UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO
MESTRADO EM SISTEMAS E COMPUTAÇÃO

SIMULAÇÃO DE RESERVATÓRIOS DE PETRÓLEO EM AMBIENTE MPSOC

Bruno Cruz de Oliveira

Maio, 2009
Natal/RN

BRUNO CRUZ DE OLIVEIRA

SIMULAÇÃO DE RESERVATÓRIOS DE PETRÓLEO EM AMBIENTE MPSOC

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para obtenção do título de Mestre em Sistemas e Computação (MSc.).

Orientador: Prof. Dr. Ivan Saraiva Silva

Maio, 2009
Natal/RN

Divisão de Serviços Técnicos

Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Oliveira, Bruno Cruz de.
Simulação de reservatórios de petróleo em ambiente MPSoC / Bruno Cruz de Oliveira. – Natal, RN, 2009.
71 f.

Orientadora: Ivan Saraiva Silva.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Sistemas-em-chip multiprocessados – Dissertação. 2. Redes em chip – Dissertação. 3. Modelos de memória – Dissertação. 4. Simulação de reservatório – Dissertação. I. Silva, Ivan Saraiva. II. Universidade Federal do Rio Grande do Norte. III. Título.

RN/UF/BCZM CDU 681.3.012(043.3)

BRUNO CRUZ DE OLIVEIRA

Simulação de Reservatórios de Petróleo em Ambiente MPSoC

Esta Dissertação foi julgada adequada para a obtenção do título de mestre em Sistemas e Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Prof. Dr. Ivan Saraiva Silva – UFRN
Orientador

Profª. Drª. Thaís Vasconcelos Batista – UFRN
Coordenadora do Programa

Banca Examinadora

Prof. Dr. Ivan Saraiva Silva – UFRN
Presidente

Prof. Dr. David Boris Paul Déharbe – UFRN

Prof. Dr. Márcio Eduardo Kreutz – UFRN

Prof. Dr. Altamiro Amadeu Susin – UFRGS

Maio, 2009

Esse trabalho foi desenvolvido com o apoio da Agência Nacional do Petróleo, através do Programa de Formação de Recursos Humanos N° 22 – Formação em Geologia, Geofísica e Informática do Setor do Petróleo e Gás Natural na UFRN – com especialização em Sistemas de Tempo Real para Otimização e Automação do Setor do Petróleo e Gás Natural

OLIVEIRA, Bruno Cruz. SIMULAÇÃO DE RESERVATÓRIOS DE PETRÓLEO EM AMBIENTE MPSOC. Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para obtenção do título de Mestre em Sistemas e Computação, Natal, 2009.

Resumo

O constante aumento da complexidade das aplicações demanda um suporte de hardware computacionalmente mais poderoso. Com a aproximação do limite de velocidade dos processadores, a solução mais viável é o paralelismo. Baseado nisso e na crescente capacidade de integração de transistores em um único chip surgiram os chamados MPSoCs (*Multiprocessor System-on-Chip*) que deverão ser, em um futuro próximo, uma alternativa mais rápida e mais barata aos supercomputadores e clusters. Aplicações tidas como destinadas exclusivamente a execução nesses sistemas de alto desempenho deverão migrar para máquinas equipadas com MPSoCs dotados de dezenas a centenas de núcleos computacionais. Aplicações na área de exploração de petróleo e gás natural também se caracterizam pela enorme capacidade de processamento requerida e deverão se beneficiar desses novos sistemas de alto desempenho.

Esse trabalho apresenta uma avaliação de uma tradicional e complexa aplicação da indústria de petróleo e gás natural, a simulação de reservatórios, sob a nova ótica do desenvolvimento de sistemas computacionais integrados em um único chip, dotados de dezenas a centenas de unidades funcionais. Para isso, um modelo de memória distribuída foi desenvolvido para a plataforma STORM (MPSoC *Directory-Based Platform*), que já contava com um modelo de memória compartilhada. Foi desenvolvida, ainda, uma biblioteca de troca de mensagens para esse modelo de memória seguindo o padrão MPI.

Palavras-chave: sistemas-em-chip multiprocessados, redes-em-chip, modelos de memória, simulação de reservatórios

OLIVEIRA, Bruno Cruz. RESERVOIR SIMULATION IN A MPSOC ENVIRONMENT. Dissertation submitted to Postgraduate Program in System and Computing from Informatics and Applied Mathematics Department of Federal University of Rio Grande do Norte like part of requisites to obtain the master's degree, Natal, 2009.

Abstract

The constant increase of complexity in computer applications demands the development of more powerful hardware support for them. With processor's operational frequency reaching its limit, the most viable solution is the use of parallelism. Based on parallelism techniques and the progressive growth in the capacity of transistors integration in a single chip is the concept of MPSoCs (Multi-Processor System-on-Chip). MPSoCs will eventually become a cheaper and faster alternative to supercomputers and clusters, and applications developed for these high performance systems will migrate to computers equipped with MP-SoCs containing dozens to hundreds of computation cores. In particular, applications in the area of oil and natural gas exploration are also characterized by the high processing capacity required and would benefit greatly from these high performance systems.

This work intends to evaluate a traditional and complex application of the oil and gas industry known as reservoir simulation, developing a solution with integrated computational systems in a single chip, with hundreds of functional unities. For this, as the STORM (MPSoC Directory-Based Platform) platform already has a shared memory model, a new distributed memory model were developed. Also a message passing library has been developed folowing MPI standard.

Keywords: multiprocessor system-on-chip, network-on-chip, memory models, reservoir simulation

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

ASIC	Application-Specific Integrated Circuit
ATA	Address Table
BOAST	Black Oil Applied Simulation Tool
CaCoMa	Cache Communication Manager
CoMa	Communication Manager
CPI	Ciclos Por Instrução
DAMa	Data Access Manager
DSP	Digital Signal Processor
FIFO	First In First Out
GCC	GNU Compiler Collection
GNU	GNU Is Not Unix
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronic Engineers
LFU	Least Frequently Used
LRU	Least Recently Used
MPI	Message Passing Interface
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
PTA	Processor Table
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SoC	System-on-Chip
SOR	Successive Over-Relaxation
SPARC	Scalable Processor Architecture
STA	Status Table
STORM	MPSoC Directory-Based Platform
TLM	Transaction Level Modeling
VCT	Virtual Cut Through

1 INTRODUÇÃO

As aplicações atuais, principalmente na área de sistemas embarcados e multimídia, demandam cada vez mais por poder de processamento. O tratamento de vídeo de alta definição em tempo-real, por exemplo, requer três bilhões de operações por segundo, enquanto que a geração sintética de vídeo pode requerer até um trilhão de operações por segundo (BERTOZZI et al, 2005). Por outro lado, na área dos sistemas de propósito geral, são igualmente numerosas as aplicação computacionalmente intensas. A indústria de hardware vem tentando ao máximo atender essa demanda. Entretanto, o aumento da freqüência dos processadores já não é uma solução viável, pois leva ao aumento do consumo de energia. Dessa forma, a indústria de hardware adotou como solução a adição de núcleos processantes (*core*) aos processadores, criando assim os chamados *multi-core*. Hoje em dia, estações de trabalho equipadas com processadores *multi-cores* de dois e até quatro núcleos são facilmente encontradas.

Apesar disso, a capacidade de integração de transistores em uma única pastilha de silício ainda supera a capacidade de utilização desses transistores em um único projeto. Como um dos resultados da grande demanda por poder de processamento, da alta disponibilidade de transistores e do desenvolvimento de novas tecnologias de projeto de hardware, surgiram as chamadas arquiteturas MPSoC (*Multiprocessor System-on-Chip*) (LOGHI et al., 2004). Um MPSoC é um sistema computacional multiprocessado integrado em chip único. Esses sistemas, além de múltiplos processadores, integram os mais diversos componentes, como co-processadores, processadores digitais de sinais, blocos de memória, blocos de entrada e saída, núcleos dedicados (ASIC – *Application Specific Integrated Circuits*) e dispositivos que gerenciam a comunicação interna ao chip (JERRAYA; WOLF, 2005).

Nestes sistemas complexos, a comunicação entre os módulos é crítica e por isso tem sido alvo de diversas investigações arquiteturais (BERTOZZI et al., 2006). A mais simples das soluções, tanto no que se refere à implementação de hardware quanto ao custo, é o uso de barramentos. Entretanto, os barramentos sofrem de um baixo grau de escalabilidade no sentido de que a inclusão de poucos elementos no sistema causa uma degradação na comunicação entre os nós. A utilização de múltiplos barramentos ou

sistemas de barramentos são, por sua vez, soluções *ad-hoc*, o que impede sua reutilização quando o sistema é modificado de alguma forma.

Inspiradas nas redes de interconexão de computadores, as redes-em-chip, ou NoC (*Network-on-Chip*) (BENINI, 2002), surgiram como alternativa aos barramentos e vem sendo alvo de diversas pesquisas nos últimos anos. Em uma NoC os módulos do sistema estão conectados a roteadores que, por sua vez, estão interconectados seguindo uma determinada topologia (malha, torus, cubo, etc). Além do alto grau de escalabilidade (ZEFERINO et al, 2002), as NoCs tem como principal característica a capacidade de comunicação paralela entre os elementos do sistema.

Algumas pesquisas recentes (BOKAR, 2007; AGARWAL; LEVY, 2007; HELD; KOEHL, 2007) tentam atender a demanda por processamento com a integração de dezenas de centenas de unidades de processamento em um único circuito integrado, são os chamados *many-core*. Nesses sistemas o uso de NoC como mecanismo de interconexão é fundamental.

Por outro lado, com a disponibilização de tantos processadores em um único chip, é natural que aplicações anteriormente executadas em supercomputadores ou em *clusters* de computadores, passem a ser consideradas para execução em MPSoCs ou *many-core*. Esta dissertação, inserida no contexto do desenvolvimento de recursos humanos para a área de petróleo e gás natural, (projeto UFRN – ANP – PRH 22), foca-se no estudo da execução, em ambiente MP-SoC, de uma aplicação computacionalmente intensa desta área.

1.1 Motivação

Na engenharia de petróleo uma das tarefas mais importantes é a previsão da produção de hidrocarbonetos de reservatórios. Essa tarefa permite quantificar reservas, avaliar e priorizar projetos de explotação e dimensionar sistemas de produção para otimizar a produção de um reservatório. Nesse sentido, a simulação computacional de reservatórios de petróleo vem sendo utilizada há décadas, e tem se tornado, a cada dia, uma importante ferramenta para a indústria de exploração de petróleo e gás natural. Entretanto, esses

simuladores são extremamente pesados e necessitam de um grande poder de processamento para obter resultados confiáveis em um tempo aceitável.

Computadores paralelos, tais como supercomputadores e clusters, têm sido a melhor alternativa para atender a demanda de processamento gerada pela simulação de reservatórios, possibilitando tratar modelos cada vez mais detalhados e representativos do problema real. Contudo, é fundamental que o código computacional tenha sido desenvolvido e implementado usando as técnicas específicas de programação paralela, levando em consideração alguns aspectos relevantes como localização de dados, sincronização e balanceamento do trabalho computacional entre processadores. O código do simulador seqüencial deve ser, portanto, reformulado e implementado visando tirar proveito das características específicas da arquitetura de cada máquina paralela.

Os MPSoCs e *many-core*, que em um futuro próximo deverão estar disponíveis no mercado, serão uma alternativa mais rápida e mais barata aos supercomputadores e clusters. Um exemplo da capacidade de processamento dos MPSoCs pode ser visto na Tabela 1.1. Essa tabela compara o desempenho e consumo de potência de um processador Intel Pentium 4 com um MPSoC (*Terascale Processor*) que se encontra sendo desenvolvido também pela Intel, e que possui 80 núcleos interligados através de uma NoC (HELD e KOEHL, 2007).

Tabela 1.1 – Comparação entre o Pentium 4 e o Terascale Processor.

	Freqüência (GHz)	**Potência Dissipada (Watts)**	**Performance (Gigaflops)**
Pentium 4	3,20	84	3,10
Terascale Processor	3,16 5,10 5,70	62 175 265	1010 1630 1810

No entanto, sistemas arquiteturalmente complexos como os *many-core* e MPSoCs apresentam diversos desafios aos projetistas. Muitos desses desafios, tais como, modelos de memória, manutenção da coerência de cache e consumo de energia, vêm sendo estudados há algum tempo. Contudo, não é apenas o projetista de hardware que tem desafios a resolver. O desenvolvimento de software para arquiteturas tão complexas é uma tarefa bastante árdua, que envolve conhecimentos em diversas áreas, como compiladores,

ligadores e técnicas de programação paralela. Essa tarefa se torna um tanto mais difícil, pois, devido aos novos paradigmas inseridos (como a comunicação através de NoC) e ao estado relativamente inicial do desenvolvimento dos MPSoCs, poucas ou nenhuma ferramentas e bibliotecas estão disponíveis para essas arquiteturas.

1.2 Objetivos

Este trabalho pretende avaliar uma tradicional e complexa aplicação da indústria de exploração de petróleo e gás natural, a simulação de reservatórios, sob a nova ótica do desenvolvimento de sistemas computacionais integrados em um único chip, dotados de dezenas a centenas de unidades funcionais. Nesta pesquisa serão avaliados aspectos tais como: modelos de memória, modelos de programação, subsistemas de interconexão e bibliotecas de comunicação. É importante deixar claro que este trabalho não tem pretensão de propor melhorias significativas no que diz respeito às técnicas de modelagem dos reservatórios de petróleo. O trabalho aqui apresentado foca-se nos aspectos de desenvolvimento da arquitetura de hardware e da aplicação para MP-SoC, como especial interesse nas medidas de desempenho obtidas, considerando-se as características do modelo de MP-SoC utilizado. Para tanto, utiliza e amplia os recursos da plataforma STORM (REGO, 2006).

A plataforma STORM tem se mostrado uma importante e eficiente ferramenta para desenvolvimento e avaliação de soluções de software e de hardware em ambiente MPSoC baseado em NoC. Com este trabalho, que implementará uma aplicação complexa da indústria de exploração de petróleo e gás natural na plataforma STORM, pretende-se propor e implementar melhorias arquiteturais e de implementação na plataforma STORM, visando, entre outros aspectos, torná-la uma ferramenta ainda mais completa com a adição de novas funcionalidades.

Este trabalho tem como objetivos específicos:

- Implementar um modelo de memória distribuída para a plataforma STORM;
- Desenvolver uma biblioteca de comunicação para a versão de memória distribuída implementada na plataforma STORM;

- Implementar simuladores de reservatórios de petróleo que explorem o máximo de paralelismo nas diferentes arquiteturas disponíveis na plataforma STORM;
- Avaliar os diferentes aspectos da arquitetura utilizada e o seu impacto no desempenho do algoritmo de simulação de reservatórios.

1.3 Organização do Texto

Este trabalho está dividido da seguinte forma: o capítulo 2 apresenta de maneira detalhada a plataforma STORM, os seus diversos módulos e os modelos de memória disponíveis. No capítulo 3 são discutidos os conceitos básicos da simulação computacional de reservatórios de petróleo. Os resultados obtidos são apresentados no capítulo 4. E por fim, as conclusões e trabalhos futuros são apresentados no último capítulo.

2 PLATAFORMA STORM

As inovações tecnológicas em conjunto com a pressão gerada pelo mercado globalizado exigem que as empresas desenvolvam novos sistemas em poucos meses. Para suprir essa necessidade, uma nova metodologia de desenvolvimento de circuitos integrados foi criada. Essa metodologia, chamada de projeto baseado em plataforma, será explicada na próxima seção.

Em seguida a plataforma STORM será apresentada em detalhes. Essa plataforma utiliza práticas de projeto baseado em plataforma e foi o principal alvo de estudo do presente trabalho.

2.1 Projeto Baseado em Plataforma

O chamado *time-to-market* – tempo decorrido desde o início do projeto de um chip até sua disponibilidade no mercado – é um fator de fundamental importância no lucro obtido com o desenvolvimento de novos sistemas. Quanto maior for o *time-to-market*, maior será o custo com o pagamento da equipe de desenvolvimento.

Outro problema com o elevado tempo de desenvolvimento é o atraso na inserção do produto no mercado. Como pode ser visto na Figura 2.1, reproduzida de (REGO, 2006), o lucro gerado por um produto está diretamente relacionado à velocidade do seu lançamento no mercado. Um atraso no lançamento, em relação ao aparecimento inicial de um tipo de produto no mercado, causa uma queda na curva de lucro do produto devido à dominação do mercado pelo produto pioneiro.

Um terceiro fator importante para a curva de lucro, e que é influenciado pelo *time-to-market*, é a vida útil do produto. O retorno financeiro de um produto deve ser obtido em poucos meses, uma vez que a vida útil de novos produtos é cada vez menor, e atrasos de poucas semanas no lançamento pode comprometer seriamente os ganhos esperados.

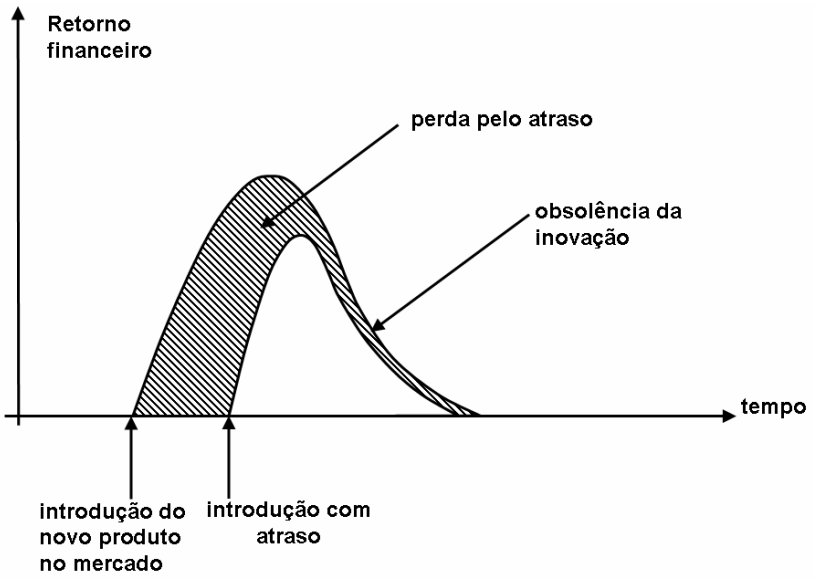

Figura 2.1 – Retorno financeiro do projeto em função do tempo.

No modelo tradicional de desenvolvimento de circuitos integrados, chamado de *full-custom*, uma equipe de engenheiros é responsável por desenhar o chip transistor a transistor. Apesar de produzir circuitos mais eficientes, essa metodologia está caindo em desuso, pois gera um alto *time-to-market* em projetos de grande complexidade. Para produzir novos sistemas cada vez mais complexos e em espaços de tempo cada vez menores, um novo conceito de desenvolvimento, chamado de projeto baseado em plataforma, vem sendo utilizado.

