Paralelização do Processo de Renderização de Documentos XSL-FO

- Tópicos Especiais em Processamento Paralelo e Distribuído I -

Mateus Raeder, Thiago Tasca Nunes
{mraeder, tnunes}@inf.pucrs.br

Resumo

A criação de documentos personalizados se tornou uma prática comum na área de impressão digital. Processos automatizados de criação e transformação destes documentos tornaram-se necessários para atender a crescente demanda do mercado. Linguagens foram desenvolvidas com a finalidade de otimizar o processo de criação de documentos personalizados e descrever conteúdo variável dentro de um documento. Neste contexto, o custo de computação se tornou muito grande, gerando uma necessidade de soluções de alto desempenho.

Em vista disso, a programação paralela se apresenta como uma alternativa atrativa para atender à necessidade existente. Este trabalho apresenta o estudo da paralelização da aplicação FO+Processor que tem como finalidade renderizar objetos descritos em determinada linguagem, no caso deste trabalho XSL-FOs, para que possam ser visualizados.

1. Introdução

Devido à crescente demanda por documentos personalizados, a impressão de grandes volumes de dados está ficando cada vez mais comum. Neste contexto, Variable Data Printing (VDP) tem se tornado uma ferramenta útil para empresas que precisam personalizar mensagens para cada cliente na promoção de materiais ou campanhas de marketing.

VDP permite a criação de documentos baseados em template com partes estáticas e variáveis. A engine de renderização deve ser capaz de transformar a porção variável num formato composto ou PDL (Page Description Language), como, por exemplo, PDF (Portable Document Format), PS (Post Script) ou SVG (Scalable Vector Graphics) para que este documento possa ser impresso. A quantidade de conteúdo a ser renderizado pode variar de acordo com os dados carregados do banco de dados. Por isso, o processo de renderização é invocado repetidamente e pode tornar-se rapidamente um gargalo.

A maioria dos ambientes que produzem publicação digital utiliza impressoras digitais em paralelo para aumentar o balanceamento, assim como a produção total de documentos.

Nestes ambientes, todas as atividades relacionadas à preparação de documentos devem ser executadas em um tempo limitado, já que os trabalhos são terminados em ordem seqüencial para serem impressos em múltiplas impressoras.

Neste cenário, técnicas de alto desempenho aparentam ser uma alternativa interessante para aumentar a vazão da fase de renderização, aumentando assim a produtividade das impressoras (que chegam a imprimir cerca de uma página por segundo).

2. Objetivos

O Objetivo deste trabalho é o de estudar e apresentar uma solução portável e modular para executar uma composição de dados variáveis utilizando engines de renderização em paralelo.

Para tanto, estudou-se o funcionamento da ferramenta FO+Processor com o intuito de identificar características relevantes que surgissem como possíveis gargalos, possibilitando a proposta de uma versão paralela para a aplicação em questão.

3. Processamento Paralelo

Desde a criação dos primeiros computadores, existe uma preocupação em torná-los mais eficientes, com o propósito de conseguir resolver problemas computacionais no menor tempo possível.

O mercado apresenta cada vez mais aplicações que necessitam de respostas rápidas em diversas áreas, como a da meteorologia (previsão do tempo), física (simulações com alta carga computacional), bioinformática (data-mining de cadeia de proteínas), computação gráfica (representação de volumes tridimensionais), etc.

O processamento paralelo surge, então, com o intuito de melhorar o desempenho de tais aplicações, proporcionando um custo relativamente pequeno para a aquisição de um ambiente paralelo com elevado poder computacional. Entretanto, novos desafios são incorporados à computação paralela. Alguns deles são: complexidade na implementação, dependência entre a implementação e o tipo de máquina para a qual se está desenvolvendo, cuidado em minimizar a comunicação exagerada e prover balanceamento de carga de forma adequada entre os recursos disponíveis. Para uma melhor compreensão sobre como superar tais desafios, fatores ligados ao paralelismo como um todo devem ser compreendidos.

3.1 Ferramentas para Programação Paralela

Para o desenvolvimento de programas paralelos, existem algumas ferramentas auxiliares. Estas ferramentas implementam os modelos de computação paralela: SPMD (Single Program Multiple Data) ou MPMD (Multiple

Program Multiple Data). O primeiro modelo pressupõe que o mesmo programa é executado por todos os processadores da arquitetura paralela em questão, mas cada um deles trabalha com um conjunto de dados distintos. Em contraste, no segundo modelo, cada processador é responsável pela execução de um programa diferente um do outro.

Nesta seção, um pouco sobre duas destas ferramentas utilizadas nesse estudo são abordadas: threads e MPI.

