High Performance XSL-FO Rendering for Variable Data Printing

Fabio Giannetti
HP Laboratories
Bristol - UK
fabio.giannetti@hp.com

Luiz Gustavo Fernandes
PPGCC - PUCRS
Porto Alegre - Brazil
gustavo@inf.pucrs.br

Rogerio Timmers
HP Brazil R&D
Porto Alegre - Brazil
rogerio.timmers@hp.com

ABSTRACT

High volume print jobs are getting more common due to the growing demand for personalized documents. In this context, Variable Data Printing (VDP) has become a useful tool for marketers who need to customize messages for each customer in promotion materials or marketing campaigns. VDP allows the creation of documents based on a template with variable and static portions. The rendering engine must be capable of transforming the variable portion into a resulting composed format, or PDL (Page Description Language) such as PDF, PS or SVG. The amount of variable content in a document is dependant on the publication layout. In addition, the features and the amount of the content to be rendered may vary according to the data loaded from the database. Therefore, the rendering process is invoked repeatedly and it can quickly become a bottleneck, especially in a production environment, compromising the entire document generation. In this scenario, high performance techniques appear to be an interesting alternative to increase the rendering phase throughput. This paper introduces a portable and scalable parallel solution for the Apache's rendering tool FOP (Formatting Objects Processor) which is used to render variable content expressed in XSL-FO (eXtensible Stylesheet Language-Formatting Objects). XSL-FO is extracted from a print job expressed in PPML (Personalized Print Markup Language), which is, in turn, obtained by the merging variable data in a template. The VDP Template is expressed using PPML/T (Personalized Print Markup Language Template).

Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation—*Markup Languages*; I.7.4 [**Document and Text Processing**]: Electronic Publishing

General Terms

Algorithms, Documentation, High Performance

SAC'06 April 23-27, 2006, Dijon, France

Keywords

VDP, PRL, PPML, PPML/T, XSL-FO, FOP, thread-safe.

1. INTRODUCTION

Personalized document creation is a common practice within the digital networked world. The automated, customized document assembly and transformation have become necessary processes to fulfill the demand. Typically, personalized documents contain areas that are common among a set of documents, and therefore static, as well as customized areas, the variable ones. In traditional variable information [9], document authoring tools allow designers to define a template on which to base a set of documents. The designer also defines empty areas (fixed sized), where the variable data will be put on. This way, the common layout is the same for all documents and although it can have variable data, it cannot respond to dynamic properties such as resizing of the fixed size of variable data.

These limitations have triggered research efforts to automate the process of creating personalized documents. Documents can be authored as templates and the production can be automated maintaining an high level of composition quality. It is the focus of this paper to explore how the generation of such documents is achieved in a production environment. Print Shops require a predictable, efficient and high quality industrial process to print and finish documents. It has been proven that using XSL-FO as the format for un-composed portions of the document makes it possible to merge the variable data later in the process, even during the printing process itself. This approach is clearly advantageous since it does not require having the variable data at design time and also enables the transmission of the documents as templates and variable data instead of the fully expanded document set.

The remainder of this paper is organized as follows: Section 2 presents the motivations for this work, Section 3 introduces the research context focusing on the definition of the used markup languages, XSL-FO and PPML as well as the XSL-FO rendering engine Apache's FOP, Section 4 describes the high performance approach to improve the rendering procedure, Section 5 introduces the testing environment, Section 6 presents an analysis of experimental results for a case study and finally some concluding remarks and future works are discussed in Section 7.

2. MOTIVATION

The aim of this research is to explore and indicate a scalable and modular solution to execute variable data composition using parallel rendering engines.

Most of the digital publishing production environments use digital presses in parallel to maximize the balance, as well as the overall document production (jobs). In such environment all the activities related to the document preparation need to be executed in a constrained time slot, since jobs are completed in a sequential order on multiple presses. In the variable data printing case, on top of the existing preprinting steps there is the need to merge the variable data into the template and rendering the un-composed document portions.

The rendering process is usually quite expensive and in case of thousands of documents can become the bottleneck. Modern digital presses also are capable of printing at an engine speed (of about 1 second per page). This rate is the minimum required to maintain the digital press working at full speed. When digital presses are used in parallel, the overall engine speed is multiplied by the number of presses, thus, it can become increasingly difficult to feed all the presses at the combined engine speed to make the most of their printing speed.