Apesar de não existir uma definição unânime para o conceito de plataforma, ele pode ser entendido como "uma abstração que cobre diversos possíveis refinamentos em baixo nível" (SANGIOVANNI-VINCENTELLI; MARTIN, 2001). Este conceito, que tem sido cada vez mais discutido tanto pelas empresas fabricantes de chips quanto pela comunidade acadêmica, está intimamente ligado ao reuso de componentes e de software. Ao invés de um simples circuito, o objetivo de uma plataforma é prover um ambiente para o desenvolvimento rápido de novos sistemas, utilizando para isto um conjunto de componentes pré-definidos Estes componentes vão desde microprocessadores a ASICs, passando por módulos de lógica reconfigurável e DSP, além de componentes de software tais como: sistemas operacionais, drivers, middlewares, bibliotecas de funções e outros.

Outro recurso que deve ser fornecido pela plataforma é o mecanismo de interconexão, como barramentos, canais dedicados e NoCs. Assim, utilizando o conceito de plataforma, o projeto de um sistema complexo consiste em selecionar e interconectar os

módulos necessários, criando uma instância da plataforma, sendo possível também selecionar componentes de software previamente desenvolvidos para acelerar o desenvolvimento de software para essa plataforma. Dessa forma, o tempo de desenvolvimento de novos produtos pode ser reduzido de meses para poucas semanas.

2.2 Plataforma STORM

A plataforma STORM (MPSoC *Directory-Based Plaftorm*) foi desenvolvida com o objetivo de estudar a viabilidade e uso de um projeto baseado em MPSoC utilizando NoC como mecanismo de interconexão. A ferramenta SystemC (IEEE, 2005) foi escolhida para o seu desenvolvimento pois se ajusta melhor às necessidades do projeto, permitindo a descrição funcional do sistema e exploração do espaço de projeto. Sua implementação foi realizada de forma que fosse possível a extração de resultados com precisão de ciclo, pois estes são mais expressivos e melhor aceitos pela comunidade acadêmica.

Na metodologia tradicional de desenvolvimento de sistemas, o projetista em um primeiro momento implementa o modelo do sistema em uma linguagem de alto nível, como C e C++, para verificar a validade dos conceitos e algoritmos em nível de sistema. Uma vez que os conceitos e algoritmos estejam validados, as partes do modelo em alto nível a serem implementadas em hardware são convertidas manualmente para HDL (*Hardware Description Language*). Este salto no nível de implementação é muito trabalhoso e passível de erros.

O SystemC é uma biblioteca de *templates* e funções para C++, que permite a construção e simulação de blocos, ou módulos, de hardware obedecendo a um relógio. Diferentemente da metodologia tradicional, o SystemC permite a utilização de diversos níveis de abstração, dessa forma, uma aplicação pode ser escrita em C++ puro e ser refinada até o nível de RTL. A Figura 2.2 mostra a diferença entre a metodologia tradicional e a metodologia SystemC.

A plataforma STORM não possui uma arquitetura definida, mas sim um conjunto de módulos e especificações sobre o seu uso, permitindo ao projetista criar diversas instâncias de arquiteturas com características diferentes. Essa propriedade permite a

integração de diferentes módulos e maior facilidade de exploração do espaço de projeto, seguindo assim, o conceito de projeto baseado em plataforma.

Atualmente dois modelos de memória estão disponíveis na plataforma STORM. O primeiro é o modelo de memória compartilhada distribuída, em que os módulos de memória presentes no sistema formam um espaço de endereçamento único e acessível por todos os processadores. O outro modelo de memória disponível é o modelo de memória distribuída, onde cada processador presente no sistema está ligado a uma memória local e precisa fazer uso de trocas de mensagens para realizar a comunicação com outros processadores. Maiores detalhes desses modelos de memória serão dados mais adiante.

A seguir serão detalhados os módulos integrados a plataforma STORM e que podem ser utilizados nos dois modelos de memória disponíveis. Em seguida os modelos de memória e seus módulos específicos serão descritos de forma detalhada.

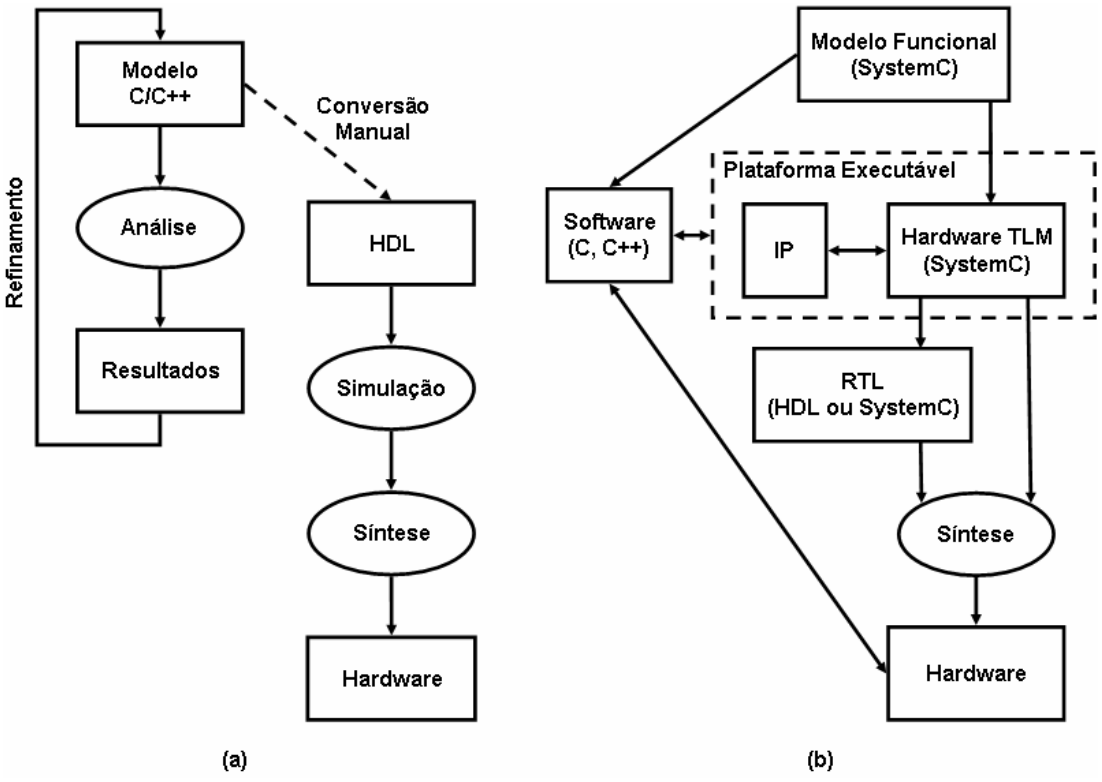

Figura 2.2 – (a) Metodologia Tradicional; (b) Metodologia SystemC.

2.2.1 Processador

A plataforma conta atualmente com um processador SPARC (*Scalable Processor ARChitecture*) V8 (SPARC, 1992) integrado. Este processador, de arquitetura RISC (*Reduced Instruction Set Computer*) desenvolvida pela Sun Microsystems, é utilizado desde sistemas embarcados até grandes servidores. Por ter arquitetura aberta, e contar com uma vasta gama de ferramentas disponíveis, este processador foi escolhido para ser integrado à plataforma. Uma das ferramentas disponíveis é o compilador GCC (GNU *Compiler Collection*), utilizado para gerar o código binário SPARC que é carregado na memória da plataforma.

Apesar de contar atualmente apenas com o processador SPARC, a plataforma STORM permite a integração de qualquer processador. Para isso, a interface com a memória cache deve ser mantida, pois é através dela que é feita a comunicação com o restante da plataforma.

2.2.2 Mecanismos de Interconexão

O sistema de interconexão é um dos principais aspectos no desenvolvimento dos SoCs. Como discutido anteriormente, os barramentos têm escalabilidade limitada e tendem a ser substituídos pelas NoCs (BENINI; DE MICHELI, 2002), que apresentam um grau maior de escalabilidade e permitem um rápido dimensionamento do sistema. Por esse motivo, foi optado pelo uso de NoC como mecanismo de interconexão da plataforma STORM, que conta atualmente com dois modelos diferentes de NoC implementados: a NoCX4 (REGO; SILVA; AZEVEDO FILHO, 2004) e a Árvore Obesa (ou ObTree) (REGO, 2006). Entretanto é possível utilizar qualquer outra topologia de NoC, desde que mantida a interface do roteador com os módulos.

A NoCX4 foi desenvolvida para a integração da arquitetura reconfigurável X4CP32 (SOARES; PEREIRA; SILVA, 2003), entretanto, ela é genérica o suficiente para a integração de um MPSoC. Essa topologia consiste de uma grelha de roteadores, onde cada um deles está ligado a um módulo da plataforma, tratando-se portanto, de uma topologia

direta. Suas principais características podem ser vistas na Tabela 2.1, e um exemplo dessa topologia é mostrado na Figura 2.3.

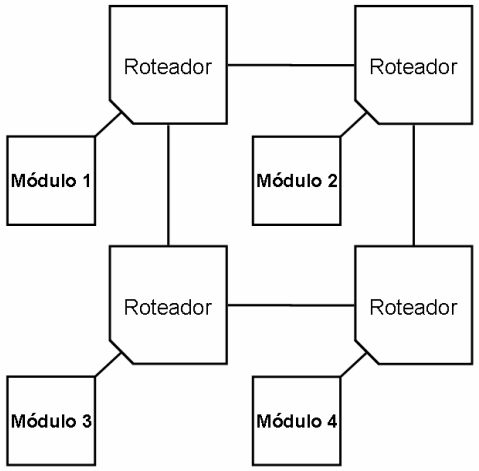

Figura 2.3 – Exemplo da topologia NoCX4.

A Árvore Obesa é uma topologia indireta criada durante o desenvolvimento da plataforma STORM. Essa topologia consiste de uma árvore, onde os roteadores são alocados em níveis. Dois tipos de roteadores são utilizados: roteadores folha, onde os módulos são ligados; e roteadores intermediários, que interligam os roteadores folha de ramos diferentes da árvore. Um exemplo dessa topologia é mostrado na Figura 2.4, e suas principais características são apresentadas na Tabela 2.1.

Tabela 2.1 – Principais características da NoCX4 e da Árvore Obesa.

Característica	NoCX4	Árvore Obesa
Topologia	Grelha	Baseada em Árvore
Chaveamento	Store-and-Forward	Virtual Cut Through (VCT)
Roteamento	Determinístico XY	Menor Caminho
Arbitragem	Distribuída	Distribuída
Controle de Fluxo	Baseado em Créditos	Baseado em Créditos
Memorização	Somente na Entrada	Somente na Entrada

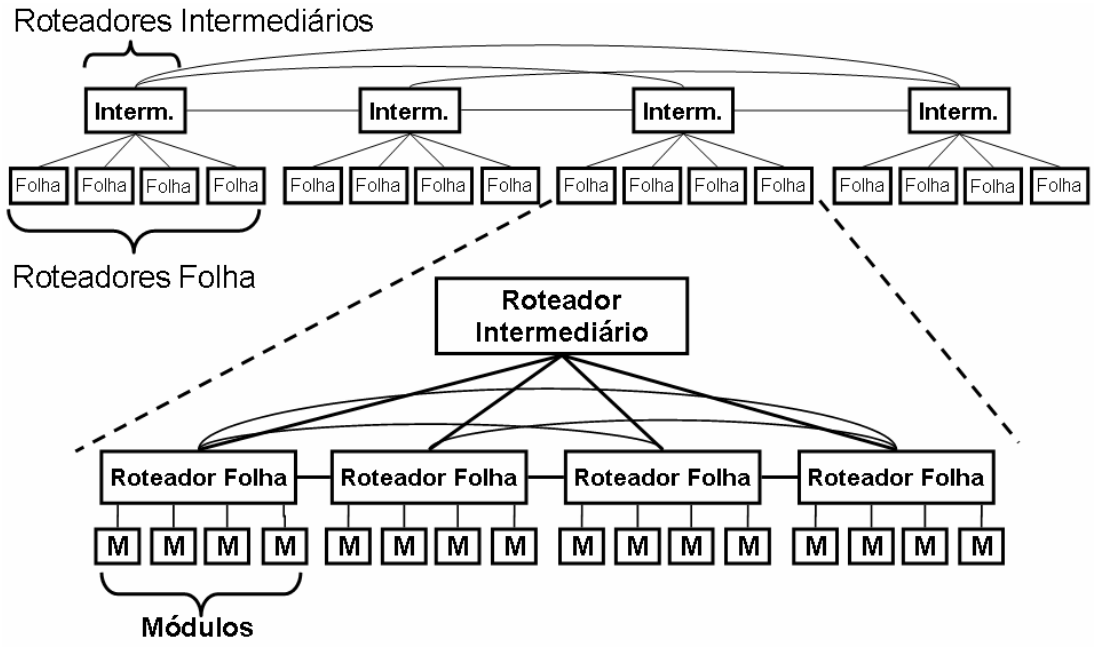

Figura 2.4 – Exemplo da topologia Árvore Obesa

2.2.3 Memória Compartilhada

No modelo de memória compartilhada da plataforma STORM cada processador possui uma cache local para o armazenamento de cópias de dados da memória compartilhada, como pode ser visto na Figura 2.5. A utilização de memória cache em sistemas multiprocessados com memória compartilhada gera um problema grave chamado de problema da coerência de cache. Cópias de um dado podem existir em caches diferentes simultaneamente, assim, uma visão inconsistente da memória poderá ocorrer se for permitido que os processadores alterem livremente os dados nas suas caches. Essa visão inconsistente pode levar ao mau funcionamento do programa em execução.

Os principais causadores de incoerência de cache são o compartilhamento de dados e a migração de processos. É importante ressaltar que a política de escrita da cache não melhora ou piora o problema da coerência de cache. Na política *write-through*, quando um dado é alterado na cache seu novo valor é enviado imediatamente para a memória compartilhada. Enquanto que na política *write-back*, um dado alterado na cache só é enviado para a memória compartilhada quando outra cache necessita desse dado ou quando

ele não é mais necessário para a que o alterou. Em ambos os casos poderá ocorrer incoerência de cache, conforme mostrado nas Tabelas 2.2 e 2.3.

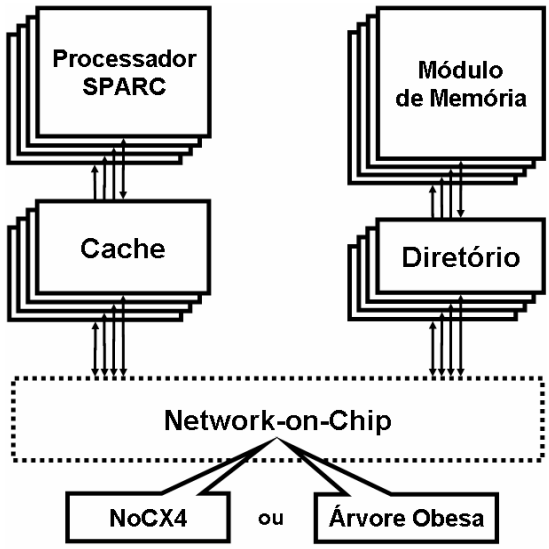

Figura 2.5 – Integração dos módulos da plataforma STORM utilizando o modelo memória compartilhada.

As soluções para o problema da coerência de cache, chamadas de esquemas de coerência de cache, são divididas em dois grandes grupos: as soluções baseadas em software, e as soluções baseadas em hardware. Apesar de apresentarem um baixo custo implementacional, as soluções baseadas em software são bastante restritas e, por isso, pouco utilizadas.

Os esquemas baseados em hardware, também chamados de protocolos de coerência de cache, mesmo sendo mais complexos e custosos, do ponto de vista implementacional, são largamente utilizados nos sistemas comerciais. Esses protocolos são divididos, de maneira geral em: protocolos de diretório e protocolos *snoop*.

Tabela 2.2 – Exemplo de incoerência em cache com política de escrita *write-through*.

Tempo	Evento	Conteúdo da Cache A	Conteúdo da Cache B	Conteúdo do Endereço X na Memória
0				1
1	CPU A lê X	1		1
2	CPU B lê X	1	1	1
3	CPU A escreve 0 em X	0	1	0

Tabela 2.3 – Exemplo de incoerência em cache com política de escrita *write-back*.

Tempo	Evento	Conteúdo da Cache A	Conteúdo da Cache B	Conteúdo do Endereço X na Memória
0				1
1	CPU A lê X	1		1
2	CPU B lê X	1	1	1
3	CPU A escreve 0 em X	0	1	1

No protocolo *snoop* cada cache é responsável por manter os estados dos blocos que possui. Para isso, a cache deve monitorar o mecanismo de interconexão para detectar ações e condições, geradas por outras caches, que possam levar a um estado de incoerência. Esse monitoramento exige que o mecanismo de interconexão permita operações de *broadcast*.

Nas soluções de diretório a responsabilidade da manutenção da coerência de cache é delegada a um controlador central que normalmente é uma parte do controlador da memória principal. Esse controlador central guarda em um local chamado de diretório as informações de localização das cópias de cada bloco e o estado dessas cópias. A cada requisição de uma cache, o controlador consulta o diretório para saber quais caches possuem cópia do bloco e, se for necessário, envia comandos de coerência para essas caches.

Como dito anteriormente, a plataforma STORM utiliza NoC como mecanismo de interconexão. Neste mecanismo as operações de *broadcast* têm um custo muito elevado, inviabilizando o uso de mecanismos de coerência de cache baseados no protocolo *snoop*. Dessa forma, uma solução para o problema da coerência de cache baseada em diretório foi desenvolvida para a plataforma STORM. Essa solução será apresentada a seguir juntamente com os módulos específicos do modelo de memória compartilhada da plataforma STORM.

a) Caches e CaCoMa

O módulo de memória cache pode ser divido em três partes: cache de instruções, cache de dados, e CaCoMa (*Cache Communication Manager*). Esta divisão em cache de instruções (icache) e cache de dados (dcache) é bastante utilizada nas arquiteturas *pipeline*

atuais para permitir o acesso simultâneo a instruções e dados. Tanto a cache de instruções quanto a cache de dados são completamente associativas. Entretanto, as demais configurações da cache de dados podem ser diferentes daquelas da cache de instrução.

As configurações das caches que podem ser alteradas são: tamanho da cache, tamanho da linha (bloco) da cache, e o algoritmo de substituição de blocos, que pode ser LRU, LFU, FIFO, FIFO segunda chance, e randômico. Para que a coerência dos dados seja mantida, a cache deve seguir o protocolo de diretório implementado pelo módulo de diretório.

A cache de instruções é acessada exclusivamente pelo estágio de busca do processador. Como apenas operações de leitura são permitidas, essa cache não precisa dar suporte às operações de diretório, e por isso tem uma implementação simplificada em relação à cache de dados. Por sua vez, a cache de dados é acessada apenas pelo estágio de memória do processador. Entretanto, sua implementação é mais complexa pois operações de leitura e escrita são permitidas, e para isso, essa cache deve suportar todas as operações de diretório.

Além de armazenar dados e instruções, o módulo de cache também é responsável pela conexão do processador com o seu roteador. Para isso, esse módulo conta com um gerenciador de comunicação, chamado de CaCoMa, que foi desenvolvido especialmente para a plataforma STORM. A forma como os módulos que compõem a cache estão interligados é mostrada na Figura 2.6.