3.1.1 Threads

A maioria dos sistemas é baseado em computadores com apenas um processador. Este processador executa diversas tarefas simultaneamente, isto é, vários processos compartilham a CPU, apoderando-se de determinadas fatias de tempo para a sua execução.

Um processo é "uma instância única de uma aplicação que está sendo executada" [1], ou seja, é o executável de um programa carregado em memória (aplicação). Threads podem ser consideradas como subprocessos. São fluxos de execução que rodam em uma aplicação. Um programa sem threads possui um único fluxo de execução, realizando uma tarefa por vez. Em um programa com a utilização de threads, por sua vez, tem-se a execução de mais de uma tarefa simultaneamente.

A execução de aplicações pode, então, ser dividida em múltiplos caminhos (threads) que rodam concorrentemente na memória compartilhada e compartilham os mesmos recursos do processo pai.

3.1.2 MPI

O MPI (Message Passing Interface) [2] é um padrão de troca de mensagens que permite a comunicação entre os processos de uma aplicação paralela.

O padrão MPI foi desenvolvido baseando-se nas diferentes bibliotecas existentes, como PVM, por exemplo, passando a ter um papel importante no contexto de computação paralela. Este padrão permite utilizar o modelo SPMD, quando todos os processos executam o mesmo código.

O programador deve dividir quais processos devem executar quais trechos de código. Porém também permite MPMD. Algumas vantagens do MPI são eficiência, portabilidade, transparência, segurança e escalabilidade.

Um programa MPI consiste em processos autônomos, que executam o mesmo código. A comunicação é feita através de duas operações básicas: send e receive.

É um método que introduz conceitos como o rank, que se trata de uma identificação única para cada processo, sempre de forma crescente. Assim, cada processador pode executar a parte do código que lhe for designada. Além disso, o MPI oferece diversas formas de implementar a comunicação ponto-a-ponto (de forma síncrona ou assíncrona, bloqueante ou não bloqueante) e a comunicação coletiva, como mensagens em Broadcast e barreiras.

4. Renderização de Documentos

Renderização de documentos é o processo de interpretação de determinados tipos de dados que objetiva transformá-los em um conteúdo visualizável. A seguir, alguns aspectos e conceitos básicos sobre renderização de documentos serão abordados.

4.1 VDP

VDP (Variable Data Printing) se refere a ligação de engines de renderização com bancos de dados, resultando em documentos personalizados. Estes documentos contêm elementos com informação variável. Esta informação variável pode ser o nome de um cliente, o nome de um produto, figuras, etc., fornecendo ao detentor deste recurso uma poderosa maneira de atingir sua clientela. Alguns exemplos de VDP são contas de bancos, contas de telefone, catálogos de marketing personalizados, etc.

Todos os exemplos citados acima partem de um template (Figura 1) previamente criado. Cada template oferece ao usuário a oportunidade de criar layouts personalizados, e, conforme se aumenta a complexidade deste layout (por exemplo, figuras e textos com tamanhos variáveis), surge a necessidade de uma linguagem eficiente para lidar com isto.

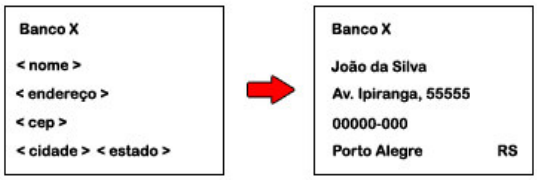


Figura 1

4.2 PPML

PPML é uma linguagem padrão para impressão digital baseada em XML (eXtensible Markup Language) [3]. Foi desenvolvida pelo PODi (Print On Demand initiative) [4], que se trata de uma associação formada por um conjunto de companhias que lideram no mercado de impressão digital. Este padrão proporciona uma forma de utilizar VDP em documentos de alta qualidade. A grande vantagem da utilização desta linguagem é que ela permite que conteúdos iguais, presentes em diferentes páginas de documentos, sejam armazenados apenas uma vez na memória da impressora e então acessados por esta quando necessário. Portanto, não existe necessidade de re-processamento por parte da impressora de conteúdos iguais, tornando mais rápido o processo de impressão.

Um arquivo PPML é composto por um ou mais conjuntos de documentos (Figura 2). Cada documento possui uma ou mais páginas, e dentro destas podem haver copy-holes contendo XSL-FOs. Copy-holes são áreas que delimitam a extensão máxima de um conteúdo (XSL-FO, imagens, etc.), todas com tamanho fixo e posição.

