Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Informática

Pós-Graduação em Ciência da Computação

Estratégias de Paralelização para Renderização de Documentos XSL-FO com Uso da Ferramenta FOP

Rogerio Timmers Zambon

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação.

Orientador: Prof. Dr. Luiz Gustavo Fernandes

Porto Alegre, outubro de 2006.

Campus Central
Av. Ipiranga, 6681 – prédio 16 – CEP 90619-900
Porto Alegre – RS – Brasil
Fone: +55 (51) 3320-3544 – Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Reconhecido pelo Parecer No. 930/98.C.N.E. Homologação Publicada no D.O.U. de 30/12/98.

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Estratégias de Paralelização para Renderização de Documentos XSL-FO com Uso da Ferramenta FOP***", apresentada por Rogério Timmers Zambon, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 27/01/2006 pela Comissão Examinadora:

Prof. Dr. Luiz Gustavo Leão Fernandes – PPGCC/PUCRS
Orientador

Prof. Dr. César Augusto Fonticielha De Rose – PPGCC/PUCRS

Prof. Dr. Paulo Henrique Lemelle Fernandes – PPGCC/PUCRS

Prof. Dr. Gerson Geraldo Homrich Cavalheiro – UNISINOS

Homologada em 11/04/07, conforme Ata No. 002 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

AV. IPIRANGA, 6681 - PRÉDIO 16/SALA 106
90619-900 - PORTO ALEGRE -RS -BRASIL
TELEFONE: 55 51 33203611 FAX 55 51 3320.3621
http: //www.pucrs.br/inf/pos e-mail: ppgcc@inf.pucrs.br

Agradecimentos

Agradeço ao professor Luiz Gustavo Fernandes, pela amizade em primeiro lugar e por acreditar no projeto ao aceitar ser meu orientador.

Ao professor Paulo Fernandes, meu orientador no primeiro ano de mestrado pela confiança e ajuda.

Aos colegas do CAP Pedro, Lucas, Márcio, Mateus, Gustavo e Thiago pela grande ajuda em tudo.

Ao professor De Rose por participar em todas as avaliações do trabalho sempre com dicas para melhorar o andamento do trabalho.

A todos na HP, que me incentivaram a participar e concluir o curso principalmente quanto à flexibilidade de horários.

Ao meu colega de mestrado Ricardo Presotto pelo companheirismo e amizade.

Ao colega Fabio Giannetti da HP Bristol, pela grande amizade e companheirismo. Também por revisar grande parte dos documentos entregues durante o curso e por auxiliar com o grande conhecimento na área de publicações digitais.

"Preserve o que é bom. Reinvente o resto."

Carly Fiorina

Resumo

Grandes volumes de trabalho para impressão são cada vez mais comuns devido ao aumento da demanda por documentos personalizados. Neste contexto, Impressão de Dados Variáveis (*Variable Data Printing - VDP*) tornou-se uma ferramenta muito útil para profissionais de *marketing* que necessitam personalizar mensagens para cada cliente em materiais promocionais e campanhas de publicidade. VDP permite a criação de documentos baseados em um modelo (*template*) contendo partes estáticas e variáveis. A ferramenta de renderização deve ser capaz de transformar a parte variável em um formato composto, ou PDL (*Page Description Language*) tais como PDF (*Portable Document Format*), PS (*PostScript*) ou SVG (*Scalable Vector Graphics*). A quantidade de conteúdo variável em um documento é totalmente dependente do modelo (*layout*) da publicação definido por um profissional da área. Além disso, o conteúdo variável a ser renderizado pode variar de acordo com os dados lidos do banco de dados. Desta forma, este processo é chamado repetidamente e pode tornar-se facilmente um gargalo, especialmente em um ambiente de produção comprometendo inteiramente a geração de um documento. Neste cenário, técnicas de alto desempenho aparecem como uma interessante alternativa para aumentar o rendimento da fase de renderização. Este trabalho introduz uma solução paralela portável e escalável para a ferramenta de renderização chamada FOP (*Formatting Objects Processor*), a qual é usada para renderizar o conteúdo variável expresso em linguagem XSL-FO (*eXtensible Stylesheet Language-Formatting Obects*).

Abstract

High volume print jobs are getting more common due to the growing demand for personalized documents. In this context, VDP (*Variable Data Printing*) 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 (*Portable Document Format*), PS (*PostScript*) or SVG (*Scalable Vector Graphics*). 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).

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Símbolos e Abreviaturas

PS	PostScript	x
VDP	Variable Data Printing	x
XSL-FO	eXtensible Stylesheet Language formatting objects	23
XML	EXtensible Markup Language	28
PODi	Print on Demand Initiative	30
PPML	Personalized Print Markup Language	30
URL	Universal Resource Identifier	30
LAN	Local Area Network	30
TIFF	*Tagged Image File Format*	31
BMP	Bit-Mapped Graphic	31
FO	Formatting Objects	31
W3C	World Wide Web Consortium	31
XSL-T	eXtensible Stylesheet Language - Transformations	31
FOP	Formatting Object Processor	31
PDL	Page Description Language	34
PCL	Printer Control Language	34
PDF	Portable Document Format	34
SVG	Scalable Vector Graphics	34
SISD	Single Instruction Stream/Single Data	38

SIMD	Single Instruction Stream/Multiple Data Stream	39
MISD	Multiple Instruction Stream/Single Data Stream	40
MIMD	Multiple Instruction Stream/Multiple Data	40
PVP	Parallel Vector Processor	41
SMP	Symmetric Multiprocessing	41
MPP	Massively Parallel Processing	41
DSM	Distributed Shared Memory	41
NOW	Network of Workstations	42
COW	Clusters of Workstations	42
ATM	Asynchronous Transfer Mode	42
NUMA	Non-Uniform Memory Access	46
COMA	Cache-Only Memory Architecture	46
UMA	Uniform Memory Access	46
NORMA	Non-Remote Memory Access	49
JVM	Java Virtual Machine	58
J2SDK	Java Standard Development Kit	58
RAM	Random Access Memory	59
CPAD	Centro de Pesquisa de Alto Desempenho	59
I/O	Input/Output	79
CPU	Central Processing Unit	80
SAC	Symposium on Applied Computing	83
DOM	Document Object Model	84

Capítulo 1

Introdução

Criação de documentos personalizados é uma prática cada vez mais comum com a evolução do mundo digital. A montagem e transformação automática de documentos tornou-se um processo necessário para atender a demanda. Tipicamente, documentos personalizados contêm áreas que são comuns em um conjunto de documentos, e portanto conteúdo estático, assim como áreas personalizadas e variáveis. No método tradicional de informação variável [PHOF03], ferramentas de documentação permitem que os *designers* definam um modelo que servirá de base para um conjunto de documentos. Além disso, o *designer* também define áreas vazias (tamanho fixo), nas quais o conteúdo variável será colocado. Assim, o *layout* comum é o mesmo para todos os documentos e embora possa conter dados variáveis, não pode responder a propriedades dinâmicas como redimensionamento do tamanho dos dados variáveis.

Estas limitações têm disparado esforços em pesquisas para automatizar o processo de criação de documentos personalizados. Documentos podem ser escritos como modelos e a produção pode ser automatizada mantendo um alto nível de qualidade de criação. Este é o foco desse trabalho, explorar como a geração de tais documentos é alcançada em um ambiente de produção. Casas de impressão (*print shops*) requerem um processo previsível, eficiente, e de qualidade industrial para imprimir e finalizar documentos. Tem sido provado que o uso de XSL-FO em partes não definidas do documento torna possível a integração dos dados variáveis tardiamente no processo, até mesmo durante a própria impressão. Esta forma é claramente mais vantajosa visto que não requer disponibilidade dos dados variáveis durante a fase de projeto (*design*) do documento. Também permite a transmissão dos documentos e dados ainda como modelos, ao invés de um documento completamente expandido.

1.1 Motivação

O objetivo dessa pesquisa é explorar e indicar uma solução escalável e modular para executar a composição de dados variáveis usando ferramentas paralelas de renderização.

A maioria dos ambientes de produção de publicações digitais usam impressoras digitais em paralelo para maximizar o equilíbrio entre os processos, assim como toda a produção de documentos (*jobs*). Em tal ambiente, todas as atividades relacionadas à preparação do documento precisam ser executadas em um determinado espaço de tempo, já que as tarefas *jobs* são completados em uma ordem seqüencial em múltiplas impressoras. No caso de impressão de dados variáveis, acima dos passos existentes de preparação para impressão, existe a necessidade de integrar os dados variáveis no modelo e renderizar as partes não definidas ou estáticas do documento.

O processo de renderização é usualmente muito extenso e no caso de milhares de documentos pode tornar-se um gargalo. Impressoras digitais modernas também são capazes de imprimir na mesma velocidade de um renderizador (cerca de 1 página por segundo). Esta taxa é o mínimo requerido para manter a impressora digital trabalhando em velocidade máxima. Quando as impressoras digitais são usadas em paralelo, a velocidade do renderizador é multiplicada pelo número de impressoras, assim, torna-se muito difícil de alimentar a todas as impressoras na velocidade necessária.

Quando o processo de renderização é centralizado em um único processador, uma quebra provavelmente irá acontecer, uma vez que a velocidade das impressoras em paralelo excede a velocidade do processo de renderização criando um gargalo. Similarmente ao conceito de usar impressoras em paralelo a fim de alcançar um melhor desempenho e mais rapidamente consumir os *jobs*, nosso objetivo é desenvolver uma proposta para paralelizar a ferramenta de renderização de documentos XSL-FO. Os resultados mostram que o sistema desenvolvido pode combinar com a velocidade de impressoras rodando em paralelo. Também temos que considerar que é necessário prover um número adequado de processadores para atingir a velocidade ideal.

Os resultados deste trabalho mostram uma solução modular e escalável para impressão de dados variáveis com renderização em tempo de impressão.

1.2 Estrutura do Volume

Este trabalho foi dividido em 6 capítulos incluindo introdução e conclusão. Os quatro primeiros descrevem as bases teóricas a esta dissertação. No Capítulo 2, é feita uma conceituação de engenharia de documentos apresentando alguns padrões, linguagens e ferramentas utilizados na área de publicações digitais. No Capítulo 3, conceitos de processamento de alto desempenho são brevemente citados também como embasamento para a dissertação. Uma análise do problema em sua versão seqüencial é descrito no Capítulo 4 assim como um posicionamento em relação às bases teóricas listadas. Além disso, neste Capítulo o ambiente utilizado para desenvolvimento da solução paralela e uma descrição dos documentos utilizados para a realização dos testes são abordados.

Nos demais capítulos estão as principais contribuições obtidas ao longo desta dissertação. No Capítulo 5, apresenta-se as estratégias de paralelização da ferramenta FOP utilizadas até atingir-se o modelo atual. Finalmente, na conclusão são tecidas as considerações finais a respeito do modelo proposto bem como os trabalhos futuros relacionados.

Capítulo 2

Engenharia de Documentos

Engenharia de Documentos está desenvolvendo-se como uma nova disciplina para especificar, esboçar, e implementar documentos eletrônicos que fornecem interfaces para processos de negócios via serviços baseados em Web [GM02].

No mundo dos negócios, documentos sempre tiveram um papel fundamental como meio de interação entre o negócio e as pessoas envolvidas. À medida que as empresas crescentemente movem suas atividades para a Internet e experimentam novas maneiras de como fazer negócios, podemos começar a tratar documentos como interfaces [RGM99].

Muitos tipos de documentos são essenciais para os negócios. Alguns como catálogos, brochuras e folhetos ajudam os compradores a localizar e selecionar produtos e serviços. Outros como guias de usuários e manuais, são feitos para proverem um uso mais efetivo dos produtos e serviços após a compra. Primeiramente, documentos na Web eram usados somente para documentos não-transacionais. Mais tarde, com o avanço das tecnologias documentos como pedidos, faturas, passaram a ter grande importância como tipos de documentos eletrônicos.

2.1 Definições

Documentos **narrativos** são tradicionalmente chamados de **publicações** e a técnica de análise e modelagem empregada é denominada **análise de documentos**. Como contraste, documentos transacionais são otimizados para uso em negócios e diferem substancialmente das publicações orientadas a usuários. O método utilizado neste caso é nomeado **modelagem de dados**.

A Engenharia de Documentos surge então como uma mescla desses métodos sendo efetivo

tanto para documentos **narrativos** como para **transacionais**. **Documentos modelos** podem ser criados e reutilizados para diferentes tipos de negócios, encorajando os criadores a balanceá-los para negócios internos e a necessidade de serem entendidos em outras áreas.

2.1.1 Análise de Documentos

Análise de documentos é conduzida com o objetivo de abstrair um modelo lógico de uma instância existente de um tipo de documento único codificando o modelo em um esquema XML (*EXtensible Markup Language*). O método de análise de documentos permite aos usuários executarem tarefas específicas com novas instâncias criadas a partir do documento. Por exemplo, quando o tipo de documento é uma publicação, o novo esquema separa descrição da estrutura do documento e o conteúdo da estrutura de apresentação do mesmo. Isto inclui fontes, tamanhos e atributos de formatação que são usados para representar ou ressaltar vários conteúdos. Assim que essa separação acontece, um ou mais estilos podem ser utilizados para formatar de maneira consistente qualquer instância válida do documento.

2.1.2 Modelagem de Dados

A Modelagem de dados é dedicado a entender e descrever uma estrutura lógica de objetos de dados que têm várias propriedades e associações umas com as outras. O objetivo típico da **modelagem de dados** é definir uma ou mais categorias ou esquemas para organizar essas propriedades e associações eficientemente para criar, revisar, apagar objetos de dados ou para encontrar aqueles com características específicas.

Análise de documentos e modelagem de dados compartilha o objetivo de criar uma descrição formal de uma classe de instâncias, porém o método é melhor aplicado quando não há um número ilimitado de instâncias idênticas.

2.1.3 Unificando Análise de Documentos e Modelagem de Dados

Análise de documentos e modelagem de dados são provenientes de diferentes disciplinas e utilizam diferentes ferramentas, terminologias e técnicas. Especialistas de cada área não sabiam como conversar e também não reconheciam uma parte comum em ambos objetivos. Ambos oferecem valiosas contribuições para criação de documentos porém têm tido pouca interação. A Engenharia de Documentos unifica estas duas perspectivas identificando e enfatizando o que têm em comum ao invés de ressaltar suas diferenças.

Antes da análise de como a Engenharia de Documentos utiliza os conceitos de análise de documentos e modelagem de dados, é importante que sejam apresentados os três tipos de informações encontradas em documentos:

- Conteúdo - informação que diz "o que isso significa".
- Estrutura - "onde é isso" ou "como isso é organizado ou montado". Agrega conteúdo e informação em mais de um componente reusável.
- Apresentação - "como isso é mostrado".

Embora o item apresentação seja o menos importante, é essencial analisá-lo com cuidado, primeiramente devido à sua correlação com a estrutura e o conteúdo. Correlações estas que seguem padrões para diferentes tipos de documentos.

Os pontos cruciais da análise e modelagem de dados harmonizados na Engenharia de Documentos são:

- Identificar a apresentação, conteúdo, e componentes estruturais definindo os relacionamentos entre si.
- Identificar componentes de "bom" conteúdo.
- Esboçar, descrever, e organizar padrões para facilitar o reuso.
- Montar modelos de documentos hierárquicos para organizar os componentes de acordo com os requerimentos de um contexto específico.

Neste contexto, XML é uma tecnologia quem tem se destacado bastante no contexto da Engenharia de Documentos. Dentre suas principais vantagens, podem ser citados:

- Permite que novos vocabulários sejam criados para tipos particulares de documentos.
- É uma linguagem hierárquica (o que facilita a organização dos componentes).
- Facilita a integração de uma variedade de paradigmas tais como banco de dados, orientação a objetos, e estrutura de documentos.

Com o crescimento da Engenharia de Documentos e a facilidade de mesclar *layout* e conteúdos provenientes de banco de dados, várias empresas começaram a desenvolver padrões baseados

em XML para controlar o processo de impressão. Documentos personalizados para diferentes campanhas de *marketing* aumentaram essa necessidade assim como a capacidade das impressoras digitais cada vez mais poderosas. Para impedir a crescente criação de modelos de composição de documentos por parte de empresas que lidam diretamente com impressão digital, foi criado um consórcio de empresas que trabalhariam unidas na definição de uma linguagem única de impressão.

Criado em 1999, o PODi (*Print on Demand Initiative*) [POD05] é uma iniciativa sem fins lucrativos cuja missão é desenvolver a indústria de impressão digital encorajando a padronização. Os membros dessa iniciativa desenvolveram uma linguagem não-proprietária denominada PPML (*Personalized Print Markup Language*) a qual utiliza XML como base.

2.2 PPML

PPML [DdB00] é uma linguagem padrão utilizada para impressão digital construída a partir de XML desenvolvida pelo PODi. PPML tem sido designado para melhorar o processo de rasterização para o conteúdo de documentos que usam linguagens tradicionais de impressão. PPML na verdade introduz o método de conteúdo reusável através do qual conteúdos usados em muitas páginas podem ser enviados para a impressora uma única vez e acessados quantas vezes for necessário. Isto permite que conteúdos de alta qualidade gráfica sejam rasterizados também uma única vez e acessados através de instruções modelo ao invés de reenviar-se todo o gráfico toda vez que o mesmo deva ser impresso. Cada objeto reusável em PPML é chamado recurso. A fim de garantir que todos os recursos estejam disponíveis e a impressora digital possa acessá-los, PPML permite referências externas URL (*Universal Resource Identifier*).

Usualmente, a impressora digital pode acessar os recursos requeridos diretamente de uma unidade disco local ou através de uma LAN (*Local Area Network*). PPML é uma linguagem hierárquica que contém documentos, páginas e objetos. Os objetos contidos são denominados reusáveis ou disponíveis. PPML também introduz o conceito de escopo, para os objetos reusáveis, de forma que o produtor PPML pode instruir o PPML consumidor sobre o tempo de vida de um objeto em particular. Esse método é bastante poderoso, eficiente e pode otimizar o requisito de cache de impressão e objetos pré-rasterizados que são reutilizados por todo o *job* e/ou somente em uma página particular. Alguns trabalhos têm sido apresentados [Bos00, MMM$^+$04] a fim de endereçar este problema que atualmente permanece aberto e está fora do escopo deste

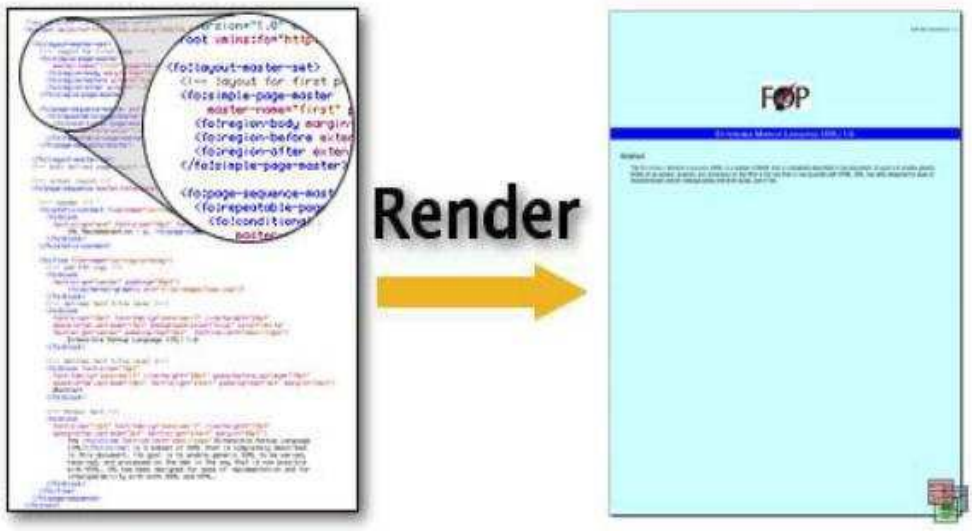


Figura 2.1: Exemplo de renderização de um XSL-FO para um formato de saída

trabalho.