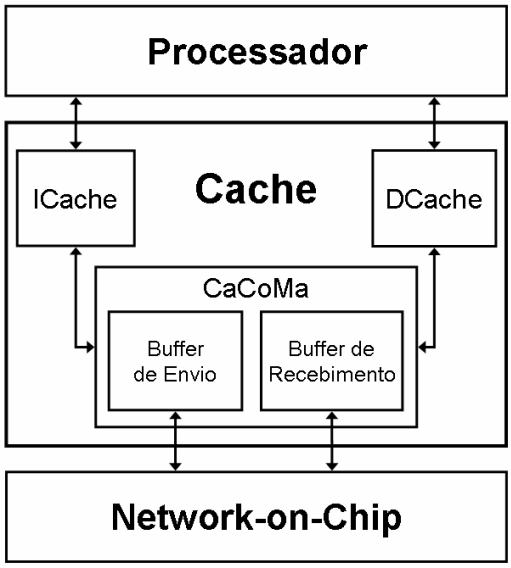

Figura 2.6 – Módulos internos da cache e suas inteligações.

O CaCoMa é responsável por enviar e receber pacotes da NoC, sendo sua responsabilidade também, a tradução de endereços lógicos de uma posição de memória para o endereço físico (da rede) do módulo de memória que contém o dado. Para isso, o CaCoMa utiliza a tabela ATA (*Address Table*) que é montada durante a inicialização da plataforma.

A ATA contém o endereço de NoC e o último endereço lógico de cada módulo de memória. Assim, para traduzir um endereço lógico X para endereço físico, o CaCoMa deve percorrer as entradas de sua ATA e retornar o endereço físico da primeira entrada cujo valor do último endereço lógico é maior que X. Um exemplo de ATA é mostrado na Tabela 2.4.

Tabela 2.4 – Exemplo de ATA.

Endereço Lógico	Endereço Físico
0x000007FF	0,1
0x00000FFF	1,0
0x000017FF	1,2

b) Diretório e Memória

No modelo de memória compartilhada, os diversos módulos de memória espalhados pela NoC formam um espaço de endereçamento único e visível por todos os processadores. Cada módulo de memória pode ter um tamanho diferente dos demais módulos de memória presentes em uma instância da plataforma. Entretanto, o tamanho do bloco é o mesmo tanto para cache quanto para os módulos de memória, podendo variar apenas em instâncias diferentes da plataforma.

Para centralizar as informações dos blocos de um módulo de memória e manter a coerência das caches, existe na plataforma STORM um módulo chamado Diretório. O Diretório mantém uma interface com um módulo de memória, sendo responsável pelo controle do funcionamento da mesma, e também com um roteador da NoC. Sendo assim, os pacotes destinados à memória são recebidos e tratados pelo diretório. O diretório implementado na plataforma STORM é caracterizado como um diretório *full-map*, uma vez que utiliza n bits por bloco (onde n é o numero de processadores) para manter

informações sobre qual cache possui cópia de qual bloco, mais um bit para sinalizar o estado deste bloco. Essas informações são guardadas na tabela chamada de STA (*Status Table*).

A STA informa quais caches de processadores contém uma cópia de um bloco da memória. Esta tabela contém B linhas e P+1 colunas, onde B é o número de blocos da memória e P é o número de processadores da instância da plataforma. Sendo assim, cada linha da tabela fornece informações de um bloco (linha 0 contém informações do bloco 0, linha 1 contém informações do bloco 1 e assim por diante) e cada coluna informa se um processador possui uma cópia de um bloco.

Uma coluna adicional indica se o bloco em questão está sujo ou não. Um bloco estar sujo significa que algum processador realizou uma operação de escrita neste bloco e, portanto, o bloco na memória não está atualizado. Esta situação acarreta numa necessidade (caso outra cache solicite uma copia deste bloco) da cache que contém a cópia suja deste bloco fazer um *write-back*, ou seja, enviar a cópia do bloco de volta para a memória. A Tabela 2.5 mostra um exemplo de uma STA para um sistema com 3 processadores.

Tabela 2.5 – Exemplo de STA.

Bloco	P0	P1	P2	Sujo
0	1	1	0	0
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0

Algumas situações exigem que o diretório envie ordens de coerência para a cache de um ou mais processadores. Sendo assim, é necessário que o diretório saiba o endereço de NoC dos processadores. Para isso, o Diretório mantém a PTA (*Processor Table*). Um exemplo de uma tabela PTA está ilustrado na Tabela 2.6.

Tabela 2.6 – Exemplo de PTA.

Processador	Endereço de NoC
P0	0,0
P1	1,0
P2	1,1

c) Coerência de Cache

Como dito anteriormente, a plataforma STORM utiliza um mecanismo baseado em diretório para manter a coerência das caches do sistema. Para o mecanismo de diretório funcionar, as caches precisam agir de forma integrada com o Diretório, informando qualquer operação sobre um bloco que possa gerar um estado de incoerência. As operações realizadas sobre blocos da cache, bem como os pacotes que eventualmente são enviados através da NoC em decorrência destas ações são listadas abaixo:

- *Read Hit*: Uma leitura de um endereço de memória foi solicitada pelo processador e o bloco correspondente ao endereço requisitado se encontra na cache. O dado é entregue ao processador e nenhum pacote é enviado ao diretório.
- *Read Miss*: Uma leitura de um endereço de memória foi requisitada pelo processador. Entretanto, o bloco correspondente ao endereço solicitado não se encontra na cache, que deve então enviar um pacote com o pedido de leitura do bloco ao diretório. O processador fica travado até que o dado seja retornado pela cache.
- *Write-Hit sem Permissão*: Um pedido de escrita em uma posição de memória foi enviado à cache pelo processador. O bloco relativo ao endereço solicitado se encontra na cache, porém sem permissão de escrita. Um pacote requisitando a permissão de escrita (*write-permit*) no bloco é enviado ao diretório pela cache. A execução do processador permanecerá bloqueada até que a escrita do dado seja confirmada pela cache.
- *Write Hit com Permissão*: O processador requisitou uma escrita em um endereço de memória cujo bloco correspondente se encontra na cache com permissão de escrita. A cache, então, confirma a escrita do dado e libera a execução do processador. Nenhum pacote é enviado ao diretório.
- *Write Miss*: Uma requisição de escrita em um endereço de memória foi enviada à cache pelo processador. Entretanto, o bloco relativo ao endereço solicitado não se encontra na cache, que deve então, pedir ao diretório uma

cópia do bloco com permissão de escrita. A execução do processador permanecerá bloqueada até que a cache confirme a escrita do dado.

Como foi visto acima, algumas operações da cache necessitam enviar requisições ao diretório. A forma como o diretório trata esses pedidos será detalhada a seguir. O caso mais simples de requisição é o pedido de substituição de pagina (*page-replacement*) sem atualização imediata (*write-back*), nesse caso o diretório simplesmente atualiza a sua tabela STA para indicar que a cache requisitante não possui mais uma cópia do bloco em questão.

Quando uma requisição de leitura chega, o diretório verifica na sua STA se o bloco está limpo. Caso esteja, a STA é atualizada para indicar que a cache requisitante possui uma cópia do bloco. E em seguida o diretório envia uma cópia do bloco para a cache. Já no caso em que o bloco está sujo, o diretório envia uma ordem de atualização imediata (*write-back*) sem invalidação para a cache que possui a cópia do bloco, guarda a requisição pendente (no caso uma leitura), e entra em modo de espera. Neste modo, apenas requisições de atualização imediata (*write-back*) são tratadas pelo diretório até que a atualização imediata (*write-back*) do bloco necessário para a requisição em espera seja recebido. Uma representação gráfica desse caso pode ser vista na Figura 2.7.

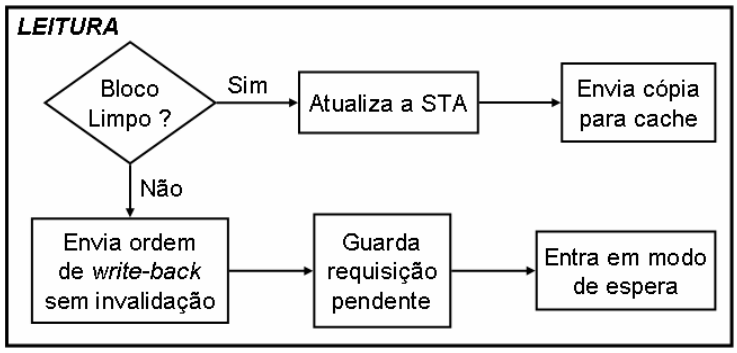

Figura 2.7 – Diagrama do funcionamento do diretório para um pedido de leitura.

Para um pedido de escrita, o diretório primeiro verifica em sua tabela STA o estado do bloco. Caso esteja sujo, o diretório age da mesma forma que em uma leitura, porém, nesse caso uma ordem de atualização imediata (*write-back*) com invalidação é enviada. Já quando o bloco está limpo, o diretório invalida as cópias das outras caches e em seguida atualiza a STA para indicar que a cache requisitante possui uma cópia do bloco e que agora o mesmo se encontra sujo. Por último, o diretório envia uma cópia do bloco com permissão

de escrita para a cache requisitante. A representação gráfica desse caso pode ser vista na Figura 2.8

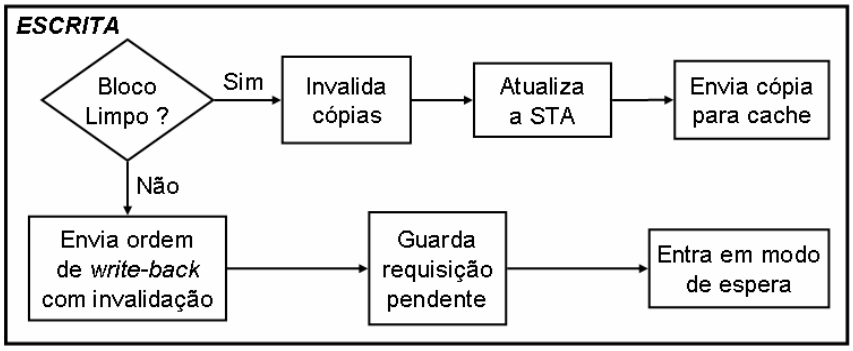

Figura 2.8 – Diagrama do funcionamento do diretório para um pedido de escrita.

No caso da permissão de escrita, o diretório verifica na STA se a cache requisitante ainda possui uma cópia do bloco, pois a mesma pode ter sido invalidada por uma ordem do diretório. Caso a cache não possua mais o bloco, o pedido é simplesmente ignorado. Se a cache ainda possui o bloco, as cópias do mesmo nas demais caches são invalidadas e em seguida a STA é atualizada para informar que o bloco se encontra sujo. Só então a permissão é enviada para a cache requisitante. A Figura 2.9 mostra uma representação gráfica desse caso.

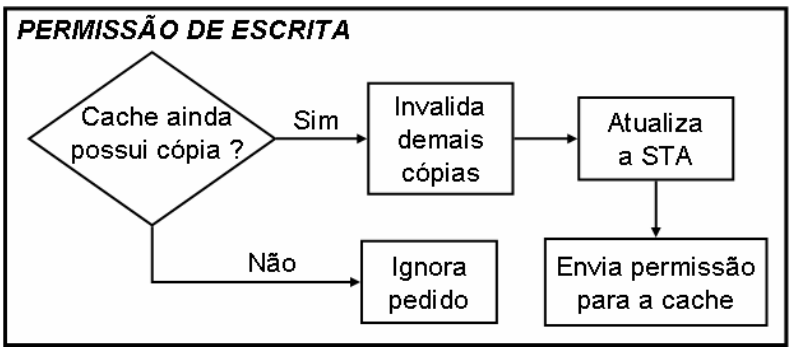

Figura 2.9 – Diagrama do funcionamento do diretório para um pedido de permissão de escrita.

O último caso que pode ser atendido pelo diretório é um pedido de atualização imediata (*write-back*), que pode ser voluntário (ocorre quando uma cache faz um substituição de página – *page-replacement* – de um bloco com permissão de escrita), ou forçado (no caso de uma resposta a uma ordem de atualização imediata – *write-back* – enviada pelo diretório), Entretanto, em ambos os casos o diretório age de forma semelhante. Se não existir uma requisição pendente, ou se o bloco sobre o qual está sendo

feito a atualização não for o esperado pela operação pendente (leitura ou escrita), o diretório atualiza o bloco na memória e altera a STA para indicar que o bloco se encontra limpo e sem cópias nas caches. No caso em que o bloco sobre o qual está sendo feito a atualização é o esperado pela requisição pendente, o diretório envia, simultaneamente, uma cópia do bloco para a memória e para a cache cuja requisição estava pendente. Em seguida, altera a STA de acordo com a operação pendente realizada e também registra que a cache que fez a atualização imediata (*write-back*) não possui mais o bloco. Uma vez que a operação em espera foi atendida, o diretório pode voltar ao modo normal de funcionamento. A representação gráfica desse caso pode ser vista na Figura 2.10.

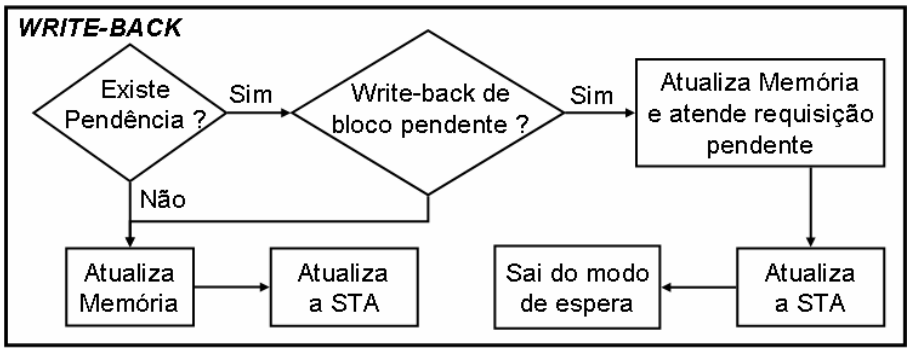

Figura 2.10 – Diagrama do funcionamento do diretório para uma operação de *write-back*.

2.2.4 Memória Distribuída

No modelo de memória distribuída implementado na plataforma STORM cada processador possui uma memória local acessada somente por ele. A comunicação entre processadores é feita através da troca de mensagens pela NoC. Para facilitar essa tarefa, uma biblioteca de troca de mensagens seguindo o padrão MPI (*Message-Passing Interface*) foi implementada e será detalhada posteriormente. A Figura 2.11 exibe o modo como os diversos módulos que compõem o modelo de memória distribuída da plataforma STORM estão interligados. O funcionamento de cada um destes módulos será detalhado a seguir.

a) Caches

Assim como no modelo de memória compartilhada, a cache do modelo de memória distribuída também se encontra dividida em cache de dados e cache de instruções para que o acesso a dados e instruções possa ser feito de forma simultânea. Tanto a cache de dados quanto a de instruções são completamente associativas. Entretanto, as demais configurações da cache de dados podem ser diferentes daquelas da cache de instruções.

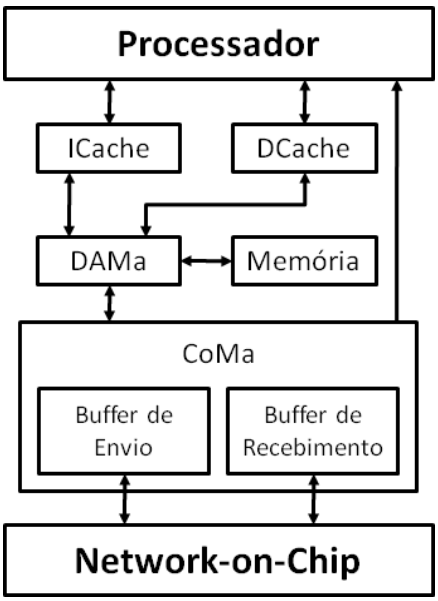

Figura 2.11 – Interligação dos módulos no modelo de memória distribuída.

As configurações das caches que podem ser alteradas são: tamanho da cache, e o algoritmo de substituição de blocos, que pode ser LRU, LFU, FIFO, FIFO segunda chance, e randômico. Outro parâmetro configurável é o tamanho da linha (bloco) das caches, sendo este, um parâmetro global que afeta todas as caches e módulos de memória presentes no sistema.

Por permitir apenas operações de leituras, a cache de instruções tem uma implementação simplificada em relação à cache de dados, que permite operações de leitura e escrita. Entretanto, como cada processador possui uma memória local que é escrita somente por ele, não é necessária a utilização de um mecanismo de manutenção de coerência de cache. Uma vez que os dados armazenados em cache não vão ser utilizados por outros processadores, optou-se por adiar a escrita em memória utilizando, para isso, a política de escrita *write-back*. Outra característica da cache de dados, que não está presente

na cache de instruções, é a capacidade de repassar dados para o processador sem armazená-los, se assim for ordenado pelo DAMa. Essa característica é importante para a troca de mensagens entre processadores e será detalhada posteriormente.

b) DAMa

O DAMa (*Data Access Manager*) é o módulo responsável pelo acesso da cache aos demais módulos do sistema. Sua principal função é fornecer uma interface padrão à cache, de forma que os módulos ligados ao DAMa possam ser acessados como se fossem módulos de memória. Essa interface permite a inclusão de novos módulos, como um módulo de entrada e saída, de forma simples e sem alteração no comportamento dos demais módulos utilizados no sistema.

Para cada módulo ligado ao DAMa é atribuída uma faixa de endereços de memória. Ao receber uma requisição da cache, o DAMa verifica a qual módulo pertence o endereço acessado, realiza a comunicação com o módulo correspondente e repassa os dados para a cache. Quando necessário, o DAMa pode enviar junto com os dados um sinal para avisar a cache que os dados não devem ser memorizados, mas apenas repassados ao processador. Esse comportamento é fundamental para o funcionamento correto de módulos que, diferentemente de módulos de memória, produzem novos dados durante seu funcionamento, como o CoMa e um módulo gerenciador de entrada e saída.

Como foi dito, cada módulo está mapeado em uma faixa de endereços de memória que é acessada pelo processador quando o mesmo deseja se comunicar com um determinado módulo. Se a cache armazenasse os dados dessa faixa de memória, o processador não perceberia mudanças no estado do módulo.

c) CoMa

O CoMa (*Communication Manager*) é responsável pelo envio e recebimento de pacotes da NoC. Como pode ser visto na Figura 2.11, o CoMa possui um *buffer* de envio e outro de recebimento de pacotes da NoC. O tamanho de cada *buffer* é configurável e independente um do outro. Um sinal de interrupção é gerado sempre que o *buffer* de recebimento do CoMa atinge um determinado nível. Esse nível é configurável e deve ser

ajustado de forma a evitar que o *buffer* de recebimento encha e acabe sobrecarregando os *buffers* dos roteadores da NoC, o que poderia prejudicar o desempenho do sistema como um todo.

Como foi visto na seção anterior, o CoMa é acessado pela cache de dados como um módulo de memória. De forma que seu conjunto de registradores, cujas funções serão descritas abaixo, se encontra mapeado em endereços de memória conforme a Tabela 2.7

- *Available SendBuffer* – informa o espaço disponível no *buffer* de envio;
- *Available RecBuffer* – informa a quantidade de dados disponíveis no *buffer* de recebimento;
- *SendBuffer Data* – utilizado para escrever dados no *buffer* de envio;
- *RecBuffer Data* – utilizado para acessar os dados armazenados no *buffer* de recebimento;

Tabela 2.7 – Conjunto de registradores do CoMa.