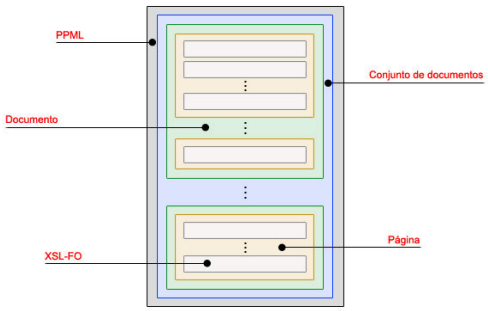


Figura 2

O documento PPML contém porções renderizáveis e porções não-renderizáveis, e caracteriza-se por apresentar primeiramente uma porção estática. As partes compostas por XSL-FO são as partes renderizáveis, e encontram-se dentro dos copy-holes. Já as partes estáticas (imagens por exemplo) não são renderizáveis.

4.3 XSL-FO

XSL-FO [5] (ou FO) é uma linguagem de formatação de objetos que proporciona aos editores a liberdade para construir documentos XML com muitos recursos visuais. É um padrão W3C [6] baseado em XML. Ele funciona bem com XSL-T (eXtensible Stylesheet Language - Transformations) [7] que mapeia o conteúdo XML para ficar formatado e dentro de uma página.

O modelo de formatação XSL-FO contém áreas de trabalho que podem ser preenchidas com texto, imagens, espaço vazio, ou outro XSL-FO aninhado. Estas áreas podem ser de quatro tipos, a saber: regiões (nível mais alto em um XSL-FO), bloco (representa níveis de bloco, como parágrafos ou item de listas), linha (um texto dentro de um bloco) ou áreas internas (partes de uma linha, como um caractere).

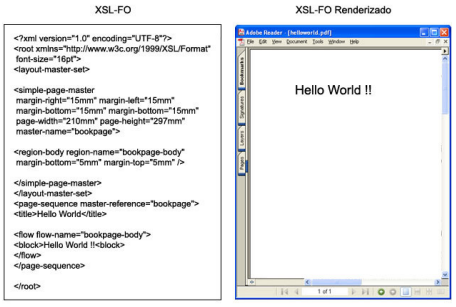


Figura 3

Um documento PPML, que serve de entrada para a renderização possui porções, que contém XSL-FO. Estas porções são as porções renderizáveis do documento, que são transformadas em conteúdo visualizável, conforme mostra o exemplo da Figura 3.

4.4 PDL

PDL (Page Description Language) é uma linguagem de comando de alto nível de abstração, utilizada para fazer com que algum dispositivo (impressora, tela do computador, etc.) imprima texto e imagens através de comandos específicos. Este tipo de linguagem otimiza o processo de impressão por diminuir a quantidade de informação a ser transmitida para o dispositivo, além de facilitar a capacidade de integração de esquemas complexos de texto e imagem. Entretanto, deve ser ressaltado que a PDL não impede a impressão ponto a ponto (pixel a pixel) no dispositivo.

O ganho de desempenho pode ser exemplificado pela possibilidade de mandar a impressora desenhar um círculo (Figura 4), ao invés de enviar ponto a ponto do mesmo ou ainda pelo fato de mandar a impressora construir uma fonte determinada partindo de suas próprias fontes, ao invés de baixar na impressora o design inteiro da fonte que será utilizada.

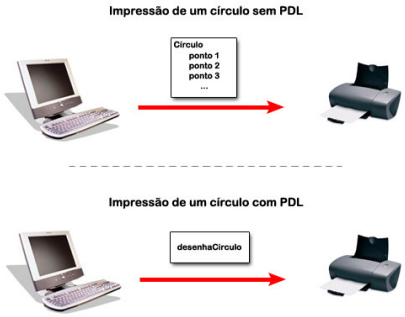


Figura 4

4.5 SVG

SVG (Scalable Vector Graphics) [8] é um padrão baseado em XML, definido através do W3C [6] com grande suporte empresarial como, por exemplo, Adobe, Apple, Canon, Corel, Ericsson, HP, entre outras. Ele é utilizado para descrever gráficos bidimensionais, suportando texto, imagens (como JPEG e PNG), formas arbitrárias de desenho, entre outras (Figura 5).

