title` can contain text and other sub-elements, whose content is structured in the same way; this fact allows an arbitrary nesting depth in the document.

```
<!ELEMENT dblp (article|
      inproceedings|proceedings|
      book|incollection|
      phdthesis|mastersthesis|
      www)*>

<!ELEMENT inproceedings
          (author|editor|title| ...)*>
<!ATTLIST inproceedings
          key CDATA #REQUIRED
          mdate CDATA #IMPLIED >

<!ELEMENT author    (#PCDATA)>
<!ELEMENT editor    (#PCDATA)>

<!ELEMENT title
          (#PCDATA|sub|sup|i|tt|ref)*>

<!ELEMENT ref (#PCDATA)>
<!ATTLIST ref href CDATA #REQUIRED>
<!ELEMENT sup
          (#PCDATA|sub|sup|i|tt|ref)*>
.  .  .
```

Figure 1. A fragment of DBLP XML DTD

For example the following is the DBLP XML representation for [20].

```
<inproceedings mdate="2003-03-04"
               key="conf/icdm/MeoP02">
  <author>Rosa Meo</author>
  <author>Giuseppe Psaila</author>
  <title>Toward XML-Based Knowledge
         Discovery Systems.</title>
  <pages>665-668</pages>
  <year>2002</year>
  <crossref>conf/icdm/2002</crossref>
  <booktitle>ICDM</booktitle>
.  .  .
</inproceedings>
```

4. The Studied Coupling Techniques

The motivations for loosely coupling Java algorithms and XML Parsers has been discussed in the previous section. Now we discuss the evaluated coupling techniques.

The general architecture adopted for loosely coupling Java algorithms and XML Parsers is presented in Figure 2. The architecture is based on the *Producer-Consumer* schema. The *Producer*, that encapsulates the XML Parser,

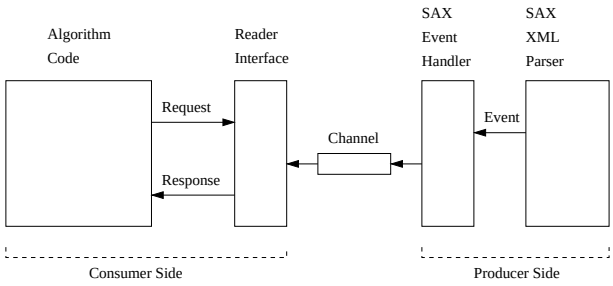

Figure 2. Loosely Coupling Architecture

encodes and possibly filters the source XML document. The encoded representation is sent to the consumer through a communication *Channel*. The *Consumer*, which embodies the algorithm code, receives the encoded description of the source XML document.

Let us describe the architecture in detail. On the *Producer Side*, the XML Parser is a *SAX XML Parser* that notifies events to the *SAX Event Handler*. The handler reacts to events generated by the parser when if recognizes syntactic elements in the source XML document; thus the handler filters and encodes descriptions of XML objects (tags, elements, attributes, text, etc.) and send them through the *Channel*. On the *Consumer Side*, the *Reader Interface* is responsible to decode the document representation obtained through the *Channel*; the *Algorithm Code* obtains XML Object descriptions (tags, elements, attributes, text, etc.) by making a request to the *Reader Interface* and obtaining a response when an XML object description is available (the request is a method call). This way, the structure of the algorithm code is not touched: input instructions are simply translated into calls to the proper method provided by the *Reader Interface*.

Notice that this architecture allows to meet requirement *1* (see Section 3), since a main memory representation of the overall XML document is not built. It is also clear that the key element in this architecture is the way the *Channel* is implemented. In the following subsections, we discuss four different implementations of the *Channel*, that meet requirement *1*.

XPath Filtering. Depending on the specific work applications are designed for, not necessarily algorithms need to read the entire source XML document. For example, data mining algorithms that looks for frequent substructures (frequent subtrees) within a document [10] need to get descriptions of the overall document; in contrast, other algorithms, such as the one extracting association rules [12, 15] works on specific and user defined XML elements, thus have to filter XML object descriptions.

In this latter case, filtering can be performed directly within the *SAX Event Handler*. We implemented an XPath

filtering mechanism, which works with *normalized XPath* expressions [16]: only subtrees rooted in elements selected by XPath expressions are encoded and sent through the *Channel*.