Endereço	Registrador
Endereço Base	Available SendBuffer
Endereço Base + 4	Available RecBuffer
Endereço Base + 8	SendBuffer Data
Endereço Base + 12	RecBuffer Data

Os registradores *Available SendBuffer*, *Available RecBuffer* e *RecBuffer Data* estão disponíveis somente para leitura e qualquer escrita nesses registradores não surtirá efeito. Já o registrador *SendBuffer Data* está disponível apenas para escrita, e qualquer leitura nesse registrador retornará o valor 0.

É importante notar que se uma operação de leitura no registrador *RecBuffer Data* inicia quando o *buffer* de recebimento está vazio, o processador e a cache ficarão travados até que um novo dado seja recebido pelo CoMa e a operação de leitura possa ser concluída. Se esse comportamento não é desejado, o programador deve primeiramente verificar a disponibilidade de dados no *buffer* de recebimento, utilizando para isso o registrador *Available RecBuffer*. Esse mesmo comportamento pode ser observado em operações de

escrita no registrador *SendBuffer Data* quando o *buffer* de envio se encontra cheio. Novamente, cabe ao programador verificar a disponibilidade de espaço em *buffer*, através do registrador *Available SendBuffer*, caso o mesmo deseje evitar esse comportamento.

Para enviar uma mensagem a outro nó da NoC, o processador deve primeiramente escrever no registrador *SendBuffer Data* uma palavra contendo o endereço de NoC do processador de destino e a quantidade de dados a ser enviada. O formato dessa palavra pode ser visto na Figura XX. Em seguida, o processador deve escrever os dados a serem enviados também no registrador *SendBuffer Data*. O CoMa, então, se encarrega de montar um pacote no formato específico da NoC e enviá-lo ao nó de destino.

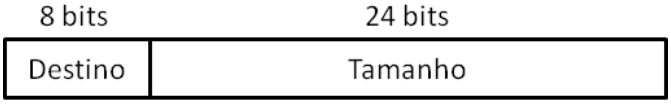

Figura 2.12 – Formato do cabeçalho de envio de mensagem.

Ao receber um pacote, o CoMa remove as informações utilizadas pela NoC e armazena a carga útil do pacote no *buffer* de recebimento. Nenhuma delimitação de pacotes é feita nesse *buffer*, e cabe ao software garantir a distinção de pacotes diferentes. Isso pode ser feito através do empacotamento dos dados na camada de software, da mesma forma que é feito em redes de computadores convencionais.

d) MPI

No modelo de memória distribuída, a comunicação entre processadores é feita através da troca de mensagens. Uma biblioteca que abstraia do programador os detalhes de baixo nível das operações de troca de mensagens é fundamental para o desenvolvimento de aplicações. O MPI é um padrão bem estabelecido e largamente utilizado para a troca de mensagens em clusters e supercomputadores com memória distribuída (MPI, 2008). Por esta razão, optou-se pela implementação de uma biblioteca para a STORM seguindo esse padrão.

Além dos benefícios citados acima, a utilização do padrão MPI permite também que aplicações existentes sejam portadas para a plataforma STORM com pouco, ou até mesmo nenhum, esforço. Essa portabilidade possibilita que novas aplicações sejam

validadas em um *cluster* de máquinas reais antes de serem simuladas na plataforma STORM, onde o tempo de simulação é elevado.

Dadas as restrições de tempo deste trabalho, apenas um subconjunto das operações especificadas no padrão MPI foi implementado. Entretanto, esse subconjunto é suficiente para implementar uma grande variedade de aplicações e pode ser expandido futuramente sem maiores complicações, uma vez que todos os tipos básicos e constantes especificadas pelo padrão MPI foram implementados. As funções implementadas e suas descrições estão listadas na Tabela 2.8.

Como a plataforma STORM ainda não conta com um sistema operacional nem com gerenciamento de memória, foram utilizados *buffers* de tamanho fixo, mas configurável, para armazenar as mensagens pendentes. Por isso, o programador deve garantir que o *buffer* foi configurado com um tamanho grande o suficiente para que a aplicação execute da maneira correta. Os outros parâmetros configuráveis são: o tamanho máximo de uma mensagem e o nó raiz do MPI. Esse nó é responsável pela inicialização e finalização do ambiente MPI.

Tabela 2.8 – Operações do padrão MPI implementadas e suas descrições.

Operação	Descrição
MPI_Comm_size	Retorna o número de processadores em um grupo
MPI_Comm_rank	Retorna o *rank* de um processador em um grupo
MPI_Init	Inicializa o ambiente MPI
MPI_Finalize	Finaliza o ambiente MPI
MPI_Recv	Recebe uma mensagem de outro nó da rede
MPI_Send	Envia uma mensagem a outro nó da rede
MPI_Bcast	Envia uma mensagem de um processador raiz para todos os outros processadores em um grupo
MPI_Reduce	Combina os dados fornecidos por cada processador de um grupo em um dado único
MPI_Pack	Empacota dados em uma região contígua de memória
MPI_Unpack	Desempacota dados contidos em uma região de memória contigua de acordo com o tipo de dado fornecido
MPI_Pack_size	Retorna a quantidade de bytes necessários para empacotar um determinado dado
MPI_Op_create	Cria uma função de combinação definida pelo usuário

3 SIMULAÇÃO DE RESERVATÓRIOS

Com a crescente demanda por recursos energéticos em todo o mundo, torna-se necessário para a sociedade otimizar o consumo de energia, buscar fontes alternativas, e sobretudo alcançar a auto-suficiência em fontes de energias vitais como o petróleo e gás-natural. Tal demanda fomentou a criação de áreas de atuação profissional, tais como a engenharia de petróleo e a engenharia de reservatórios, que tratam de todos os ramos relacionados à indústria do petróleo e, em particular, aos relacionados à exploração e produção.

Dentre as funções de um engenheiro de petróleo estão a quantificação de reservas, o dimensionamento de sistemas de produção, e a criação de projetos de extração de hidrocarbonetos. A execução de projetos de extração, que podem envolver investimentos de centenas de milhões de dólares, deve ter seus riscos calculados e minimizados. Entretanto, fatores físicos e químicos complexos como variações regionais das propriedades dos fluxos e características da permeabilidade relativa da rocha devem ser levados em consideração no cálculo do risco. A complexidade dessa tarefa levou à criação da simulação de reservatório.

Um simulador de reservatório é um modelo computacional capaz de gerar previsões precisas do comportamento de um reservatório de petróleo sob diferentes condições de operação (ERTEKIN; ABOU-KASSEN; KING, 2001). Esse modelo tem como base a teoria de fluxo em meios porosos (SOARES, 2002), que possui um campo de aplicação bastante abrangente, sendo empregada não só na engenharia de reservatórios, mas também em diversas outras áreas da ciência e engenharia como: modelagem de aqüíferos, mecânica dos solos, fluxo de contaminantes e análises de fluxo sanguíneo no interior do corpo humano.

Para a elaboração desse modelo computacional, é necessário o entendimento do comportamento dos fluidos em rocha reservatório, bem como o entendimento de seus componentes e de suas fases enquanto matéria. A partir desse entendimento é possível a elaboração de um modelo de reservatório que possa, por fim, ser simulado.

Em reservatórios de petróleo reais, diversos componentes formam as fases óleo, gás e água. Em função das condições de pressão e temperatura tais componentes se dissolvem

e vaporizam, migrando de uma fase para outra. Nessas situações podem ou não existir os mesmos componentes em cada uma das fases e, em conseqüência, alguma fase pode até mesmo deixar de existir (bastante comum com a fase gasosa).

3.1 Modelo do Reservatório

Para desenvolver um simulador de reservatório é necessário gerar um modelo do comportamento dos fluxos no reservatório de acordo com o fenômeno físico predominante. Diversos modelos foram desenvolvidos na tentativa de obter resultados cada vez mais próximos do comportamento real de um reservatório. Estes modelos tornam-se mais complexos à medida que características (como reações químicas, variação de temperatura e miscibilidade de fluxos) dos reservatórios são adicionadas. Alguns dos modelos de fluxo mais utilizados em simulações de reservatórios são: o modelo *black-oil*, o modelo composicional, os modelos térmicos e os modelos de fluxo miscível (MATTAX e DALTON, 1990).

O modelo *black-oil* considera o sistema com apenas três componentes – óleo, água e gás – e três fases, também designadas por óleo, água e gás. Assume-se que o componente óleo só existe na fase óleo, o componente água só existe na fase água e o componente gás pode encontra-se na fase gás, livre no reservatório, ou na fase óleo, associado ao componente óleo. Esse modelo considera também que a temperatura do reservatório é constante e que não há reações químicas entre os componentes. A sua utilização é recomendada para reservatórios que possuem óleos pesados e com baixa volatilidade, como os encontrados no Brasil (CORDAZZO, 2006).

Por obter resultados satisfatórios em espaços de tempo aceitáveis, o modelo *black-oil* – mesmo não sendo o modelo mais completo existente – é o mais utilizado pelos simuladores comerciais de reservatórios (FACHI; HARPOLE; BUJNOWSKI, 1982; CMG, 1995; SCHLUMBERGER, 2009). Os demais modelos são mais completos, entretanto, para a maioria dos casos práticos, são altamente custosos do ponto de vista computacional. Este trabalho, por não pretender apresentar contribuição específica no que ser refere a modelagem de reservatórios, abordará apenas o modelo *black-oil.*

3.2 Formulação Matemática.

As equações de fluxo para o modelo *black-oil* são obtidas a partir de equações básicas, que são: equações de conservação de massa, equações de estado e a lei de Darcy. Essas equações são descritas detalhadamente em (DAKE, 1978) e (ERTEKIN; ABOU-KASSEM; KING, 2001) e serão apresentadas a seguir de forma resumida.

3.2.1 Equações de Estado

As equações de estado descrevem o comportamento das propriedades do meio poroso e dos fluidos presentes no sistema. Estas propriedades podem variar em função das pressões e saturações dos fluidos, sendo responsáveis pelo comportamento não linear do sistema. Algumas destas propriedades importantes para o modelo *black-oil* são descritas a seguir.

a) Propriedades da Rocha

As propriedades da rocha que interessam ao problema de fluxo em reservatórios de petróleo são a porosidade, ϕ, e a permeabilidade absoluta, k, que apresentam valores distintos para cada ponto do reservatório.

A porosidade ϕ é definida em um determinado volume da rocha como sendo a relação entre o volume de vazios e o volume de rocha total considerado. Devido à compressibilidade da rocha, normalmente assumida como constante, a porosidade é dependente da pressão p no reservatório. A porosidade da rocha em função da pressão é expressa da seguinte forma:

$$\phi(p) = [1 + c_r(p - p^{ref})] \tag{3.1}$$

Onde p^{ref} é a pressão de referência na qual a porosidade é ϕ^{ref}. A pressão de referência é normalmente a pressão atmosférica ou a pressão inicial do reservatório.

A permeabilidade absoluta k representa uma medida da facilidade apresentada pelo meio para o fluido escoar através de seus poros, e é representada através de um tensor de permeabilidades absolutas K, que contém os valores de permeabilidades segundo as direções x, y e z, admitindo um sistema de coordenadas cartesiano, mostrado abaixo:

$$K = \begin{bmatrix} k_x & k_{xy} & k_{xz} \\ k_{yx} & k_y & k_{yz} \\ k_{zx} & k_{zy} & k_z \end{bmatrix} \tag{3.2}$$

Em simulação de reservatórios, é comum admitir que as direções preferenciais de fluxo coincidem com as direções principais do tensor de permeabilidade absolutas, podendo ser substituído por um tensor diagonal representado por:

$$K = \begin{bmatrix} k_x & & \\ & k_y & \\ & & k_z \end{bmatrix} \tag{3.3}$$

Esta hipótese de fluxo segundo as direções principais nem sempre ocorre, trazendo alguns problemas nos resultados da simulação, dependendo da orientação da malha utilizada na discretização do modelo (MATTAX; DALTON, 1990). Admite-se que não há variação das permeabilidades absolutas durante o processo de simulação.

b) Propriedades dos Fluidos

Algumas propriedades dos fluidos são importantes para a avaliação do comportamento do fluxo nas condições específicas dos reservatórios. Estas propriedades são: compressibilidade, fator volume de formação, densidade e viscosidade. Estas propriedades são intrínsecas ao fluido e podem variar com a pressão e temperatura no reservatório.

Para um fluido l, a compressibilidade c_l é definida como uma medida da variação do volume do fluido em relação à pressão. Admitindo uma temperatura constante no reservatório, tem-se as relações:

$$c_l = -\frac{1}{V_l}\frac{\partial V_l}{\partial p_l} \tag{3.4}$$

$$c_l = \frac{1}{\rho_l}\frac{\partial \rho_l}{\partial p_l} \tag{3.5}$$

Onde V_l e ρ_l são o volume e a densidade em condições de reservatório, respectivamente.

Uma definição bastante utilizada no modelo *black-oil* é o fator volume de formação B_l, representado pela relação entre o volume do fluido em condições de reservatório, V_l, sobre o volume do fluido em condições padrão, V_{lsc}. Admite-se para a condição padrão a pressão de 1 atm e temperatura de 20°C. Sendo, então, o fator volume de formação expresso da seguinte forma:

$$B_l = \frac{V_l}{V_{lsc}} \tag{3.6}$$

Para fluidos levemente compressíveis, como água e óleo pesado, a expressão para a variação do volume de formação com a pressão no reservatório pode ser obtida integrando-se a Equação 3.5, admitindo uma temperatura constante, resultando em:

$$B_l(p_l) = \frac{B_l^{ref}}{[1 + c_l(p_l - p^{ref})]} \tag{3.7}$$

Onde B_l^{ref} é o fator volume de formação correspondente a uma pressão de referencia p^{ref}, normalmente 1 atm.

A densidade, ou massa especifica, também varia com a pressão. Partindo da Equação 3.6, a densidade em condições de reservatório pode ser escrita em função da densidade em condições padrão, ρ_{lsc}, como:

$$\rho_l = \frac{\rho_{lsc}}{B_l} \tag{3.8}$$

Utilizando as Equações 3.7 e 3.8, a variação da densidade em função da pressão no reservatório pode ser expressa por:

$$\rho_l(p_l) = \rho_l^{ref}[1 + c_l(p_l - p^{ref})] \tag{3.9}$$

Onde ρ_l^{ref} é a densidade de referência a uma pressão p^{ref}.

Outra propriedade importante é a viscosidade μ_l, que representa a dificuldade que o fluido apresenta para escoar quando submetido a um gradiente de pressão. Considerando a temperatura do reservatório constante, a água e o óleo pesado têm variação de viscosidade muito pequena, crescendo à medida que a pressão aumenta.

c) Propriedades Rocha-Fluido

Algumas propriedades variam em função de características conjuntas da rocha e do fluido considerado, como saturação S_l, permeabilidade relativa k_{rl}, e pressão de capilaridade P_c, descritas a seguir.

A saturação S_l em um sistema multifásico representa o percentual do volume poroso ocupado pelo fluido, e é dada pela relação entre o volume ocupado, em condições de reservatório, e o volume de vazios. No modelo *black-oil* as saturações das fases seguem a equação de restrição abaixo.

$$S_o + S_a + S_g = 1 \tag{3.10}$$

A permeabilidade relativa k_{rl} representa um fator de redução da permeabilidade absoluta devido à presença de outros fluidos no espaço poroso, sendo calculada em função da saturação.

A pressão de capilaridade P_c consiste na diferença de pressão entre a fase não-molhada e a fase molhada. Em sistemas óleo/água a água é a fase molhada, já em sistemas gás/óleo a fase molhada é o óleo, resultando em:

$$P_{coa}(S_a) = p_o - p_a \tag{3.11}$$

$$P_{cgo}(S_g) = p_g - p_o \tag{3.12}$$

As diferenças P_{coa} e P_{cgo} são, respectivamente, as pressões de capilaridade óleo-agua e gás-oleo. Sabe-se que P_{coa} e P_{cgo} variam com a saturação da água e do gás, respectivamente.

3.2.2 Lei de Darcy

A lei de Darcy é uma expressão empírica que descreve o fluxo de fluidos através de um meio poroso, estabelecendo uma relação entre a velocidade superficial e o gradiente do potencial (BEAR, 1972; DAKE, 1978). O potencial Φ é definido pela fórmula abaixo:

$$\Phi_l = p_l - \gamma_l Z \tag{3.13}$$

Onde p_l é a pressão a uma profundidade Z, calculada em relação a um ponto de referência e positiva para baixo, γ_l é o peso específico do fluido definido por $\gamma_l = \rho_l g$ onde g é a aceleração da gravidade e ρ_l é a densidade do fluido. Assumindo um fluxo multifásico unidimensional, a velocidade superficial, ou velocidade de Darcy, segundo a direção x pode ser escrita para o fluxo l através da expressão:

$$u_{lx} = -\frac{k_x k_{rl}}{\mu_l}\frac{\partial \Phi_l}{\partial x} \tag{3.14}$$

Onde k_x é a permeabilidade absoluta na direção x, μ_l é a viscosidade e k_{rl} é a permeabilidade relativa.

Substituindo a Equação 3.13 na Equação 3.14, tem-se:

$$u_{lx} = -\frac{k_x k_{rl}}{\mu_l}\left(\frac{\partial p_l}{\partial x} - \gamma_l \frac{\partial Z}{\partial x}\right) \tag{3.15}$$

Na equação acima a densidade é considerada constante, embora seja sempre atualizada em função da pressão calculada no passo de tempo anterior, durante o processo de simulação. Esta consideração pode ser aplicada também a fluidos levemente compressíveis (ERTEKIN; ABOU-KASSEM; KING, 2001).

No caso multidimensional, admitindo um sistema de coordenadas cartesiano, a Equação 3.14 pode ser escrita da seguinte forma:

$$u_l = -K\frac{k_{rl}}{\mu_l} r\Phi_l \tag{3.16}$$

Onde u_l é o vetor de velocidade de Darcy, composto pelos componentes de velocidade nas direções x, y e z, e K é o tensor diagonal de permeabilidades absolutas representado por 3.3.

3.2.3 Conservação de Massa

A lei da conservação de massa estabelece o balanço de massa para um determinado fluido, tomando como referencia um elemento de volume de controle. Para o sistema de coordenadas cartesiano, o elemento de controle representado pela Figura 3.1 é utilizado na dedução das equações de balanço de massa, assumindo que o volume do elemento não varia com o tempo.