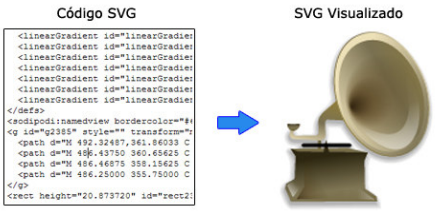


Figura 5

Este padrão foi desenvolvido, primeiramente, com o intuito de ser utilizado em páginas da internet. Devido a grande quantidade de facilidades gráficas oferecidas por

ele, sua aplicação estendeu-se a diversas outras áreas e suas versões posteriores passaram a se preocupar não apenas com o lado da internet. Entre as diversas áreas em que se pode aplicar o padrão destaca-se a área de impressão. Neste âmbito, SVG apresenta algumas vantagens como independência de resolução da impressão, além de poder ser integrado ao padrão PPML definido pelo PODi [4] para a impressão de dados variáveis.

4.6 FOP e FO+Processor

A engine de renderização utilizada neste estudo trata-se da ferramenta FOP (Formatting Objects Processor) [9], que é a mais popular e difundida ferramenta de renderização do mercado, não apenas porque é uma aplicação open source, mas também porque providencia uma variedade de formatos de saída, é flexível e facilmente extensível.

A ferramenta FOP é uma aplicação Java que lê um objeto de entrada e renderiza-o (transforma-o) para diferentes formatos de saída específicos (PDL), como PDF, PS e SVG. O arquivo de entrada deve ser formatado pelo padrão XSL-FO e, como resposta, gera um arquivo no formato de saída requisitado, com o conteúdo descrito pelos objetos de formatação (Figura 6).

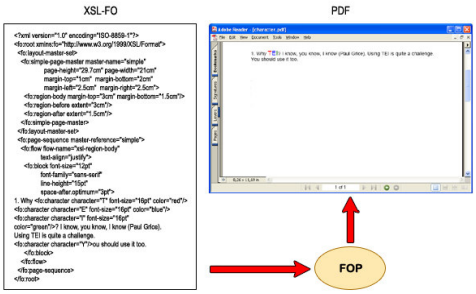


Figura 6

Não sendo responsável pela leitura e análise (parsing) do documento de entrada (no caso um arquivo PPML), a engine FOP é complementada por outra ferramenta, que tem como função realizar a busca das porções compostas por XSL-FO nestes documentos. Para isto, em conjunto com a ferramenta FOP tem-se a presença da ferramenta FO+Processor (alvo deste estudo).

Como já mencionado anteriormente, documentos PPML são constituídos de partes renderizáveis e partes não-renderizáveis. Estas são as imagens e as porções estáticas do documento, aquelas são as porções compostas por XSL-FO (porções variáveis). O FO+Processor (Formatting Objects Plus Processor) trata-se de uma aplicação que interage com o FOP, ficando encarregada de extrair do documento PPML de entrada suas partes variáveis e encaminhá-las à engine de renderização.

A ferramenta FO+Processor também é implementada utilizando-se a linguagem de programação Java.

5. FO+Processor Seqüencial

Para a geração do documento final, as porções variáveis do documento PPML devem ser enviadas ao FOP, para então serem transformadas em PDL. A ferramenta FO+Processor é a responsável por extrair estas porções e entregá-las à engine de renderização (FOP). A seguir, será explicado em detalhes como é realizado este processo.

O documento final (arquivo de saída) do FO+Processor é formado pela mesma estrutura do documento PPML de entrada: as partes não-renderizáveis são apenas copiadas *ipsis litteris* e as partes variáveis têm os XSL-FOs substituídos pelos seus correspondentes renderizados (Figura 7).

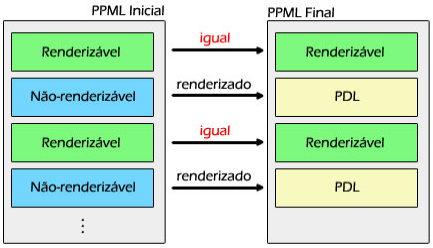


Figura 7

A ferramenta FO+Processor pode ser subdividida em dois módulos principais: FOExtractor e FOIterator.

O FOExtractor é responsável pela análise do documento PPML de entrada, separando as partes renderizáveis das não renderizáveis. O FOIterator então, envia para o FOP a parte renderizável.

6. FO+Processor Paralelo

O fato do FO+Processor ter sua execução bloqueada enquanto a ferramenta de renderização (FOP) transforma a porção variável em um formato PDL surge como um possível gargalo, podendo aparecer como o maior problema no resultado final do seu tempo de execução.

Pode-se perceber que a aplicação apresentaria uma melhora significativa em sua velocidade e em seu desempenho se, ao invés de ter sua execução bloqueada enquanto aguarda a resposta do FOP, o FOExtractor pudesse continuar seu fluxo de execução normal (fazendo o parsing do documento PPML de entrada em busca de mais XSL-FOs), paralelamente à renderização dos XSL-FOs já encontrados.