When the rendering process is centralized at a single processor, a break point is likely to happen, since the engine speed of the presses in parallel is exceeding the rendering process speed creating a bottleneck. Similarly to the concept of using presses in parallel to achieve better performance and quickly consume jobs, our aim is to develop a proposal for parallelizing the XSL-FO rendering engines. Results show that the system developed can match the presses combined engine speed at the rendering stage. We have also considered that it is necessary to provide the best compromise between the number of processors and overall speed. Therefore, the concept of "unit" has emerged. A unit is, the minimum set of parallel processors that, it is provided accordingly to the configuration of the presses, to maintain the combined engine speed.

We have explored the two following situations:

1. Print Shop A has two digital presses consuming jobs in parallel: one "unit" containing a certain number of parallel rendering engines is used to match the combined engine speed;

2. Print Shop B has four digital presses consuming jobs in parallel: two "units" containing the same number of parallel rendering engine is used to match the approximately doubled engine speed.

The results in this paper shows a modular and scalable solution for variable data printing with late binding and rendering (composition) at printing time.

3. BACKGROUND

The combination of PPML and XSL-FO has been chosen to represent document templates with high degree of flexibility, reusability and printing optimization. The synergy achieved by this combination assures the expressibility of invariant portions of the template as re-usable objects and variable parts as XSL-FO fragments. After the variable data is merged into the template, various document instances are formed. The final step is to compose or render the XSL-FO parts into a Page Description Language (PDL), enabling the digital press to rasterize the page. The rendering process is carried out by Apache's FOP [3]. Before describing the processing environment, it is relevant to highlight the most important aspects of these XML formats: PPML and XSL-FO.

PPML [2] is a standard language for digital print built from XML (eXtensible Markup Language) developed by PODi (Print on Demand Initiative) [8] which is a consortium of leading companies in digital printing. PPML has been designed to improve the rasterizing process for the content in documents using traditional printing languages. PPML, in fact, introduces the reusable content method where, contents which are used on many pages, can be sent to the printer once, and accessed as many times as needed from the printer's memory. This allows high graphical content to be rasterized once as well, and to be accessed by sending layout instructions instead of re-sending the whole graphic every time it is printed. Each reusable object in PPML is called resource. In order to guarantee all the resources are available and the digital press can retrieve them, PPML allows external referencing. It is also common practice to give responsibility to express all the necessary resources to successfully complete the print job to the job ticketing language.

Usually, the digital press can access to the required resources directly from the local mass storage unit or from the local LAN, as a way of maintaining the performance rate. PPML is a hierarchical language that contains documents, pages and objects. The contained objects are classified as re-usable or as disposable. PPML also introduces the concept of scoping, for the re-usable objects, so the PPML producer can instruct the PPML consumer about the lifespan of a particular object. This approach is very powerful and efficient and can optimize the press requirement of caching and pre-rasterizing objects that are re-used all across the job and/or only in a particular page. Unfortunately, PPML lacks of strategies and constructs to give different importance to the objects since the rasterizing process can have significant differences and impact on the overall time to produce the document. Some work has been already presented [1, 5] in order to address this issue, which currently remains open and is outside the scope of this paper.

The variable data is merged inside the PPML Object element and it is formatted using XSL-FO. The containing element, expressed as PPML, is named "copy-hole", which is a defined area in the PPML that can contain the variable data expressed in XSL-FO, or a non variable content such as images. XSL-FO (also abbreviated as FO) is a W3C [11] standard introduced to format XML content for paginated media. XSL-FO ideally works in conjunction with XSL-T (eXtensible Stylesheet Language - Transformations) [12] to map XML content into formatted and ready to be mapped into a pagination model. When the XSL-FO is completed with both the pagination model and the formatted content, the XSL-FO rendering engine, executes the composition step to lay out the content inside the pages and obtain the final composed document. The composition is a complex step and requires typesetting capabilities, as well as layout expertise and resolution. The XSL-FO rendering engine used in our solution is FOP.