O conteúdo variável é integrado dentro do objeto PPML e é formatado através do uso de XSL-FO. O objeto que contém o XSL-FO é denominado *"copy-hole"*, que é uma área definida no PPML a qual pode conter um conteúdo variável expresso na própria linguagem XSL-FO ou conteúdo não variável como imagens TIFF (*Tagged Image File Format*), BMP (*Bit-Mapped Graphic*), etc. XSL-FO (também abreviado como FO - *Formatting Objects*) é um padrão definido pelo consórcio W3C (*World Wide Web Consortium*), o qual conta com empresas envolvidas com Internet e Web, [W3C] introduzido para formatar conteúdo XML em mídias paginadas. De modo ideal, funciona em conjunto com XSL-T (*eXtensible Stylesheet Language - Transformations*) [XT05] para mapear conteúdo XML em um modelo de página. Quando o XSL-FO é completado com ambos: modelo de paginação e conteúdo formatado, o renderizador XSL-FO executa o passo de composição do conteúdo dentro das páginas obtendo assim o documento final conforme ilustrado na Figura 2.1. A composição é um passo complexo e requer ordem de impressão assim como conhecimento do modelo. A ferramenta de renderização XSL-FO usada em nossa solução é o FOP (*Formatting Object Processor*).

Podemos dizer que PPML é utilizado para definição do *layout* da página e XSL-FO contém a parte renderizável pela ferramenta FOP dentro da página. PPML é hierárquico. Como podemos ver em 2.2, o elemento raiz pode conter elementos Tarefas (*JOB*), que podem conter Documentos (*DOCUMENTS*), que contêm Páginas (*PAGE*), as quais contêm marcas (*MARKS*)

os quais são denominados *copy-holes*.

```
1  PPML...>...
2    <JOB...>...
3      <DOCUMENT...>...
4        <PAGE...>...
5           <MARK...>...</MARK>
6           <MARK...>...</MARK>
7        <PAGE>
8        <PAGE...>...</PAGE>
9        ...
10      <DOCUMENT>
11     <DOCUMENT>...
```

Figura 2.2: Estrutura hierárquica em um documento PPML

2.3 XSL-FO

Em um documento PPML podemos encontrar *copy-holes* cujo conteúdo pode ser uma imagem, espaço em branco, ou um conteúdo de texto. Neste trabalho, estamos particularmente interessados em conteúdos de textos representados em XSL *Formatting Objects*, ou simplesmente XSL-FO, visto que é a linguagem de entrada para a ferramenta de renderização FOP. XSL-FO é um vocabulário que descreve como as páginas irão aparecer para o leitor. Existem 56 elementos XSL-FO todos listados em [W3C] sendo 99% deles inicializados pelo prefixo *fo*.

Os objetos de formatação (*FOs*) diferem basicamente naquilo que cada um deles representa. Por exemplo, o objeto *fo:list-item-label* é um marcador localizado na frente de uma lista. Pode ser um número, uma bolinha ou um caracter qualquer. Um *fo:list-item-body* contem o texto de um item na lista.

Um FO quando processado, pode ser quebrado em mais de uma página e para facilitar a impressão, foi dividido em quatro áreas principais também hierárquicas como PPML :

- Regiões: nível mais alto da hierarquia. Pode-se imaginar como uma região de uma página contendo cabeçalho, texto e rodapé. FOs que produzem regiões são do tipo *fo:region-body*, *fo:region-after*.

- Blocos: representam um bloco de texto como um parágrafo. *fo:block* e *fo:list-block* são exemplos.

- Linhas: esta área representa uma linha de texto dentro de um parágrafo.

- Entre linhas: são partes de uma linha como um simples caracter, uma referência de rodapé, ou uma equação matemática. *fo:external-graphic*, *fo:inline*, etc.

Em um documento PPML, um *copy-hole* contendo XSL-FO é facilmente identificado pelo delimitador <fo:root> </fo:root>, como mostrado em 2.3.

```
1  <fo:root>
2    <fo:layout-master-set>
3    <fo:simple-page-master page-width="162.00089pt" page-height="67.18196pt"
4  aster-name="simplePageMaster">
5        <fo:region-body />
6    </fo:simple-page-master>
7    <fo:page-sequence-master master-name="simplePageMasterSequence">
8        <fo:single-page-master-reference master-reference="simplePageMaster" />
9    </fo:page-sequence-master>
10   </fo:layout-master-set>
11   <fo:page-sequence master-reference="simplePageMasterSequence">
12   <fo:flow flow-name="xsl-region-body">
13       <fo:block-container width="162.00089pt" height="67.18196pt">
14       <fo:block language="en" hyphenate="true" font-family="Helvetica"
15         color="device-color(0,0,0,'http://www.hp.com/devicecmyk',0,0,0,1)">
16           <fo:block space-before.optimum="12pt" font-size="11pt">
17           <fo:inline>
18               We here at
19               <fo:inline font-weight="bold">MINI of Portland</fo:inline>
20               <fo:inline> want to make your MINI experience Great!</fo:inline>
21           </fo:inline>
22           </fo:block>
23       </fo:block>
24       </fo:block-container>
25   </fo:flow>
26   </fo:page-sequence>
27 </fo:root>
```

Figura 2.3: Exemplo de um *copy-hole* contendo conteúdo renderizável XSL-FO

A combinação de PPML e XSL-FO tem sido escolhida para representar modelos de documentos com alto grau de flexibilidade, reusabilidade e otimização de impressão. A sinergia alcançada por essa combinação garante que a parte não variável do modelo seja expressa como reusáveis, e a parte variável como fragmentos XSL-FO. Após a inserção dos dados variáveis no

documento, várias instâncias de documentos são formadas. O passo final é compor ou renderizar as partes em XSL-FO em uma linguagem de descrição de página (*PDL - Page Description Language*) que nada mais é do que dispor os comandos de uma página impressa para comandos que a impressora possa executá-los. PCL (*Printer Control Language*) da *HP* e Postscript da *Adobe* são dois dos PDLs mais utilizado atualmente. O processo de renderização é processado pela ferramenta FOP [FOP05].

2.4 FOP

FOP é um dos mais comuns processadores no mercado não somente porque é uma aplicação de código aberto, mas também porque provê uma grande variedade de formatos de saída além de flexibilidade. É uma aplicação Java que lê objetos de formatação (*FO*) renderizando para diferentes formatos de saída tais como PDF, PostScript, SVG , que é o foco dos resultados de renderizações realizadas nesse trabalho, entre outros.

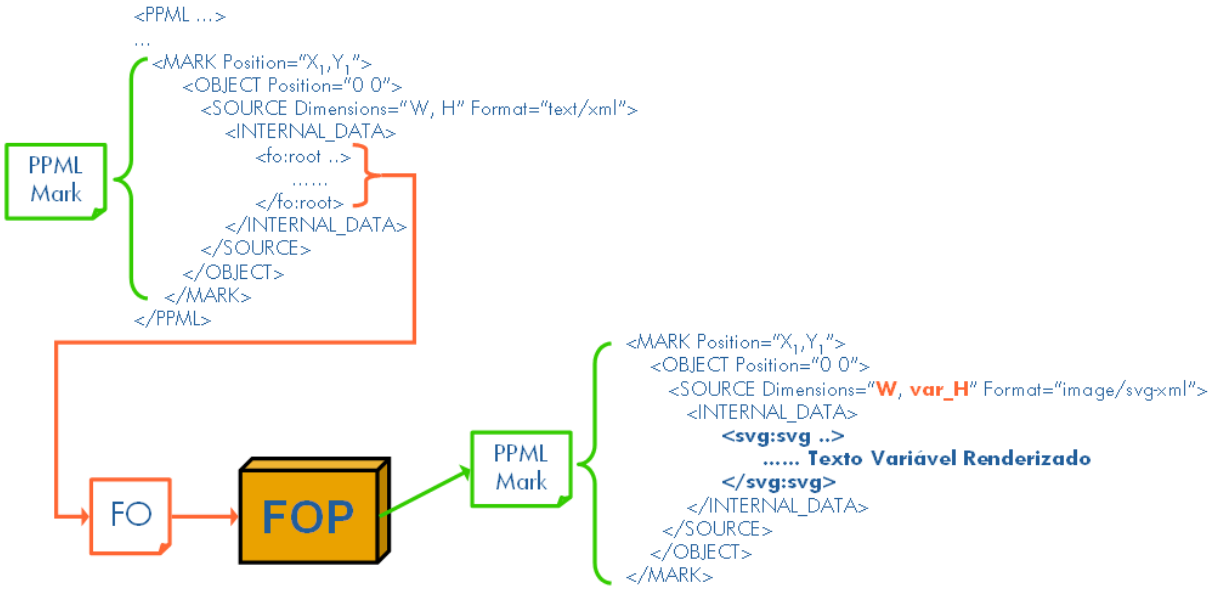


Figura 2.4: Processo de renderização de XSL-FO para SVG

A Figura 2.4 mostra como o processo de renderização é feito com o uso da ferramenta FOP partindo-se de um documento PPML contendo *copy-holes* em XSL-FO. Ao ser localizado no documento uma marca que indica um *copy-hole*, <MARK Position='X1,Y1'>, a área delimitada pelas entradas *fo:root* é enviada para a ferramenta FOP que devolverá o mesmo conteúdo

renderizado em SVG. O texto renderizado é realocado na mesma posição onde encontrava-se o XSL-FO no documento PPML.

O processo de renderização é tipicamente composto por três diferentes passos como ilustrado pela Figura 2.5.

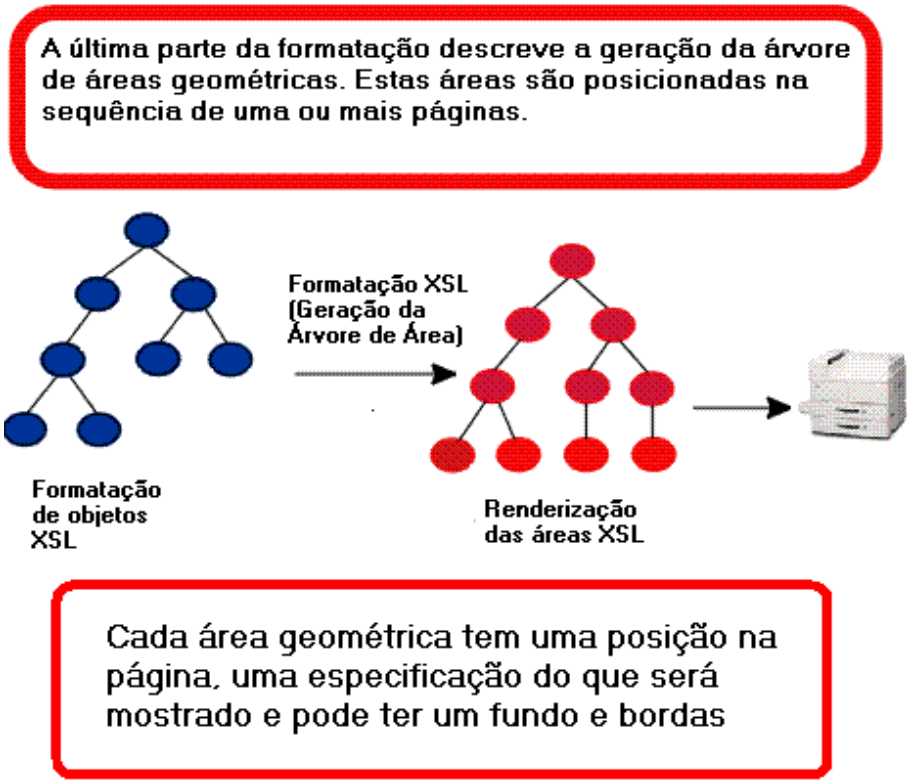


Figura 2.5: Fases do processo de renderização

1. Geração de uma árvore de objetos de formatação e resolução de propriedades;

2. Geração de uma árvore de trabalho (*area tree*) representando o documento modelado composto por uma hierarquia retangular tendo as folhas como elementos de texto ou imagens;

3. Conversão ou mapeamento da árvore de trabalho (*area tree*) para o formato de saída.

As vantagens deste método estão na completa independência entre a representação do documento XSL-FO e a construção interna da árvore de trabalho. Deste modo, é possível mapear a *area tree* para diferentes conjuntos de PDLs.

Capítulo 3

Processamento de Alto Desempenho

A área de processamento de alto desempenho vem se tornando ao longo dos anos cada vez mais necessária para que se possa obter, de forma efetiva, a solução de grandes problemas científicos. Em tais problemas, muitas vezes, os computadores tradicionais não conseguem produzir um resultado necessário dentro de limites de tempo razoáveis, comprometendo assim a viabilidade das soluções para estes problemas. Por outro lado, sistemas computacionais de alto desempenho, principalmente aqueles com arquitetura paralela, oferecem um maior potencial para a abordagem. Tais sistemas devem ser utilizados de forma que se possa efetivamente aproveitar a maior capacidade computacional disponível.

Este Capítulo, apresenta de forma resumida alguns dos principais conceitos abordados na área de processamento de alto desempenho tais como, modelos de arquiteturas, programação e algoritmos, assim como alguns fatores de desempenho utilizados para medir o ganho em relação à versão seqüencial.

3.1 Modelos de Arquiteturas de Processamento Paralelo

Muito já foi desenvolvido em termos de *hardware* paralelo, e várias classificações foram propostas [AG94, Dun90, HB84]. A mais conhecida pela comunidade computacional é a classificação de Flynn [Fly72], que apesar de antiga é bastante respeitada. Já a classificação de Duncan [Dun90], mais recente, representa o esforço de acomodar novas arquiteturas que surgiram após a taxonomia de Flynn.

3.1.1 Classificação de Flynn

Segundo Flynn, o processo computacional deve ser visto como uma relação entre fluxos de instruções e fluxos de dados. Um fluxo de instruções equivale a uma seqüência de instruções executadas (em um processador) sobre um fluxo de dados aos quais estas instruções estão relacionadas [Dun90] [Fly72].

As arquiteturas de computadores são divididas em 4 classes cada uma apresentando um esquema genérico de acordo com o fluxo de dados e instruções (Figura 3.1).

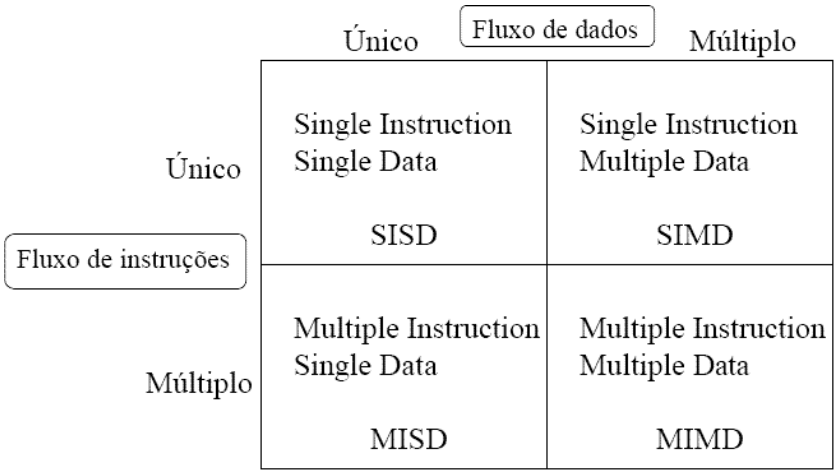


Figura 3.1: Taxonomia de arquiteturas (Flynn)

3.1.1.1 SISD

Single Instruction Stream/Single Data Stream (fluxo único de Instruções/fluxo único de dados) corresponde ao tradicional modelo Von Neumann. Um processador executa seqüencialmente um conjunto de instruções sobre um conjunto de dados (Figura 3.2).

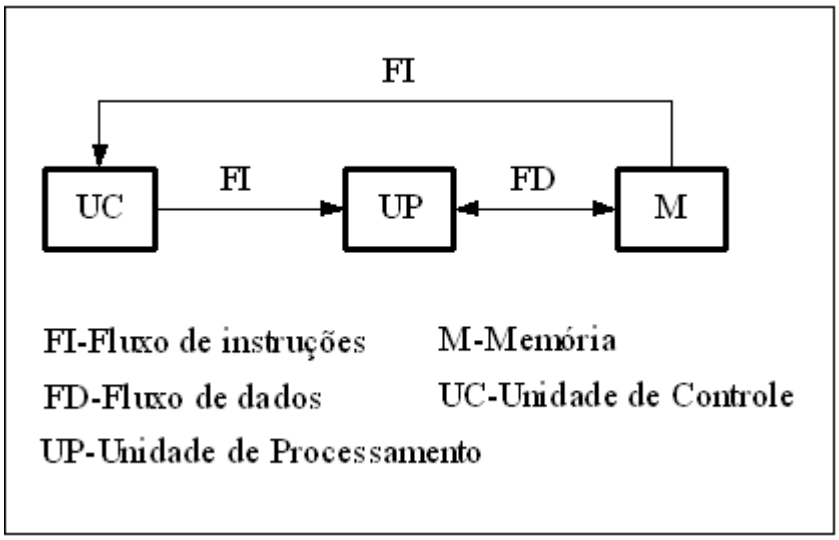


Figura 3.2: Modelo computacional SISD

3.1.1.2 SIMD

Single Instruction Stream/Multiple Data Stream (fluxo único de instruções/fluxo múltiplo de dados). Envolve múltiplos processadores controlados por uma única unidade mestre executando simultaneamente a mesma instrução em diversos conjuntos de dados (Figura 3.3). Arquiteturas SIMD são utilizadas para manipulação de matrizes e processamento de imagens.