Portanto, o foco da paralelização da ferramenta FO+Processor é otimizar este modo de execução, procurando uma maneira na qual a aplicação não necessite aguardar até que o FOP lhe envie uma resposta para continuar a análise do documento.

6.1 Implementação Paralela com 3 Módulos

Uma nova abordagem da ferramenta foi utilizada para a resolução do problema apresentado. A implementação

paralela divide-se basicamente em três componentes principais: PPMLConsumer, Broker e FOP.

6.1.1 PPMLConsumer

O componente PPMLConsumer (Figura 8) é responsável pela leitura e identificação de XSL-FOs no documento PPML de entrada e também pela escrita destas porções variáveis já renderizadas no arquivo de saída.

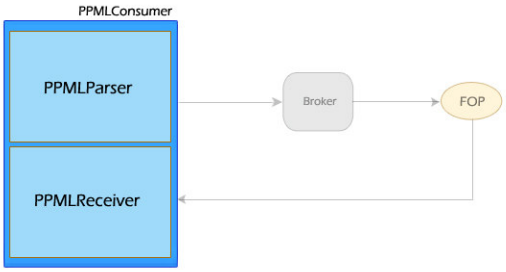


Figura 8

Para um melhor aproveitamento deste módulo, foi dividido em duas threads. A primeira – chamada de PPMLParser – lê o arquivo de entrada e faz o parsing dele, armazenando todo o conteúdo estático que está sendo lido em um buffer temporário até que encontre uma tag que lhe indique o início de uma porção renderizável (XSL-FO). Como cada parte do documento que não corresponde a um XSL-FO deve ser incluída no arquivo final sem modificações, este buffer é inserido em uma fila – com exceção do primeiro bloco que, por tratar-se sempre de uma porção estática, é escrito diretamente no arquivo de saída.

Ao encontrar a tag de início de um XSL-FO, o PPMLParser começa a armazenar este bloco de dados variáveis em outro buffer. Ao encontrar a tag referente ao final deste XSL-FO, atribui-se um número de seqüência nele para indicar a posição em que aparece no arquivo de entrada, uma vez que é indispensável que apareçam na mesma seqüência (seus correspondentes renderizados) no arquivo de saída.

O buffer, agora, contém todo o XSL-FO, juntamente com sua identificação, e está pronto para ser transmitido para o componente Broker. O processo de leitura do documento de entrada prossegue normalmente, até que se encontre outro XSL-FO.

Quem recebe os XSL-FOs previamente processados pelo componente FOP é a segunda thread do PPMLConsumer, chamada de PPMLReceiver, que também responsável pela escrita do documento final.

6.1.2 Broker

O próximo componente – denominado Broker (Figura 9) – é responsável por receber os XSL-FOs do componente PPMLConsumer e armazená-los em uma fila. Estes objetos devem ser repassados para um componente FOP que esteja livre para renderizá-los. Como o Broker deve aguardar um envio do PPMLConsumer e deve enviar os XSL-FOs para um componente FOP, também foi dividido em duas threads: BrokerReceiver e BrokerSender.

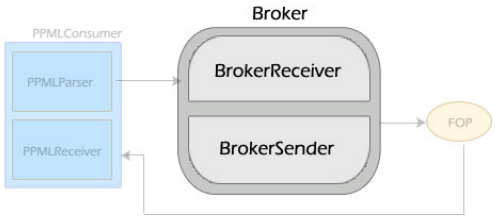


Figura 9

Ao receber do PPMLConsumer um XSL-FO, a thread BrokerReceiver armazena-o em uma fila, para que a outra thread do componente Broker também o conheça (Figura 10). A thread BrokerSender então, é responsável por verificar se há algum XSL-FO na fila para ser renderizado. Se houver, retira-o da fila e envia-o para o primeiro componente FOP que estiver livre.

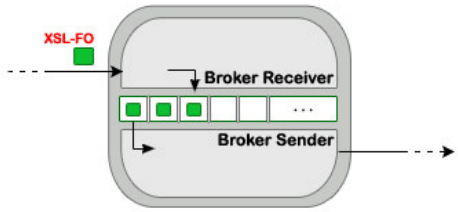


Figura 10

6.1.3 FOP

O último componente trata-se da engine de renderização (FOP). O FOP renderiza os XSL-FOs recebidos do componente Broker e, ao término deste processo, deve enviar o resultado para a thread PPMLReceiver do componente PPMLConsumer e avisar ao componente Broker que está livre para receber (se for o caso) outro XSL-FO, completando assim o ciclo da arquitetura implementada.