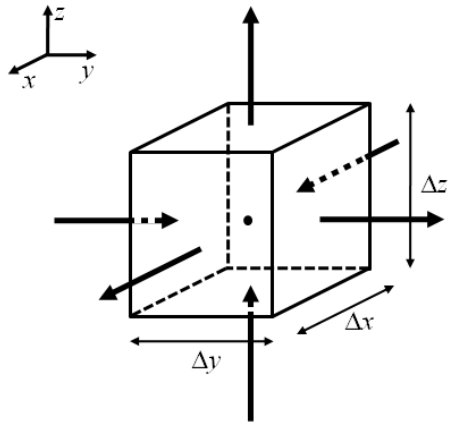

Figura 3.1 – Elemento de Controle

Admitindo que o fluxo de massa entra ou sai do elemento, paralelo aos eixos do sistema cartesiano, através das faces do elemento, tem-se a seguinte expressão para o balanço de massa do componente l:

$$(m_e - m_s)_l + \left(m_p\right)_l = (m_a)_l \tag{3.17}$$

Onde:

m_e – massa que entra no elemento através das suas faces;

m_s – massa que sai do elemento através das suas faces;

m_p – massa que entra ou sai no elemento através de um poço no seu interior;

m_a – massa acumulada no elemento durante um intervalo de tempo Δt;

No lado esquerdo da Equação 3.17, os termos de massa que entra e sai do elemento por suas faces e o termo de massa devido ao poço – injetor ou produtor – localizado no seu interior, são chamados termos de fluxo e termo de poço, respectivamente, enquanto o termo do lado direito é chamado termo de acumulação.

Os termos de fluxo podem ser escritos, em relação ao ponto central de coordenadas (x, y, z), da seguinte forma:

$$(m_e)_l = (\dot{m}_{lx}A_x)_{x-\frac{\Delta x}{2}} + \left(\dot{m}_{ly}A_y\right)_{y-\frac{\Delta y}{2}} + (\dot{m}_{lz}A_z)_{z-\frac{\Delta z}{2}} \tag{3.18}$$

$$(m_s)_l = (\dot{m}_{lx}A_x)_{x+\frac{\Delta x}{2}} + \left(\dot{m}_{ly}A_y\right)_{y+\frac{\Delta y}{2}} + (\dot{m}_{lz}A_z)_{z+\frac{\Delta z}{2}} \tag{3.19}$$

Onde A_x, A_y, e A_z são áreas das faces do elemento segundo as direções x, y e z respectivamente, e $(\dot{m}_{lx})_{x\pm\frac{\Delta x}{2}}$ representa o fluxo de massa do componente l por unidade de área que entra ou sai do elemento pela face $x \pm \frac{\Delta x}{2}$ por unidade de tempo, podendo ser definido como:

$$\dot{m}_{lx} = \rho_l u_{lx} \tag{3.20}$$

Onde ρ_l é a densidade do componente l e u_{lx} é a velocidade de Darcy, que representa uma vazão por unidade de área. Definição análoga vale para o fluxo de massa segundo as direções y e z.

Definindo q_{ml} como sendo o fluxo de massa do componente l que entra ($q_{ml} > 0$) ou sai ($q_{ml} < 0$) no elemento devido ao poço (injetor ou produtor) localizado no seu interior, tem-se:

$$\left(m_p\right)_l = q_{ml}\Delta t \tag{3.21}$$

Escrevendo o fluxo de massa q_{ml} em termos de vazão q_l, dada em volume por unidade de tempo, tem-se:

$$q_{ml} = \rho_l q_l \tag{3.22}$$

O termo de acumulação representa a variação da massa acumulada entre os tempos t e $t + \Delta t$ sendo escrito por:

$$(m_a)_l = (m_{vl})^{t+\Delta t} - (m_{vl})^t \tag{3.23}$$

Onde $m_{vl} = S_l \phi \rho_l V_b$ é a massa do componente l contida no elemento, $V_b = \Delta x \Delta y \Delta z$ é o volume do elemento, ϕ é a porosidade do elemento e S_l é a saturação do componente, resultando em:

$$(m_a)_l = (S_l \phi \rho_l \Delta x \Delta y \Delta z)^{t+\Delta t} - (S_l \phi \rho_l \Delta x \Delta y \Delta z)^t \tag{3.24}$$

Substituindo as Equações 3.18, 3.19, 3.21 e 3.24 na Equação 3.17, rearranjando os termos e dividindo o resultado pelo volume do elemento, tem-se:

$$-\left[\frac{(\dot{m}_{lx})_{x+\frac{\Delta x}{2}} - (\dot{m}_{lx})_{x-\frac{\Delta x}{2}}}{\Delta x}\right] - \left[\frac{\left(\dot{m}_{ly}\right)_{y+\frac{\Delta y}{2}} - \left(\dot{m}_{ly}\right)_{y-\frac{\Delta y}{2}}}{\Delta y}\right]$$

$$-\left[\frac{(\dot{m}_{lz})_{z+\frac{\Delta z}{2}} - (\dot{m}_{lz})_{z-\frac{\Delta z}{2}}}{\Delta z}\right] + \frac{q_{ml}}{\Delta t} = \left[\frac{(S_l \phi \rho_l)^{t+\Delta t} - (S_l \phi \rho_l)^t}{\Delta t}\right] \tag{3.25}$$

Escrevendo a equação acima em termos de derivadas parciais e substituindo os termos de fluxo de massa pelas Equações 3.20, deduzida para cada direção, e 3.22, tem-se:

$$-\frac{\partial}{\partial x}(\rho_l u_{lx}) - \frac{\partial}{\partial y}\left(\rho_l u_{ly}\right) - \frac{\partial}{\partial z}(\rho_l u_{lz}) + \frac{\rho_l q_l}{V_b} = \frac{\partial}{\partial t}(S_l \phi \rho_l) \tag{3.26}$$

Ou

$$-\frac{\partial}{\partial x}(\rho_l u_{lx} A_x)\Delta x - \frac{\partial}{\partial y}\left(\rho_l u_{ly} A_y\right)\Delta y - \frac{\partial}{\partial z}(\rho_l u_{lz} A_z)\Delta z + \rho_l q_l = V_b \frac{\partial}{\partial t}(S_l \phi \rho_l) \tag{3.27}$$

A Equação 3.27 representa a forma final da equação de balanço de massa para o componente l, admitindo um sistema de eixos cartesiano. Esta equação pode ser obtida em relação a outros sistemas de eixos, tornando-se mais conveniente de acordo com as características do problema que se deseja resolver (ERTEKIN; ABOU-KASSAM; KING, 2001).

3.2.4 Equações de Fluxo

Tendo estabelecido as equações básicas para o problema, a forma final das equações que regem o fluxo em um sistema óleo-água-gás são obtidas combinando-se estas equações. Considerando a densidade dos componentes em condições padrão, ρ_{lsc}, constante e substituindo a Equação 3.8 e a Equação 3.15 escrita para as direções y e z, na Equação 3.27, chega-se a:

$$\frac{\partial}{\partial x}\left[A_x k_x \frac{k_{rl}}{\mu_l B_l}\left(\frac{\partial p_l}{\partial x} - \gamma_l \frac{\partial Z}{\partial x}\right)\right]\Delta x + \frac{\partial}{\partial y}\left[A_y k_y \frac{k_{rl}}{\mu_l B_l}\left(\frac{\partial p_l}{\partial y} - \gamma_l \frac{\partial Z}{\partial y}\right)\right]\Delta y +$$

$$\frac{\partial}{\partial z}\left[A_z k_z \frac{k_{rl}}{\mu_l B_l}\left(\frac{\partial p_l}{\partial z} - \gamma_l \frac{\partial Z}{\partial z}\right)\right]\Delta z = V_b \frac{\partial}{\partial t}\left(\frac{S_l \phi}{B_l}\right) - q_{lsc} \qquad l = a, o, g \qquad (3.28)$$

Onde, $q_{lsc} = q_l / B_l$, é a vazão do termo de poço em condições padrão.

A Equação 3.28 representa o sistema de equações que rege o problema do fluxo tridimensional para o modelo *black-oil* trifásico em reservatórios, usando um sistema de coordenadas cartesiano. O sistema é composto por três equações diferenciais parciais dependentes do tempo, uma para cada componente l, cujas incógnitas são as pressões e saturações de cada componente, no caso p_o, p_a, p_g, S_o, S_a e S_g. Como o número de incógnitas é maior que o número de equações, é necessário introduzir algumas relações constitutivas para que seja possível resolver o problema.

As relações constitutivas que relacionam as incógnitas da Equação 3.28 são a equação de restrição das saturações 3.10 e as equações da pressão de capilaridade 3.11 e 3.12, reescritas abaixo:

$$S_o + S_a + S_g = 1 \qquad (3.29)$$

$$P_{cgo}(S_g) = p_g - p_o \qquad (3.30)$$

$$P_{coa}(S_a) = p_o - p_a \qquad (3.31)$$

Usando estas equações, é possível reescrever o sistema 3.28 eliminando três incógnitas e deixando-o apenas em função das três variáveis primárias, pressão de óleo p_o, saturação do óleo S_o e saturação de água S_a, resultando no sistema de equações diferenciais final com três equações e três incógnitas:

$$\frac{\partial}{\partial x}\left[A_x k_x \frac{k_{ro}}{\mu_o B_o}\left(\frac{\partial p_o}{\partial x} - \gamma_o \frac{\partial Z}{\partial x}\right)\right]\Delta x +$$

$$\frac{\partial}{\partial y}\left[A_y k_y \frac{k_{ro}}{\mu_o B_o}\left(\frac{\partial p_o}{\partial y} - \gamma_o \frac{\partial Z}{\partial y}\right)\right]\Delta y +$$

$$\frac{\partial}{\partial z}\left[A_z k_z \frac{k_{ro}}{\mu_o B_o}\left(\frac{\partial p_o}{\partial z} - \gamma_o \frac{\partial Z}{\partial z}\right)\right]\Delta z =$$

$$V_b \frac{\partial}{\partial t}\left(\frac{S_o \phi}{B_o}\right) - q_{osc}$$

$$\frac{\partial}{\partial x}\left[A_x k_x \frac{k_{ra}}{\mu_a B_a}\left(\frac{\partial p_o}{\partial x} - \frac{\partial P_{coa}}{\partial x} - \gamma_a \frac{\partial Z}{\partial x}\right)\right]\Delta x +$$

$$\frac{\partial}{\partial y}\left[A_y k_y \frac{k_{ra}}{\mu_a B_a}\left(\frac{\partial p_o}{\partial x} - \frac{\partial P_{coa}}{\partial x} - \gamma_a \frac{\partial Z}{\partial y}\right)\right]\Delta y +$$

$$\frac{\partial}{\partial z}\left[A_z k_z \frac{k_{ra}}{\mu_a B_a}\left(\frac{\partial p_o}{\partial x} - \frac{\partial P_{coa}}{\partial x} - \gamma_a \frac{\partial Z}{\partial z}\right)\right]\Delta z =$$

$$V_b \frac{\partial}{\partial t}\left(\frac{S_a \phi}{B_a}\right) - q_{asc}$$

$$\frac{\partial}{\partial x}\left[A_x k_x \frac{k_{rg}}{\mu_g B_g}\left(\frac{\partial P_{cog}}{\partial x} + \frac{\partial p_o}{\partial x} - \gamma_g \frac{\partial Z}{\partial x}\right)\right]\Delta x +$$

$$\frac{\partial}{\partial y}\left[A_y k_y \frac{k_{rg}}{\mu_g B_g}\left(\frac{\partial P_{cog}}{\partial x} + \frac{\partial p_o}{\partial x} - \gamma_g \frac{\partial Z}{\partial y}\right)\right]\Delta y +$$

$$\frac{\partial}{\partial z}\left[A_z k_z \frac{k_{rg}}{\mu_g B_g}\left(\frac{\partial P_{cog}}{\partial x} + \frac{\partial p_o}{\partial x} - \gamma_g \frac{\partial Z}{\partial z}\right)\right]\Delta z =$$

$$V_b \frac{\partial}{\partial t}\left[\frac{(1 - S_o - S_a)\phi}{B_g}\right] - q_{gsc} \qquad (3.32)$$

3.3 Discretização

As equações diferenciais parciais que regem o fluxo em um sistema óleo-água-gás obtidas na seção anterior são muito complexas para serem resolvidas de forma analítica, sendo necessário o emprego de técnicas de discretização numérica para obter uma solução aproximada.

Diversos métodos numéricos de solução de equações diferenciais parciais estão disponíveis na literatura, sendo os seguintes os mais utilizados em simuladores de reservatórios:

- Método de Volumes Finitos
- Método de Elementos Finitos
- Método de Diferenças Finitas

O papel de qualquer um dos métodos numéricos acima é obter uma solução aproximada de uma equação diferencial parcial através da construção e solução de um sistema de equações algébricas. As descrições destes métodos, detalhadas em (ZWILLINGER, 1998), (AMES, 1977) e (MATTAX; DALTON, 1990), fogem do escopo deste trabalho e por isso não serão mostradas.

Em simulações de reservatório existem dois níveis de discretização: no espaço e no tempo. A discretização do tempo consiste em dividir o domínio do tempo em intervalos, onde, em cada passo de tempo, são calculadas as incógnitas do problema.

Para aproximar derivadas no espaço é necessário definir uma malha de discretização, como a mostrada na Figura 3.2, com o objetivo de tratar o eixo coordenado contínuo através de pontos isolados. Cada bloco da malha possuiu um valor médio para cada uma das características (saturação, pressão, porosidade, etc) do reservatório.

O processo de discretização do sistema de equações diferenciais parciais leva a um sistema de equações lineares. O próximo, e último, passo para a construção de um simulador de reservatórios é a solução desse sistema de equações lineares através de métodos numéricos que serão discutidos na próxima seção.

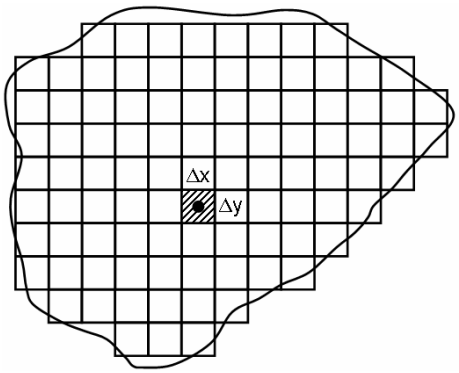

Figura 3.2 – Exemplo de discretização espacial em duas dimensões.

3.4 Solução de Sistemas Lineares

A discretização do sistema de equações diferenciais parciais pelos métodos citados na seção anterior leva a um sistema de equações lineares. De acordo com (SILVA et al, 2003), cerca de 80% do tempo total de execução de um simulador de reservatório é gasto na etapa de solução do sistema de equações lineares. Por isso, é de fundamental importância empregar métodos de solução eficientes, capazes de resolver o problema rapidamente, utilizando pouca memória, sem perder a precisão na solução.

Existem vários métodos para solução de sistemas de equações lineares, divididos, basicamente, em dois grupos: métodos diretos e métodos iterativos. Os métodos diretos utilizam algoritmos de fatoração para encontrar a solução exata do sistema em um número fixo de passos. Já os métodos iterativos tentam aproximar, a cada iteração, uma solução inicial (pouco precisa) à solução real do sistema. As iterações são realizadas até que a solução aproximada seja suficientemente precisa.

Para resolver problemas de larga escala, como os da simulação de reservatório, a utilização de métodos iterativos é mais adequada que a utilização de métodos diretos (FANCHI; HARPOLE; BUJNOWSKI, 1982). Isso se deve à maior paralelização alcançada pelos métodos iterativos e, principalmente, à utilização de uma quantidade de memória significativamente menor que a necessária pelos métodos diretos. Dessa forma, apenas alguns algoritmos desse grupo serão mostrados a seguir.

3.4.1 Métodos Iterativos

Antes de apresentar os algoritmos vamos apresentar as definições necessárias. Um sistema de equações lineares pode ser escrito da seguinte forma:

$$\begin{array}{rl} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n & = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n & = b_2 \\ \vdots & \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n & = b_n \end{array} \tag{3.33}$$

Alternativamente, esse mesmo conjunto de equações pode ser expresso numa forma mais compacta utilizando a notação de matriz, como abaixo.

$$A\bar{x} = \bar{b} \tag{3.34}$$

Onde A é a matriz de coeficientes, $\bar{x}$ e $\bar{b}$ são vetores coluna como mostrado abaixo.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \quad \bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \bar{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{3.35}$$

a) Método de Jacobi

No método iterativo de Jacobi, uma solução aproximada para o componente x_i^{k+1} na iteração $k+1$ é encontrada em função dos demais elementos da iteração anterior, k, conforme a Equação 3.36. Dessa forma, o vetor $\bar{x}^k$ não pode ser sobrescrito até o final da iteração $k+1$.

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1\ j\neq i}^{n} a_{i,j}x_j^{(k)}\right) \qquad i = 1, 2, \ldots, n \tag{3.36}$$

b) Método de Gauss-Seidel

O método de Gauss-Seidel é obtido a partir do método de Jacobi, diferenciando apenas que os componentes do vetor solução que já foram calculados na iteração $k+1$ são

imediatamente utilizados no cálculo dos demais componentes desta mesma iteração, resultando na seguinte expressão:

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j<i}^{n} a_{i,j}x_j^{(k+1)} - \sum_{j>i}^{n} a_{i,j}x_j^{(k)}\right) \qquad i = 1, 2, \dots, n \qquad (3.37)$$

No método de Jacobi, observa-se que os elementos x_i^{k+1} podem ser calculados independentes da ordem em que cada elemento é calculado. No método de Gauss-Seidel a ordem em que o elemento x_i^{k+1} é calculado, segundo a Equação 3.37, é importante, pois seu valor depende dos elementos já calculados nesta mesma iteração. Normalmente o número de iterações no método Gauss-Seidel é menor que no método de Jacobi.

c) Método SOR (*Successive Over-Relaxation*)

O método SOR é obtido a partir de uma sobre-relaxação do método de Gauss-Seidel. Seu funcionamento consiste na obtenção de uma média ponderada entre a iteração de Gauss-Seidel e a iteração anterior para cada componente, resultado em:

$$x_i^{k+1} = \omega x_i^{k+1} + (1 - \omega)x_i^k \qquad (3.38)$$

Onde x_i^{k+1} é o componente da iteração de Gauss-Seidel, obtido pela Equação 3.38, e ω é o fator de sobre-relaxação. O uso de fator ω adequado acelera a convergência do método, reduzindo o número de iterações necessárias para resolver o sistema. Quanto ω é igual a 1, o método SOR é igual ao método de Gauss-Seidel.

3.4.2 Opções do Trabalho

Um simulador de reservatórios de petróleo consiste em um modelo matemático que tem como objetivo prever o comportamento dos fluidos (água, gás, óleo) no decorrer do tempo em um determinado reservatório de petróleo. Essa ferramenta é de grande importância para o gerenciamento de reservatórios, sendo utilizada em todas as fases, desde a descoberta até o abandono do campo. A precisão de um simulador de reservatório é diretamente proporcional à quantidade de propriedades dos fluidos e da rocha utilizadas

para a obtenção do modelo matemático. Entretanto, a utilização de um número maior de características leva também ao aumento do custo computacional da simulação. Por isso, é importante encontrar um equilíbrio entre precisão e custo computacional.

As complexas equações resultantes do modelo gerado não podem ser resolvidas analiticamente e, por isso, técnicas de discretização numérica são utilizadas para obter uma solução aproximada. Nesse processo de discretização, o intervalo de tempo que se deseja simular é divido em subintervalos de tamanho fixo e a geometria do reservatório é subdividida, de forma aproximada, através de uma malha de blocos.