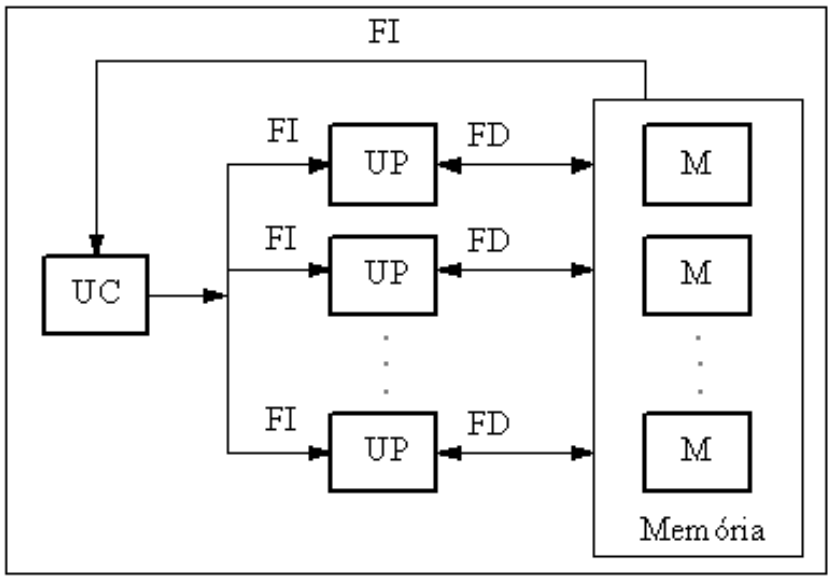


Figura 3.3: Modelo computacional SIMD

3.1.1.3 MISD

Multiple Instruction Stream/Single Data Stream (Fluxo múltiplo de instruções/Fluxo único de dados). Envolve múltiplos processadores executando diferentes instruções em um único conjunto de dados. Diferentes instruções operam a mesma posição de memória ao mesmo tempo, executando instruções diferentes. Esta classe é considerada vazia, por ser tecnicamente impraticável. (Figura 3.4).

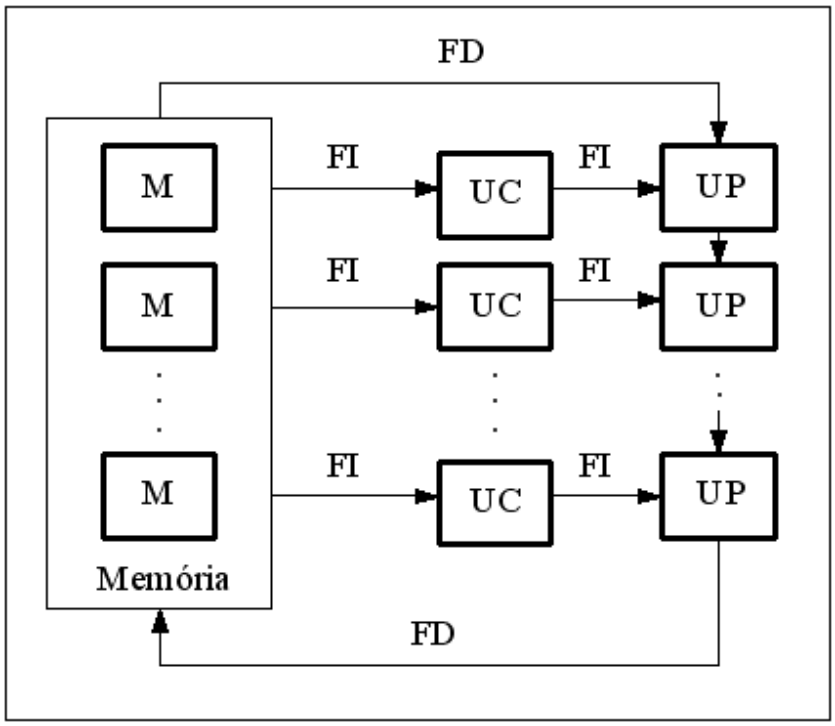


Figura 3.4: Modelo computacional MISD

3.1.1.4 MIMD

Multiple Instruction Stream/Multiple Data Stream (fluxo múltiplo de instruções/fluxo múltiplo de dados). Envolve múltiplos processadores executando diferentes instruções em diferentes conjuntos de dados, de maneira independente (Figura 3.5). Esta classe engloba a maioria dos computadores paralelos.

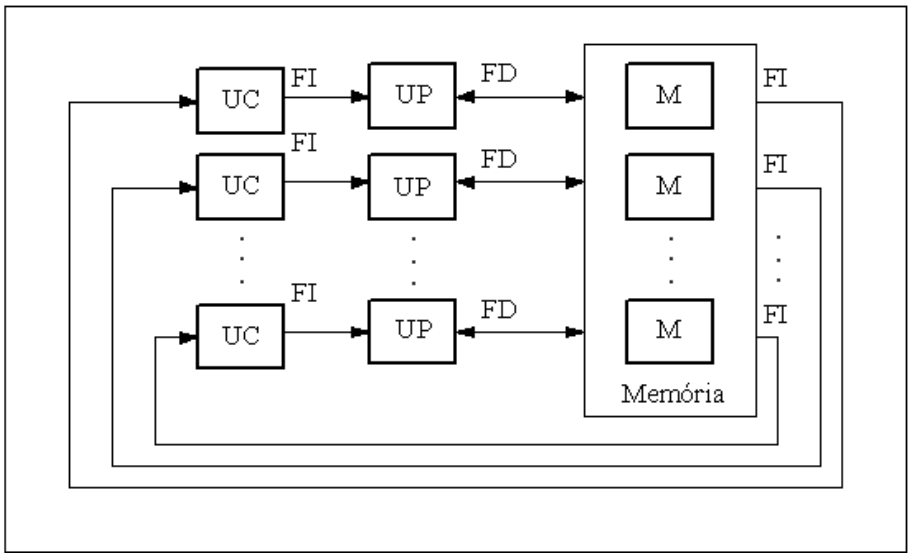


Figura 3.5: Modelo computacional MIMD

Dentro da classificação MIMD enquadram-se os seguintes modelos de arquiteturas:

Máquinas Vetoriais (PVP - *Parallel Vector Processor*) - máquinas que possuem processadores compostos de vários pipelines vetoriais com alto poder de processamento. Cray e NEC são exemplos de máquinas vetoriais.

Multiprocessadores Simétricos (SMP - *Symmetric Multiprocessing*) - são sistemas constituídos de vários processadores comerciais, conectados a uma memória compartilhada, na maioria dos casos através de um barramento de alta velocidade.

Máquinas Massivamente Paralelas (MPP - *Massively Parallel Processing*) - diversos microprocessadores interligados através de uma rede de interconexão normalmente proprietária. Cada nó de processamento da malha de interconexão pode possuir mais de um processador e podem existir máquinas com milhares destes nós. A diferença em relação aos dois últimos modelos de máquinas é que estas não possuem uma memória compartilhada.

Memória Compartilhada Distribuída (DSM - *Distributed Shared Memory*) - sistemas em que, apesar de a memória encontrar-se fisicamente distribuída através dos nós, todos os processadores podem endereçar todas as memórias. Isso se deve à implementação de um único espaço de endereçamento.

Redes de Estações de Trabalho (NOW - *Network of Workstations*) - são sistemas constituídos por várias estações de trabalho interligadas por tecnologia tradicional de rede como Ethernet e ATM (*Asynchronous Transfer Mode*). Na prática, uma rede local de estações que já

existe é utilizada para execução de aplicações paralelas.

Agregados (COW - *Clusters of Workstations*) - neste grupo enquadram-se máquinas cujo princípio básico é o emprego de uma rede de custo baixo, porém de alto desempenho, interligando nodos que podem possuir mais de um processador. Podem ser vistas como uma evolução das redes de estações de trabalho NOW, pois também são constituídas por várias estações de trabalho interligadas, mas com a diferença de terem sido projetadas com o objetivo de executar aplicações paralelas.

Grades computacionais - são ambientes para computação distribuída de alto desempenho que permitem o compartilhamento de recursos heterogêneos. Uma grade é uma coleção de recursos computacionais distribuídos sobre uma rede, que estão disponíveis a um usuário ou a uma aplicação. Grade computacional é uma infra-estrutura de software e hardware que provê serviços seguros, consistentes, de acesso penetrante a um custo relativamente acessível.

3.1.2 Classificação de Duncan

A classificação de Duncan [Dun90] surgiu da necessidade de acomodar arquiteturas mais recentes. Duncan exclui arquiteturas que apresentem apenas mecanismos de paralelismo de baixo nível (pipeline, múltiplas unidades funcionais e processadores dedicados para entrada e saída), que já se tornaram lugar comum nos computadores modernos, e mantém os elementos da classificação de Flynn, no que diz respeito ao fluxo de dados e instruções.

A classificação de Duncan apresentada na Figura 3.6, divide as arquiteturas em dois grupos principais: arquiteturas síncronas e assíncronas.

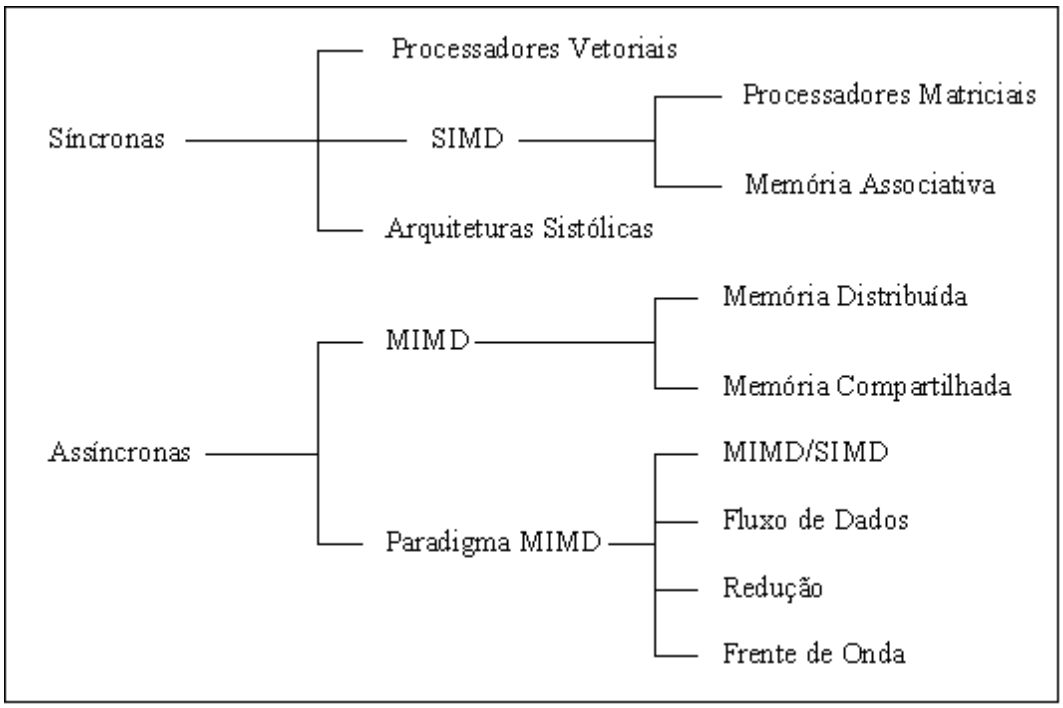


Figura 3.6: Classificação de Duncan

3.1.2.1 Arquiteturas Síncronas

Arquiteturas paralelas síncronas coordenam suas operações concorrentes sincronamente em todos os processadores, através de relógios globais, unidades de controle únicas ou controladores de unidades vetoriais [Dun90]. Tais arquiteturas apresentam pouca flexibilidade para a expressão de algoritmos paralelos [Ble94].

- Processadores Vetoriais: são caracterizados por possuírem um hardware específico (múltiplas unidades funcionais organizadas utilizando pipeline) para a otimização de operações efetuadas sobre vetores.

- Arquiteturas SIMD: arquiteturas SIMD apresentam múltiplos processadores, sob a supervisão de uma unidade central de controle, que executam a mesma instrução sincronamente em conjuntos de dados distintos.

- Arquiteturas Sistólicas: têm como principal objetivo fornecer uma estrutura eficiente para a solução de problemas que necessitem de computação intensiva junto a grande quantidade de operações de E/S. Essas arquiteturas se caracterizam pela presença de vários

processadores, organizados de maneira pipeline, que formam uma cadeia na qual apenas os processadores localizados nos limites desta estrutura possuem comunicação com a memória.

3.1.2.2 Arquiteturas Assíncronas

Estas arquiteturas caracterizam-se pelo controle descentralizado de hardware, de maneira que os processadores são independentes entre si. Essa categoria é formada pelas máquinas MIMD, sejam elas convencionais ou não [Dun90].

- Arquiteturas MIMD: relacionam arquiteturas compostas por vários processadores independentes, onde se executam diferentes fluxos de instruções em dados locais a esses processadores.

- Paradigma MIMD: essa classe engloba as arquiteturas assíncronas que, apesar de apresentarem a característica de multiplicidade de fluxo de dados e instruções das arquiteturas MIMD, são organizadas segundo conceitos tão fundamentais a seu projeto quanto suas características MIMD. Estas características próprias de cada arquitetura, dificultam a sua classificação como puramente MIMD. Por isso, tais arquiteturas se denominam paradigmas arquiteturais MIMD.

3.2 Compartilhamento de Memória

Um outro critério para a classificação de máquinas paralelas é o compartilhamento da memória.

Memória compartilhada é assim denominada quando dois ou mais processos compartilham uma mesma região de memória. É a maneira mais rápida dos processadores efetuarem uma troca de dados, porém um lugar da memória não pode ser modificado por uma tarefa enquanto outra estiver acessando. A Figura 3.7 mostra como o acesso à memória pelos processadores é feito. Máquinas SMP utilizam este modelo.

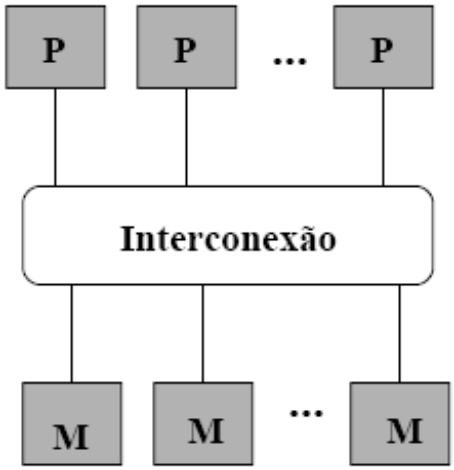


Figura 3.7: Arquitetura com memória compartilhada

Em arquiteturas de **memória distribuída**, cada processador possui sua própria memória local (Figura 3.8), sendo então fracamente acoplados. Em virtude de não haver compartilhamento de memória, os processos comunicam-se via troca de mensagens, que se trata da transferência explicita de dados entre os processadores.

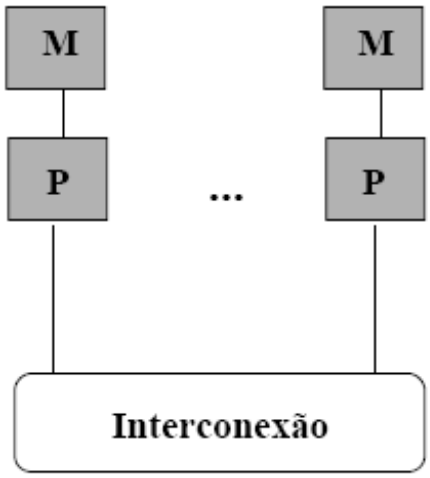


Figura 3.8: Arquitetura com memória distribuída

Dependendo de uma máquina paralela utilizar-se ou não de uma memória compartilhada por todos os processadores, pode-se diferenciar: **Multiprocessadores** ou **Multicomputadores**.

3.2.1 Multiprocessadores

Esse tipo de máquina possui apenas um espaço de endereçamento, de forma que todos os processadores P são capazes de endereçar todas as memórias M. Essas características resultam do fato de esse tipo de máquina paralela ser construída a partir da replicação apenas do componente **processador** de uma arquitetura convencional conforme mostra a Figura 3.9. Daí o nome múltiplos processadores.

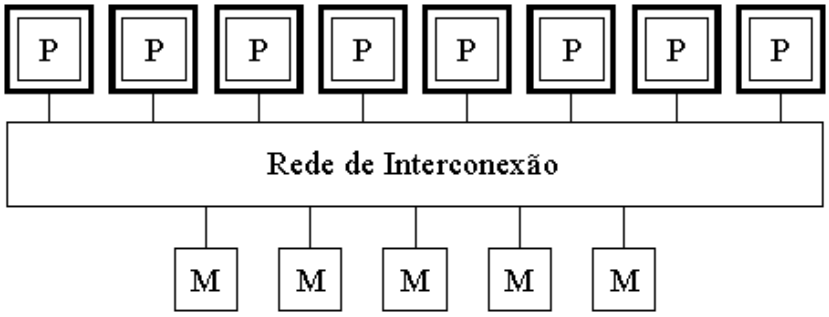


Figura 3.9: Multiprocessadores

Em relação ao tipo de acesso às memórias do sistema, multiprocessadores podem ser classificados como: UMA (*Uniform Memory Access*), NUMA (*Non-Uniform Memory Access*) e COMA (*Cache-Only Memory Architecture*).

3.2.1.1 UMA

A memória usada nessas máquinas é centralizada e encontra-se à mesma distância de todos os processadores, fazendo com que a latência de acesso à memória seja igual para todos os processadores do sistema (uniforme) (Figura 3.10). Como o barramento é a rede de interconexão mais usada nessas máquinas e suporta apenas uma transação por vez, a memória principal é normalmente implementada com um único bloco.

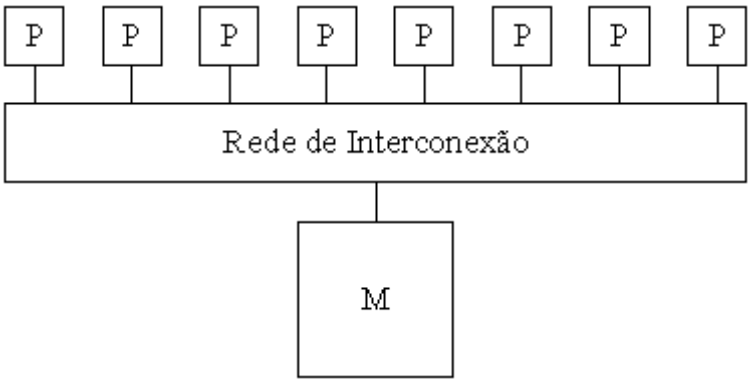


Figura 3.10: Classificação UMA

3.2.1.2 NUMA

A memória usada nessas máquinas é distribuída, implementada com múltiplos módulos que são associados um a cada processador (Figura 3.11). O espaço de endereçamento é único, e cada processador pode endereçar toda a memória do sistema. Se o endereço gerado pelo processador encontrar-se no módulo de memória diretamente ligado a ele (local) o tempo de acesso a ele será menor que o tempo de acesso a um módulo que está diretamente ligado a outro processador (remoto) que só pode ser acessado através da rede de interconexão. Por esse motivo, essas máquinas possuem um acesso não uniforme à memória.