Todos os módulos em conjunto são apresentados na Figura 11.

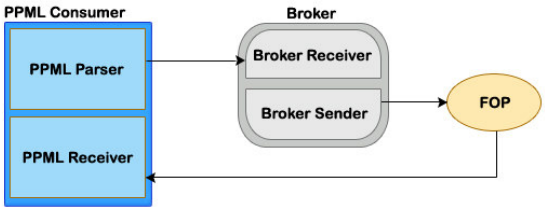


Figura 11

Nesta versão, então, necessita-se da utilização de no mínimo três unidades ativas distintas: uma para o PPMLConsumer, uma para o Broker e uma para o FOP. A cada novo processo adicionado na execução, aumenta-se o número de componentes FOP. Ou seja, se 4 processos forem utilizados, ter-se-á um componente PPMLConsumer, um Broker e dois FOPs. Se houverem 5 processos para a execução da aplicação paralela, a configuração será de um componente PPMLConsumer, um Broker e três FOPs, e assim sucessivamente

6.2 Implementação Paralela com Buffer de XSL-FOs e 4 Módulos

Nesta versão, foram realizadas algumas otimizações em relação àquela apresentada anteriormente. Dentre elas estão o envio de buffers de XSL-FOs baseado no seu tamanho em bytes e a divisão do componente PPMLConsumer em dois processos distintos.

6.2.1 Buffer de XSL-FO

A situação de apenas um XSL-FO ser enviado por vez do componente PPMLConsumer para o Broker pode não compensar o ganho de processamento na renderização em paralelo de um XSL-FO pequeno, pois a engine de renderização FOP trabalha de forma muito rápida. Para resolver tal problema, poder-se-ia adotar uma estratégia definindo um número x pré-determinado de XSL-FOs a ser enviado para cada FOP. Entretanto, o tamanho dos XSL-FOs pode variar muito, existindo aqueles muito grandes, os quais demorariam a serem processados. Com isso, caso um conjunto de XSL-FOs grandes fosse enviado aos FOPs a comunicação poderia tornar-se muito custosa. Também ocorreria o risco de prejudicar o balanceamento de carga do sistema, a medida que alguns FOPs ficariam sem trabalho por terminar a renderização muito rápido, enquanto outros teriam conjuntos gigantescos de XSL-FOs para renderizar.

Com isto, na tentativa de equilibrar o peso das tarefas entre os componentes FOP presentes na aplicação, pensou-se em agrupar XSL-FOs de forma a enviar aproximadamente a mesma carga de trabalho para todos. Utilizou-se então, uma técnica para agrupar em buffers certa quantidade de dados, baseada no tamanho em bytes da porção variável que cada FOP receberá por vez para renderizar.

Neste contexto, então, sempre que o PPMLParser identificar uma porção renderizável a ser enviada, ele adiciona a um buffer de XSL-FOs. Quando este buffer atingir um número y de bytes, o mesmo é transmitido ao Broker que se responsabiliza por armazená-lo. O Broker, então, envia o buffer inteiro a algum FOP livre do sistema. Este FOP desempacota o conteúdo e renderiza todos os XSL-FOs recebidos, armazenando os resultados renderizados em um buffer de saída que é enviado ao PPMLReceiver. O PPMLReceiver recebe os XSL-FOs renderizados e escreve-os no arquivo de saída, levando em conta a posição que devem ser colocados.

6.2.2 Divisão em 4 Módulos

O módulo composto pelas threads PPMLParser e PPMLReceiver pode ficar sobrecarregado, pois lê o arquivo de entrada, faz o parsing do arquivo procurando por XSL-FOs, armazena estas porções variáveis em um buffer, envia o buffer para o Broker, recebe os XSL-FOs renderizados e escreve o arquivo de saída.

Perante tal situação, separou-se as threads do componente PPMLConsumer em dois componentes diferentes: PPMLParser e PPMLReceiver. Com isto, a responsabilidade de cada componente permanece a mesma em relação a que apresentava, porém, não existe mais a disputa pelo processador, e cada componente é um novo processo da aplicação (Figura 12).