FOP is one of the most common processor in the market not only because it is an open source application, but

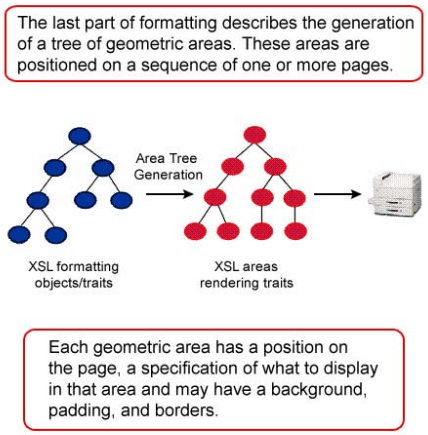


Figure 1: Phases of the rendering process

also because it provides a variety of output formats and it is flexible and easily extendable. It is a Java application that reads formatting objects and renders to different output formats such as PDF, plain text, PostScript, SVG which is the focus of the rendering results obtained in this paper, among others. Figure 1 illustrates the various phases of the rendering process. This process is typically composed by three distinctive steps:

1. generation of a formatting object tree and properties (or traits) resolution;
2. generation of an area tree representing the laid out document composed by a hierarchy of rectangular having as leafs text elements or images;
3. converting or mapping the area tree to the output format.

The advantages of this approach are related to the complete independence between the XSL-FO representation of the document and the internal area tree construction. Using this approach makes it possible to map the area tree to a different set of PDLs.

4. HIGH PERFORMANCE APPROACH

This section describes the main features of the high performance approach we propose for XSL-FO rendering for Variable Data Printing. It is relevant to notice that the approach described in this paper considers FOP as a black box and it is not aiming at parallelizing the various rendering steps previously described. In the future work section we will explore the possibilities of acting inside FOP's code to enhance the speed of the rendering process leveraging, where possible, the common processing parts from the document specific ones.

In the sequential FOP version presented early in this paper, the output document generated after the rendering process is composed by the same PPML structure but the XSL-FOs, which are replaced by its corresponding part rendered. In this version, the part of the document that is not rendered (static part) is automatically copied to the output PPML when the document is being parsed up to the moment a XSL-FO is detected. Therefore, when a XSL-FO is located it is sent out to the FOP to be rendered, and the rendered content saved back at the same part of the document it was before in the output PPML. It happens one after another until the entire document is parsed.

Since the rendering process is called repeatedly, the main idea behind the high performance approach is to allow the execution in parallel of several instances of FOP tool. However, in the parallel version there are three main problems. The first one is that several FOs are being rendered in parallel and need to be written in the output file in the same order they were parsed from the original document. The second one, is that if the FOs are sent to a FOP module while it is rendering a XSL-FO object, the sender module will have to wait for the FOP to finish the current render procedure so the communication can be completed. That way, the overall time will be compromised. The third problem is that if the module that is parsing the input document, extracting FOs and sending them out to the FOPs is also writing the output file (that takes a considering amount of time), it will be overloaded and probably will not be able to deal with communication to receive the rendered objects.

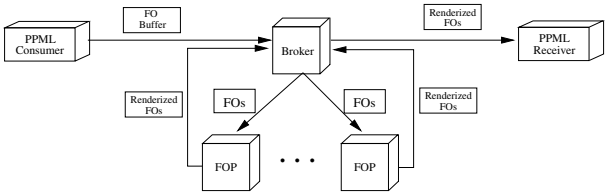


Figure 2: Proposed architecture

In order to solve the first problem, an ID is assigned to each FO rendered, allowing it to be identified and saved in the correct position. For solving the second problem, a new module - called Broker - was created. Its function is queueing the FOs and sending them out to the first idle FOP. Finally for the third problem, we have split up the parse module in two parts making a module responsible for each part of the process. The PPML Consumer is the module that will parse out the document and extract the FOs to send them out to the FOP modules. On the other hand, the PPML Receiver is the module that will receive the rendered FOs and write them at the output file. The PPML Receiver also will have to get the static parts of the PPML to re-write them in the output file. So, this module has two threads. The first, parse the document extracting only the static parts and the second receives the rendered FOs and write the output file. Figure 2 represents our proposed architecture.