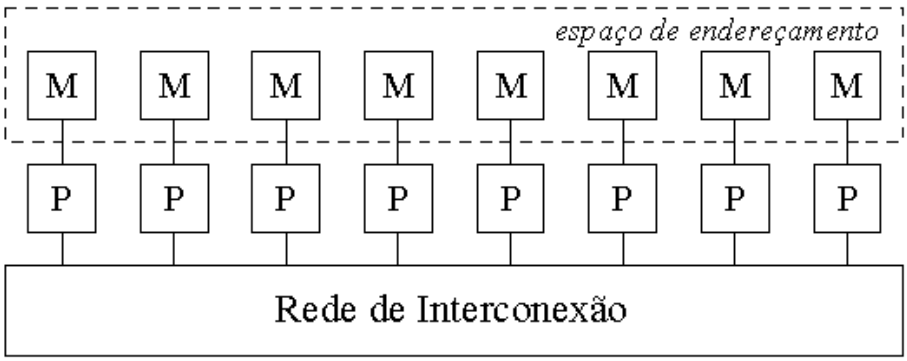


Figura 3.11: Classificação NUMA

3.2.1.3 COMA

Em uma máquina COMA, todas as memórias locais estão estruturadas como memórias *cache* e são chamadas de COMA *caches* (Figura 3.12). Essas *caches* têm muito mais capacidade que

uma *cache* tradicional. Arquiteturas COMA têm suporte de *hardware* para a replicação efetiva do mesmo bloco de cache em múltiplos nós fazendo com que essas arquiteturas sejam mais caras de implementar que as máquinas NUMA.

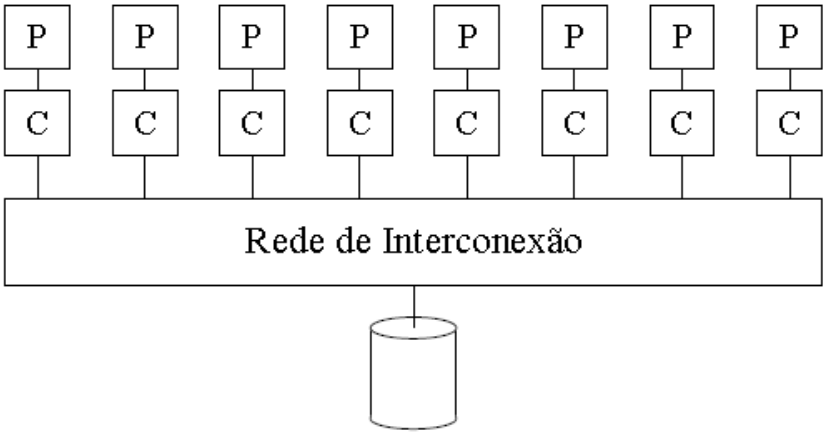


Figura 3.12: Classificação COMA

3.2.2 Multicomputadores

Cada processador P possui uma memória local M (Figura 3.13), a qual só ele tem acesso. As memórias dos outros processadores são consideradas memórias remotas e possuem espaços de endereçamento distintos. Como não é possível o uso de variáveis compartilhadas nesse ambiente, a troca de informações com outros processos é feita por envio de mensagens pela rede de interconexão. Por essa razão, essas máquinas também são chamadas de sistemas de troca de mensagens.

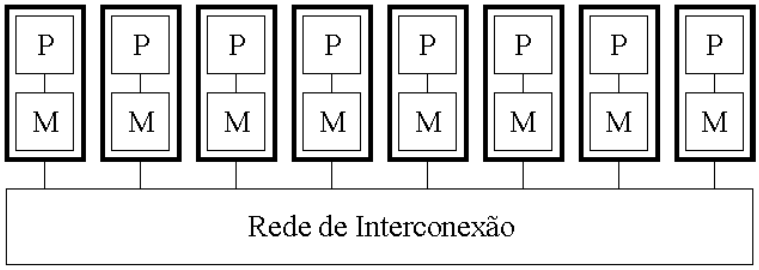


Figura 3.13: Multicomputadores

Em relação ao tipo de acesso às memórias do sistema, multicomputadores podem ser classi-

ficados como: NORMA (*Non-Remote Memory Access*).

3.2.2.1 NORMA

Como uma arquitetura tradicional inteira foi replicada na construção dessas máquinas, os registradores de endereçamento de cada nó só conseguem endereçar a sua memória local.

3.3 Modelos de Programação Paralela

Os modelos de programação paralela existem como uma camada de abstração sobre a arquitetura do hardware e da memória do computador [Bar05]. No entanto, esses modelos não são específicos de uma determinada arquitetura nem de um tipo de memória. Geralmente, a escolha do modelo a ser utilizado depende do programador, do tipo de hardware disponível e das características da aplicação.

3.3.1 Paralelismo Implícito e Explícito

No paralelismo explícito, a linguagem de programação contém mecanismos para paralelização do programa. Desta forma, o programador pode utilizar seu conhecimento empírico para explorar ao máximo o potencial de paralelização de suas aplicações. No entanto, de acordo com ([KL88]) a utilização de mecanismos explícitos pode levar a uma exploração inadequada do potencial de paralelismo. Além disso, conforme [KB88], grande parte do trabalho necessário para paralelização de programas é muito difícil para ser realizado por pessoas. Por exemplo, somente compiladores são confiáveis para realização da análise de dependências em sistemas paralelos com memória compartilhada. Por outro lado, deve-se ressaltar que o paralelismo explícito diminui a complexidade dos compiladores paralelizadores, pois elimina a necessidade da detecção automática do paralelismo em tempo de compilação.

No paralelismo implícito, a linguagem de programação não contém mecanismos para paralelização dos programas. A principal vantagem deste método consiste na liberação do programador do envolvimento com a paralelização de suas aplicações. Além disso, o paralelismo implícito aumenta a portabilidade de programas entre sistemas paralelos, eliminando a necessidade da alteração do código fonte em função da arquitetura paralela a ser utilizada. Outra característica interessante da exploração automática consiste no aproveitamento tanto dos programas seqüenciais já existentes quanto dos ambientes de desenvolvimento (depuração) direcionados para o

paradigma seqüencial.

3.3.2 Paralelismo de Dados

O paralelismo de dados representa o uso de múltiplas unidades para se aplicar a mesma operação simultaneamente em um dado conjunto de elementos. Segundo [Qui94], K unidades de processamento adicionais geram um aumento de vazão de K vezes no sistema. Por vazão, entende-se o número de resultados obtidos por ciclo de tempo. A execução deste tipo de algoritmo pode ser verificada, por exemplo, em algoritmos paralelos de multiplicação de matrizes.

3.3.3 Paralelismo de Controle

O paralelismo de controle, diferentemente do paralelismo de dados onde o paralelismo é atingido através de diversas unidades de processamento executando uma única instrução, atinge o paralelismo através da aplicação de diferentes operações a diferentes conjuntos de dados simultaneamente. Conforme [Qui94], o fluxo de dados sobre este processo pode ser arbitrariamente complexo. No paralelismo de controle, a computação é dividida em passos, chamados segmentos ou estágios, que são distribuídos entre os processadores. Cada segmento realiza uma parte do processamento, e pode ser possível que a entrada de um segmento seja a solução gerada na saída do segmento anterior. Por exemplo, a modelagem de um ecossistema, onde cada programa calcula a população de um determinado grupo que depende dos vizinhos como mostrado na Figura 3.14.

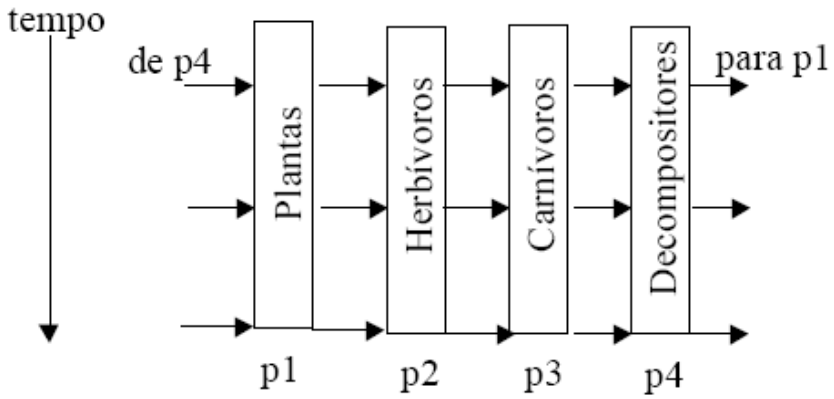


Figura 3.14: Exemplo de paralelismo de controle

3.3.4 Troca de Mensagens

O desenvolvimento de programas paralelos e distribuídos encontra na programação baseada em troca de mensagens, uma abordagem eficaz para explorar as características das máquinas de memória distribuída. Com o uso de *clusters* e de bibliotecas de suporte às trocas de mensagens, como o padrão MPI (*Message Passing Interface*), aplicações eficientes e economicamente viáveis podem ser construídas. MPI é uma biblioteca que contém funções para implementar programas que executam trocas de mensagens em uma ambiente distribuído. Estes programas rodam em um cluster e o ambiente MPI se encarrega da distribuição destes processos.

Existem implementações de MPI para diversas plataformas de hardware e software. Isto quer dizer que é possível montar um *cluster* com nós de diferentes arquiteturas e usar MPI para resolver um problema de maneira distribuída. Uma utilidade imediata disto seria utilizar arquiteturas especializadas em um tipo de processamento para resolver partes do problema que devem assim ser abordados. No entanto isto imediatamente nos leva a nos perguntarmos como podemos trocar mensagens entre máquinas de arquiteturas diferentes, que possuem tipos internos de dados diferentes. Para resolver este problema, MPI define seus próprios tipos básicos, que são independentes da arquitetura real da máquina. MPI se encarrega de converter esses tipos de dados para os tipos de dados internos.

3.4 Modelos de Algoritmos Paralelos

Existem vários modelos de programação paralela que podem ser escolhidos pelo programador para estruturar ou organizar o desenvolvimento de programas. A escolha de um ou de outro depende das características da aplicação, dos recursos computacionais disponíveis para quem vai desenvolver o programa e do tipo de paralelismo encontrado no problema. Nas próximas seções, é explicado, resumidamente, alguns dos paradigmas mais comumente utilizados na implementação de programas paralelos [NBvO01].

3.4.1 Divisão e Conquista

Um algoritmo de divisão e conquista primeiramente divide o problema original em diversos subproblemas, que são mais fáceis de se resolver do que o original, e então resolve os subproblemas, geralmente recursivamente. Finalmente o algoritmo mescla as soluções dos subproblemas para construir uma solução de um problema original.

3.4.2 *Pipeline*

No paradigma pipeline um número de processos forma um *pipeline* virtual. Os processos podem formar esses *pipelines* de uma maneira linear, multidimensional, cíclica ou acíclica. Um fluxo contínuo de dados entra no primeiro estágio do *pipeline* e os processos são executados nos demais estágios complementares, de forma simultânea. Cada processo no *pipeline* pode ser visto como um consumidor de uma seqüência de dados precedendo-o no *pipeline* e como produtor de dados sucedendo-o no *pipeline*.

3.4.3 Mestre/Escravo

Neste paradigma, um ou mais processos mestre executam as tarefas essenciais do programa paralelo e dividem o resto das tarefas para os processos escravos. Quando um processo escravo termina sua tarefa, ele informa o mestre que atribui uma nova tarefa para o escravo. Este paradigma é bastante simples, visto que o controle está centralizado em um processo mestre. Sua desvantagem é que o mestre pode se tornar o gargalo na comunicação. Isso acontece quando as tarefas são muito pequenas (ou escravos são relativamente rápidos).

3.4.4 *Pool* de Trabalho

Neste modelo, um *pool* (conjunto) de tarefas é disponibilizado por uma estrutura de dados global e um determinado número de processos é criado para executar esse conjunto de tarefas. No início só existe um único pedaço de tarefa; gradativamente os processos buscam pedaços da tarefa e imediatamente passam a executá-los, espalhando o processamento. O programa paralelo termina quando o *pool* de trabalho fica vazio.

3.4.5 Fases Paralelas

Neste modelo, a aplicação consiste em um número de etapas, onde cada etapa é dividida em duas fases: uma fase de computação, quando os múltiplos processos executam processamentos independentes; seguida de uma fase de interação, quando os processos executam uma ou mais operações de interação síncrona, tais como barreiras ou comunicações bloqueantes.

3.5 Critérios de Avaliação

Uma característica fundamental da computação paralela trata-se do aumento de velocidade de processamento através da utilização do paralelismo. Neste contexto, duas medidas muito importantes, dentre várias outras, para a verificação da qualidade de algoritmos paralelos são aceleração (*speedup*) e eficiência.

Aceleração é o aumento de velocidade observado quando se executa um determinado processo em p processadores em relação à execução deste processo em um único processador.

$$Speedup = \frac{T1}{Tp}$$

Onde, $T1$ = tempo de execução em 1 processador (serial)

Tp = tempo de execução em p processadores (paralela)

O ganho de *speedup* deveria tender a p, que seria o seu valor ideal 1. Outra medida importante é a **eficiência**, que trata da relação entre o *speedup* e o número de processadores. Tal medida é obtida através da seguinte fórmula:

$$Eficiência = \frac{speedup}{Np}$$

Np é o número de processadores utilizados para executar o programa paralelo.

Dada as fórmulas acima, nota-se que o *speedup* ideal deve ser igual a quantidade de processadores utilizados no programa paralelo. A eficiência deve estar entre zero e um, pois indica um valor relativo. Se for alcançado um *speedup* ideal também é alcançada a eficiência ideal que é igual a 1 (indicando 100% de eficiência).

3.6 Fatores de Desempenho

3.6.1 Granularidade

A granularidade de um sistema paralelo corresponde ao tamanho das unidades de trabalho submetidas aos processadores. Isto acaba influenciando na determinação do porte e da quantidade de processadores, uma vez que existe uma relação entre esses dois fatores.

Em uma linguagem seqüencial, a unidade de paralelismo é todo o programa. Em uma linguagem paralela, entretanto, a unidade de paralelismo pode ser definida, em ordem decrescente

de granularidade, como um processo, um objeto, um comando, uma expressão ou uma cláusula [Hwa93].

O nível de granularidade varia de fina (muito pouco processamento por comunicação de *byte*) a grossa. Quanto mais fina a granularidade, menor a aceleração devido à quantidade de sincronização exigida.

3.6.2 Portabilidade

Portabilidade é a capacidade que um *software* tem de ser compilado ou executado em diferentes arquiteturas de sistemas computacionais (diferentes arquiteturas de *hardware* ou de sistema operacional).

3.6.3 Escalabilidade

Escalabilidade é a capacidade de evoluir um *software* ou fazer com que o mesmo obtenha recursos adicionais sem perda de desempenho em sua funcionalidade.

Capítulo 4

Definições Gerais

Neste Capítulo é apresentado uma análise do problema enfrentado atualmente com a versão seqüencial da ferramenta FOP. Além disso, um posicionamento em relação ao embasamento descrito nos Capítulos anteriores assim como uma descrição do ambiente de testes e *hardware* utilizados na obtenção dos resultados.

4.1 Análise do Problema

Impressoras digitais atualmente encontradas no mercado têm velocidade de rasterização que chegam a cerca de 60 páginas por minuto, que significa cerca de uma página por segundo. Isto é possível se a página já esteja representada em um formato que a impressora possa consumir, ou seja, já tenha passado pelo processo de renderização e também rasterização.

O conteúdo variável de uma página representado em XSL-FO varia de acordo com a publicação desenhada pelo *designer*. Isto significa que uma página pode conter somente um único XSL-FO a ser renderizado em um documento PPML ou vários. É importante que o processo de renderização sustente este tipo de desempenho médio para que a impressora consiga atingir seu potencial máximo de impressão.

Na Figura 4.1, podemos notar que após o processo de renderização há ainda outro processo denominado rasterização que irá justamente converter o documento PPML na linguagem da impressora. Entretanto, este processo é bem mais rápido do que a fase de renderização não ameaçando o desempenho da impressão. De modo contrário, dependendo da quantidade de *copy-holes* contendo dados variáveis em XSL-FO, a fase de renderização pode tornar-se um gargalo.

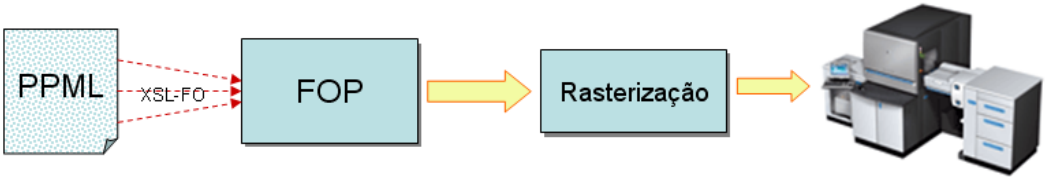


Figura 4.1: Processo de impressão de documentos em impressoras digitais

Na versão seqüencial da ferramenta FOP, somente um XSL-FO pode ser enviado por vez ficando o processo de renderização parado até que o XSL-FO enviado seja completamente renderizado e realocado em sua posição de origem no documento PPML (Figura 4.2). Em casas de impressão de grande porte, o número de *copy-holes* com conteúdo variável pode facilmente chegar a milhões. Por esse motivo, é comum disparar a renderização horas antes do processo de impressão para que não se perca tempo. Muitas vezes esse processo é executado durante a noite para que a impressão ocorra sem problemas durante o dia.

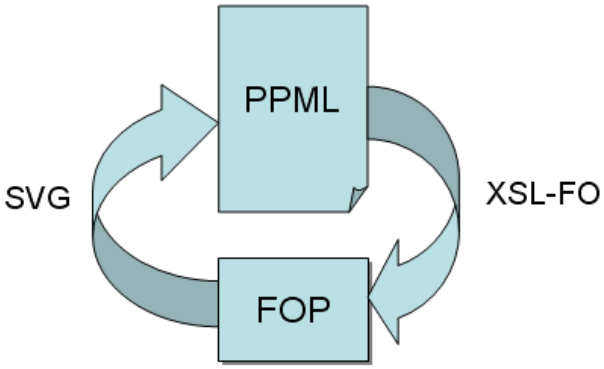


Figura 4.2: Renderização de um XSL-FO em um documento PPML

Devido à grande quantidade de dados a serem impressos, uma impressão de uma campanha de publicidade para um grande cliente pode durar muitas horas. Neste cenário, qualquer ganho de desempenho significa muito tempo do total de horas utilizado. Isso é fundamental para que um cliente decida por uma casa de impressão e não outra na hora de solicitar o serviço. Em busca desse ganho de desempenho, a proposta de paralelização da fase de renderização através do uso da ferramenta FOP torna-se uma solução de grande significado, pois aumentaria em muito a velocidade com que os documentos são renderizados dando a possibilidade de um aproveitamento maior da real capacidade de impressão das atuais impressoras digitais disponíveis no

mercado.

4.1.1 Arquitetura Atual

No Capítulo 2, vimos que a ferramenta FOP renderiza FOs em diferentes formatos de saída, e também que um documento PPML pode conter vários FOs . Entretanto, para que um XSL-FO seja enviado para o FOP é necessário que o mesmo seja retirado do documento PPML, enviado para a renderização, e realocado já no formato SVG na mesma posição onde encontrava-se o XSL-FO anteriormente ao processo de renderização. Na arquitetura seqüencial apresentada na Figura 4.3, é possível notar que um extrator foi adicionado justamente para que esse mecanismo de busca fosse possível. De maneira seqüencial, os FOs são extraídos pelo extrator, o qual envia para o FOP o conteúdo a ser renderizado aguardando o retorno no formato SVG . Nesta arquitetura, o extrator também é responsável por realocar o conteúdo renderizado de volta no documento PPML, que é o documento final a ser impresso.

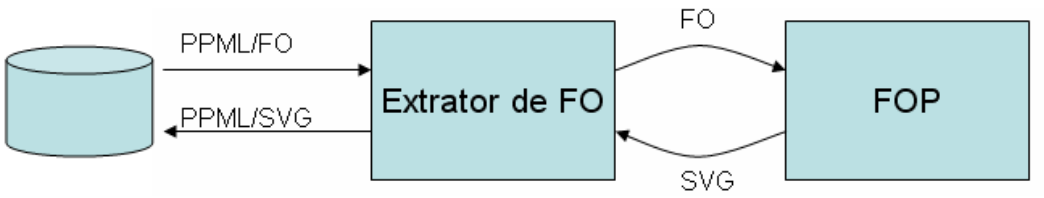


Figura 4.3: Versão seqüencial da ferramenta FOP

A arquitetura mostra que o arquivo PPML é lido e salvo em um dispositivo de disco ou qualquer outra mídia de entrada/saída. Isto será melhor apresentado no Capítulo 5, porém é importante destacarmos este fator que é de fundamental relevância para ambos os modelos: seqüencial e paralelo.

4.2 Posicionamento

O primeiro problema que um programador de aplicações de alto desempenho tem que lidar é com a escolha entre arquiteturas multiprocessadas ou multicomputadores. Máquinas multiprocessadas, como apresentado na Seção 3.1 do Capítulo 3, utilizam um esquema global de acesso à memória, e geralmente precisam de um bom barramento para interconexão entre processadores

e memória. Hoje em dia, tais máquinas estão perdendo espaço para plataformas de multicomputadores como *clusters* ou grades computacionais. Estas máquinas apresentam um esquema de memória distribuída, e no caso de *clusters*, são conectados por uma rede rápida dedicada. O desenvolvimento de aplicações para esses tipos de máquinas é bem diferente. O primeiro modelo é baseado no paradigma de memória compartilhada e o segundo é tipicamente baseado no paradigma de troca de mensagens.

Programar para plataformas com memória distribuída é mais complexo porque cada processador da arquitetura tem uma memória local e não pode acessar diretamente a memória de outros processadores. Neste cenário, a aplicação deve ser dividida em módulos, também chamados de processos, que não compartilham o mesmo espaço de endereçamento entre eles. Assim, os processos não podem trocar informações através de variáveis compartilhadas. A alternativa é prover uma série de comunicações primitivas as quais baseiam-se em duas funcionalidades principais: enviar e receber dados através de uma interconexão de rede. Apesar da grande complexidade, paradigma de programação por troca de mensagens tem a grande vantagem de um alto grau de portabilidade, visto que tais programas podem ser executados sobre plataformas de memória compartilhada sem nenhuma mudança considerando que uma inevitável perda de eficiência pode ser aceita. Por outro lado, programas com memória compartilhada têm um baixo grau de portabilidade, pois não podem ser executados em plataformas com memória distribuída. Tal fato somente será possível através de uma completa conversão dos programas para o paradigma de troca de mensagens.

Considerando que portabilidade e escalabilidade são funcionalidades desejáveis em implementações de alto desempenho, decidimos adotar a linguagem de programação *Java* em nossa implementação. *Java* não é freqüentemente utilizada para esse tipo de aplicações [GHM98, MMG+00] por duas razões: é uma linguagem interpretada e é baseada em um ambiente virtual (JVM - *Java Virtual Machine*), que garante a portabilidade. Estes dois fatores são responsáveis por um custo computacional que na maioria das vezes é considerado muito significativo pelos desenvolvedores de aplicações de alto desempenho. Entretanto, nesta implementação, portabilidade e compatibilidade com diferentes sistemas operacionais são cruciais.

Para o desenvolvimento deste trabalho foi utilizado o *Java Standard Development Kit* (J2SDK, versão 1.4.2) e o modelo de programação paralela por passagem de mensagens com utilização da biblioteca MPI [SOHL+96] para realizar a comunicação entre os processos. Mais especificamente, foi escolhida a implementação mpich (versão 1.2.6) juntamente com *mpi.Java* [mpi05]

(versão 1.2.5) que é uma implementação *Java* orientada a objetos para o padrão MPI . O modelo de algoritmo paralelo escolhido foi o mestre/escravo, visto que em todas as arquiteturas desenvolvidas há sempre um módulo mestre e escravos, no caso as ferramentas FOP rodando em paralelo. Os experimentos foram realizados em processadores rodando *Linux* (distribuição *Slackware*, *kernel* 2.4.29), visto que era a configuração de *hardware* disponível para testes. Entretanto, é importante mencionar que *mpi.Java* também é compatível com o sistema operacional *Windows*, assegurando a portabilidade.

4.3 Plataformas de Hardware

Os resultados apresentados neste trabalho foram obtidos em dois diferentes agregados: Amazônia e Ombrófila. Ambos instalados no CPAD (Centro de Pesquisa de Alto Desempenho) sob coordenação do professor César De Rose, que disponibiliza a infra-estrutura para realização de pesquisas em projetos cadastrados na área de Alto Desempenho.

4.3.1 Amazônia

Amazônia (Figura 4.4) é um agregado heterogêneo com 31 nós com as seguintes configurações:

- 8 HP Compaq dc5000 MT com processadores Pentium IV de 2.8GHz com 1GB de memória RAM.
- 8 HP NetServers E800 cada um com 2 processadores Intel Pentium III 1GHz e 256MB de memória RAM
- 8 HP NetServers E60 cada um com 2 processadores Intel Pentium III 550MHz e 256MB de memória RAM
- 2 HP workstation zx2000 cada um com 1 processador Intel Itanium2 900MHz e 1GB de memória RAM
- 5 HP Integrity rx2600 cada um com 2 processadores Intel Itanium2 1.5GHz com 2GB de memória RAM

Utiliza uma rede de alto desempenho *Myrinet* para comunicação das aplicações e uma rede *Fast-Ethernet*.

Figura 4.4: Amazônia

Para os testes realizados neste agregado, foram utilizadas 8 máquinas com duplo processador *Pentium* IV 1Ghz com 1GB de memória RAM conectadas por uma rede *FastEthernet* de 100MB.

4.3.2 Ombrófila

O agregado Ombrófila (Figura 4.5) é composto de 16 máquinas HP e-pc com processador *Pentium* III de 1GHz, 256 MB de memória e 20GB de disco. Utiliza uma rede *Fast-Ethernet* para comunicação das aplicações.

Figura 4.5: Ombrófila

Para os testes realizados neste agregado, foram utilizadas 8 máquinas conectadas por uma rede 100MB *FastEthernet*.

4.4 Casos de Estudo

Arquivos PPML podem conter ou referenciar uma grande quantidade de diferentes objetos que vão de vários tipos de imagens a documentos PDF e PostScript, e linguagens baseadas em XML como SVG e XSL-FO. Contudo, neste trabalho o foco principal não é destacar o potencial da linguagem PPML, mas sim a capacidade da ferramenta FOP em sua versão paralela de renderizar uma grande quantidade de XSL-FOs . Logo, para a realização dos testes os mesmos documentos foram replicados n vezes em um único *job* em um arquivo PPML, ou seja, os mesmos XSL-FOs com o mesmo conteúdo são enviados para o FOP.

O primeiro arquivo PPML de entrada, chamando **Mini**, contem um *job* com **mil** documentos a serem renderizados. Cada documento é composto por duas páginas como mostra a Figura 4.6 distribuídas da seguinte forma:

- Página 1: 1 *copy-hole* com XSL-FO composto por 4 blocos de texto e aproximadamente 107 palavras.
- Página 2: 3 *copy-holes* com XSL-FO, respectivamente composto por 6 blocos de texto e aproximadamente 130 palavras, 2 blocos de texto e aproximadamente 43 palavras e 1 bloco de texto com 36 palavras.
- Número médio de palavras por bloco: 24,3

Os número total de XSL-FOs a serem renderizados contidos nos *copy-holes* no documento PPML somam quatro mil. Lembrando que cada referência *fo:* dentro de um *copy-hole* é considerado um FO renderizado. Os documentos PPML são instâncias do modelo mostrado na Figura 4.6.

O segundo teste (**CAP**) tem **dois mil** documentos. O documento tem duas páginas, cada uma com as seguintes características:

- Página 1: 3 *copy-holes* com XSL-FO, todos com 1 bloco de texto, respectivamente com 4 palavras, 6 palavras e 7 palavras.
- Página 2: 3 *copy-holes* com XSL-FO, respectivamente com 4 blocos de texto e 56 palavras, 1 bloco de texto e 6 palavras e 1 bloco de texto com 2 palavras.
- Número médio de palavras por bloco: 9

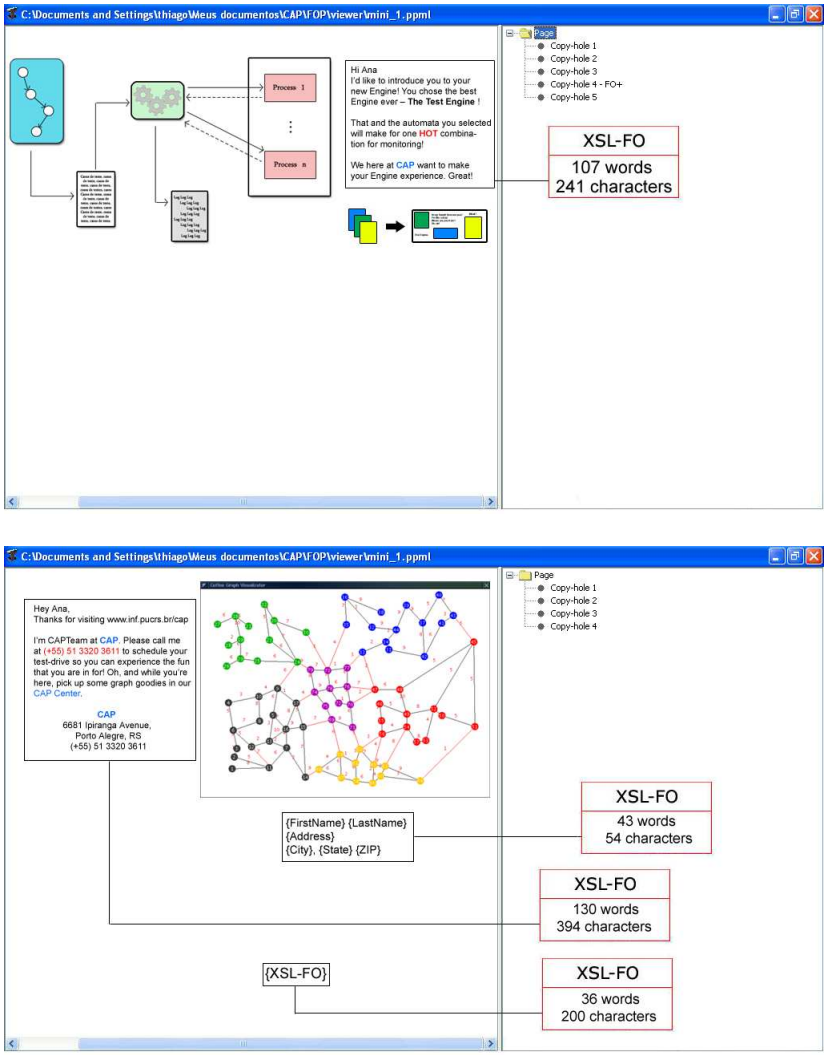


Figura 4.6: Exemplo de documento gerado pelo PPML **Mini**

O número total de XSL-FOs a serem renderizados chegam a 12000. O modelo mostrado na Figura 4.7 foi usado para criar este arquivo de entrada.

O terceiro teste é denominado **Appl**. Tem um *job* com **mil** documentos. Cada documento contem três páginas como segue:

- Página 1: 2 *copy-holes* com XSL-FO, ambos compostos somente por 1 bloco de texto cada, respectivamente com 11 palavras e com 13 palavras.
- Página 2: 1 *copy-hole* com XSL-FO, quem contem 1 bloco de texto e 32 palavras.
- Número médio de palavras por bloco: 18,67

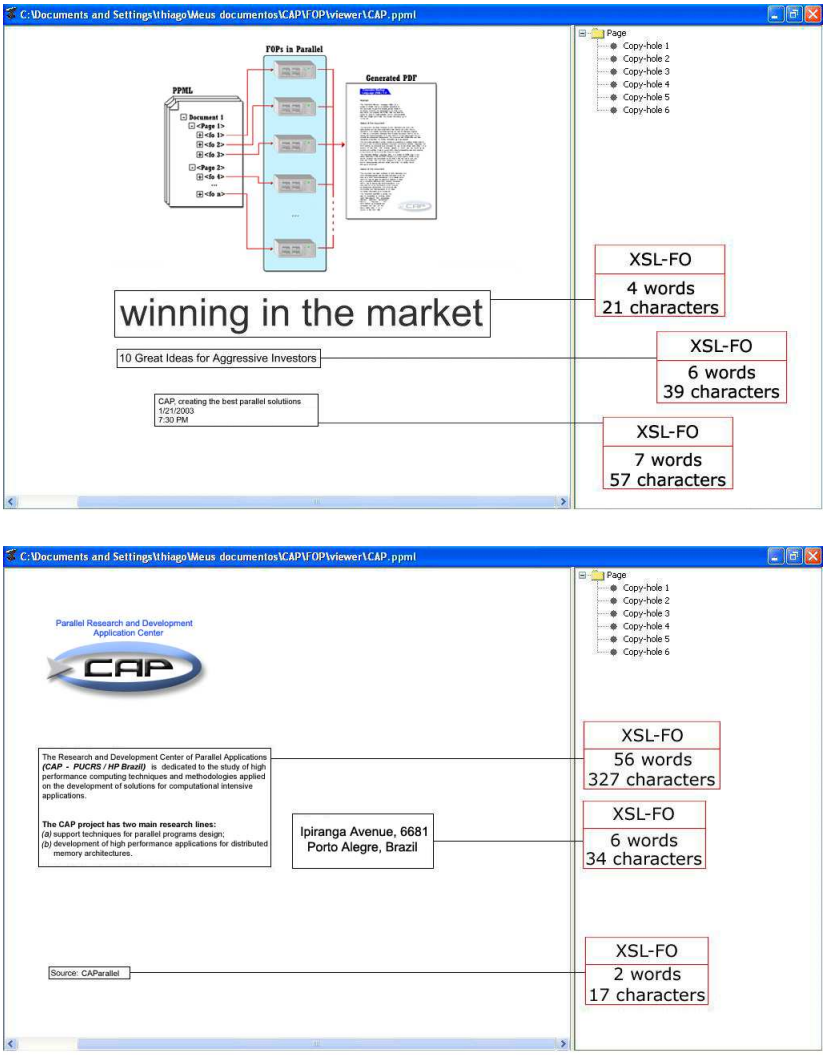


Figura 4.7: Exemplo de documento gerado pelo PPML **CAP**

Assim, o número de fragmentos XSL-FO a serem renderizados chega a 3000. Tal entrada foi gerada usando o modelo mostrado na Figura 4.8.

O último teste é idêntico ao terceiro mas com um *job* de **dois mil** documentos, o que resultará em 6000 XSL-FOs a serem renderizados. Este último teste também foi gerado pelo modelo mostrado na Figura 4.8.

O tamanho dos arquivos a serem lidos e salvos da unidade de disco é apresentado na tabela 4.1. O tamanho do arquivo de saída afeta diretamente o tempo de E/S (Entrada/Saída) (Seção 5.4.1) gasto para salvar o arquivo final no disco.

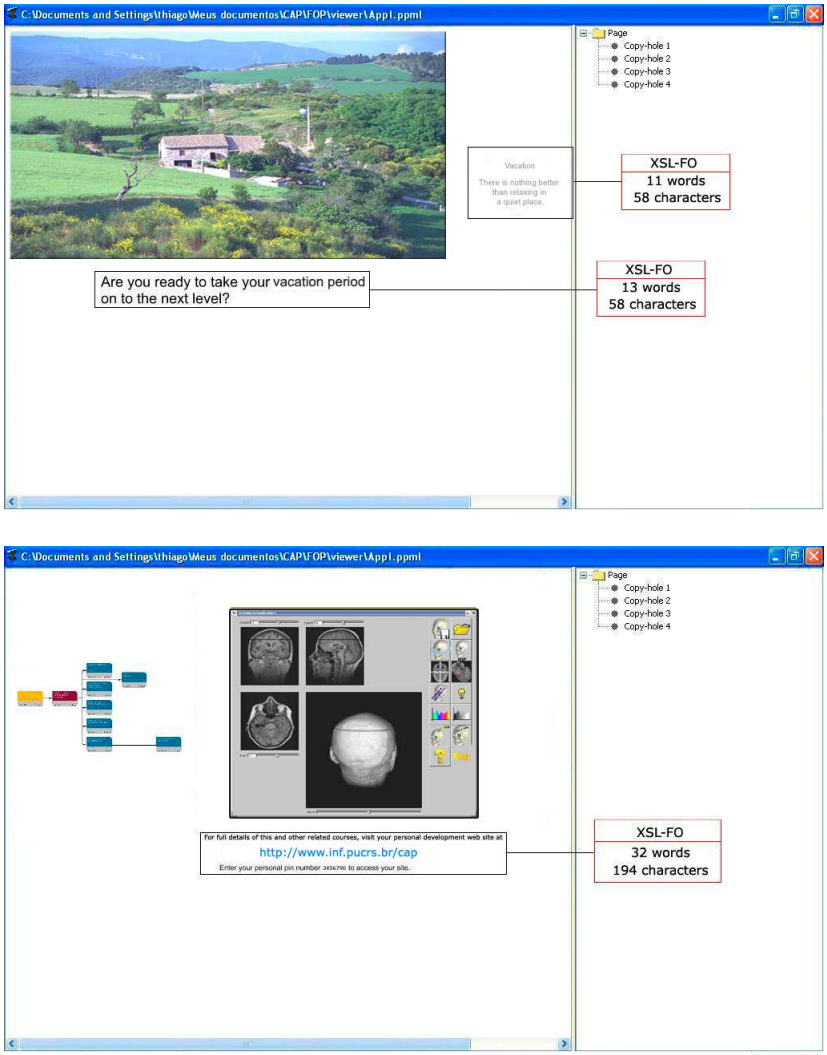


Figura 4.8: Exemplo de documento gerado pelo PPML **Appl**

Arquivo	Documentos	Tamanho não renderizado	Tamanho renderizado
Mini	1000	11MB	23MB
CAP	2000	24MB	33MB
Appl	1000	17MB	32MB
Appl	2000	33MB	62MB

Tabela 4.1: Tamanho dos arquivos PPML utilizado nos testes

Capítulo 5

Estratégias de Alto Desempenho

Neste Capítulo são apresentadas as estratégias de paralelização adotadas para a renderização de documentos XSL-FO através do uso da ferramenta FOP. Para cada estratégia é descrito, resumidamente, como a implementação se desenvolveu seguido dos resultados obtidos em cada arquitetura apresentada.

5.1 Estratégia Inicial

Tanto na versão seqüencial do FOP como na solução de alto desempenho, o documento de saída gerado após a renderização, é composto pela mesma estrutura do PPML de entrada, porém com os FOs substituídos por sua correspondente versão renderizada, conforme descrito no Capítulo 4, Seção 4.1.

Na versão seqüencial, a parte do documento PPML que não é renderizável (parte estática) é automaticamente copiada para o PPML de saída até o momento em que um *copy-hole* com conteúdo XSL-FO é encontrado. Este é enviado para o FOP que retorna SVG salvo no PPML de saída. Entretanto, na versão de alto desempenho este processo de envio e espera pela renderização não é possível, já que o extrator de FOs não pára a busca por XSL-FOs assim que o primeiro é encontrado. Pelo contrário, ao encontrá-lo já o envia para o FOP e segue a busca no documento por mais *copy-holes* contendo XSL-FOs . Para lidar com o recebimento de vários FOs enviados pelo extrator, que na arquitetura mostrada na Figura 5.1 aparece como consumidor PPML, foram adicionados ao esquema FOPs rodando em paralelo. Com vários FOs sendo enviados para o FOP e SVGs retornando para serem realocados no PPML (para que o consumidor PPML soubesse onde realocá-los) fez-se necessário a criação de um identificador

único para conteúdo enviado para renderização. Assim, o arquivo PPML de saída é gerado da seguinte forma: o consumidor PPML varre o documento em busca de *copy-holes* com conteúdo renderizável. Aquilo que não é renderizável já vai sendo gravado no documento de saída em memória. Quando um XSL-FO é encontrado, um identificador é gerado e o XSL-FO enviado para o FOP, e no documento de saída abre-se uma lacuna esperando que o SVG com o identificador correspondente retorne para que seja realocado em sua posição. Como a busca por XSL-FOs prossegue, o arquivo segue sendo gerado. À medida que os FOs renderizados vão retornando, entram em uma fila para que o consumidor verifique qual o identificador correspondente à primeira lacuna no documento. Caso seja encontrado, o SVG é imediatamente re-inserido fechando aquela lacuna. Caso não seja encontrado, a fila de FOs renderizados cresce até que o esperado seja enviado por um dos FOPs . Quando não há mais FOs na fila, o documento é transposto da memória para a unidade de disco.

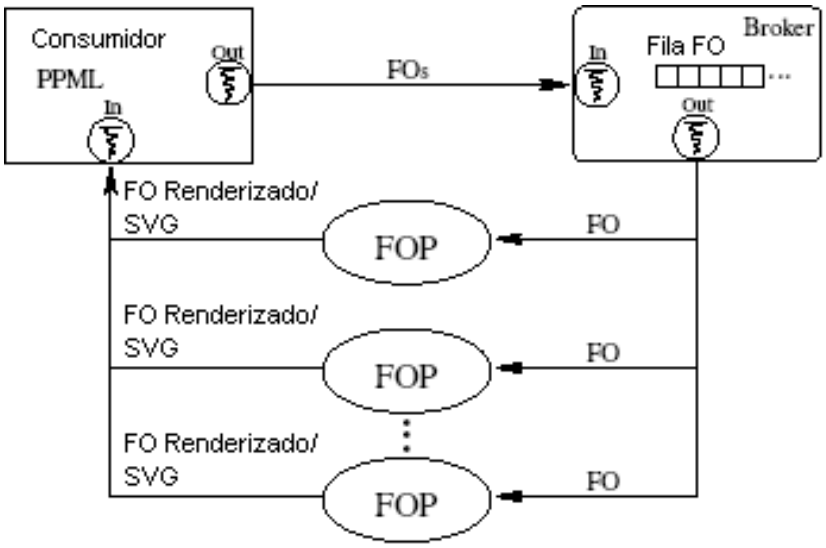


Figura 5.1: Solução inicial

5.1.1 Implementação

Para a implementação dessa arquitetura, três módulos são necessários: o consumidor PPML (*PPML consumer*), *broker* e a própria ferramenta FOP. Nos dois primeiros módulos, fez-se necessário o uso de *threads* para lidar com as várias requisições de comunicação em paralelo. A Figura 5.1 mostra duas *threads* de entrada e saída no módulo *broker* e uma *thread* para receber

os FOs renderizados no consumidor PPML. Este último, é responsável por analisar o arquivo PPML de origem removendo os FOs e enviá-los para o *broker*. A *thread* de recebimento salva o conteúdo estático (parte não renderizada) do PPML em memória, e recebe os FOs renderizados enviados pelos módulos FOP realocando-os em sua posição de origem. O *broker* é responsável por receber e enfileirar os FOs a serem renderizados. Estes FOs devem ser enviados para o componente FOP que requisitou trabalho. De forma a obter um melhor desempenho, este componente foi dividido em duas *threads*:

1. *receiver* (*in*): responsável por receber e enfileirar os FOs a serem renderizados;

2. *sender* (*out*): verifica se existe algum FO esperando para ser enviado na fila de FOs e o envia para o primeiro módulo FOP ocioso.

O módulo FOP é o responsável por renderizá-los, e quando este processo é finalizado, o resultado é enviado de volta para a *thread* de recebimento do consumidor PPML, que também notifica os módulos FOP de que está pronta para receber outro FO .

Um comentário final sobre a implementação do processo de renderização XSL-FO está relacionado ao uso das *threads*. Sistemas de programação concorrente usando *threads* introduzem problemas relacionados ao acesso simultâneo de recursos compartilhados. Um sistema é denominado *thread-safe* se este está salvo para chamar múltiplas *threads* mesmo que em paralelo. O contrário pode causar comportamentos imprevisíveis e gerar resultados inesperados, corromper estruturas de dados internas, etc. Em Java, uma implementação chamada *thread-safe* é alcançada com:

1. uso de métodos sincronizados;

2. imutabilidade de dados encapsulados, ou seja, não é possível modificar nenhum campo depois que o objeto for criado.

5.1.2 Resultados

Alguns experimentos foram executados a fim de que as vantagens e desvantagens da abordagem descrita na Seção anterior fossem apontadas. Esta Seção apresenta os resultados destes experimentos que utilizaram o documento XSL-FO **Mini** apresentados na Seção 4.4 como entrada. Buscando um parâmetro de comparação, a versão seqüencial da ferramenta de renderização foi

executada utilizando um processador do agregado Amazônia descrito no Capítulo 4 Seção 4.3, resultando em um tempo de execução de 350,05 segundos. Cada tempo de execução apresentado nesta Seção foi obtido após 5 execuções descartando o maior e o menor valor encontrado.

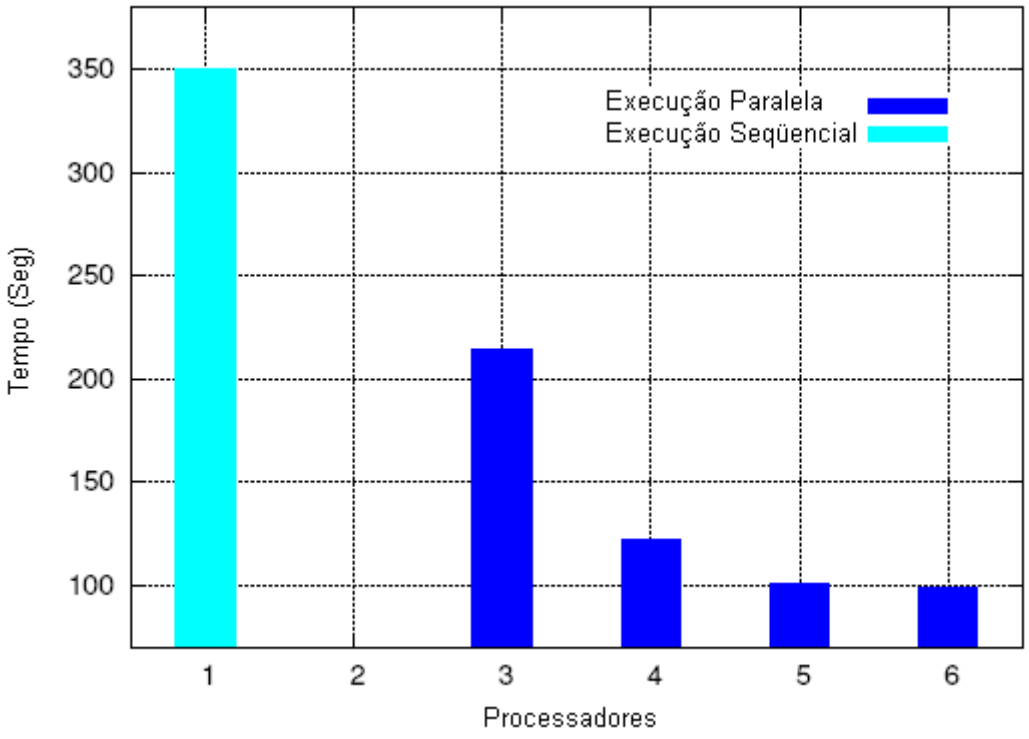


Figura 5.2: Resultados: seqüencial e versão rodando em paralelo com até 6 processadores

O primeiro conjunto de experimentos foi executado com a seguinte configuração dos módulos: um consumidor PPML, um *broker*, e de um a quatro módulos FOP. Em cada configuração, cada módulo foi exclusivamente designado para um processador do agregado. Os resultados deste experimento são mostrados na Figura 5.2. Como pode ser observado, o tempo de execução cai de 350,05 segundos para menos de 100 segundos (mais precisamente 98,23) com quatro módulos FOP executando em paralelo em diferentes processadores.

A análise dos resultados revelam as diferenças entre a versão seqüencial e a versão de alto desempenho usando somente três processadores (somente 1 módulo FOP). Embora o segundo não apresente módulos FOP rodando em paralelo, um melhor tempo de execução é alcançado apesar do custo de comunicação introduzido pelo agregado. Isto pode ser explicado pela modificação adicionada no procedimento de leitura e escrita dos arquivos de entrada e saída descritos

na Seção 5.1.1. Os benefícios reais da versão de alto desempenho começam a aparecer no experimento com quatro processadores. Neste caso, existem dois módulos FOP executando em paralelo e o tempo de execução cai quase à metade da configuração anterior (121,92 segundos).

Uma pequena diferença entre o tempo de execução com três ou quatro módulos FOP (100,09 para 98,23 segundos) é outra informação interessante que podemos extrair do gráfico. Isso é um forte indício de que o módulo *broker* começa a ter problemas para escalar quando tem que lidar com mais de três módulos FOP rodando em paralelo. De modo a validar esta hipótese, mais experimentos foram executados com configurações de 5 à 14 módulos FOP (7 à 16 módulos incluindo o Consumidor PPML e o *broker*).

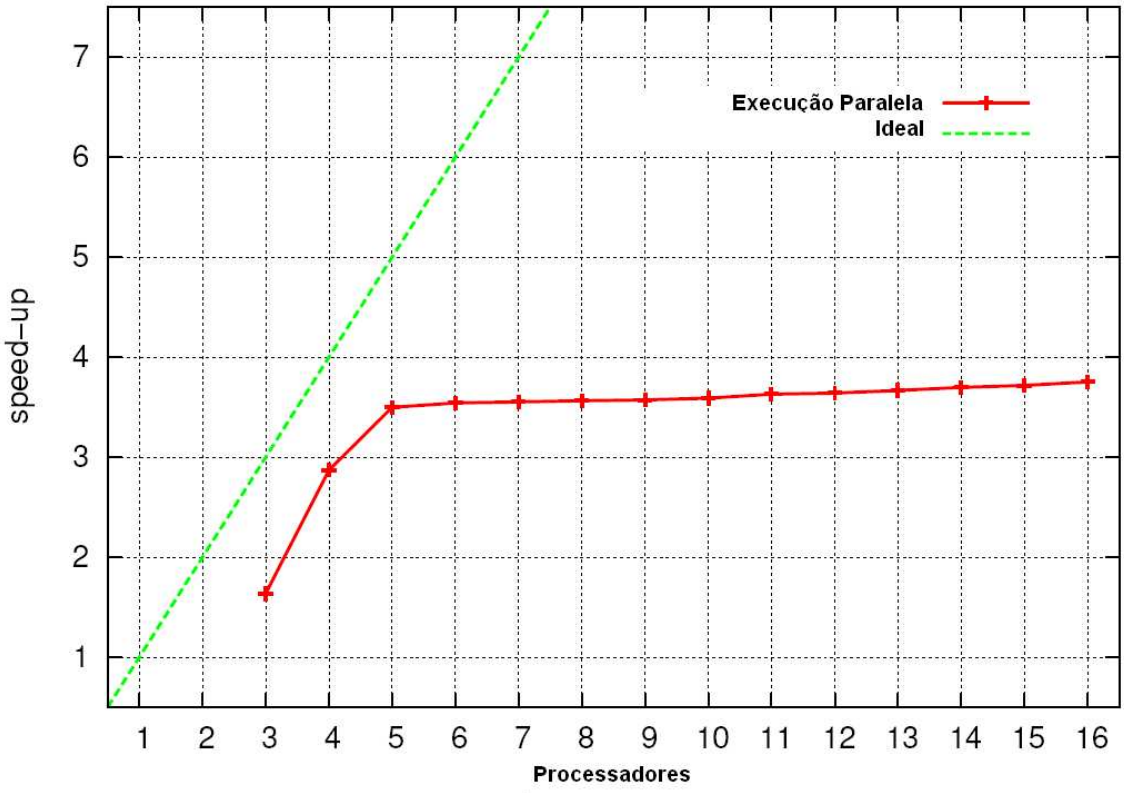


Figura 5.3: Comparação entre o ganho de desempenho (*speedup*) ideal e o alcançado pela execução da solução de alto desempenho com até 16 processadores

A Figura 5.3 evidencia a queda na eficiência à medida que o número de processadores aumenta. O gráfico mostra que o maior ganho obtido para os experimentos executados foi com 4 (71%) e 5 (69%) processadores. A partir disso, a eficiência cai gradativamente comprovando que não há ganho em usarmos todos os processadores do agregado, como pode ser observado nos

percentuais apresentados na Tabela 5.1. Isto se deve provavelmente ao aumento da comunicação entre o módulo FOP e o módulo *broker*, visto que com muitas tarefas concorrentes para executar, este torna-se o gargalo do sistema tendo que prover a comunicação com todos os módulos FOP.

	Número de CPUs													
	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tempo (seg)	214,09	121,92	100,09	98,83	98,55	98,25	98,00	97,53	96,47	96,15	95,44	94,69	94,18	93,30
Eficiência (%)	54,50	71,77	69,94	59,03	50,74	44,53	39,68	35,89	32,98	30,33	28,21	26,40	24,77	23,44

Tabela 5.1: Tabela de eficiência e tempo de execução por processador

A fim de confirmar tal suposição, medimos o tempo que os módulos FOP ficam aguardando pela comunicação. A Figura 5.4 apresenta os resultados comparando o tempo total de execução para cada configuração com o tempo gasto com a comunicação dos módulos FOP. Com uma configuração de 6 a 14 módulos FOP, o tempo gasto com comunicação por um módulo FOP representa cerca de 70% do tempo de execução.

Podemos colher outra análise importante do gráfico apresentado na Figura 5.5, o qual mostra a diferença entre o tempo de execução do módulo FOP mais rápido e do mais lento para cada configuração executada. É possível notar que à medida que o número de módulos FOP aumenta, a diferença entre o mais rápido e o mais lento cresce até que atinja uma diferença de aproximadamente 15 segundos. Nesta situação, o módulo *broker* pode não responder ao grupo de módulos FOP igualmente e por esta razão, alguns módulos FOP gastam mais tempo esperando por comunicação com o *broker* do que os demais.

Levando-se em consideração as análises feitas até agora, a configuração ideal de módulos FOP por *broker* para um conjunto de dados de entrada de mesma característica é de 1 *broker* e 3 módulos FOP. Tais descobertas estão alinhadas com o objetivo de identificar um conceito de **unidade** composto por um *broker* e um certo número de renderizadores. Esta **unidade** será usada em paralelo de acordo com a velocidade das impressoras utilizadas nas *Print Shops*. Assim, para melhorar o desempenho desta abordagem, a melhor solução seria ter uma configuração com múltiplos *brokers*, na qual o módulo consumidor PPML coordena um conjunto de módulos *broker* cada um lidando com seu próprio grupo de módulos FOP. A Figura 5.6 representa este novo esquema descrito na Seção 5.2 a seguir.

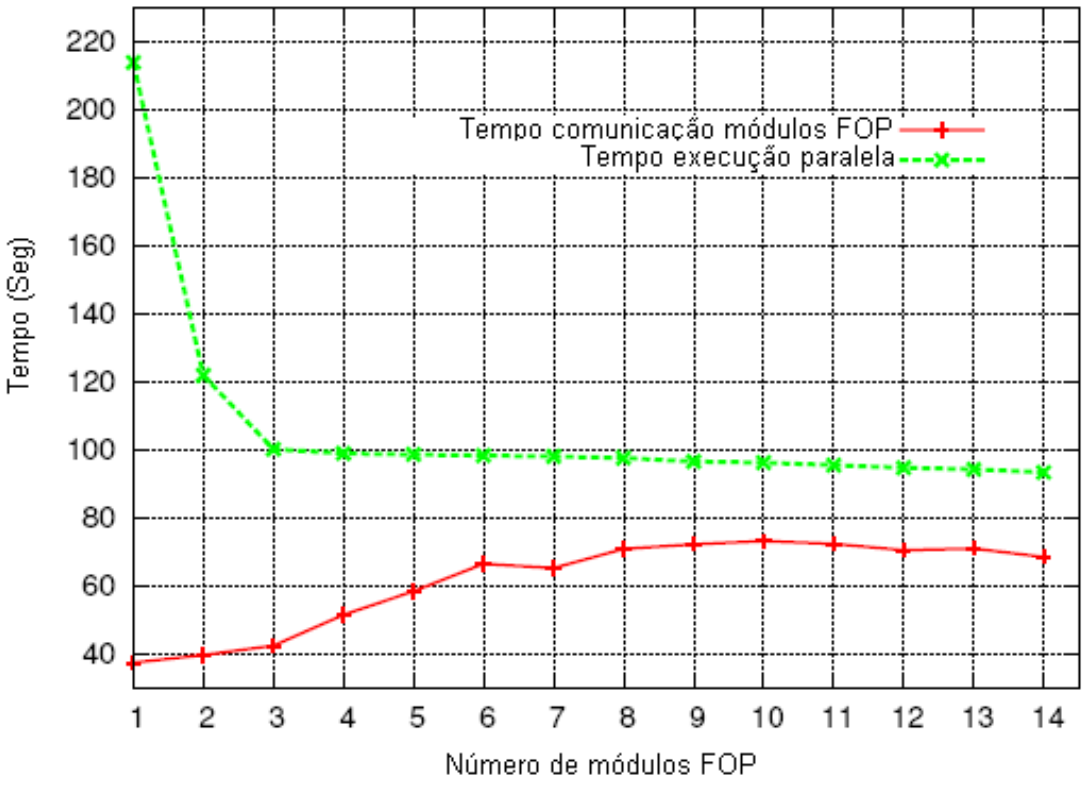


Figura 5.4: Tempo de comunicação módulos FOP

5.2 Múltiplos Brokers

Com base nos resultados apresentados anteriormente na Seção 5.1.2 e em cima da análise de que quanto maior o número de módulos FOP o módulo *broker* pode não responder igualmente devido ao tempo gasto com a comunicação, a estratégia de utilização de múltiplos *brokers* foi implementada.

5.2.1 Implementação

Basicamente, esta estratégia conforme mostrado na Figura 5.6 replica o número de *brokers* fazendo com que o consumidor PPML tenha mais opções livres ao enviar os FOs . Portanto, a funcionalidade das *threads* anteriormente explicada na Seção 5.1.1 é mantida adicionando-se somente a possibilidade do consumidor PPML enviar FOs para diferentes módulos *brokers*.

Cada *broker* seria responsável por um número fixo de módulos FOP, os quais seriam responsáveis por renderizá-los retornando-os para o consumidor PPML.

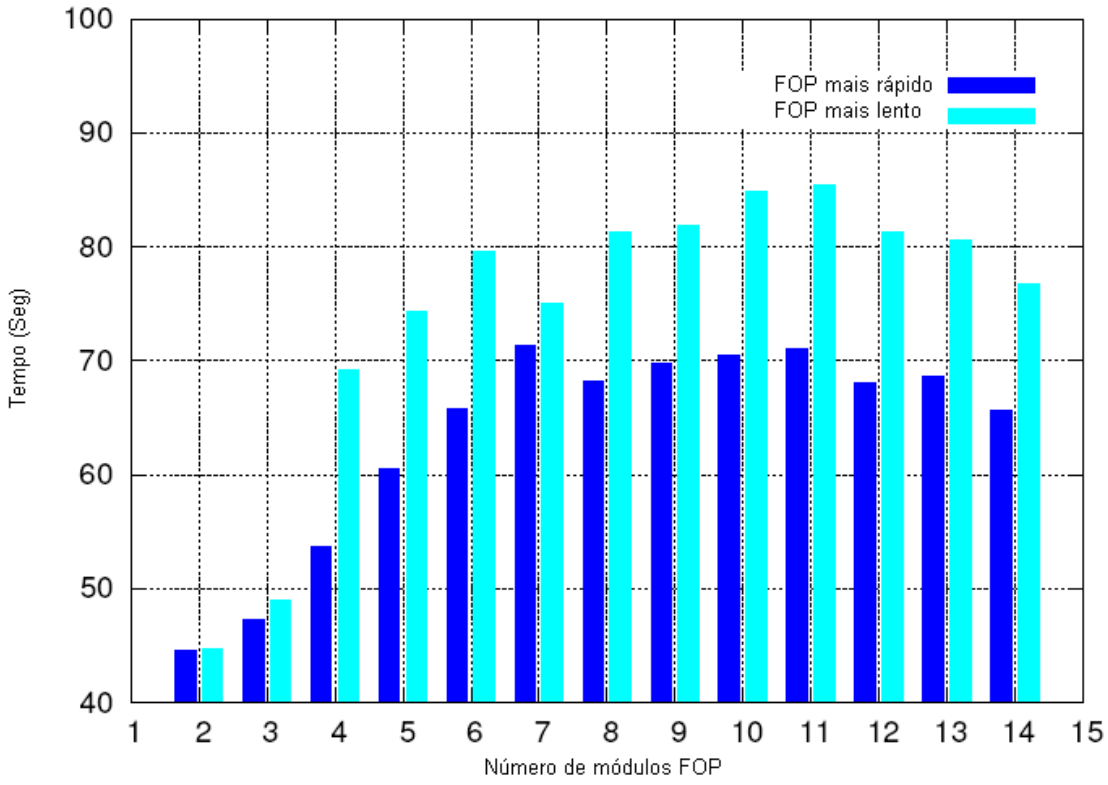


Figura 5.5: Módulos FOP não balanceados

O objetivo desta estratégia era provar que mesmo com o aumento do tempo gasto com a comunicação devido ao incremento do número de *brokers* comunicando-se com módulos FOP e conseqüentemente do número de FOs transitando, o módulo *broker*, identificado como um possível gargalo, pudesse ser aliviado e como conseqüência melhores resultados alcançados.

5.2.2 Resultados

Os resultados apresentados na Tabela 5.2, porém não confirmaram as expectativas. Como pode-se notar, com diferentes combinações de *Brokers* (B) e módulos FOP (F), o tempo de execução em segundos foi pior em todos os casos.

Indo mais a fundo na detecção da causa dessa queda no desempenho, mediu-se o tempo que o módulo FOP gastava recebendo e enviando os FOs , e foi possível identificar que o tempo de recebimento caiu, porém o tempo de envio aumentou sensivelmente. Isto demonstra que com a adição dos múltiplos *brokers* o gargalo transferiu-se para a fase posterior, que no caso é consumidor PPML o qual recebe os vários FOs renderizados e monta o arquivo PPML de saída.

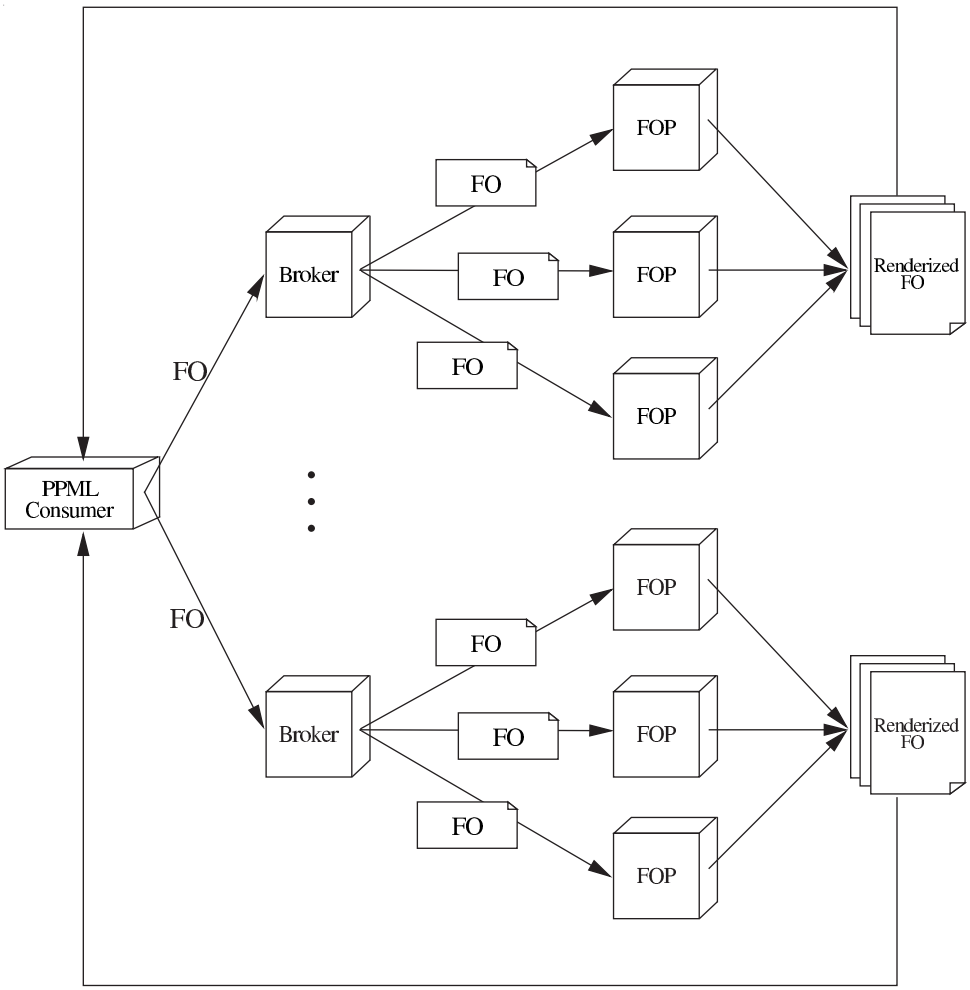


Figura 5.6: Múltiplos *brokers*

Configuração	Tempo(seg)*Multi-broker*	Tempo 1 *Broker* (seg)
2B 3F	113,99	111,06
2B 4F	117,32	112,25
3B 3F	116,74	111,18
3B 4F	113,25	110,78

Tabela 5.2: Tabela comparando a execução com diferentes configurações (*brokers* e módulos FOP) e 1 *broker*

Como o recebimento dos FOs renderizados é feito através de uma única *thread* implementada no consumidor PPML, há uma concorrência com o processo de busca e envio de FOs que é gerenciado também pelo mesmo módulo. Logo, com o aumento na velocidade de renderização

dos FOs tanto a montagem do arquivo final quanto a busca por novos FOs a serem renderizados que são tarefas que exigem muito do processo acabam concorrendo e conseqüentemente a *thread* de recebimento nem sempre está pronta para receber os FOs dos módulos FOP.

Portanto, uma possível solução para este problema seria implementar o recebimento dos FOs separadamente em outro processo de modo que não haja concorrência. A estratégia apresentada a seguir na Seção 5.3 mostra como isto seria arquitetado.

5.3 Divisão do Consumidor PPML

Diferentemente das soluções anteriores em que o módulo responsável por varrer o documento PPML a procura de FOs era o mesmo responsável pela tarefa de recebimento dos FOs renderizados enviados pelos módulos FOP, nesta arquitetura o objetivo foi justamente separar este processo de recebimento colocando-o em um processo separado a fim de evitar sobrecarga do módulo consumidor. Além disso, um *buffer* de FOs foi adicionado ao módulo *broker* o qual anteriormente enfileirava FOs enviando-os para os FOPs à medida que estavam livres para o processamento. Com essa nova funcionalidade, primeiro o *buffer* é preenchido com vários FOs variando a quantidade de acordo com o tamanho do *buffer* e também do FO , e somente após estar cheio é enviado para um módulo FOP que irá processá-los.

5.3.1 Implementação

A Figura 5.7 mostra a adição de um novo processo na arquitetura denominado recebedor PPML (*receiver*), o qual anteriormente fazia parte do módulo consumidor PPML. O processo de renderização se dá então da seguinte maneira: o consumidor PPML continua responsável por varrer o arquivo PPML de origem retirando os FOs a serem enviados para os *brokers*. Estes são enviados para um *buffer* de FOs no módulo *broker* que ao atingir o tamanho especificado os distribui entre os módulos FOP para renderização. Durante o processo de varredura no PPML, a parte não-renderizável do documento vai sendo armazenada em um vetor, e assim que um *copy-hole* contendo FOs é localizado é enviado para a fila. Seguindo o processo normalmente, o *broker* envia os FOs para os módulos FOP que os renderizam, e estes após a renderização os enviam para o *receiver*. Neste momento, para que o arquivo de saída seja montado substituindo os FOs por SVGs , o *receiver* acessa o vetor preenchido pelo consumidor PPML a fim de que a parte não renderizada seja copiada para o arquivo de saída e, de acordo com o identificador

do FO , este seja corretamente substituído pelo código SVG . Este processo se repete até que não hajam mais FOs a serem renderizados.

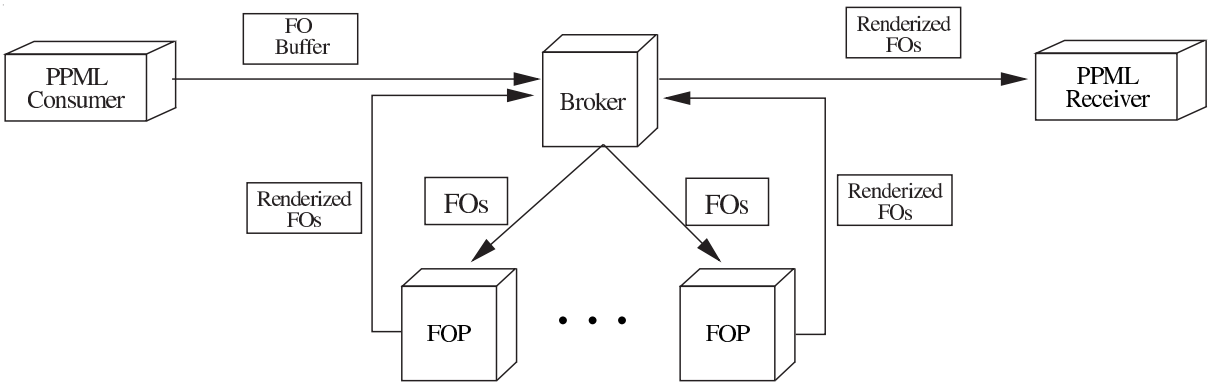


Figura 5.7: Arquitetura da solução de divisão do consumidor PPML

5.3.2 Resultados

Esta Seção apresenta os resultados destes experimentos que utilizaram os documentos XSL-FO apresentados na Seção 4.4 como entrada. Maiores detalhes sobre estes resultados podem ser encontrados em [FGZea06].

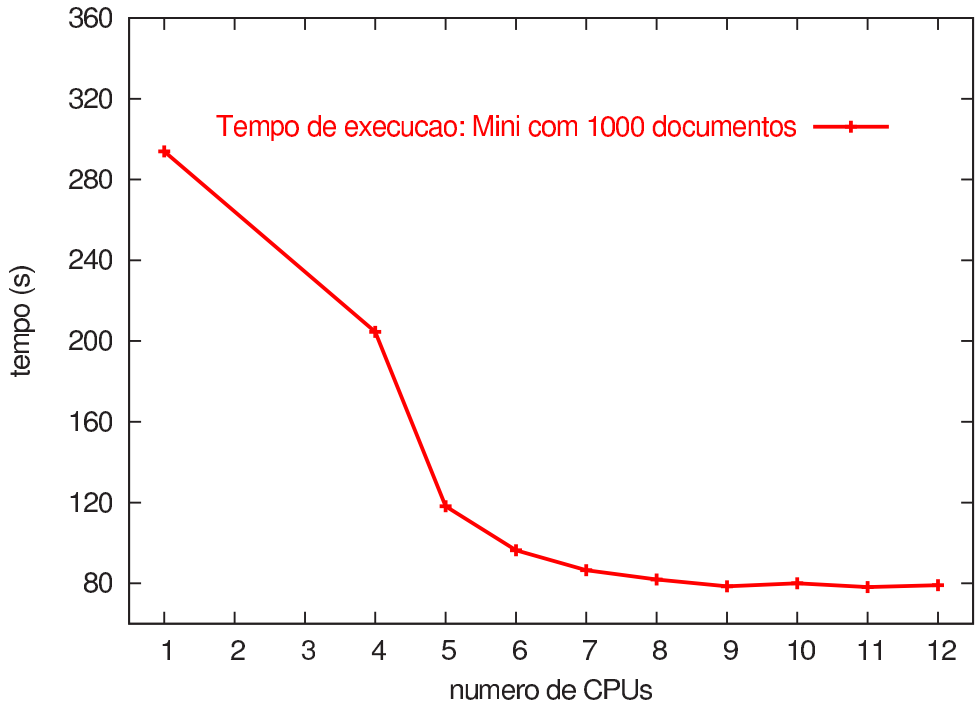
Buscando um parâmetro de comparação, a versão seqüencial da ferramenta de renderização foi executada utilizando um processador do agregado Ombrófila descrito na Seção 4.3, resultando nos tempos apresentados nas figuras 5.8, 5.9 e 5.10. Cada tempo de execução apresentado nesta Seção foi obtido após 5 execuções descartando o maior e o menor valor encontrado.

Para entender melhor os gráficos e tabelas apresentados, conforme descrito na implementação esta solução apresenta em sua arquitetura a divisão do consumidor PPML o que significa a adição de um novo processo. Assim, onde aparecem 4 processadores em paralelo temos a seguinte configuração:

- 1 consumidor PPML
- 1 recebedor PPML
- 1 Broker
- 1 FOP

Desta forma, para termos pelo menos 2 módulos FOP trabalhando realmente em paralelo é necessário no mínimo 5 processadores.

O primeiro experimento foi executado utilizando o arquivo de entrada **Mini**, o qual contém 1000 documentos. Este é o menor *job* utilizado nos testes, porém representa uma alta densidade em termos de números de palavras por bloco de texto. Neste caso, o melhor tempo de execução foi de 79,07 segundos (usando 12 processadores), mas esta configuração apresenta eficiência baixa (30,98%). Na verdade, de 7 à 12 processadores o ganho em termos de tempo de execução não é muito significativo, indicando que o sistema pode não ter vantagens quando há mais de 4 módulos FOP rodando em paralelo. A Figura 5.8 mostra os resultados para este caso de teste.

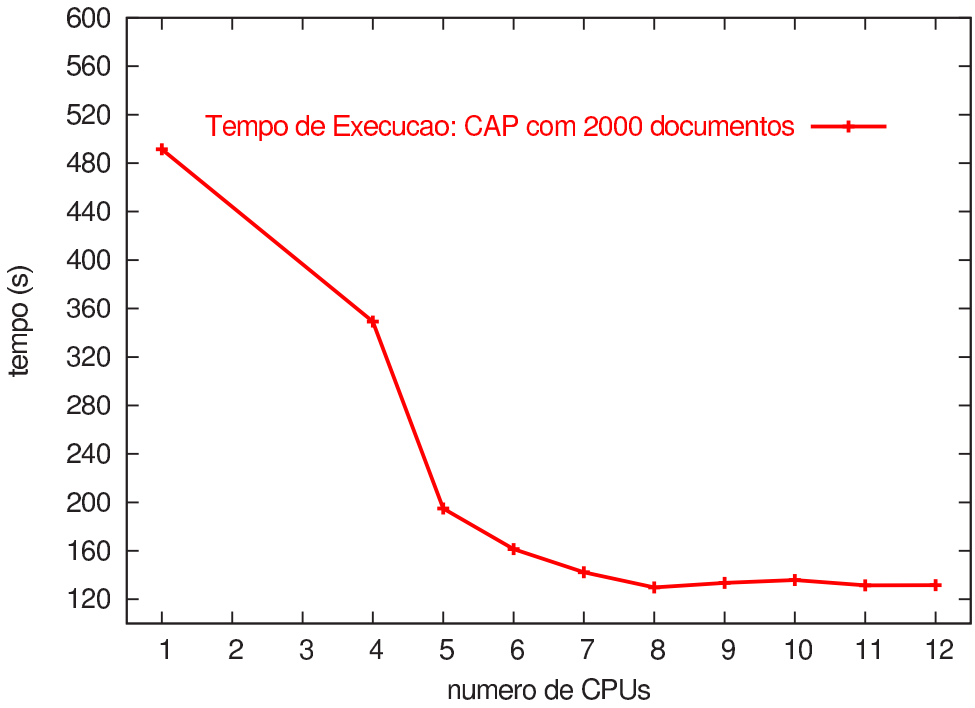


	Número de Processadores									
	1	4	5	6	7	8	9	10	11	12
Tempo(seg)	293,96	204,52	118,19	96,34	86,51	81,88	78,51	79,98	78,18	79,07
Eficiência(%)	100,00	35,94	49,74	50,85	48,54	44,88	41,60	36,75	34,18	30,98

Figura 5.8: Resultados com arquivo de entrada **Mini** com 1000 documentos

No segundo experimento, foi utilizado o arquivo de entrada **CAP**. Este é mais denso em termos de elementos a serem renderizados. O tempo seqüencial neste caso foi de 491,51 segundos para renderizar 2000 documentos. O melhor tempo de execução (129,73 segundos) foi alcançado

com 8 processadores, porém novamente o ganho de 7 à 12 processadores não é significativo em termos de tempo de execução. Os resultados deste experimento são mostrados na Figura 5.9.

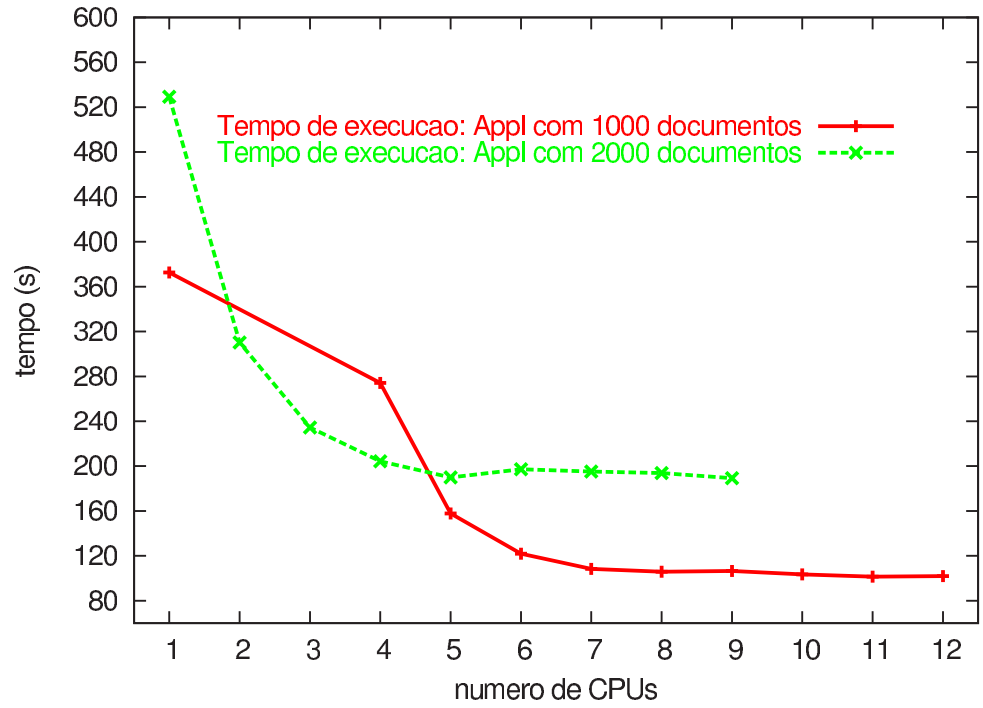


	Número de Processadores									
	1	4	5	6	7	8	9	10	11	12
Tempo(seg)	491,51	349,23	194,96	161,39	142,35	129,73	133,52	135,80	131,51	131,62
Eficiência(%)	100,00	35,18	50,42	50,76	49,32	47,36	40,90	36,19	33,98	31,12

Figura 5.9: Resultados com arquivo de entrada **CAP** com 2000 documentos

Para o último experimento, foi utilizado o mesmo modelo somente trocando o número de documentos contidos no *job* de entrada **Appl** com 1000 e 2000 documentos. Tal procedimento permitiu uma análise de escalabilidade da solução em paralelo quando a quantidade de trabalho é aumentada. O experimento com 1000 documentos apresentou a melhor execução com 11 processadores (101,37 segundos). Por outro lado, para renderizar 2000 documentos, a execução mais rápida foi obtida com 10 processadores (190,10 segundos). Os resultados mostram que a solução paralela escalonou bem quando a quantidade de documentos a serem renderizados dobrou. Os resultados são mostrados na Figura 5.10.

Comparando os três casos de testes aqui apresentados, um comportamento em comum foi detectado: rodando a aplicação com mais de 7 processadores não apresenta melhoras no tempo



	Número de Processadores									
	1	4	5	6	7	8	9	10	11	12
Tempo(seg)	372,68	274,15	157,70	121,97	108,41	105,88	106,45	103,48	101,37	101,88
Eficiência(%)	100,00	33,98	47,26	50,92	49,11	44,00	38,90	36,01	33,42	30,48

	Número de Processadores									
	1	4	5	6	7	8	9	10	11	12
Tempo(seg)	742,17	529,57	316,11	245,91	203,79	198,18	198,71	190,10	195,79	192,80
Eficiência(%)	100,00	35,03	46,96	50,30	52,03	46,81	41,50	39,04	34,46	32,08

Figura 5.10: Resultados com arquivo de entrada **Appl** com 1000 e 2000 documentos

de execução que justificariam o uso de mais processadores. Acredita-se que a razão disso é devido ao módulo *Broker* alcançar seu limite quando lida com 4 módulos FOP. Caso o número ultrapasse este valor, o módulo não consegue distribuir os FOs eficientemente entre os módulos FOP tornando-se um gargalo.

5.4 Análise Complementar

Nos testes apresentados neste Capítulo, não levam em consideração dois fatores de grande importância nos resultados: o tempo de entrada e saída e a variação do tamanho do *buffer*. Esta

Seção mostra uma análise complementar considerando estes dois fatores.

5.4.1 Entrada/Saída

Um fator relevante nos resultados mostrados é o dispositivo de entrada e saída (I/O - *Input/Output*). Em todos os testes realizados, o tempo gasto com I/O está presente nos resultados. Entretanto, como o dispositivo de I/O é o mesmo para ambos os casos, seqüencial e paralelo, para que se tenha uma idéia do ganho real obtido na paralelização da ferramenta FOP, é essencial que o tempo de I/O seja analisado. Para isso, mais alguns testes foram executados para que fossem coletados os tempos de I/O em ambas as versões. Como era esperado, o tempo de I/O foi muito parecido como mostrado na Tabela 5.3.

Arquivo PPML	Número de Documentos	Tempo(seg) seqüencial	Tempo(seg) paralelo
Mini	1000	40,36	39,83
CAP	2000	60,04	57,00
Appl	1000	57,00	53,00

Tabela 5.3: Tabela comparando o tempo de I/O entre as versões seqüencial e paralela da ferramenta FOP

Portanto, se removermos o tempo gasto com I/O nos casos de testes realizados, verificamos que o ganho real com o paralelismo é muito grande conforme mostrado nas a tabelas 5.4, 5.5, e 5.6. Em todos os casos a eficiência foi maior do que 75% chegando até a atingir 89,45% para renderizar o PPML Appl de 1000 documentos com 7 processadores.

	Sem tempo de I/O									
CPU	1	4	5	6	7	8	9	10	11	12
Tempo(seg)	253,96	165,33	79,19	57,34	47,51	42,87	39,51	40,98	39,18	40,07
Eficiência(%)	100,00	38,40	64,14	73,81	76,36	74,05	71,41	61,97	58,92	52,81
	Com tempo de I/O									
Tempo(seg)	293,96	204,51	118,19	96,34	86,51	81,87	78,51	79,98	78,18	79,07
Eficiência(%)	100,00	35,94	49,74	50,85	48,54	44,88	41,60	36,75	34,18	30,98

Tabela 5.4: Comparativo de tempo e eficiência de renderização Mini 1000 documentos com e sem tempo de I/O

	Sem tempo de I/O									
CPU	1	4	5	6	7	8	9	10	11	12
Tempo(seg)	431,51	292,23	133,96	102,39	83,68	67,73	73,52	66,80	70,51	70,62
Eficiência(%)	100,00	36,91	64,42	70,24	73,66	79,64	65,21	64,59	55,63	50,92
	Com tempo de I/O									
Tempo(seg)	491,51	349,23	194,96	161,39	142,35	129,73	133,52	135,80	131,51	131,62
Eficiência(%)	100,00	35,18	50,42	50,76	49,32	47,36	40,90	36,19	33,98	31,12

Tabela 5.5: Comparativo de tempo e eficiência de renderização CAP 2000 documentos com e sem tempo de I/O

	Sem tempo de I/O									
CPU	1	4	5	6	7	8	9	10	11	12
Tempo(seg)	315,68	221,15	103,70	66,97	50,41	51,88	52,45	50,48	47,53	47,17
Eficiência(%)	100,00	35,69	60,88	78,56	89,45	76,05	66,87	62,53	60,38	55,77
	Com tempo de I/O									
Tempo(seg)	372,68	274,15	157,70	121,97	108,41	105,88	106,45	103,48	101,37	101,88
Eficiência(%)	100,00	33,98	47,26	50,92	49,11	44,00	38,90	36,01	33,42	30,48

Tabela 5.6: Comparativo de tempo e eficiência de renderização Appl 1000 documentos com e sem tempo de I/O

5.4.2 *Buffers*

Na Figura 5.7, que descreve a arquitetura da solução de divisão do consumidor PPML, nota-se que entre o consumidor PPML e o *Broker* há um *buffer* de FOs . Tendo em vista que um único FO é um dado muito pequeno, o *buffer* foi criado para acumular um número significativo de FOs a serem enviados para os módulos FOP de modo que justificasse o tempo de comunicação gasto neste processo. Desta forma, o consumidor PPML varre o arquivo PPML retirando os FOs e enviando-os para o *buffer* até que este atinja um tamanho especificado, sendo então enviado para a renderização. Nos testes realizados neste trabalho, o tamanho do *buffer* foi fixado em 64 KBytes. Este mesmo tamanho é assumido para o *buffer* de saída do *broker* para o recebedor PPML que realoca os SVGs nas posições corretas no PPML. Portanto, a variação deste *buffer* pode interferir diretamente nos tempos encontrados tanto para mais quanto para menos.

Um mínimo de testes utilizando-se outros tamanhos de *buffer* (32KB e 128KB) foram realizados. Contudo, estes testes serviram somente para identificar qual tamanho base em KBytes seria utilizado em todos os testes. Como resultado, o tamanho de 64KB mostrou um desempenho superior, mas nada pode-se afirmar visto que foi um teste isolado, sem variação do tamanho dos documentos de entrada, entre outras possíveis variáveis.

Capítulo 6

Considerações Finais

Com o ganho de aproximadamente 30% após a execução da primeira estratégia que foi implementada simplesmente para validar uma idéia sem nenhuma preocupação com otimização de código e demais técnicas computacionais, já foi possível concluir que o ganho de termos uma versão paralela de renderização de documentos XSL-FO seria válido. Considerando-se que um cliente de grande porte imprime milhões de documentos para distribuir aos seus clientes e este processo leva por volta de 24 horas, um ganho de 50% na eficiência já reduziria em 12 horas o tempo total de renderização. Isto é um ganho muito grande tratando-se de mercado. Alguns resultados ainda estão por ser obtidos. Novas alternativas ainda estão por serem exploradas como apresentado na Seção 6.1. Espera-se um ganho ainda maior ao executarmos a solução em uma máquina SMP . Todavia, acreditamos que a solução já esteja validada e os resultados futuros são ainda mais promissores.

Este trabalho rendeu uma publicação em uma conferência internacional (SAC - *Symposium on Applied Computing*), além da colaboração e reconhecimento do laboratório da HP em Bristol que vem demonstrando cada vez mais interesse na utilização dos modelos apresentados neste trabalho em um de seus produtos.

6.1 Trabalhos Futuros

Os resultados apresentados neste trabalho indicam que ainda é possível se alcançar resultados melhores na renderização de documentos XSL-FO usando técnicas computacionais de alto desempenho. Na primeira implementação foi usado *threads* e o paradigma de programação por troca de mensagens para diminuir o tempo de execução de 350,05 para 93,30 segundos para

uma tarefa contendo mil documentos. Embora o ganho de desempenho possa ser considerado satisfatório, a principal contribuição deste trabalho foi indicar a melhor configuração entre as estudadas.

A primeira estratégia apresentada com um único *broker* traz à tona o problema de saturação do módulo *broker*, que lida com o recebimento de FOs renderizados ao mesmo tempo que verifica módulos FOP ociosos os quais requisitam novos FOs a serem processados. Baseado em tal fato, uma segunda estratégia foi implementada contendo múltiplos módulos *brokers* que não apresentou, num primeiro momento, os resultados esperados. Porém, com a adição de um *buffer* no *broker* permitindo o envio não somente de um único FO para ser renderizado, mas vários ao mesmo tempo, foi obtido mais um ganho de desempenho conforme apresentado nos resultados no Capítulo 5, Seção 5.2.2.

Entretanto, sabe-se que em ambas alternativas o arquivo PPML de saída é gerado em memória até que seja inteiramente finalizado quando é descarregado para o ambiente físico. Além da limitação de tamanho que pode ser encontrada em testes futuros, já que, por exemplo, um PPML com dois mil documentos pode em casos somente com FOs simples gerar um arquivo de saída em torno de 23MB, existe o problema da ordenação dos FOs que deve ser mantida conforme no PPML original.

Considerando todos os casos nesta linha potencial de pesquisa, como trabalhos futuros temos alguns pontos interessantes que podemos destacar. Para solucionar o problema mencionado acima da geração do arquivo de saída, o uso de DOM (*Document Object Model*) [DOM05] pode ser melhor investigado a fim de que o documento seja montado dinamicamente à medida em que os FOs renderizados são enviados dos módulos FOP para o consumidor PPML. Fora isso, nessa primeira versão implementada tanto os *brokers* quanto os módulos FOP foram implementados utilizando-se de primitivas MPI síncronas. Com isso, o recebedor PPML caso não esteja pronto para receber um FO renderizado faz com que o módulo FOP fique trancado no envio até que o mesmo esteja pronto para o recebimento. Neste caso, primitivas assíncronas podem utilizadas de mode que os módulos FOP não fiquem trancados caso tal situação ocorra.

Balanceamento de carga é uma outra possibilidade a ser pesquisada. Nos exemplos de arquivos PPML utilizados neste trabalho optou-se por utilizar FOs de mesmo tamanho, já que o objetivo era obter um volume significativo de documentos e não complexidade. Entretanto, é bastante comum termos diferentes tipos de FOs com diferentes complexidades em documentos PPML os quais conseqüentemente exigem um tempo maior ou menor de renderização. Tal

fato possibilita que em implementações futuras seja possível dimensionar o tamanho de um FO através de seu tipo de modo que seja possível enviar os maiores FOs para os processadores de maior capacidade balanceando, assim, o processamento.

Somando-se ainda a essas alternativas, o dimensionamento correto do *buffer* surge como mais uma estratégia a ser explorada. A solução com múltiplos *brokers* apresentada na Seção 5.2 considera um *buffer* cujo tamanho foi fixado em 64KB. Contudo, este valor não foi definido de forma estatística, pois no momento o que se buscava era a validade da solução e se haveria ganho de desempenho em relação às demais estratégias. Logo, testes com *buffers* de maior tamanho devem ser executados para que se possa ter uma relação entre ganho de desempenho e tamanho do *buffer* e, por conseguinte, eleger a melhor opção baseada em resultados.

Experimentos em plataformas multi-processadas (SMP) para as quais a implementação teria que sofrer algumas adaptações, porém a estratégia é idêntica.

O módulo FOP tratado não como uma caixa-preta é uma idéia a longo prazo, porém não descartada. Após todos os experimentos realizados nos trabalhos aqui mencionados, dependendo da resposta obtida em um ambiente real de impressão talvez não seja necessário tal modificação. Entretanto, uma versão preparada para uso de *threads* (*thread safe*) do FOP já está pronta para ser utilizada. FOP em sua versão liberada atualmente usa variáveis estáticas para configuração de dados e leitura de imagens. Entretanto, uma versão não disponibilizada já contorna esses problemas e será a base para o desenvolvimento de uma versão paralela futuramente.

Referências Bibliográficas

[AG94] G.S. Almasi and A. Gottlieb. *Highly parallel computing, 2a. ed.* The Benjamin Cummings Publishing Company, Inc., 1994.

[Bar05] B.M. Barney. MPI performance topics. Extracted from http://www.llnl.gov/computing/tutorials/mpi/ at Nov 19th, 2005.

[Ble94] R.A Blech. An overview of parallel processing. Extracted from http://www.lerc.nasa.gov/othergroups/IFMD/2620/tutorialPP.html at Oct 20th, 1994. Slides presented at the Parallel Computing with PVM Workshop.

[Bos00] 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.

[DdB00] P. Davis and D. de Bronkart. PPML (Personalized Print Markup Language). In *Proceedings of the XML Europe 2000*, Paris, France, 2000. International Digital Enterprise Alliance.

[DOM05] DOM. Document Object Model. Extracted from http://www.w3.org/DOM/ at Jan 19th, 2005.

[Dun90] R. Duncan. A survey of parallel computer architectures. *IEEE Computer*, pages 5–16, 1990.

[FGZea06] L.G. Fernandes, F. Giannetti, R.T. Zambon, and et al. High performance XSL-FO rendering for variable data printing. In *ACM Symposium on Applied Computing (SAC)*, Dijon, France, 2006. Artigo aceito.

[Fly72] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C(21):pp.948–960, 1972.

[FOP05] FOP. Formatting Objects Processor. Extracted from http://xml.apache.org/fop at May 13th, 2005.

[GHM98] 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.

[GM02] R. Glushko and T. McGrath. Document Engineering for e-Business. In *ACM Symposium on Document Engineering*, 2002.

[HB84] K. Hwang and F.A. Briggs. *Computer architecture and parallel processing.* McGraw-Hill International Editions, 1984.

[Hwa93] K. Hwang. *Advanced computer architecture - parallelism, scalability, programmability.* McGraw-Hill International Edition, 1993.

[KB88] A.H. Karp and R.G. Babb. A comparison of 12 parallel Fortran dialects. *IEEE Software*, 5(5):52–67, 1988.

[KL88] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Software*, 5(1):23–32, 1988.

[MMG+00] 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.

[MMM+04] F. Meneguzzi, L. Meirelles, F. Mano, J. Oliveira, and A. Silva. Strategies for document optimization in digital publishing. In *ACM Symposium on Document Engineering*, pages 163–170, Milwaukee, USA, 2004. ACM Press.

[mpi05] mpiJava. The mpiJava home page. Extracted from http://www.hpjava.org/mpiJava.html at May 13th, 2005.

[NBvO01] P. Navaux, M. Barreto, R. Ávila, and F. Oliveira. *ERAD - Escola Regional de Alto Desempenho*, chapter Execução de aplicações em ambientes concorrentes, pages 2–9. 2001.

[PHOF03] 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.

[POD05] PODi. Print on Demand Initiative. Extracted from http://www.podi.org at May 13th, 2005.

[Qui94] M. Quinn. *Parallel computing: theory and practice*. McGraw-Hill, 1994.

[RGM99] J. Tenenbaum R. Glushko and B. Meltzer. An XML framework for agent-based e-commerce. *Communications of the ACM*, 42(3):106–114, 1999.

[SOHL+96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: the complete reference*. MIT Press, 1996.

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

[XT05] XSL-T. XSL-Transformations. Extracted from http://www.w3.org/TR/1999/REC-xslt-19991116 at May 13th, 2005.