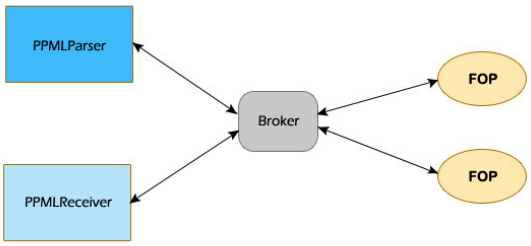


Figura 12

7. Resultados Obtidos

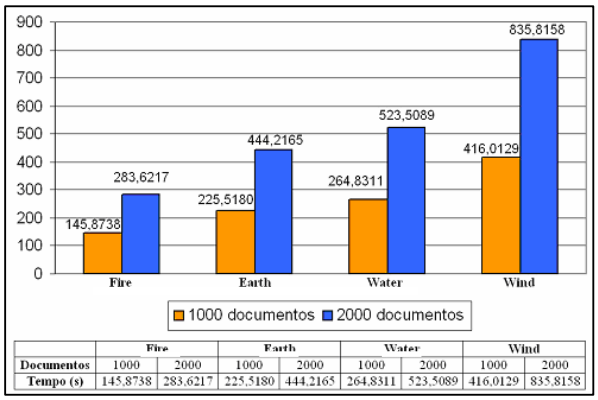
Para a execução do FO+Processor paralelo, utilizou-se o cluster denominado “Ombrófila”, encontrado no Centro de Pesquisa em Alto Desempenho (CPAD) [10], na PUCRS. Este cluster é composto por 32 máquinas, interligadas por uma rede Fast-Ethernet, sendo 14 delas epc10 com 256Mb de memória RAM e processador Pentium de 1GHz e as outras 18, epc40 com 256Mb de memória RAM e processador Pentium de 1.6Ghz.

A comunicação entre os processos deu-se através de troca de mensagens, com a utilização da biblioteca MPI MPICH. Também utilizou-se Threads em algumas partes do programa.

Com a finalidade de validar a implementação e verificar vantagens e possíveis problemas da solução paralela, fez-se alguns experimentos com alguns casos de teste. A Figura 13 apresenta um detalhamento sobre os casos de teste executados.

Figura 13

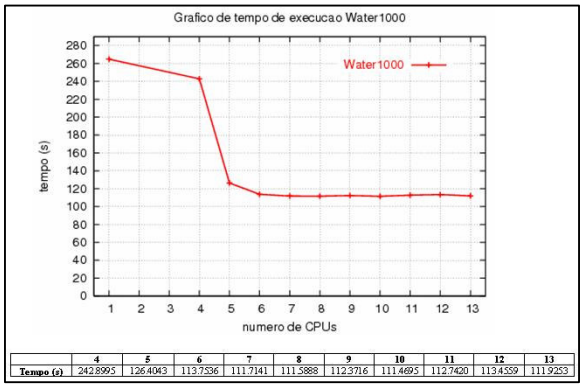
A seguir, apresentam-se os tempos da versão seqüencial.



	Fire		Earth		Water		Wind	
Documentos	1000	2000	1000	2000	1000	2000	1000	2000
Tempo (s)	145,8738	283,6217	225,5180	444,2165	264,8311	523,5089	416,0129	835,8158

Figura 14

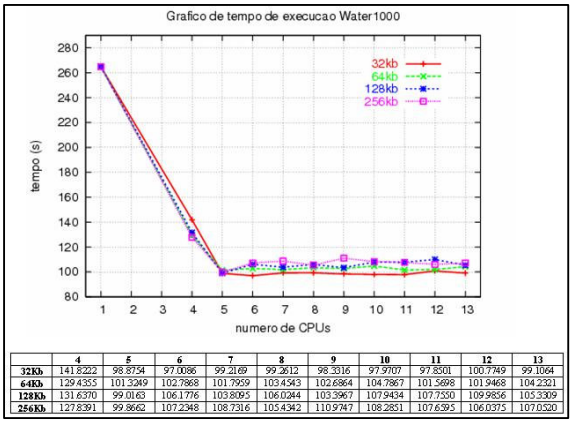
A Figura 15 mostra o gráfico do caso de teste Water 1000 para a implementação com 3 módulos.



	4	5	6	7	8	9	10	11	12	13
Tempo (s)	242.8995	126.4043	113.7536	111.7141	111.5888	112.3716	111.4695	112.7420	113.4559	111.9253

Figura 15

O gráfico da Figura 16 mostra os tempos obtidos com a versão com buffers de XSL-FOs e 4 módulos.