The Broker function, as said before, is to receive and queue FOs to be rendered. These FOs are sent to the FOP components requesting for work. The FOP module renders the XSL-FOs and when finished sent it to the Broker, in such a way that the Broker knows when the FOP can receive another FO to be rendered. In order to gain performance, this module has been divided in two threads:

- *receiver*: responsible for receiving and queueing FOs;
- *sender*: check whether there is some FO waiting in

the queue to be sent out to the first idle FOP module. Also transmits rendered FOs to PPML Receiver.

Finally, in order to maximize the parallel version performance, the communication issue must be treated. We have realized that if the FOPs receive a single FO per time, renders it and send it back to the Broker, the communication cost is not compensated by the time of rendering just one FO. This problem can be minimized through the use of communication buffers. Thus, the PPML Consumer send buffers filled with work (FOs to be rendered) to the Broker. It then splits these buffers in smaller ones that will be queued and sent to each FOP. The size of these buffers is critical: they should not be too large, because the communication cost does not compensate the time of rendering FOs in parallel, neither too small, because there will be too much communication.

A final comment on high performance implementation of the XSL-FO rendering process is related to the usage of threads. Programming concurrent systems using threads introduces issues related to the simultaneous access of shared resources (e.g., output stream). A system is called thread-safe if it is safe to invoke its methods from multiple threads, even in parallel. Non-thread-safe objects may behave unpredictably and generate unexpected results, corrupt internal data structures, etc. Thread-safety is typically achieved in Java with (both employed in our implementation):

1. use of synchronized statements (or synchronized methods);
2. immutability of encapsulated data (i.e. it is not possible to modify any field after you have created the object).

5. TESTING ENVIRONMENT

This section introduces the requirements for a platform to support a portable and scalable high performance implementation. The tests performed for all the solutions presented in this paper have been done in the same environment using the same input data according to this section.

5.1 Platform

The first issue a high performance application designer must deal with is to choose between multiprocessor or multicomputer architectures. Multiprocessor machines use a global memory access scheme, and usually need an expensive interconnection bus between processors and memory (e.g., crossbar). Nowadays, these machines are loosing space for the multicomputer platforms such as clusters or computational grids. These machines present distributed memory scheme and, in the case of clusters, are connected through a dedicated, fast network. Developing programs for the two platforms is quite different. The first one is based on a shared memory programming paradigm and the second one is typically based on message passing paradigm.

Programming for distributed memory platforms is more complex because each processor of the architecture has a local memory and cannot directly access others processors memories. In this scenario, the application must be divided in modules, also called processes, which do not share the same address space with each other. Thus, processes cannot exchange data through shared variables. The alternative is to provide a set of communication primitives which are based on two main functionalities: sending and receiving data through/from an interconnection network. Despite of the greater complexity of the message passing programming paradigm, it presents the significant advantage of a high degree of portability. Such kind of programs can be executed over shared-memory platforms without any changes considering that an unavoidable loss of efficiency can be accepted. Shared memory programs, on the other hand, have a lower degree of portability since they cannot be carried out over distributed memory platforms. This only happens through a complete conversion of the program to the message passing paradigm.

Considering that portability and scalability are two desirable features for high performance implementations, we decided to adopt the Java programming language in our high performance implementation. Java is not frequently used for high performance applications [4, 6] for two reasons: it is an interpreted language and it is based on a virtual environment (Java Virtual Machine - JVM), which allows portability. These two features are responsible for a computational overhead, which most of the time is considered too significant by high performance applications designers. However, in our implementation, portability and compatibility with different operational systems are crucial. That, plus unique features like multi-thread primitives, justify the adoption of Java.

We used the Java Standard Development Kit (J2SDK, version 1.4.2) plus the standard Message Passing Interface (MPI) [10] to provide communication among processes. More specifically, we chose the mpich implementation (version 1.2.6) along with mpiJava [7] (version 1.2.5) which is an object-oriented Java interface to the standard MPI. The experiments were carried out over processors running Linux (Slackware distribution, kernel 2.4.29), as this is the standard hardware configuration available to us. However, it is important to mention that mpiJava is also compatible with Windows operating system platforms, assuring the implementation portability. The target hardware platform is a cluster composed by 12 Pentium III 1Ghz processors with 256 MB RAM connected through 100 Mb FastEthernet.

5.2 Input Data

Documents to be rendered can have a multitude of different layouts, as well as FOs that compose a document. In this paper, we have chosen to show the behavior of our implementation using four rendering stress tests which contains a large amount of data to be rendered. The first input PPML file, which is called **Mini**, contains a job with **one thousand** documents to be rendered. Each document is composed by two pages as follows:

- Page 1: 1 copy-hole with XSL-FO composed by 4 text blocks and approximately 107 words;
- Page 2: 3 copy-holes with XSL-FO, respectively composed by 6 text blocks and approximately 130 words, 2 text blocks and approximately 43 words and 1 text block with 36 words;
- Average number of words per block: 24.3.

The resulting XSL-FO fragments contained in the PPML copy-holes to be rendered amounts to four thousands. The PPML documents are instances of the template shown in Figure 3.

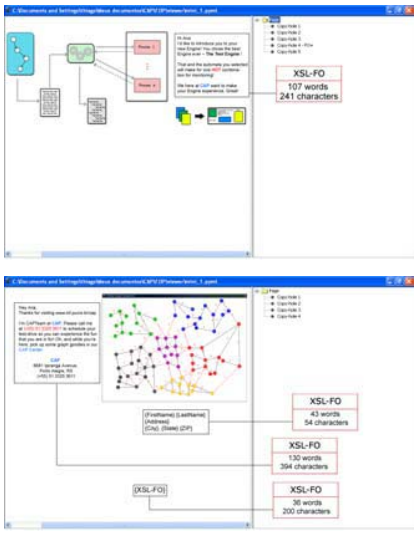


Figure 3: Document generated by the Mini XSL-FO

The second test, which is called **Cap**, has **two thousand** documents. The documents have two pages, each one as follows:

- Page 1: 3 copy-holes with XSL-FO, all of them with 1 text block, respectively with 4 words, 6 words and finally 7 words;
- Page 2: 3 copy-holes with XSL-FO, respectively with 4 text blocks and 56 words, 1 text block and 6 words and 1 text block with 2 words;
- Average number of words per block: 9.

The number of XSL-FO fragments to be rendered get to 12000. The template shown in Figure 4 was the one used to create this input file.

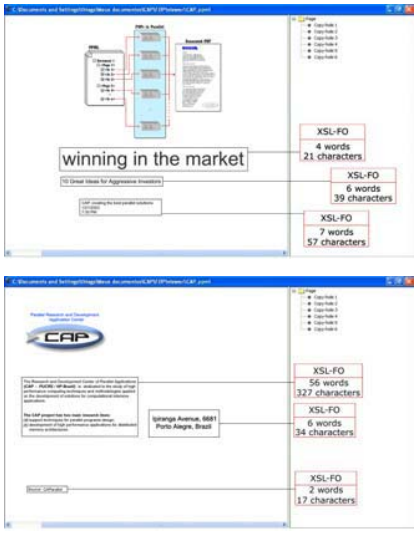


Figure 4: Document generated by the Cap XSL-FO

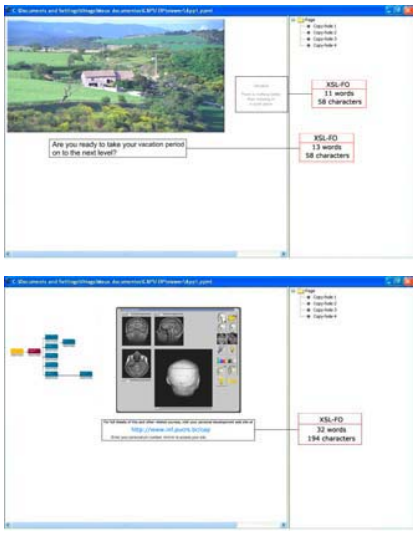


Figure 5: Document generated by the Appl XSL-FO

The third test is called **Appl**. It has a job with **a thousand** documents. Each document contains three pages as follows:

- Page 1: 2 copy-holes with XSL-FO, both composed by only 1 text block each, respectively with 11 words and with 13 words;
- Page 2: 1 copy-hole with XSL-FO, that contains 1 text block and 32 words;
- Average number of words per block: 18.67.

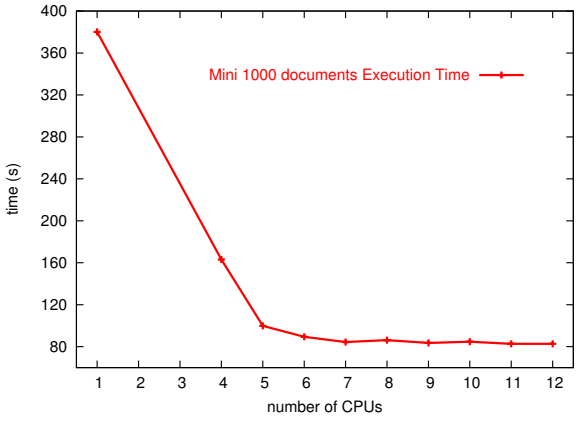
In this case, the number o XSL-FO fragments to be rendered gets to 3000. This input has been generated using the template shown in Figure 5. The last test is the same as the third one, but it has a job with **two thousand** documents. That will end up with 6000 XSL-FO fragments to be rendered. This last test was also generated by the template shown in Figure 5.

6. RESULTS

In order to verify the advantages and drawbacks of the approach described in the previous section, some experiments were carried out. This section presents the results of the high performance rendering approach for the XSL-FO document introduced in section 4. Seeking to provide the reader a comparison parameter, the sequential version of the render tool was executed over a single processor of the cluster described in section 5.1, resulting in the execution times showed in Figures 6, 7 and 8 for one CPU. Each execution time presented in this section is obtained after a mean of five executions, discarding the highest and lowest times.

The first experiment we have carried out was using the **Mini** input job, which contains 1000 documents. This input job is the smallest one, but it presents a high density in terms of the numbers of words inside each text-block. In this case, the best execution time was 82.6377 seconds (using 12 CPUs), but this configuration presents a low efficiency (38.33%). In fact, from 7 CPUs up to 12 the gain

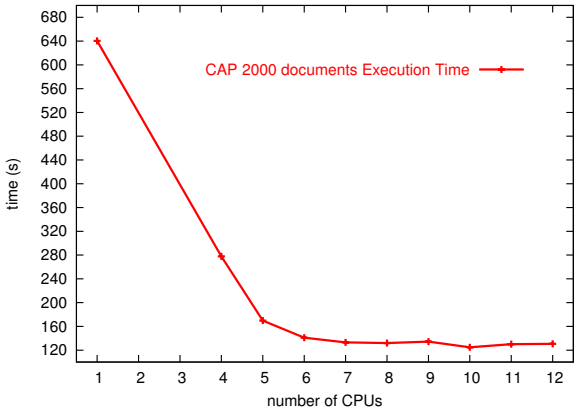
in terms of execution time is not significant, indicating that the whole system could not take advantage of more than 4 FOP modules running in parallel. Figure 6 shows the results for this test case.



	number of CPUs - Mini 1000									
	1	4	5	6	7	8	9	10	11	12
T	380.0776	163.0715	99.8171	89.4505	84.3465	86.2389	83.5292	84.6751	82.7083	82.6377
E	100.00	58.27	76.15	70.82	64.37	55.09	50.56	44.89	41.78	38.33

Figure 6: Results for the Mini 1000 input job

In our second experiment, we used the **Cap** input job. This is the most dense input job in terms of elements to be rendered. The sequential time in this case was 640.3371 seconds to render 2000 documents. The best execution time (124.8848 seconds) was achieved with 10 CPUs, but again the gain from 7 CPUs up to 12 is not significant in terms of execution time. In Figure 7 the results for this experiment are shown.

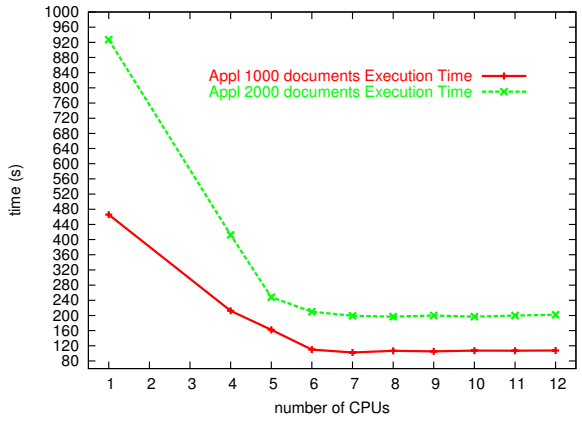


	number of CPUs - Cap 2000									
	1	4	5	6	7	8	9	10	11	12
T	640.3371	277.8193	169.6794	141.0473	133.0431	132.0088	134.3758	124.8848	130.1075	130.5597
E	100.00	57.62	75.48	75.67	68.76	60.63	52.95	51.27	44.74	40.87

Figure 7: Results for the Cap 2000 input job

For the last experiment, we used the same template only changing the number of documents contained within the input job (**Appl** with 1000 and 2000 documents). This procedure allowed us to analyze the scalability of the proposed parallel solution when the workload is increased. The experiment with 1000 documents presented the best execution time with 7 CPUs (102.4167 seconds). On the other hand, for 2000 documents, the faster execution time was obtained with 10 CPUs (196.9459 seconds). The results show that our parallel solution have scaled well, when the amount of documents to be rendered doubles. The results are shown if Figure 8.

Comparing the three test cases previously discussed, a common behavior was detected: running the application with more than 7 CPUs does not seem to present major improvements on the execution time that would justify the use of more processing units. We believe that the reason for that is that the Broker module reaches its limits when dealing with four FOP modules. If the number of FOPs is greater than four, the Broker module cannot handle efficiently the distribution of FOs among the FOPs, becoming the system's bottleneck.



	number of CPUs - Appl 1000									
	1	4	5	6	7	8	9	10	11	12
T	465.6895	212.2404	162.2920	109.9721	102.4167	106.7142	105.2449	107.3861	107.0634	107.8098
E	100.00	54.85	57.39	70.58	64.96	54.55	49.16	43.37	39.54	36.00
	number of CPUs - Appl 2000									
	1	4	5	6	7	8	9	10	11	12
T	927.0293	412.5711	248.1964	209.7896	199.2431	199.8891	199.8978	196.9459	199.9694	201.9489
E	100.00	56.17	74.70	73.65	66.47	58.86	51.53	47.07	42.14	38.25

Figure 8: Results for the Appl input job (1000 and 2000)

7. CONCLUSION AND FUTURE WORK

The results presented in this paper indicate that it is possible to achieve better results in rendering XSL-FO documents using high performance computations techniques. In this first implementation, we have used threads and the message passing programming paradigm to decrease the execution time to render three different jobs containing several thousands documents.

Although the gain of performance could be considered satisfactory, the main contribution of this work was to indicate the best configuration among the tested ones: from 4 to 12 parallel processors. This has paved the way to indicate that a multiple Broker solution using four FOP modules per Broker is the optimum solution (see Figure 9). This configuration tackles the saturation of the Broker module problem, avoiding it to be too busy receiving rendered FOs and not be able to send new FOs to idle FOP modules.

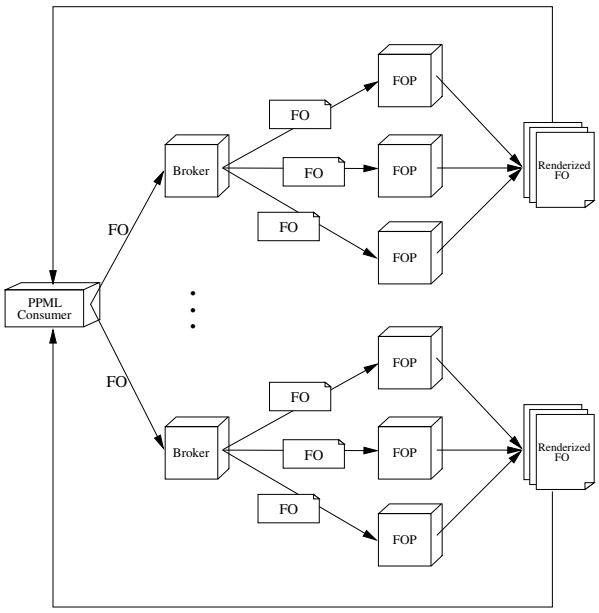


Figure 9: Illustrative example of the Multi-broker architecture

Analyze the PPML template in order to identify the possible number of FOs and their distribution in order to set the number of Broker modules to be loaded in the available machines, represents a possible optimization step that we plan to investigate. The incipient idea behind that is that the Brokers set-up, among the available set of processors, could be initialized as result of: either an user decision or an automatic decision of the application based on a previous analysis of the PPML Template. The second alternative is more robust, but its implementation is much more complex due to the difficulties to handle dynamic reconfigurations in distributed memory platforms.

Results so far have also generated another potential line of improvement. Depending on the features of the documents to be rendered, the number of transmissions between modules changes drastically. For example, it is better to send a group of FOs to the rendering modules than one FO per time, if the FOs are small. In this case, the communication overhead may be higher than the time needed for rendering a single FO. This situation can decrease the speed of the high performance version. An alternative would be to send a group of several small FOs for each FOP module. Another case is when the documents to be rendered have only large FOs, in this case it is better to send one FO at the time, because groups will increase the amount of time taken for the transmission through the network. Considering this scenario, an alternative is to set a parameter to the application informing the mean size of the FOs of a PPML document, fixing the size of communication buffers based on this information until the limit supported by the network.

Considering all cases in this line of potential research, our future work, is to implement a smarter parallel version in a way that it can automatically identify different situations, as exposed above, aiming at the best possible performance. This implies the development of a decision making environment with the capabilities of choosing the number of Brokers and the size for the communication buffers.

8. ACKNOWLEDGEMENTS

This work was developed in collaboration with HP Brazil R&D. Authors would like to thank Lucas J. Baldo, Gustavo S. Serra, Márcio R. Farias Soares and Pedro A. Madeira de Campos Velho from the CAP (Parallel Applications Research and Development Center) team for their technical support. All experiments were carried out over the CPAD (Research Center in High Performance Computing) infrastructure.

9. ADDITIONAL AUTHORS

Additional authors: Thiago Nunes (CAP - PUCRS, email: `tnunes@inf.pucrs.br`), Mateus Raeder (CAP - PUCRS, email: `mraeder@inf.pucrs.br`) and Márcio Castro (CAP - PUCRS, email: `mcastro@inf.pucrs.br`)

10. REFERENCES

[1] D. D. Bosschere. Book ticket files & imposition templates for variable data printing fundamentals for PPML. In *Proceedings of the XML Europe 2000*, Paris, France, 2000. International Digital Enterprise Alliance.

[2] P. Davis and D. deBronkart. PPML (Personalized Print Markup Language). In *Proceedings of the XML Europe 2000*, Paris, France, 2000. International Digital Enterprise Alliance.

[3] FOP. Formatting Objects Processor. Extracted from http://xml.apache.org/fop/ at May 13th, 2005.

[4] V. Getov, S. F. Hummel, and S. Mintchev. High-performance parallel programming in Java: exploiting native libraries. *Concurrency: Practice and Experience*, 10(11–13):863–872, 1998.

[5] F. R. Meneguzzi, L. L. Meirelles, F. T. M. Mano, J. B. de S. Oliveira, and A. C. B. da Silva. Strategies for document optimization in digital publishing. In *ACM Symposium on Document Engineering*, pages 163–170, Milwaukee, USA, 2004. ACM Press.

[6] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, M. Snir, and R. Lawrence. Java programming for high performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.

[7] mpiJava. The mpiJava Home Page. Extracted from http://www.hpjava.org/mpiJava.html at May 13th, 2005.

[8] PODi. Print on Demand Initiative. Extracted from http://www.podi.org/ at May 13th, 2005.

[9] L. Purvis, S. Harrington, B. O'Sullivan, and E. C. Freuder. Creating personalized documents: an optimization approach. In *ACM Symposium on Document Engineering*, pages 68–77, Grenoble, France, 2003. ACM Press.

[10] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: the complete reference*. MIT Press, 1996.

[11] W3C. The World Wide Web Consortium. Extracted from http://www.w3.org/ at May 13th, 2005.

[12] XSL-T. XSL-Transformations. Extracted from http://www.w3.org/TR/1999/REC-xslt-19991116, section References, at May 13th, 2005.