4.1. File Channel

The first implementation we considered for the *Channel* is a *File* channel. In practice, the channel is a temporary file. The *SAX Event Handler* writes the encoded XML object descriptions into the temporary file; thus, during the parsing, the consumer is stopped until parsing ends. Once parsing is terminated, the consumer is activated and the *Reader Interface* reads data from the temporary file.

This solution meets requirements *2* and *3* (Section 3); in particular, requirement *3* is met since, in case of multi-scan algorithms, it is possible to read the temporary file instead of parsing the document again.

Optimization. The critical aspect of this solution is the width of the temporary file: the larger the temporary file, the greater the number of disk access attempts that slow down performance. For this reason, we decided to adopt the following encoding for XML object description in the temporary file.

1. A special character denotes the type of XML object described by next bytes. ASCII character 01 (decimal code) denotes a *Start Element* (called *start element marker*); ASCII character 02 denotes a *Text* (called *text marker*); ASCII character 03 denotes an *End Element* (called *end element marker*); ASCII character 04 denotes the *Document End* (called *document end marker*).

2. A *start element marker* is followed by the element name followed by a possibly empty list of pairs *(attribute name, attribute value)*. element name, attribute name and attribute value are separated by the ASCII 00 character.

3. A *text marker* is followed by a byte sequence constituting the textual content of an element.

4. An *end element marker* is self explaining; no specific character follows.

5. After the *document end marker* no character follows.

With this encoding, we describe the original XML document in a more compact form. For example, consider the following XML fragment.

```
<elementName1 attribute1="value 1"
              attribute2='value 2'>
   text 1
   <elementName2 attribute3="value 3"/>
   <elementName3/>
</elementName1>
```

Including new line characters (decimal ASCII code 10), it is described by 159 characters. Its encoded version, that we report hereafter, needs 121 characters, i.e., the 23.9% smaller. To understand the encoded version, we used the notation `\0`, `\1`, `\2`, `\3`, `\4`, `\10` to denote ASCII characters 00, 01, 02, 03, 04, 10 (new line) respectively.

```
\1elementName1\0attribute1\0value 1\0
attribute2\0value 2\2\10
   text 1\10
   \1elementName2\0attribute3\0value 3
\3\1elementName3\3\3\4
```

Notes about Implementation. When implementing the *SAX Event Handler* and the *Reader Interface*, we had to pay attention to avoid creating useless objects. In fact, the simple allocation of a `String` object for each *Start Element* event, may cause the allocation of millions of temporary objects, causing the frequent activation of the Java Garbage Collector, which makes the entire process very inefficient. Therefore, we implemented the handler in such a way no local objects are created, and only pre-allocated class objects, as well as local elementary type variables are used.

The same is for the *Reader Interface*: only pre–allocated `StringBuffer` objects and vectors of `byte` are exploited.

As far as input/output from file is concerned, buffered `InputStream` and `OutputStream` objects are exploited; this way, we reduce the number of disk access attempts.

4.2. Pipe Channel

The second technique we considered for implementing the *Channel* is based on a *pipe*. The producer and the consumer are executed in time sharing, in two distinct threads. The *SAX Event Handler* writes the encoded XML object descriptions into the pipe. The *Reader Interface* reads the encoded version from the pipe, and, of course, when the pipe is empty it is stopped until a character is available. We implemented this version on top of the standard `Pipe` class provided with the JDK 1.5 distribution.

This solution meets requirements *1* and *2* (see Section 3), while does not meet requirement *3*, since the encoded version is not kept, and simply flows through the pipe. Thus, it is not suitable to optimize multi-scan algorithms.

Optimization. The need to reduce the number of characters sent through the pipe is the same as for the file-based implementation. Therefore, we adopted the same encoding technique discussed in Section 4.1.

Nevertheless, we adopted the same implementation strategy to avoid the generation of useless objects: both the *SAX Event Handler* and the *Reader Interface* do not allocate temporary objects.

4.3. Shared Object Channel

Another implementation we considered is still based on two distinct threads for producer and consumer. The two threads share an object defined on a specific class we named `SharedObject`. An object belonging to this class describes an XML object found in the source file. Access to this object is performed by means of two methods, named `setValue` and `getValue`, which perform a mutually exclusive access (they are defined as `synchronized`): the producer set an XML object description and is suspended until the consumer reads the description; the consumer tries to read an XML object description, but is suspended until the producer provides a new description.