A aplicação das técnicas de discretização do espaço e do tempo gera, para cada bloco da malha, um sistema de equações lineares que deve ser resolvido para cada subintervalo de tempo. Dessa forma, a resolução de sistemas de equações lineares é uma atividade extremamente importante na simulação de reservatório e chega a ocupar 80% do tempo da simulação (SILVA et al, 2003).

Dada a importância dos métodos numéricos de solução de sistemas de equações lineares no contexto dos simuladores de reservatório e as restrições de tempo de implementação e de simulação do presente trabalho, optou-se pela implementação de um desses métodos e não um simulador de reservatório completo. É importante destacar que essa opção não causa nenhuma perda na qualidade dos resultados, uma vez que o objetivo deste trabalho é estudar o impacto de aplicações computacionalmente massivas, como as da indústria do petróleo e gás natural, em ambiente MPSoC e não propor melhorias nas técnicas de modelagem e simulação de reservatórios. Detalhes sobre o método de solução de sistemas de equações lineares implementado e a paralelização utilizada serão dados no próximo capítulo.

4 EXPERIMENTAÇÃO

Como foi visto no capítulo anterior, a resolução de sistemas de equações lineares é um passo de grande importância na simulação de reservatório, podendo chegar a ocupar 80% do tempo da simulação. Dada a importância da resolução de sistemas de equações e devido às restrições de tempo de implementação e simulação impostas a este trabalho, optou-se pela implementação do método de resolução de sistemas de equações lineares SOR, detalhado na seção 3.4.1, e não pela implementação de um simulador de reservatório completo.

O método SOR tem como característica convergir mais rapidamente para uma solução que os demais métodos mostrados na seção 3.4.1. Esse fator foi de fundamental importância para a sua escolha devido às restrições de tempo desse trabalho e ao alto custo das simulações da plataforma STORM. Outro fator determinante é a sua utilização pelo simulador de reservatórios BOAST (*Black Oil Applied Simulation Tool*). Esse simulador, que é bastante utilizado até hoje, foi desenvolvido pelo Departamento de Energia dos Estados Unidos e está documentado em (FANCHI; HARPOLE; BUJNOWSKI, 1982).

4.1 Paralelização

Como foi visto na seção 3.4.1, a cada iteração a Equação 3.38 é utilizada para encontrar o novo valor de cada uma das n incógnitas do sistema. Assim, o algoritmo foi paralelizado de forma que cada um dos p processadores é responsável por encontrar, a cada iteração, o novo valor de n/p incógnitas do sistema, conforme exemplificado pela Figura 4.1. Essa divisão de tarefas foi utilizada tanto no modelo de memória compartilhada quanto no modelo de memória distribuída.

$$\underbrace{x_1 \quad x_2 \quad x_3}_{p_1} \quad \underbrace{x_4 \quad x_5 \quad x_6}_{p_2} \quad \underbrace{x_7 \quad x_8 \quad x_9}_{p_3}$$

Figura 4.1 – Divisão de tarefas entre processadores.

A Equação 3.38 utiliza a Equação 3.37 para encontrar os valores de x_i^{k+1}. Esta equação, por sua vez, necessita dos valores já atualizados das incógnitas cujos índices são menores que i. Devido à forma como as tarefas foram divididas, esses valores, ou a maioria deles, são calculados por outros processadores gerando então dependência de dados. Assim, para obter o melhor paralelismo possível, cada processador calcula primeiramente a parte da Equação 3.37 que utiliza os valores das incógnitas da iteração passada e fica na espera dos dados necessários para a continuação de sua tarefa. Quando um processador termina de calcular os novos valores das incógnitas sob sua responsabilidade, todos os outros processadores podem prosseguir seus cálculos até que uma nova dependência seja alcançada. Esse processo continua até que todos os processadores concluam suas tarefas.

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j<i}^{n} a_{i,j}x_j^{(k+1)} - \sum_{j>i}^{n} a_{i,j}x_j^{(k)}\right) \qquad i = 1, 2, \ldots, n \qquad (3.37)$$

Tomemos o processador p_2, da Figura 4.1, calculando o novo valor de x_4 como exemplo. Primeiramente p_2 calcula o segundo somatório da Equação 3.37 utilizando os valores das incógnitas x_5, x_6, x_7, x_8 e x_9 encontrados na iteração passada. Em seguida p_2 deve calcular o primeiro somatório da Equação 3.37 utilizando os valores atualizados de x_1, x_2 e x_3. Entretanto, esses valores são atualizados pelo processador p_1, devendo p_2 então esperar a conclusão das tarefas de p_1 para poder concluir o calculo de x_4.

No modelo de memória compartilhada, todos os processadores executam a mesma aplicação e, por isso, a divisão de tarefas é feita no código. Para isso, cada processador precisa de um identificador único fornecido pela função *get_id*, que retorna valores de 0 a $n - 1$, onde n é o número de processadores presentes no sistema. A divisão de tarefas é feita então por estruturas condicionais baseadas no valor do identificador único.

Nesse modelo de memória, toda troca de dados entre os processadores é realizada através do uso de variáveis compartilhadas e *mutexes*. Assim, para paralelizar o algoritmo do método SOR, um *mutex* inicialmente travado foi atribuído a cada processador presente no sistema. Quando um processador deseja ler os dados produzidos por outro processador, ele deve tentar obter o *mutex* correspondente, ficando bloqueado até que esse *mutex* seja liberado. Então, quando um processador termina suas tarefas, ele imediatamente libera o seu *mutex* fazendo com que o primeiro processador travado a espera dos dados seja

liberado para executar. Uma vez liberado o *mutex*, o primeiro processador a travar novamente o *mutex* vai ler os dados compartilhados e em seguida destravar novamente o *mutex* para que os demais processadores possam acessar os dados compartilhados também.

Todos os processadores do sistema esperam o processador mestre calcular o erro da iteração para poder começar a iteração seguinte. No início de cada iteração, o *mutex* de cada processador é novamente travado para que os dados compartilhados sejam lidos apenas quando os seus cálculos estiverem concluídos. Esse processo é repetido até que um número limite de iterações seja atingido ou até que o erro calculado pelo mestre esteja dentro de uma faixa tida como aceitável.

No modelo de memória distribuída, cada processador executa sua própria aplicação, tornando o processo de divisão de tarefas mais simples e claro. Ainda assim, quando utilizando a biblioteca MPI, cada processador recebe um identificador único, chamado *rank*, que pode ser usado para dividir as tarefas de forma semelhante à realizada no modelo de memória compartilhada, desde que uma cópia do mesmo código binário seja executada por cada processador presente no sistema.

Nesse modelo de memória, a comunicação entre os processadores é realizada através de trocas explícitas de mensagens. Assim, na paralelização do método SOR, quando um processador necessita de um dado gerado por outro processador ele tenta receber uma mensagem desse processador. Como as funções do MPI utilizadas são bloqueantes o processador fica travado até que a mensagem desejada chegue. Dessa forma, quando um processador termina suas tarefas, os dados calculados devem ser enviados para todos os outros processadores do sistema através da função de *broadcast* do MPI.

Novamente, ao término da iteração o processador mestre calcula o erro enquanto os outros processadores aguardam. Se o número máximo de iterações for atingido ou o erro calculado for menor que o limite considerado aceitável, uma mensagem é enviada para todos os processadores avisando do término da execução. Se nenhuma das condições for atendida, uma mensagem iniciando a próxima iteração é enviada para todos os processadores.

4.2 Dados de Entrada

Simuladores de reservatório reais recebem como entrada dados obtidos em campo pelos mais diversos equipamentos e dados obtidos em laboratório. Esses dados, quando na forma bruta, estão em diversas escalas e precisam ser tratados antes de serem enviados ao simulador. Entretanto, como o algoritmo escolhido para ser simulado é um método de resolução de sistemas lineares, a etapa de tratamento de dados não é necessária e dados genéricos podem ser passados como entrada.

O método SOR recebe como entrada a matriz de coeficientes do sistema, A, e o vetor de termos independentes do sistema, $\bar{b}$, mostrados na Equação 3.34. Esses dados foram gerados randomicamente obedecendo às condições de convergência do método SOR descritas em (YOUNG, 1954). O algoritmo então foi executado com esses dados em um computador convencional variando o fator de sobre-relaxação, ω, até que o valor ótimo fosse encontrado. O sistema gerado tem 200 equações, pois, devido ao alto custo da simulação da plataforma STORM, simulações com quantidades maiores de equações levam tempos inviáveis para as restrições de tempo desse trabalho.

4.3 Instâncias da Plataforma STORM

Foram utilizadas 14 instâncias da plataforma STORM, sendo seis utilizando o modelo de memória compartilhada e oito utilizando o modelo de memória distribuída. Em todas as instâncias simuladas as caches têm 512 bytes de tamanho. Pesquisas anteriores (GIRÃO et al, 2007; OLIVEIRA, 2006) mostraram que caches maiores que esse valor geram um tráfego apenas ligeiramente menor na NoC, não justificando assim a sua utilização.

No modelo de memória compartilhada, todos os processadores acessam o módulo de memória. Assim, visando minimizar o tempo médio de acesso desse módulo, o mesmo foi posicionado em um nó central da malha em todas as instâncias. A Figura 4.2 mostra a disposição dos módulos nas instâncias simuladas utilizando modelo de memória compartilhada.

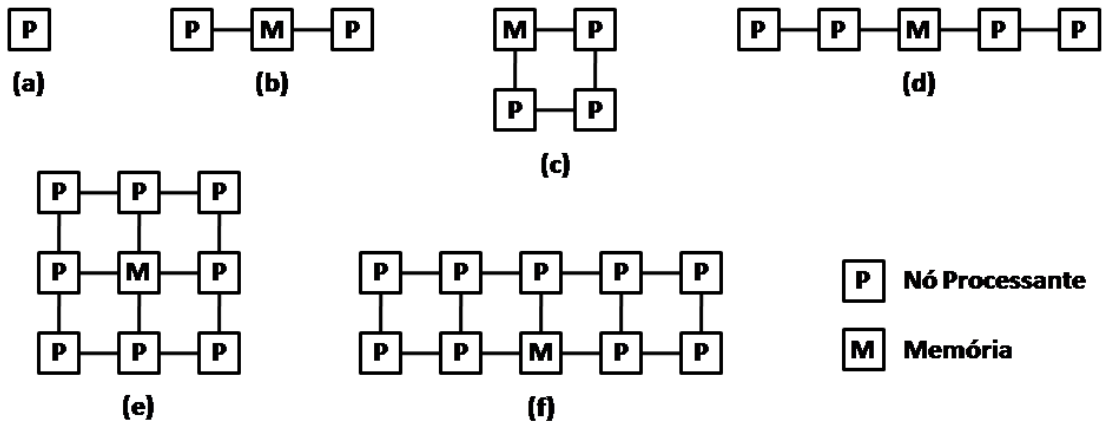

Figura 4.2 – Instâncias simuladas da plataforma STORM com modelo de memória compartilhada.

Para o modelo de memória distribuída foram simuladas as instâncias mostradas na Figura 4.3. Como pode ser visto, duas instâncias com três (c, d) e quatro (e, f) nós processantes foram simuladas. As instâncias (d), (f) e (g) foram utilizadas para obter condições de tráfego na NoC semelhantes as das instâncias (c), (d) e (e) do modelo de memória compartilhada. No caso das instâncias (d) e (g), o espaço destinado ao módulo de memória, necessário apenas no modelo de memória compartilhada, foi ocupado por um nó inativo. A aplicação executada nesse nó apenas pára o processador evitando que o mesmo gere tráfego na rede e acabe por interferir nos resultados. As demais instâncias dos dois modelos de memória foram criadas visando minimizar a distância máxima entre os nós sem a necessidade de utilizar nós inativos.

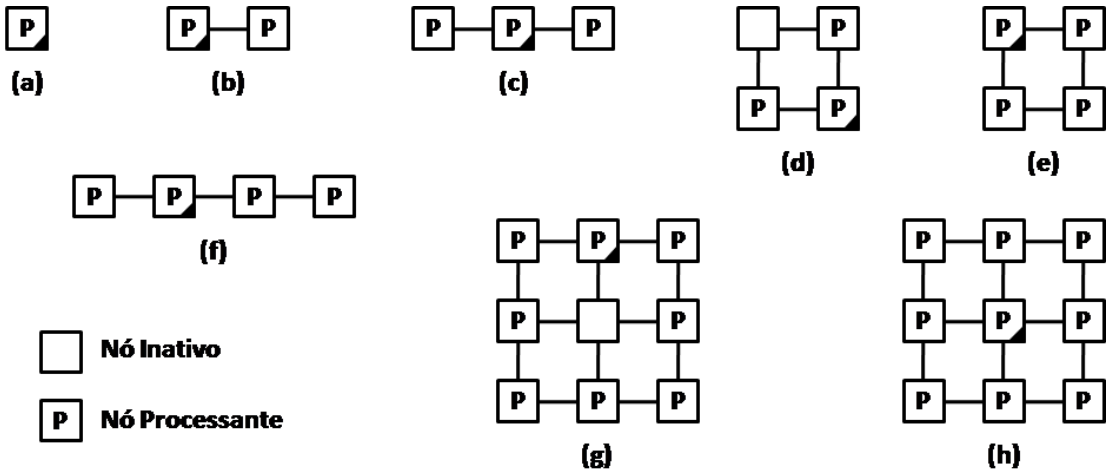

Figura 4.3 – Instâncias simuladas da plataforma STORM com modelo de memória distribuída.

No modelo de memória distribuída, o nó raiz da biblioteca MPI é acessado por todos os outros nós processantes do sistema para a troca de informações internas da biblioteca. Além disso, esse nó foi utilizado pela aplicação como ponto inicial das mensagens de *broadcast*. Sendo assim, este trabalho optou por posicioná-lo de forma central na malha visando minimizar a sua distância média para os demais nós do sistema e, conseqüentemente, reduzir o custo total de execução da aplicação.

4.4 Resultados

Como foi dito anteriormente, foram utilizadas duas instâncias para três (c, d) e quatro (e, f) nós processantes no modelo de memória distribuída, sendo as instâncias (d) e (f) semelhantes às instâncias utilizadas no modelo de memória compartilhada. A Tabela 4.1 mostra que a diferença dos resultados dessas instâncias com mesma quantidade de nós processantes é insignificante. Sendo assim, apenas os valores obtidos pelas instâncias (c) e (e) serão utilizados ao apresentar os resultados para três e quatro nós processantes no modelo de memória distribuída.

Tabela 4.1 – Comparação das instâncias com mesmo número de nós processantes.

	(c)	**(d)**	**(e)**	**(f)**
Ciclos Simulados	314768623	314824799	258013265	258046145
CPI Médio	2,757	2,757	2,776	2,776
Carga Total Injetada	280320	280320	334020	334020
Latência Média	36,163	36,163	36,575	36,998
SpeedUp	1,454	1,453	1,773	1,773

A Figura 4.4 apresenta os resultados de carga total injetada na NoC para cada modelo de memória. Como esperado, o modelo de memória compartilhada gera um tráfego muito superior ao gerado pelo modelo de memória distribuída. A Figura 4.5 mostra a carga injetada no modelo de memória compartilhada separada em carga gerada pela troca de dados e carga gerada pela coerência de cache. Como pode ser visto, a carga injetada nesse modelo de memória é dominada pela troca de dados, pois a carga gerada pela coerência de

cache é praticamente insignificante. Esse resultado está em conformidade com os resultados obtidos em trabalhos anteriores (GIRÃO et al, 2007; OLIVEIRA, 2006).

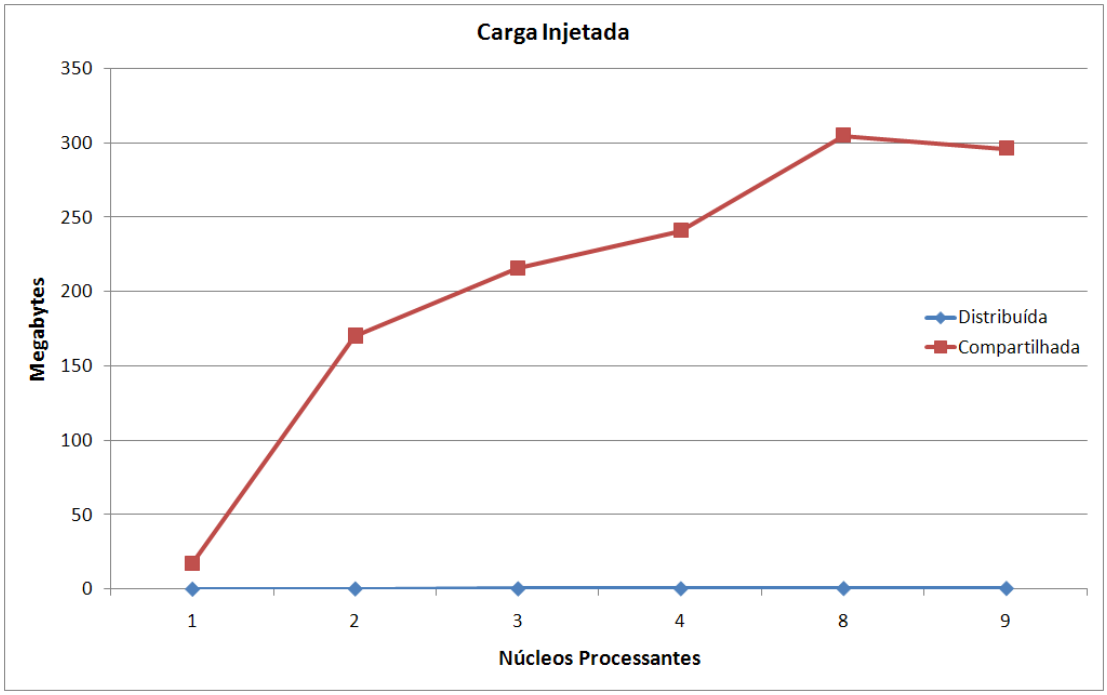

Figura 4.4 – Carga total injetada na NoC.

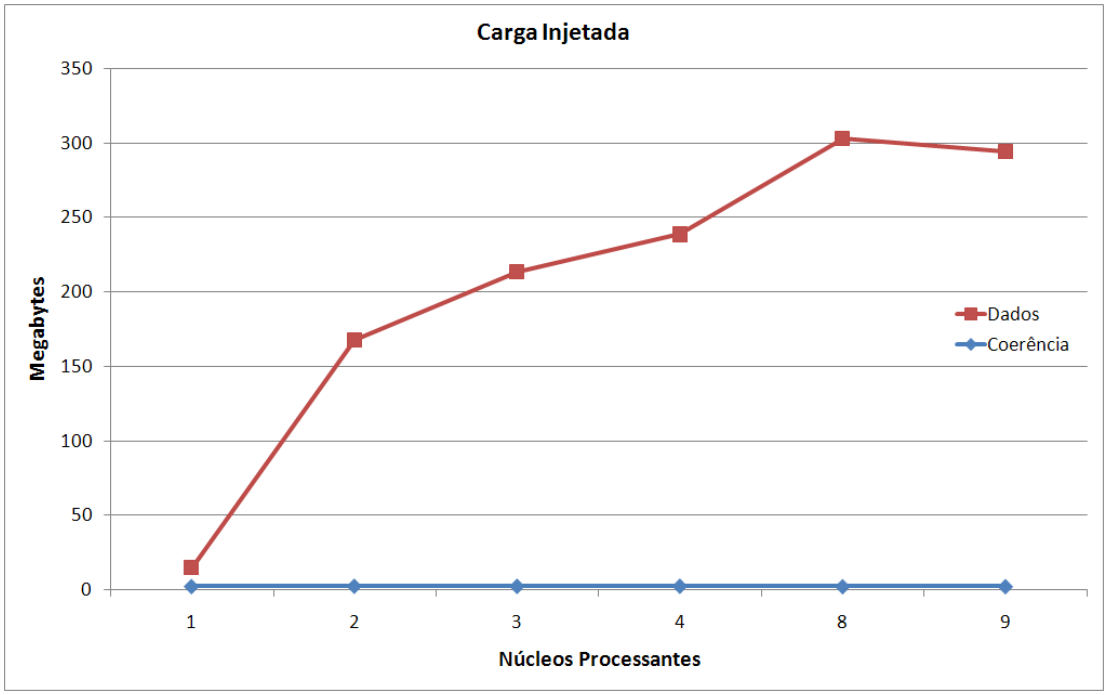

Figura 4.5 – Carga total injetada no modelo de memória compartilhada.