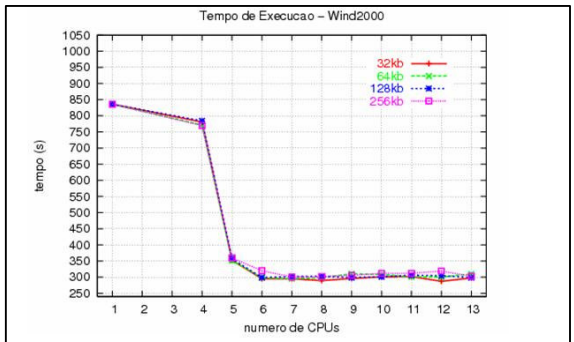


	4	5	6	7	8	9	10	11	12	13
32Kb	141.8222	98.8754	97.0086	99.2169	99.2612	98.3316	97.9707	97.8501	100.7749	99.1064
64Kb	129.4355	101.3249	102.7868	101.7959	103.4543	102.6864	104.7867	101.5698	101.9468	104.2321
128Kb	131.6370	99.0163	106.1776	103.8095	106.0244	103.3967	107.9434	107.7550	109.9856	105.3309
256Kb	127.8391	99.8662	107.2348	108.7316	105.4342	110.9747	108.2851	107.6595	106.0375	107.0520

Figura 16

Como pôde-se perceber, houve uma melhora significativa no tempo de execução da aplicação, tanto na versão com 3 módulos quanto na segunda versão (com buffer de FOs e 4 módulos).

A seguir, o gráfico da Figura 17 ilustra os tempos obtidos com a segunda versão para o maior caso de teste (Wind2000).



	4	5	6	7	8	9	10	11	12	13
32kb	780.6578	351.9972	295.4205	295.2089	289.2129	295.9342	301.0186	301.5967	287.0866	297.1698
64kb	771.2319	351.1376	298.2857	295.3347	298.1006	310.4149	307.9302	301.2263	300.4275	309.7562
128kb	783.9990	357.0458	298.7687	301.5960	302.5130	298.7054	301.2455	305.7956	304.5428	298.7003
256kb	769.9124	359.5012	319.8948	300.4844	301.5463	305.7111	311.5069	312.0312	318.8420	302.7778

Figura 17

8. Considerações Finais

O ganho apresentado comprova que é possível atingir melhores resultados na renderização de documentos a partir da computação de alto desempenho. O speedup não se apresentou tão satisfatório, porém, o grande ganho do estudo e da implementação deu-se na diminuição significativa do tempo de execução. Diversas técnicas e metodologias foram aplicadas para a implementação da versão paralela apresentada. Entre elas destacam-se modelos teóricos de algoritmos, estudos sobre a granularidade, estudos sobre a forma de comunicação, utilização de threads e da biblioteca MPI mpich. Este esforço gerou resultados satisfatórios, entretanto algumas melhorias futuras podem ser consideradas para um ganho ainda maior de desempenho.

Supondo a utilização de apenas um componente Broker para lidar com muitos FOPs, existe a possibilidade dele ficar saturado. O Broker – responsável por receber XSL-FOs não renderizados, verificar quais FOPs estão livres para receber trabalho e enviar o trabalho – pode não conseguir realizar todas suas funções em tempo hábil, provocando a subutilização de diversos componentes FOP. Desta maneira, vários módulos FOP ficarão livres, enquanto a fila do Broker irá conter tarefas a serem realizadas. Considerando esta situação, uma melhoria a ser implementada na versão paralela atual é a possibilidade de replicação de módulos Broker, a fim de permitir que cada um deles seja responsável por um determinado número de FOPs, com o qual consiga lidar de forma a não ficar saturado de trabalho. Assim, possivelmente, o desempenho geral da aplicação não será comprometido.

Referências

[1] MSDN. **MSDN home page**. Extraído de http://www.msdn.microsoft.com/, em setembro de 2005.

[2] MPI. **Message Passing Interface (MPI) Forum Home Page**. Extraído de www.mpi-forum.org, em maio de 2006.

[3] XML. **Extensible Markup Language (XML)**. Extraído de http:// www.w3.org/XML, em maio de 2006.

[4] PODi. **Print on Demand Initiative**. Extraído de http://www.podi.org/, em maio de 2005.

[5] XSL-FO. **The Extensible Stylesheet Language Family (XSL)**. Extraído de http://www.w3.org/Style/XSL, seção XSL Formatting Objects, em maio de 2006.

[6] W3C. **The World Wide Web Consortium**. Extraído de http://www.w3.org/, em maio de 2005.

[7] XSL-T. **XSL-Transformations**. Extraído de http://www.w3.org/TR/1999/REC-xslt-19991116, seção References, em maio de 2005.

[8] SVG. **Scalable Vector Graphics**. Extraído de http://www.w3.org/Graphics/SVG/, em maio de 2006.

[9] FOP. **Formatting Objects Processor**. Extraído de http://xml.apache.org/fop/, em maio de 2005.

[10] Centro de Pesquisa em Alto Desempenho. **CPAD home page**. Extraído de http://www.cpad.pucrs.br/ em setembro de 2005.