Observe that this solution has a main advantage: it needs a very limited amount of main memory, so that memory access may result significantly optimized. However, it has a major drawback: each thread accesses the critical region as many times as many XML objects are found in the document; consequently, the degree of parallelism reached by this solution might not be high. Anyway, it is important to experiment this solution, to understand the weight of mutually exclusive access to the critical region.

This solution meets requirements *1.* and *2.* (see Section 3, while requirement *3.* is not met. As far as the optimization of code is concerned, in this implementation objects provided by the SAX parser are exploited as property values in the shared object and no extra objects are generated.

4.4. Shared Vectors Channel

The last evaluated coupling technique is an evolution of the previously discussed shared object channel. Instead of sharing one single object, in this solution two vectors of objects (200 elements each) are shared; when a vector is owned by one thread, the other vector is owned by the other thread, and vice versa.

Let us describe this technique in detail. A class named `ShareVectors` provides two methods, named `setValue` and `getValue`, used to set and get XML object descriptions. The producer is allowed to load XML object descriptions on the first vector; when this is full, then the producer can load XML object descriptions on the second vector; when also this vector is full, it is suspended until the consumer has not read all XML object descriptions from the first vector, and so on.

When the consumer tries to download XML object descriptions from the first vector, it is suspended until it is full; at this point, the consumer thread is woken up and downloads data from the first vector; then, when this becomes empty, it tries to access the second vector, and so on.

We implemented the class in such a way threads access the critical region only when they try to get the ownership of a vector: once they get this, they freely access the vector, without entering the critical region. This way, we obtain a high degree of parallelism between the threads.

As far as requirements reported in Section 3 are concerned, requirements *1* and *2* are met, while requirement *3* is not met. As far as code optimization is concerned, the same strategy as for the previous coupling techniques has been adopted, i.e., objects provided by the SAX parser are directly exploited as property values for XML objects described within the shared vectors; furthermore, the vectors are pre-allocated before starting parsing.

5. Performance Study

We ran several experiments in order to evaluate performance of each proposed coupling technique. Experiments were performed on a PC powered by an AMD Athlon(tm) XP 3000+ (clock 2 GHz and cache size 512 KB) and equipped with 1 GByte main memory; the installed operating system is Linux (Fedora distribution, with kernel `2.6.9-1.667`): the Java Distribution is the `1.5.0_02` (runtime environment `1.5.0_02-b09`).

The data set on which experiments were ran is the DBLP XML Records document [3], downloaded on October 19, 2005; its length is 284 MByte (exactly, 298225014 Byte).

In order to evaluate the overhead provided by the different loosely coupling techniques, we ran a simple application based on SAX parsing that only counts the number of nesting levels and the number of XML elements in the documents; since the execution time of this application is negligible w.r.t. the parsing time, we can effectively evaluate the overhead introduced by the different coupling techniques. Table 2 reports about this experiment.

Notice that the Xerces XML Parser [7] (version 2.0) adopted in our experiments is slightly sensitive to validation: about 21 seconds in non-validating mode, and about 26 seconds (only 4 seconds more) in validating mode. In our experiments, we always configured the parser to work in *validating mode*.

The execution times were measured by means of the `time` Unix command. For each test, we performed 5 runs, the worst and the best ones were discarded, then we computed the average of the three remaining values.

5.1. The Experiments

In order to evaluate the performance of the four studied coupling techniques, described in detail in Section 4, we

Exec. Time (No valid.)	*Exec. Time (DTD Valid.)*
21.521 sec	25.998 sec

Table 2. Basic SAX Parsing Performance

Num.	*XPath Expression*	*Elements*
1.	`/*`	6964761
2.	`/dblp/article`	2657631
3.	`/dblp/article/author`	776453
4.	`/dblp/book`	12601

Table 3. The Experiment: XPath Expressions and selected XML Elementss

ran experiments with four different kinds of filters over the source XML document. Table 3 reports the XPath expressions used for each experiment and the number of selected XML elements.

Notice that elements selected by experiment number *3.* are all contained in the set of elements selected by experiment number *2.*; this way, we ensure monotonicity among experiments, both in terms of selected elements and in terms of intermediate representation dimensions (for file-based and pipe-based implementations).

5.2. Evaluation

We now evaluate performances shown by the four considered implementations for channels. Figure 3 depicts the results.

File-based Channel. First of all, consider the performance shown by the *File-based* implementation of the *Channel* (see Table 4).

The first element we evaluate, is the dimension of the temporary file. Without filters, it is 231496530 Bytes, against the 298225014 Bytes of the source XML file; note that our encoding technique is able to save 66728484 Bytes, obtaining a compression rate of 22.4%. If we now consider the dimension of the temporary file with filters (experiments from *2.* to *4.*), we notice that their dimension is not excessive. Consequently, performance significantly benefits: to run experiments *2.*, *3.* and *4.* only 11 seconds, 4 seconds and 1 second (respectively), must be added to the parsing time, which is about 26 seconds, to write the temporary file.

Observe that for all the experiments, we evaluated separately the *write* time and the *read* time; furthermore, Table 4 reports the aggregated parsing and write times, for the sake of clarity. Figure 4 shows the relationship among total execution time, write time and read time: it is clear that the most significant overhead to execution time is always given by the write phase.

Exp.	*Temp. File (Bytes)*	*Exec. Time (secs)*	
1.	231496530	84.189	(full)
		61.083	(parsing + write)
		35.085	(write)
		23.106	(read)
2.	82345310	45.173	(full)
		36.457	(parsing + write)
		11.359	(write)
		9.026	(read)
3.	31218774	32.504	(full)
		30.287	(parsing + write)
		4.289	(write)
		2.217	(read)
4.	447521	27.344	(full)
		26.802	(parsing + write)
		0.804	(write)
		0.542	(read)

Table 4. The Performance: File-based Channel Implementation

In Experiment *1.*, notice the inefficiency introduced by the file system and disk access: more than 30 seconds to write the encoded file, and 25 extra seconds to read it.

In the case of one single scan of the document, the file-based implementation performs well when filters are applied, while it is not adequate for reading the entire document. However, it is a very good solution for multi-scan algorithms when filters are applied and are effective: experiment *2* shows that each extra scan costs only 7 seconds, compared to the 26 seconds of parsing time; Figure 4 also shows that the write phase becomes significantly expensive only when almost the entire XML document is encoded.

Observe that the encoding strategy we adopted for XML object descriptions might be further improved, thus further reducing the size of the intermediate file and, consequently, the execution time; we plan to do that in the figure.

Pipe-based Channel. Consider now the second implementation, i.e. the *Pipe-based Channel*. The execution times are reported in Table 5: column *overhead* is the overhead execution time given by the channel to XML parsing. Note that the number of bytes that flows through the pipe is the same as the length of temporary files generated by the File-based channel, since the adopted encoding technique is the same.

Observe that experiment *4.* is very fast (less than 1 second added to parsing time), so it is possible to suppose that this is a good coupling technique. However, the remaining experiments show that it is the contrary. In fact, even though it is a pure main memory solution, this channel is always much slower than the file-based channel. This means that

Exp.	*Exec. Time*	*Overhead*
1.	166.302 sec	140.304
2.	74.803 sec	48.805
3.	43.662 sec	17.664
4.	26.754 sec	0.756

Table 5. The Performance: Pipe-based Channel Implementation

Exp.	*Exec. Time*	*Overhead*
1.	305.490 sec	279.492
2.	130.794 sec	104.796
3.	57.967 sec	31.969
4.	27.350 sec	1.352

Table 6. The Performance: Shared Object-based Channel Implementation

the implementation of pipes in Java is computationally expensive when large volumes of data are passed through the pipe. Thus, this solution cannot be adopted in practice.

Observe that even under the hypothesis that the compression rate of the encoded description of XML objects is improved, we cannot expect significant improvements: in effect, we can expect to save only a few seconds, but the execution times remain far from the corresponding ones shown by the File-based channel.

Shared Object-based Channel. Consider now the third coupling technique, i.e. the *Shared Object-based Channel*. The execution times are reported in Table 6.

Observe that experiment *4.* is fast (only 1 second added to parsing time), even though the degree of parallelism provided by this solution is not high. However, the other experiments show that the reduced degree of parallelism makes this implementation not suitable; if compared to the File-based and Pipe-based channel, the Shared object-based channel always performs worse.

Shared Vectors-based Channel. Finally, we can discuss the performance shown by the last evaluated coupling technique, i.e. the *Shared Vectors-based Channel*. The execution times are reported in Table 7.

The reader can notice that this coupling technique is very efficient. In the worst case, i.e., experiment *1.* where the consumer reads the overall document, it only doubles the parsing time. If we consider experiments from *2.* to *4.*, it is always faster than the File-based Channel. This is due to the high degree of parallelism reached by the strategy of the two shared vectors (notice that the overhead of 2.6 seconds in experiment *4.* is due to data structure initialization).

Consequently, for one scan algorithms, this is the best coupling technique, since it does not require extra disk

Exp.	*Exec. Time*	*Overhead*
1.	52.241 sec	26.243
2.	36.868 sec	10.87
3.	30.819 sec	4.821
4.	28,601 sec	2.603

Table 7. The Performance: Shared Vectors-based Channel Implementation

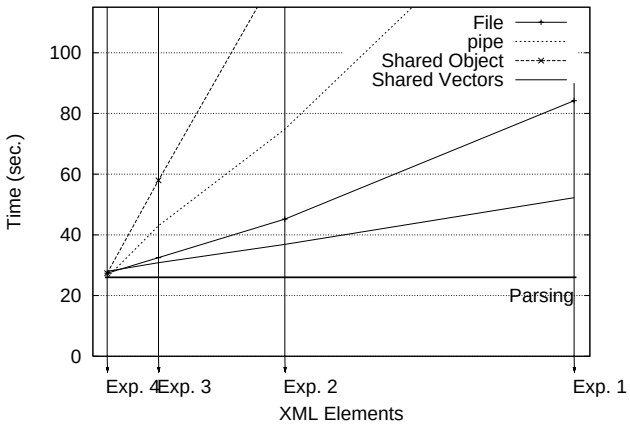

Figure 3. Execution Times

space to work, as required by the File-based channel. In contrast, for multi-scan algorithms, this coupling technique is not suitable, since it has to repeat parsing for each scan.

5.3. Discussion

The execution times observed in our experiments allow us to discuss in depth how to loosely couple algorithms and XML Parsers.

- *Scalability.* All the presented coupling techniques are scalable w.r.t. the number of selected XML elements. This means that a fine-tuned code is suitable to deal with large documents, such as the DBLP XML Records, even though implemented in Java.
- *Techniques to avoid.* The Pipe-based Channel and the Shared-Object Channel are not suitable for large volumes of data. They cannot be used in practice.
- *Using the Shared Vectors Channel.* The Shared Vectors Channel is very suitable for one-scan algorithms that filters the XML elements. Observe that even in the case of experiment *2.*, where more than 2 Million elements are selected, it adds only 11 sec. to the parsing time (about 26 sec.). It is reasonable to think that when an application has to deal with a so large number

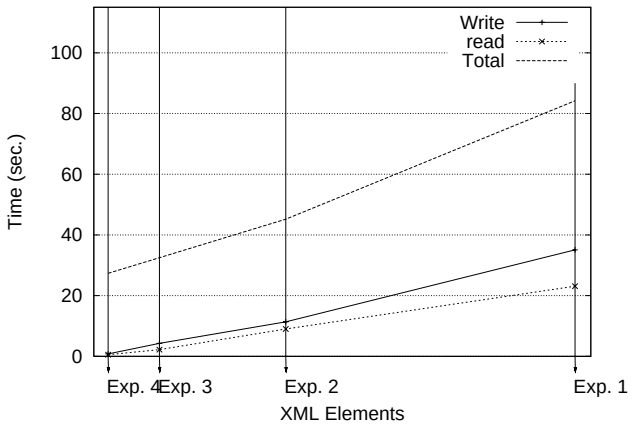

Figure 4. File-based Channel: Relationship among Write Phase, Read Phase and Overall Execution Time

of XML elements, 11 sec. are negligible w.r.t. the rest of the application execution time.

Furthermore, observe that with a smaller number of selected XML elements (as in the case of experiment *4.*), the cost of this coupling technique is negligible in general. Consequently, any kind of algorithm and application can be effectively coupled to XML parsers adopting this loosely coupling strategy.

As a practical example, we considered the implementation of the well known classifier C4.5 [22], in the Weka suite [6]. We built an XML document containing a training set with 1700000 samples; each sample is described by 5 attributes (including the class attribute) and the average number of Bytes necessary for each sample is 83 Bytes; the dimension of the XML document containing the training set is 132 MByte. The overall execution time is 347 seconds, 19 of which necessary to parse the XML document (about 12 seconds) and transfer the XML object descriptions over the channel (about 7 seconds). As the reader can notice, 19 seconds are only the 5.5% of the overall execution time, where the overhead caused by the shared vectors channel is only the 2% of the overall execution time. Note that the algorithm has been coupled with the *Reader Interface* (see Section 4) by specializing the loading class provided by the tool and by overriding its methods, without changing the algorithm.

- *Using the File-based Channel.* Finally, consider the File-based channel. It is clear that it is not suitable for one-scan algorithms and applications. However, it is good for multi-scan algorithms and applications. In fact, Figure 5 shows that, w.r.t. the Shared Vectors channel, the File-based channel becomes faster after the third scan (while they are comparable for two scans): this is due to the fact that only reading time is added by each scan, while in the Shared Vectors channel the entire process (scanning and data exchange) must be repeated for each scan.

  Furthermore, in case of filtering, notice that the File-based channel becomes advantageous from the second scan: in fact, the time needed to read the temporary file is much smaller than the parsing time.

- *When to Use Loosely Coupling.* To conclude, observe that loosely coupling is a good solution for algorithms and applications that need to work with XML and other formats, as well as for algorithms and applications not originally developed to work with XML data sources. In effect, we can observe that algorithms like the ones shown in [10] and [25] do not take advantage of loosely coupling, since they have to scan several times the full document and are specifically designed to work with XML data sources.

  For example, [10] presents an algorithm that discovers frequent tree fragments, thus it is an XML-specific algorithm; generally, these algorithms should be developed from scratch by tightly coupling them to an XML parser.

  In contrast, several algorithms exist that can be improved by adopting loosely coupling, even though they were designed to work on source XML documents. They are the algorithms (such as [12]) that translate large documents into main memory DOM representations, and scan it several times. We showed that XML documents such as the DBLP XML Records cannot be dealt with in main memory. Thus, the only way to make these algorithms effective with such large documents without rewriting them is to adopt a loosely coupling technique.

6. Conclusions

In this paper, we considered the problem of identifying a technique for loosely coupling Java algorithms with SAX XML Parsers. The work is motivated by the need for coupling existing algorithms and applications, in origin not developed to work with XML documents, to event-based XML Parsers. We chose to consider Java algorithms and applications since they are knowing a wide diffusion in a large number of application domains.

The paper shows that loosely coupling is possible and effective. In particular, two coupling techniques, the File-based channel and the Shared Vectors channel, are suitable and can be applied in practice.

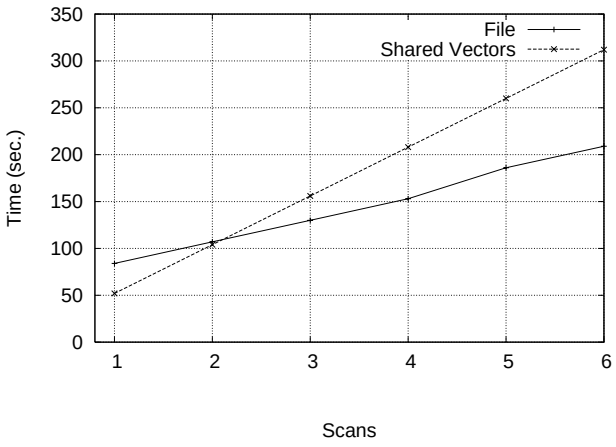

Figure 5. Execution Times

The study reported in this paper is only the first step of the research work we plan to do toward a better comprehension of problems concerned with efficient coupling of algorithms and XML Parsers. In fact, in the future, we plan to further investigate the problem. In particular, a first step to do is to consider the problem of loosely coupling C and C++ algorithms. Furthermore, we plan to study the effectiveness of binary representations for XML documents, in order to understand if these representations are really suitable to improve performance in contexts that require fast execution times, such as the development of web services on top of existing information systems.

References

[1] *Common warehouse metamodel (CWM) specification.* OMG Object Management Group, http://www.omg.org/cwm.

[2] *CRoss-Industry Standard Process for Data Mining*. CRISP-DM Special Interest Group, http://www.crisp-dm.org.

[3] *Database and Logic Programming Bibliography: XML Records*. University of Trier - Germany, http://dblp.uni-trier.de/xml/.

[4] *Predictive Modeling Mark-up Language (PMML)*. DMG Data Mining Group, http://www.dmg.org/.

[5] Soap 1.2 specifications. Technical report, World Wide Web Consortium, http://www.w3.org/.

[6] *Weka Machine Learning Project*. The University of Waikato, http://www.cs.waikato.ac.nz/ ml/index.html.

[7] *XERCES 2 Java XML Parser*. The Apache Software Foundation, http://xerces.apache.org/xerces2-j/.

[8] Xml path language (xpath), version 1.0. Technical report, World Wide Web Consortium, http://www.w3.org/TR/1999/REC-xpath-19991116, 11 1999.

[9] F. Alcamo, F. Domenichini, and F. Turini. An xml based environment in support of the overall kdd process. In *In Proc. of the International Conference on Flexible Query Answering Systems*, Warsaw (Poland), 2000.

[10] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *the Second SIAM International Conference on Data Mining*, Arlington, VA (USA), 2002.

[11] J. Boulicaut, M. Klemettinen, and H. Mannila. Modeling kdd processes within the inductive database framework. In *Proceedings of DaWaK 1999 Intl. Conference on Data Warehousing and Knowledge Discovery*, Florence, Italy, September 1999.

[12] D. Braga, A. Campi, S. Ceri, M. Klemettinen, and P. Lanzi. A tool for extracting xml association rules. In *14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2002)*, pages 57–, Washington, DC (USA), 2002.

[13] M. Champion. How much pain for xml's gain? In *XML Conference and Exposition*, Washington, DC (USA), 2004.

[14] K. Chiu and W. Lu. A compiler-based approach to schema-specific xml parsing. In *the WWW-2004 First International Workshop on High Performance XML Processing*, New York, NY (USA), 2004.

[15] Q. Ding, K. Ricords, and J. Lumpkin. Deriving general association rules from xml data. In *Proc. ACIS Fourth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03)*, Lbeck (Germany), 2003.

[16] M. Harren, M. Raghavachari, O. Shmueli, M. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. Xj : Facilitating xml processing in java. In *the 14th World Wide Web Conference WWW-2005*, Chiba (Japan), 2005.

[17] M. Leventhal. Is now the time for binary xml? report on current w3c activity. In *XML Conference and Exposition*, Washington, DC (USA), 2004.

[18] J. Lunteren, J. Bostian, B. Carey, T. Engbersen, and C. Larsson. Xml accelerator engine. In *the WWW-2004 First International Workshop on High Performance XML Processing*, New York, NY (USA), 2004.

[19] M. Mehta, R. Agrawal, and J. Rissanen. Sliq: A fast scalable classifier for data mining. In *Intl. Conf. on Extending Database Technology*, Avignon, France, March 1996.

[20] R. Meo and G. Psaila. Toward xml-based knowledge discovery systems. In *IEEE International Conference on Data Mining (ICDM 2002)*, pages 665–668, Maebashi City (Japan), 2002.

[21] M. Nicola and J. John. Xml parsing: a threat to database performance. In *the 2003 ACM CIKM International Conference on Information and Knowledge Management*, pages 175–178, New Orleans, Louisiana (USA), 2003.

[22] R. Quinlan. *C4.5 Programs for Machine Learning*. Morgan Kauffmann, Los Altos, California, 1993.

[23] T. Takase, H. Miyashita, T. Suzumura, and M. Tatsubori. An adaptive, fast, and safe xml parser based on byte sequences memorization. In *the 14th World Wide Web Conference WWW-2005*, Chiba (Japan), 2005.

[24] H. Yu, J. Yang, and J. Han. Classifying large data sets using svms with hierarchical clusters. In *KDD*, pages 306–315, 2003.

[25] M. J. Zaki and C. Aggarwal. Xrules: An effective structural classifier of xml data. In *Proc. 2003 ACM SIGKDD Conference*, Washington DC (USA), 2003.