Na Figura 4.6 os resultados de carga injetada para cada modelo de memória são apresentados em escalas diferentes. A escala da esquerda (Megabytes) corresponde ao modelo de memória compartilhada, enquanto que o modelo de memória distribuída utiliza a escala da direita (Kilobytes). Nesse gráfico fica claro que, apesar de ser baixa, a carga gerada pelo modelo de memória distribuída se comporta de forma semelhante à carga gerada pelo modelo de memória compartilhada.

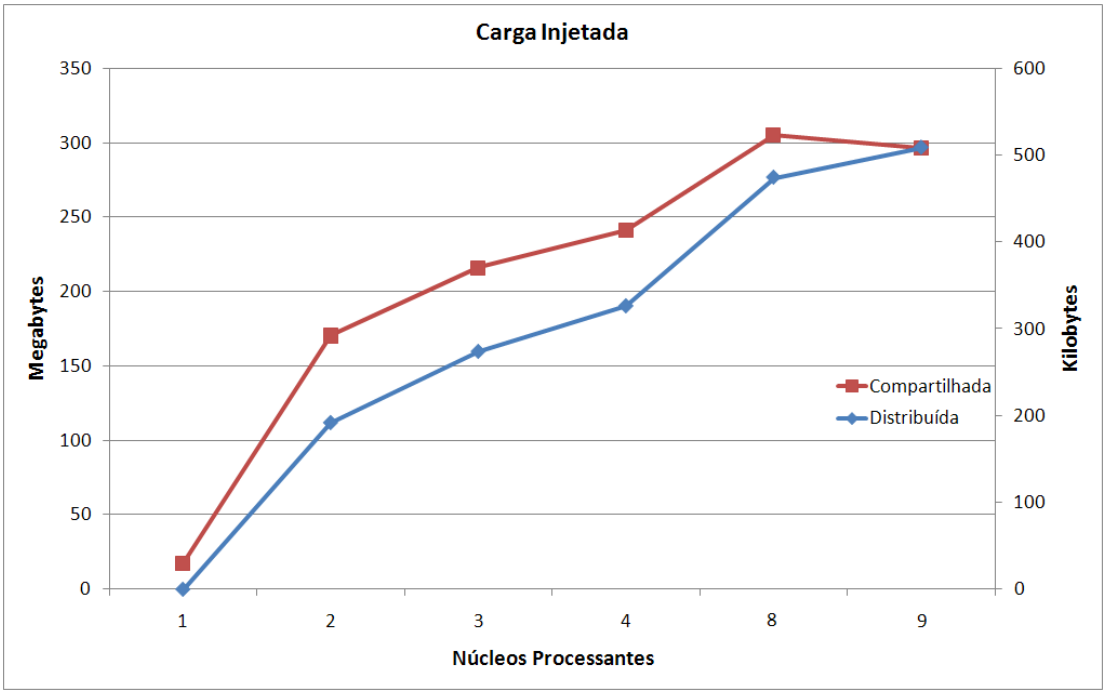

Figura 4.6 – Carga total injetada na NoC em escalas diferentes.

A Figura 4.7 mostra que o número de ciclos por instrução é consideravelmente menor no modelo de memória distribuída. Podemos notar, ainda, que esse valor se manteve praticamente constante para o modelo de memória distribuída enquanto que para o modelo de memória compartilhada esse valor cresce à medida que nós processantes são adicionados ao sistema. Esse comportamento pode ser explicado pela concorrência no acesso aos dados que é característico do modelo de memória compartilhada.

A quantidade de ciclos que cada instância simulada levou para executar a aplicação é exibida na Figura 4.8. Podemos notar que esse valor é semelhante em ambos os modelos de memória simulados, apesar da vantagem obtida pelo modelo de memória distribuída nos resultados de ciclos por instrução e carga injetada exibidos anteriormente. Esse resultado pode ser creditado ao *overhead* causado pela execução da biblioteca de troca de

mensagens, pois, como podemos observar na Figura 4.9, a quantidade média de instruções executadas por processador é maior no modelo de memória distribuída.

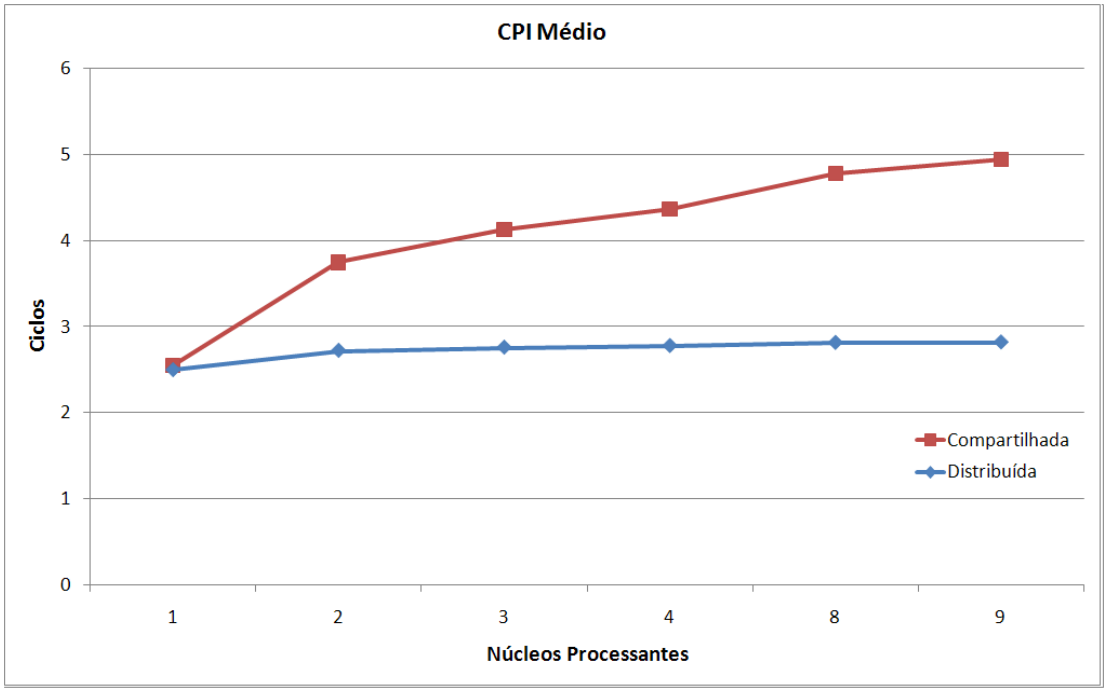

Figura 4.7 – Número de médio de ciclos por instrução.

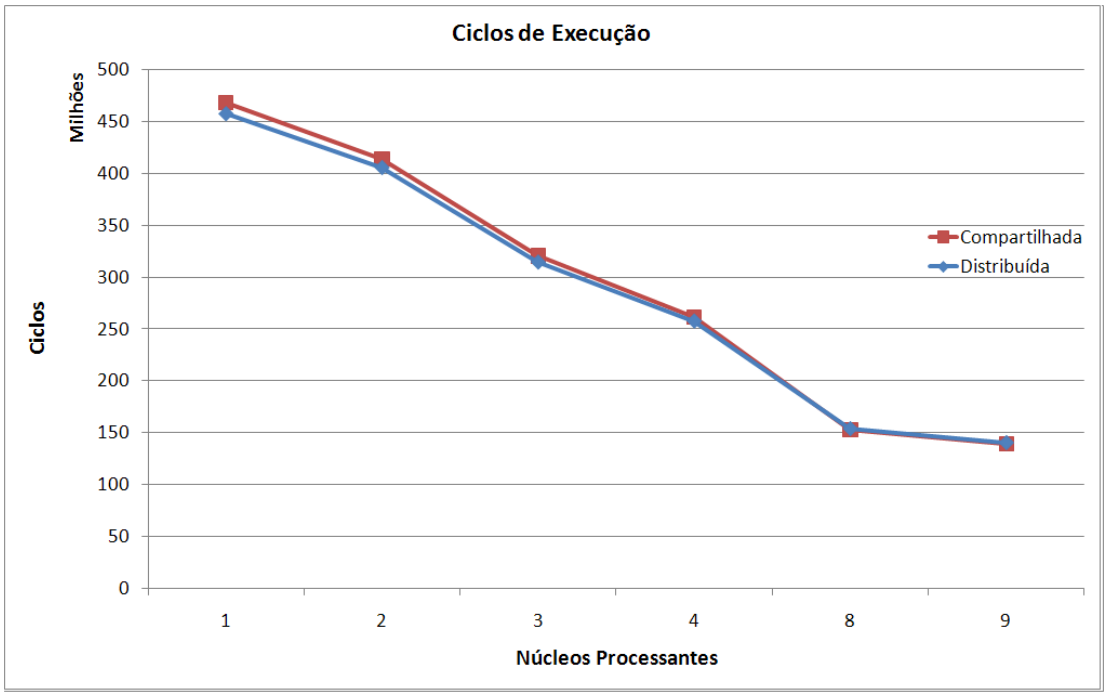

Figura 4.8 – Número total de ciclos simulados.

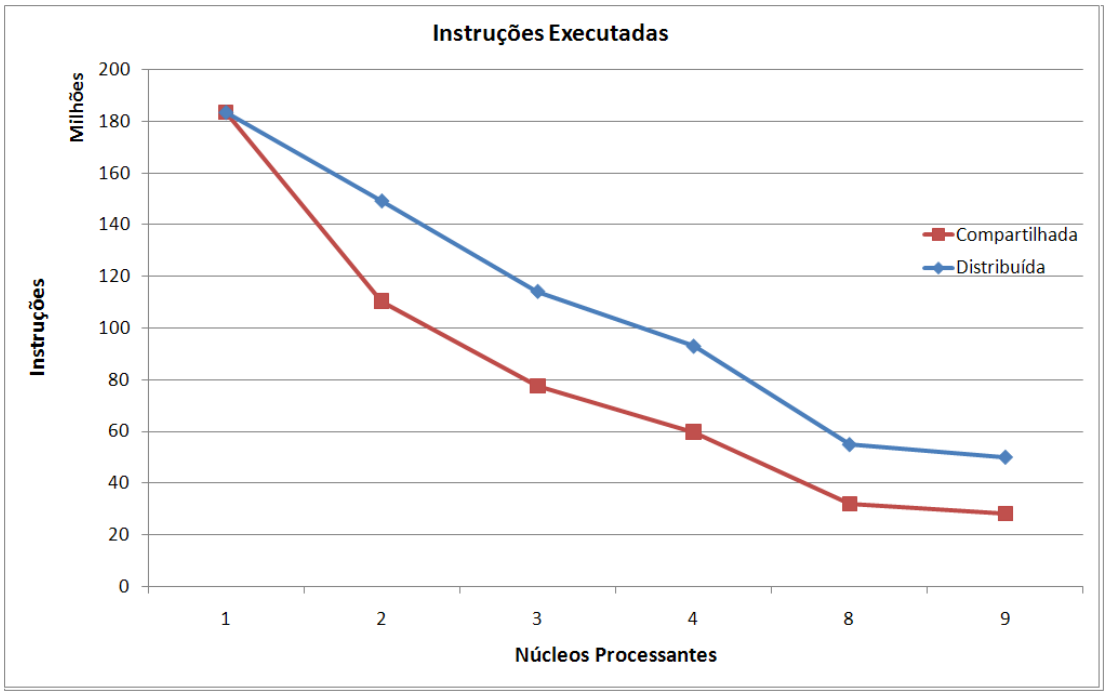

Figura 4.9 – Número médio de instruções executadas por processador.

A Figura 4.10 mostra o tempo gasto, em segundos, para simular cada instância. Como pode ser visto, o tempo necessário para simular a mesma aplicação cresce à medida que a quantidade de módulos do sistema aumenta. Este resultado demonstra apenas a forma como o custo de simulação da plataforma STORM cresce e não deve ser utilizado como parâmetro de comparação dos modelos de memória, pois o mesmo depende da carga da máquina no momento em que a simulação foi realizada. A Figura 4.11 mostra a quantidade de ciclos simulados por segundo.

Os resultados de latência média dos pacotes na rede são exibidos na Figura 4.12. Esses valores foram obtidos calculando a média da latência dos pacotes que chegam a cada roteador, e em seguida calculando a média desses valores. Podemos ver no gráfico que a latência média dos pacotes no modelo de memória distribuída é aproximadamente três vezes maior que a latência média do modelo de memória compartilhada. Esse fato ocorre por uma combinação de dois fatores: o chaveamento *store-and-forward* utilizado pela NoC, e o tamanho dos pacotes de NoC.

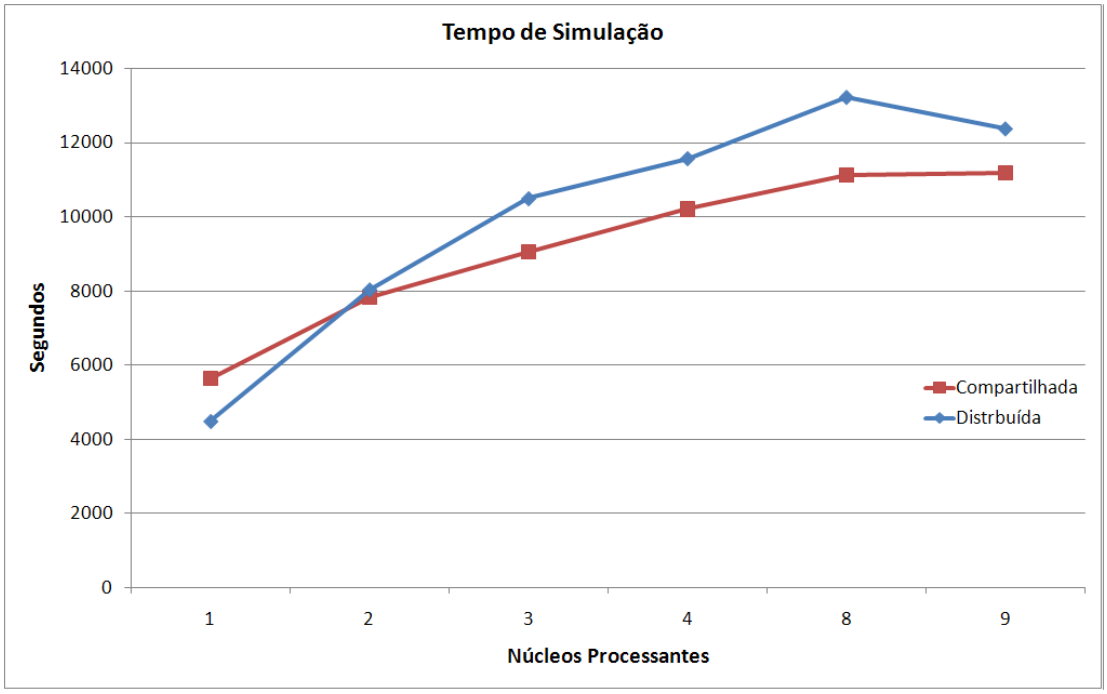

Figura 4.10 – Tempo de simulação.

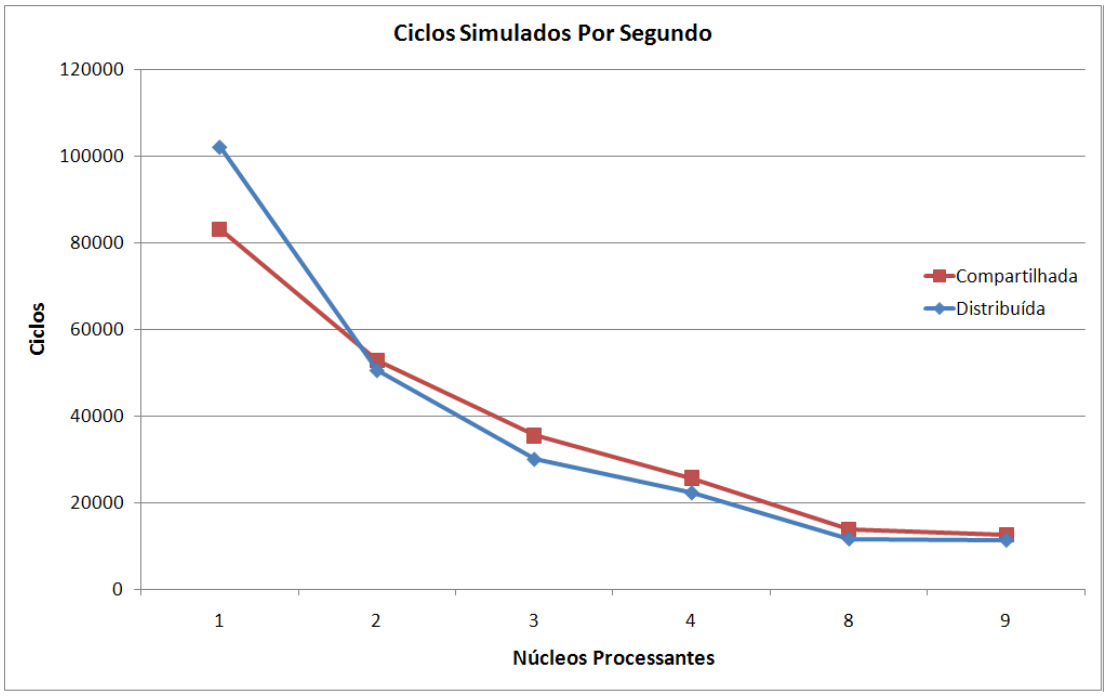

Figura 4.11 – Ciclos simulados por segundo.

No modelo de memória compartilhada, o tráfego é totalmente gerado pelos pedidos de dados das caches. Dessa forma, o tamanho médio do pacote de NoC é proporcional ao tamanho do bloco de cache. Já no modelo de memória distribuída, o tamanho do pacote depende do tamanho das mensagens enviadas pela aplicação. Como na aplicação simulada é necessária a troca de vetores entre os processadores do sistema, os pacotes de NoC tendem a ser maiores no modelo de memória distribuída. No chaveamento *store-and-forward* os pacotes devem ser armazenados inteiramente nos roteadores intermediários. Assim, a latência tende a aumentar quando o tamanho do pacote aumenta.

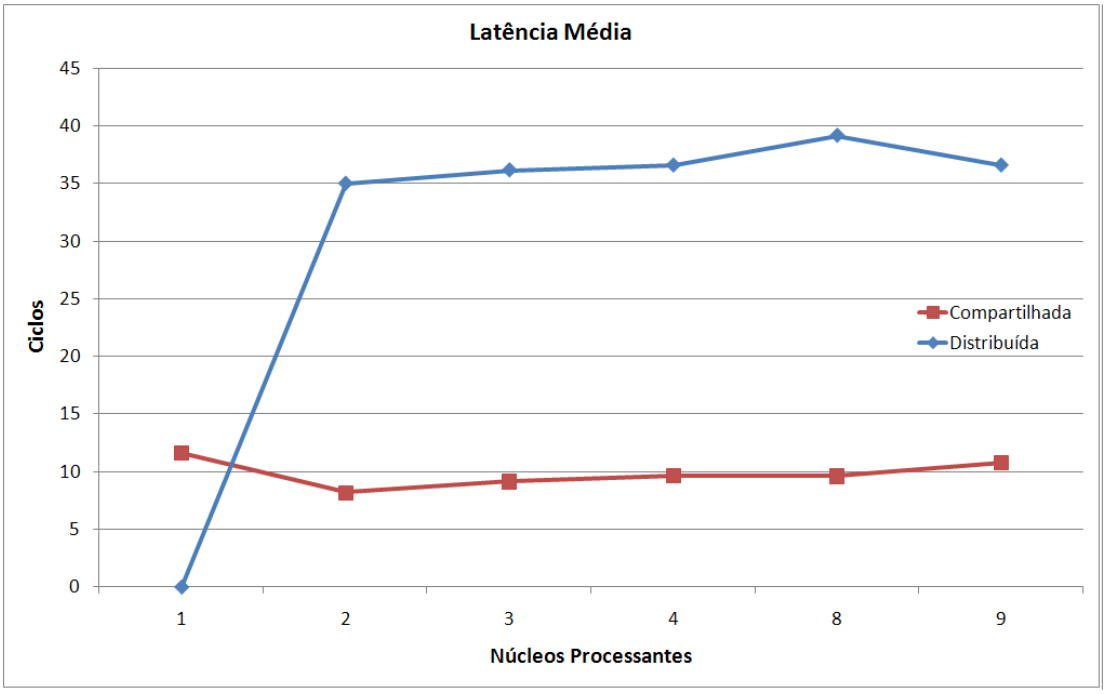

Figura 4.12 – Latência média.

A Figura 4.13 mostra o *speedup* absoluto da aplicação simulada nos dois modelos de memória. Podemos ver que o ganho de performance proporcionado pela paralelização não foi grande e que os dois modelos de memória obtiveram resultados semelhantes Entretanto, a paralelização desse aplicação é limitada pela dependência de dados característica desse algoritmo. Para verificar se o sistema impôs alguma limitação à paralelização da aplicação, um algoritmo de multiplicação de matrizes foi executado no modelo de memória compartilhada. Podemos ver na Figura 4.14 que esse algoritmo permite uma paralelização mais eficiente, e que seu *speedup*, mostrado na Figura 4.13, é

próximo do linear. Dessa forma, podemos concluir que a performance foi realmente limitada pelas características da aplicação e não pelo sistema.

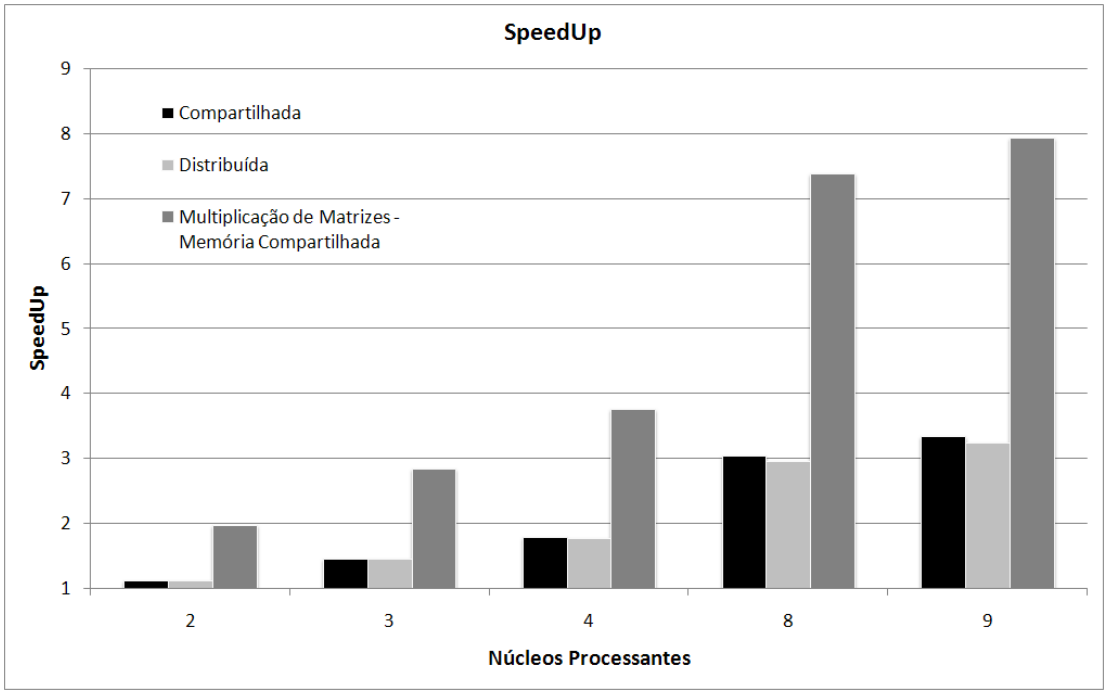

Figura 4.13 – SpeedUp

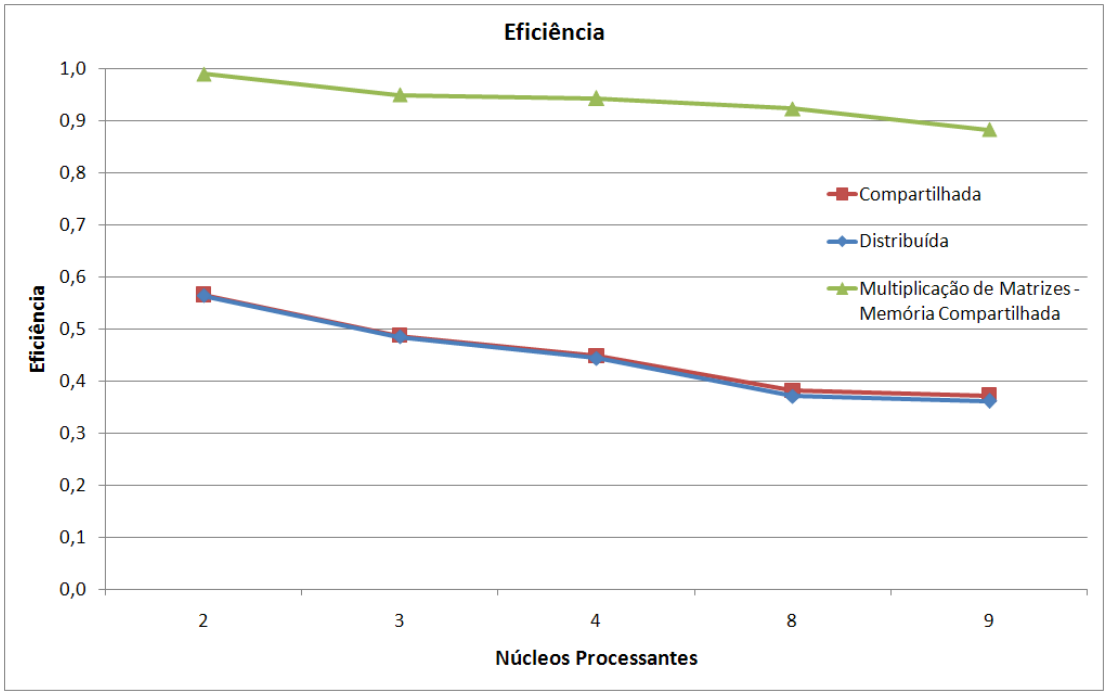

Figura 4.14 – Eficiência.

5 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve por objetivo avaliar uma tradicional e complexa aplicação da indústria de exploração de petróleo e gás natural, a simulação de reservatórios, sob a ótica do desenvolvimento de sistemas computacionais integrados em um único chip, dotados de dezenas a centenas de unidades funcionais. Contudo, esse trabalho não tinha pretensão de propor melhorias significativas no que diz respeito às técnicas de modelagem dos reservatórios de petróleo, mas sim, avaliar o uso MPSoCs e *many-cores* como futura alternativa ao uso supercomputadores e *clusters* para executar aplicações computacionalmente tão intensas quanto essas simulações.

Nessa avaliação, aspectos gerais do sistema, como modelo de memória e desempenho da NoC, foram analisados. Como a plataforma STORM já contava com um modelo de memória compartilhada, foi então desenvolvido, no decorrer desse trabalho, um modelo de memória distribuída para a mesma. Uma biblioteca de troca de mensagens seguindo o padrão MPI também foi desenvolvida para esse modelo de memória, e versões seqüenciais e paralelas da aplicação foram implementadas para os dois modelos de memória utilizados nos experimentos. Todo o trabalho de codificação, incluindo correções de erros eventuais, encontrados no modelo já implementado, representou a codificação de aproximadamente 12 mil linhas de código SystemC. Para a obtenção de resultados foram realizadas 14 simulações envolvendo 14 instâncias da plataforma STORM.

Os resultados obtidos mostraram que a carga injetada na NoC durante a execução da aplicação no modelo de memória distribuída não chega a 0,2% da carga injetada no modelo de memória compartilhada. Entretanto, mesmo sendo substancialmente menor, essa carga se comporta de maneira semelhante à carga gerada pelo modelo de memória compartilhada. Pôde ser visto também que o modelo de memória distribuída obteve resultados melhores na quantidade média de ciclos necessários para a execução de uma instrução e que, mesmo com o aumento de nós processantes no sistema, essa quantidade se manteve praticamente inalterada, enquanto que no modelo de memória distribuída houve uma degradação desse valor.

Apesar dos resultados citados anteriormente serem favoráveis ao modelo de memória distribuída, os resultados de tempo de execução mostraram que os dois modelos

de memória levaram praticamente a mesma quantidade de ciclos para executar a aplicação. Vimos que esse resultado é justificado pela quantidade maior de instruções executadas no modelo de memória distribuída, causado pelo uso da biblioteca de troca de mensagens, e pelo aumento da latência de rede quando esse modelo de memória é utilizado. Entretanto, esses resultados podem ser melhorados com a correção de deficiências da NoC e com a adição de suporte a operações de acesso direto à memória ao DAMa.

Durante o desenvolvimento desse trabalho foram detectadas duas deficiências no modelo de NoC utilizado na plataforma STORM. A primeira se refere ao tamanho dos pacotes, que possuem tamanhos fixos pré-definidos, fazendo com que as mensagens maiores enviadas pela aplicação tenham que ser divididas em vários pacotes de rede, gerando então um overhead no envio e recebimento de mensagens. A segunda deficiência é o chaveamento *store-and-forward*. Nesse chaveamento, os pacotes são totalmente armazenados nos roteadores antes de serem enviados para os roteadores seguintes. Esse comportamento resulta num aumento da latência à medida que o tamanho dos pacotes aumenta.

A arquitetura inicialmente concebida para o modelo de memória distribuída contava com o suporte a operações de acesso direto à memória, que não chegou a ser implementado por limitações de tempo desse trabalho. Esse suporte previa que o DAMa pudesse ser programado para transferir dados entre os módulos conectados a ele, liberando o processador dessa tarefa e diminuindo o tempo necessário para enviar uma mensagem.

O modelo de memória distribuída obteve resultados expressivos, mas algumas alterações na plataforma STORM são necessárias para que se possa avaliar de forma mais conclusiva esse modelo de memória. Dessa forma, podemos citar como trabalhos futuros imediatos: a substituição do modelo de NoC utilizada na plataforma STORM por um modelo que não possua as deficiências citadas anteriormente, a adição do suporte a operações de DMA ao DAMa, e a otimização do tamanho dos buffers dos roteadores da NoC para o modelo de memória distribuída.

Como trabalhos futuros mais distantes, podemos citar a implementação de alguns mecanismos de sistema operacional, como o gerenciamento de processos e o gerenciamento de memória, para a plataforma STORM, e o desenvolvimento de um módulo de entrada e saída. Esse módulo pode ser mais facilmente desenvolvido para o modelo de memória distribuída devido às características do DAMa, explicadas na seção

2.2.4. Podemos citar ainda, a avaliação do consumo de potência de módulos importantes do sistema, como a NoC, a memória e o processador. Essa avaliação pode ser feita através da inclusão de bibliotecas e o uso de ferramentas adequadas. Esses trabalhos são de fundamental importância para o desenvolvimento dos MPSoCs como sistemas de uso geral.

BIBLIOGRAFIA

AGARWAL, A.; LEVY, M. The kill rule for multicore. In: Design Automation Conference, 44, 2007, San Diego, *Proceedings*… New York: ACM Press, 2007, p. 750–753.

AMES, W. F. *Numerical Methods for Partial Differential Equations*. 3 Ed. San Diego: Academic Press, 1977. 433 p.

BEAR, J. *Dynamics of Fluids in Porous Media*. New York: Elsevier, 1972. 764 p.

BENINI, L.; DE MICHELI, G. Networks-on-chip: a new SoC paradigm. *IEEE Computer*, Los Alamitos, v. 35, n. 1, p. 70-78, Jan. 2002.

BERTOZZI, D. et al. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, Piscataway, v.16, n. 2, p.113 – 129, Fev. 2005.

BERTOZZI. S. et al. Supporting task migration in multi-processor systems-on-chip: A feasibility study. In: Design, Automation and Test in Europe, 2006. *Proceedings*... Leuven: European Design and Automation Association, 2006.

BORKAR, S. Thousand core chips - a technology perspective. In: Design Automation Conference, 44, 2007, San Diego, *Proceedings*… New York: ACM Press, 2007, p. 746–749.

CMG Ltd. *IMEX Reference Manual, version 96.00*, [S.l.: s.n.], 1995.

CORDAZZO, J. *Simulação de reservatórios de petróleo utilizando o método EbFVM e multigrid algébrico*. 2006. Tese (Doutorado em Engenharia Mecânica) – Programa de Pós-Graduação em Engenharia Mecânica. UFSC, Florianópolis. 2006.

DAKE, L. P. *Fundamentals of Reservoir Engineering*. Amsterdam: Elsevier; 1978.

ERTEKIN, T.; ABOU-KASSEM, J. H.; KING, G. R. *Basic applied reservoir simulation*. SPE Textbook Series Vol. 7. Richardson: SPE, 2001.

FANCHI, J. R.; HARPOLE, K. J.; BUJNOWSKI, S. W. *BOAST: A three-dimensional, three-phase black oil applied simulation tool*. Oklahoma: [s.n.], 1982.

GIRÃO, G.; OLIVEIRA, B. C.; REGO, R. S. L. S.; SILVA, I. S. Cache Coherency Communication Cost in a NoC-based MPSoC Platform. In: Symposium on Integrated Circuits and Systems Design, 20, 2007, Rio de Janeiro. *Proceedings*... New York : Association for Computing Machinery, 2007. v. 1. p. 288-293.

HELD, J. B. J.; KOEHL, S. *From a few cores to many: A terascale computing research overview*. Intel Technical White Paper, Setembro 2007. Disponível em: <http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf>

IEEE Standard Society. *IEEE Open SystemC Language Reference Manual*. [S.l., s.n.], 2006. 441 p.

JERRAYA, A. A.; WOLF, W. The what, why, and how of MPSoCs. In:_____ (Ed.). *Multiprocessor System-on-Chip*. San Francisco: Morgan Kauffman, 2005. p. 1-18.

LOGHI. M. et al. Analyzing on-chip communication in a MPSoC environment. In: Design, Automation and Test in Europe, 2004. *Proceedings*... Washington: IEEE Computer Society, 2004.

MATTAX, C. C.; DALTON, R. L. *Reservoir Simulation*. SPE Monograph Series Vol. 13. Richardson: SP, 1990.

MPI Forum. *MPI: A Message-Passing Interface Standard, Version 2.1*. Junho, 2008. Disponível em: < http://www.mpi-forum.org/docs/mpi21-report.pdf>

OLIVEIRA, B. C. O. *Manutenção da Consistência de Dados em uma Plataforma MP-SoC Baseada em NoC: Projeto do Diretório*. 2006. Trabalho de Conclusão de Curso. (Graduação em Ciência da Computação) – Departamento de Informática e Matemática Aplicada, UFRN. Natal, 2006.

REGO, R. S. L. S.; SILVA, I. S.; AZEVEDO FILHO, A. When reconfigurable architecture meets network-on-chip. In: Symposium on Integrated Circuits and Systems Design, 17, 2004, *Proceedings*... New York: ACM Press, 2004, p. 216 - 221.

REGO, R. S. L. S. *Projeto e implementação de uma plataforma MPSoC usando SystemC*. 2006. Dissertação (Mestrado em Sistemas e Computação) – Programa de Pós-Graduação em Sistemas e Computação, UFRN, Natal, 2006.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, Los Alamitos, v. 18, n. 6, p. 23-33, Nov./Dec. 2001.

SCHLUMBERGER Ltd. *ECLIPSE Reservoir Engineering Software*. Disponível em: <http://www.slb.com>. Acesso em 03/04/2009.

SILVA, F. A. et al. Parallelizing Black Oil Reservoir Simulation Systems for SMP Machines. In: Annual Symposium on Simulation, 36, 2003, *Proceedings*... Washington: IEEE Computer Society, 2003, p. 224 - 230.

SOARES, A. A. M. *Simulação de reservatórios de petróleo em arquiteturas paralelas com memória distribuída*. 2002. Dissertação (Mestrado em Ciências e Engenharia Civil) – Programa de Pós-Graduação em Engenharia Civil. UFPE, Recife, 2002.

SOARES, R.; PEREIRA, A.; SILVA, I. S. X4CP32: A new parallel/reconfigurable general-purpose processor. In: Symposium on Computer Architecture and High Performance, 15, 2003, *Proceedings...* Washington: IEEE Computer Society, 2003, p. 260–268.

YOUNG, D. Iterative Methods For Solving Partial Difference Equations of Elliptic Type. *Transactions of the american mathematical society*, [s.l.], v. 76, n. 1, p. 92 – 111, 1954.

SPARC International, Inc. *The SPARC architecture manual, version 8*. [S.l.]: Prentice Hall, 1992.

ZEFERINO, C. A. et al. A study on communication issues for systems-on-chip. In: Symposium on Integrated Circuits and Systems Design, 15, 2002, *Proceedings...* Washington: IEEE Computer Society, 2002, p. 121–126.

ZWILLINGER, D. *Handbook of Differential Equations.* 3 Ed, Boston: Academic Press, 1998. 801 p.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo