UM CLUSTER HÍBRIDO COM MÓDULOS DE CO-PROCESSAMENTO EM HARDWARE (FPGAS) PARA PROCESSAMENTO DE ALTO DESEMPENHO

por

Severino José de Barros Júnior
Dissertação de Mestrado
Orientador: Manoel Eusébio de Lima

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CIN - CENTRO DE INFORMÁTICA
PÓS GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2014

SEVERINO JOSÉ DE BARROS JÚNIOR

UM CLUSTER HÍBRIDO COM MÓDULOS DE CO-PROCESSAMENTO EM HARDWARE (FPGAS) PARA PROCESSAMENTO DE ALTO DESEMPENHO

Dissertação de Mestrado apresentada à Universidade Federal de Pernambuco, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do Título de Mestre.

Orientador: Manoel Eusebio de Lima

RECIFE

2014

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

B277c Barros Júnior, Severino José de
Um cluster híbrido com módulos de co – processamento em hardware (FPGAS) para processamento de alto desempenho. / Severino José de Barros Júnior. – Recife: O Autor, 2014.
116 f.: il., fig., tab.

Orientador: Manoel Eusébio de Lima.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2014.
Inclui referências.

1. Engenharia da computação. 2. Arquitetura de computador. 3. FPGA. 4. Computação de alto desempenho. I. Lima, Manoel Eusébio de (orientador). II. Título.

621.39 CDD (23. ed.) UFPE- MEI 2014-183

Severino José de Barros Júnior

UM CLUSTER HÍBRIDO, COM MÓDULOS DE CO-PROCESSAMENTO EM HARDWARE (FPGAS), PARA PROCESSAMENTO DE ALTO DESEMPENHO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 10/09/2014

BANCA EXAMINADORA

_____

Prof. Dr. Carlos André Guimarães Ferraz
Centro de Informática / UFPE

_____

Prof. Dr. Remy Eskinazi Sant'Anna
Escola Politécnica de Pernambuco / UPE

_____

Prof. Dr. Manoel Eusebio de Lima(Orientador)
Centro de Informática / UFPE

Agradecimentos

Agradeço primeiramente à minha família por todo apoio, carinho e dedicação dada a mim durante toda minha vida. É a base a qual tenho sorte e orgulho de fazer parte. Gostaria também de agradecer ao Professor Manoel Eusebio, cujo apoio, confiança, incentivo e compreensão são de caráter ímpar, além de proporcionar excelentes oportunidades de conhecimento com as quais permitiu meu crescimento acadêmico e profissional. Também agradeço aos professores das disciplinas que cumpri durante esta pós-graduação, pela contribuição dada ao aprendizado e experiência úteis para realização deste trabalho. Grato aos companheiros de Projeto, como João Paulo, Bruno Pessôa, Rodrigo Camarotti, Augusto Benvenutto, Victor Medeiros, Marcelo Marinho, Joelma França, Antonius Pyetro, Luis Figueroa e todos os outros que compartilharam comigo as rotinas e bons momentos de convivência e amizade no laboratório HPCIn. Sou grato também ao projeto HPCIn financiado pelo FINEP/Petrobrás(CENPES) que disponibilizou toda a infra-estrutura técnica para a realização deste projeto. Por último, mas não menos importante, agradeço à minha então namorada Adha Santos, pois apesar de termos dividido apenas os momentos finais dessa jornada, soube ter uma enorme compreensão, dando todo carinho e amor que ela podia, nos momentos bons e nos mais difíceis.

Resumo

Organizações que lidam com sistemas computacionais buscam cada vez mais melhorar o desempenho de suas aplicações. Essas aplicações possuem como principal característica o processamento massivo de dados. A solução utilizada para execução desses problemas é baseada, em geral, em arquiteturas de processadores de uso geral, cuja principal característica é sua estrutura de hardware baseada no Paradigma de *Von Neumann*. Esse paradigma possui uma deficiência conhecida como "Gargalo de *Von Neumann*", onde instruções que poderiam ser executadas de forma simultânea, devido à sua independência de dados, acabam sendo processadas sequencialmente, prejudicando o potencial desempenho dessa classe de aplicações. Para aumentar o processamento paralelo dos sistemas, as Organizações costumam adotar uma estrutura baseada na associação de vários *PCs*, conectados a uma rede de alta velocidade e trabalham em conjunto para resolver um grande problema. A essa associação é atribuída o nome de cluster, a qual cada integrante *PC*, chamado de nó, realiza uma parte da computação de um grande problema de forma simultânea, proporcionando a ideia de um paralelismo explícito da aplicação como um todo. Mesmo com um aumento significativo de elementos de processamento independentes, este crescimento é insuficiente para atender à enorme quantidade de demanda de computação de dados em aplicações complexas. Ela exige uma divisão de grupos de instruções independentes, distribuídos entre os nós. Esta estratégia dá a idéia de paralelismo e assim um melhor desempenho. No entanto, o desempenho em cada nó permanece degradado, devido ao estrangulamento seqüencial presente nós processadores. A fim de aumentar o paralelismo das operações em cada nó, soluções híbridas, compostas por *CPUs* convencionais e coprocessadores foram adotadas. Um desses coprocessadores é o *FPGA* (*Field Programmable Gate Array*), que geralmente é conectado ao *PC* através do barramento *PCIe*. O projeto descrito na dissertação propõe uma metodologia de desenvolvimento para este aglomerado híbrido, de modo a aumentar o desempenho de aplicações científicas que requerem uma grande quantidade de processamento de dados. A metodologia é apresentada e dois exemplos são discutidos em detalhes.

Palavras-chave: HPC. Cluster Híbrido. FPGA. OpenMP. MPI

Abstract

Actually Companies and organizations that deal with big computing systems are constantly seeking for performance improvement on their applications. These applications have, as main characteristic, the massive data processing. The main approach applied on these problems is based, generally, on general-purpose processors, with architectures based on *Von Neumann* paradigm. This paradigm has a drawback known as "*Von Neumann bottleneck*," where instructions, which could be executed simultaneously due to their independence of data, end up being processed sequentially, decreasing in this class of applications. In order to increase the parallel processing capability, many organizations adopt an approach based on association of multiple PCs, connected through a high-speed network, working together to solve big problems. This approach is called *cluster* and each *PC* member is called a node. These nodes compute only parts of the problem simultaneously, providing the idea of explicit parallelism of the application. Even with a significant increase of independent processing elements, this growth is insufficient to meet the huge amount of data computation demand in complex applications. It requires a split of independent instructions groups, distributed among the nodes. This strategy gives the idea of parallelism and so better performance. However, the performance on each node remains degraded, due to the sequential bottleneck present on processors. In order to increase the parallelism of operations on each node, hybrid solutions, composed by conventional CPUs and co-processors have been adopted. One of these co-processors is the FPGA (Field Programmable Gate Array), which is generally connected to the PC through the PCIe bus. The project described in this dissertation proposes a development methodology for this hybrid cluster, in order to increase the performance of scientific applications that require a large amount of data processing. The methodology is presented and two examples are discussed into details.

Kerwords: HPC. Hybrid Cluster. FPGA. OpenMP. MPI

Índice de Figuras

Lista de Tabelas

Sumário

Lista de Abreviaturas e Siglas

API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CDT	C/C++ Development Tools
CENPES	Centro de Pesquisas e Desenvolvimento da Petrobras
CLB	Configurable Logic Block
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECC	Error Correction Code
FPGA	Field Programmable Gate Arrays
GFLOPS	Giga Floating-Point Operations Per Second
GPU	Graphic Processing Unit
HAU	Hardrware Abstraction Unit
HDL	Hardware Description Language
HPC	High Performance Computing
HPCIn	High Performance Computing CIn-UFPE
IDE	Integrated Development Environment
LUT	Look Up Table
MIPS	Milhões de Instruções por segundo
MPI	Message Passing Interface
NNUS	Nonuniform Node Uniform System
OpenCL	Open Computing Language
OpenMP	Open MultiProcessing
OpenSSH	Open Source Secure SHell
OpenSSL	Open Source toolkit for SSL/TLS
PCIe	Peripheral Component Interconnect express
PE	Process Element

PETSc	Portable, Extensible Toolkit for Scientific Computation
RTM	Reverse Time Migration
SAS	Serial Attached SCSI
SATA	Serial ATA
SMP	Symmetric Multiprocessing
TDP	Thermal Design Power
Top500	Lista de computadores com maior poder computacional do mundo
UNNS	Uniform Node Nonuniform System
VHDL	VHSIC Hardware Description Language

1. Introdução

Organizações que lidam com sistemas computacionais que processam quantidades massivas de dados, sejam governamentais ou privadas, buscam cada vez mais por um melhor desempenho de suas aplicações. Esse desempenho pode representar menor tempo de processamento, e consequentemente resultados rápidos, que muitas vezes podem ser críticos, em situações de decisão, tais como, em sistemas financeiros ou de segurança (identificação de contaminação em ambientes, identificações de pessoas suspeitas, etc.), ou ainda desempenho no consumo de potência, etc.

Este conjunto de aplicações, de uma forma geral, abrange uma gama complexa de atividades em diversas áreas do conhecimento científico, tais como: mecânica, geofísica, aviônica, setor financeiro, biologia, climatologia, entre outras. Um exemplo típico deste tipo de abordagem é o processamento de dados captados por satélites espaciais, cuja ordem de grandeza dessa massa de dados chega a mais de 300 exa-byte por dia (INTA, 2012). Na área financeira, temos processos com movimentações em bilhões de dólares, e pequenos lucros obtidos com a compra e venda de um grandioso número de ações, que precisam ser negociados em uma fração de segundos, gerando lucros de dezenas de milhões por ano para seus operadores (SADOGHI, 2010). Em biologia, sequências de *DNA* e de proteínas são modeladas através de algoritmos de bioinformática abrangendo estruturas proteicas extremamente complexas e custosas em processamento (ULIANA, 2013). Bibliotecas para cálculos de equações lineares, como o *PETSc (Portable, Extensible Toolkit for Scientific Computation)* (PETSC, 2014), realizam operações entre vetores e matrizes com dezenas de milhares de elementos numéricos. Algoritmos de simulação geofísica também merecem destaque, pois seus resultados de estimativa de um terreno, provenientes de um grande conjunto de informações extraídas de áreas de difícil acesso, permitem que empresas de minérios e hidrocarbonetos possam extrair esses substratos valiosos de forma eficiente (ROCHA, 2010).

Essas aplicações possuem como característica determinante um processamento massivo de dados, com um grande potencial de paralelismo em suas operações aritméticas. Isto possibilita tratar uma enorme quantidade deste processamento através de instruções similares, de forma paralela, em várias unidades de processamento, por independência de dados. Ambientes como estes permitem, portanto, soluções escaláveis na adoção de processamento paralelo, seja com relação a arquitetura interna dos processadores, seja com

relação a estrutura externa de máquinas conectadas através de uma conexão rápida (ROCHA, 2010; ESPENSHADE, 2009).

Neste contexto, este trabalho propõe uma metodologia de desenvolvimento de sistemas para aplicações que requeiram um processamento massivo de dados, voltados para uma arquitetura de *cluster* de computadores, interligados através de uma rede de alta velocidade. Nestes computadores, nós da rede, placas aceleradoras, baseadas em dispositivos lógicos programáveis, denominados de *Field Programmable Gate Arrays (FPGAs)*, são usadas para acelerarem o processamento dos núcleos aritméticos dos algoritmos e assim, aumentar substancialmente o desempenho computacional das aplicações. Embora essa metodologia possa ser aplicada de forma genérica, este trabalho demonstra seu desempenho em aplicações voltadas para a resolução de multiplicação de matrizes densas, características de aplicações como a biblioteca *BLAS (Basic Linear Algebra Subprograms)* (PETSC, 2014), e o processamento da modelagem sísmica *RTM 3D (Reverse Time Migration)*. Este último foi o modelo de um projeto desenvolvido pelo Grupo HPCIn do CIn/UFPE, em cooperação com o CENPES/Petrobras. Em ambos os casos foi utilizado aritmética ponto flutuante com precisão simples como requisito das aplicações.

.

1.1 **Motivação**

Várias aplicações científicas utilizam algoritmos da biblioteca *BLAS*, particularmente multiplicação de matrizes, com aritmética de ponto flutuante, o que dependendo do volume de dados a serem processados, ou seja, do tamanho das matrizes, pode-se requerer grandes sistemas de armazenamento e processamento de dados. Na mesma linha de problema, os algoritmos da exploração sísmica são, em geral, computacionalmente muito custosos. A busca por petróleo em regiões com estruturas geológicas mais complexas, como o pré-sal, passaram a exigir a necessidade do emprego de algoritmos sísmicos ainda mais custosos, como o *RTM* (ROCHA, 2010) por exemplo.

Devido ao alto custo computacional destes algoritmos, é importante que várias alternativas de plataformas computacionais existentes na atualidade sejam exploradas. Assim, além do uso dos processadores de propósito geral, em especial os *multicores*, utilizados como hospedeiros (*host*) na maioria dos servidores comerciais, novas arquiteturas passaram a serem avaliadas, principalmente como unidades de coprocessamento aritméticos, como as *GPUs (Graphic Processing Unit)*, os processadores *Cell* e os *FPGAs*.

Neste contexto, a motivação deste trabalho pode ser sintetizado pela necessidade de se desenvolver uma metodologia capaz de desenvolver aplicações que aproveitam bem os recursos de uma plataforma computacional escalável, paralela e de alto desempenho.

1.2 Objetivos desta dissertação

O objetivo deste trabalho é propor uma metodologia de desenvolvimento para um *cluster* híbrido escalável, com nós constituídos por uma *CPU* de propósito geral (*host*) e uma placa aceleradora composta de um *FPGA* e módulos de memória. Em cada nó, a placa aceleradora é interligada ao *host* através de um barramento *PCIe*.

A arquitetura de cada nó permite que os *FPGAs*, aqui representando coprocessadores aritméticos, possam contribuir de sobremaneira para a aceleração de aplicações científicas, na execução dos núcleos aritméticos das aplicações. Todas as aplicações deverão ser adequadamente particionadas e distribuídas nos nós do cluster. Como modelo híbrido, as *CPUs* convencionais, em cada nó, deverão trabalhar de forma síncrona e paralela com os *FPGAs*, desenvolvendo ações complementares ao processamento, tais como: transferência de dados entre o *host* e a placa aceleradora, montagem de tabelas e matrizes resultantes do processamento, além de todo o tratamento de comunicação entre nós do cluster.

Portanto, espera-se que a metodologia seja capaz de aplicar as adaptações necessárias para a aceleração de aplicações massivas, permitindo que as mesmas possam aproveitar ao máximo os recursos disponíveis no cluster híbrido, isto é, com a maior quantidade de elementos de processamento sendo executados simultaneamente. Para tanto, é preciso definir um conjunto de etapas para analisar a natureza da aplicação, os trechos de maior custo computacional, o fluxo de dados e suas dependências. Com a coleta dessas informações, é possível definir o comportamento dos elementos de computação paralela e a distribuição dos dados entre os nós, proporcionando assim um aumento de desempenho, preservando as funcionalidades gerais da aplicação. Esta metodologia garante a transparência de comportamento da aplicação para o usuário.

Para validar a metodologia, foram escolhidas duas aplicações como estudos de caso, que envolvem a comparação de desempenho entre a execução da aplicação em cluster apenas utilizando apenas *CPUs* como nós processadores, e a execução conjunta com o auxílio de coprocessadores baseados em *FPGAs*.

1.3 Estrutura da Dissertação

A estrutura deste trabalho está organizada na forma descrita a seguir. O Capítulo 2 aborda os conceitos utilizados para o desenvolvimento do projeto proposto nesta dissertação. No Capítulo 3 são apresentados e discutidos trabalhos relacionados. O capítulo 4 dá ênfase ao desenvolvimento do cluster proposto e da metodologia de processamento. O Capítulo 5 apresenta exemplos com a metodologia proposta, bem como apresenta a discussão dos resultados das implementações. Por fim, o Capitulo 6 apresenta as conclusões da dissertação, seus limites e sugere os trabalhos futuros que podem vir a enriquecer este projeto.

2. Fundamentação Teórica

Este capítulo introduz conceitos relevantes para o bom entendimento da dissertação, tais como: a ideia de um cluster e arquitetura de conexão entre seus nós, protocolo de comunicação entre seus nós; características da arquitetura interna de um dispositivo FPGA, que o torna indicada como unidade coprocessadora em sistemas como este; comparação de desempenho entre uma arquitetura *FPGA* e uma *GPU*; e uma análise do melhor dos dois mundos, *CPU* de propósito geral e *FPGAs*, quando se tece considerações sobre modelos híbridos de processamento.

2.1 Clusters

Com o objetivo de melhorar a eficiência das aplicações científicas que requeriam um processamento massivo de dados, organizações buscaram, ao longo do tempo, alternativas que disponibilizassem um incremento da capacidade de execução de tarefas em paralelo.

Inicialmente foram feitos esforços para a construção de modelos os quais uma única máquina possuísse uma arquitetura com alto poder de computação massiva. Um desses modelos comercialmente adotados foi o processador vetorial (CRAY, 2014), cuja principal característica era a realização de uma grande quantidade de instruções utilizando o mesmo dado de entrada. Essas instruções representavam operações aplicadas a um vetor de dados, muito comum em aplicações envolvendo processamento de imagens e gráficos. Com isso, os elementos de processamento geravam resultados de forma independente, permitindo assim um aumento da profundidade do pipeline individual, consequentemente aumentando a frequência de operação e o desempenho geral do sistema. Supercomputadores como o *NEC SX-6* e *Cray X1* continham essa característica estrutural, e estavam entre as máquinas que lideravam a lista Top500 em suas épocas (DONGARRA, 2005). O grande problema desses supercomputadores era o alto custo de aquisição e manutenção, pois sua arquitetura requeria um sistema operacional, componentes proprietários de hardware, ferramentas de compilação e modelos de programação próprios, dificultando a sua utilização em larga escala.

O declínio da utilização dos processadores vetoriais começou a partir do desenvolvimento de novas técnicas de computação paralela. Com o objetivo de aumentar o poder de paralelismo a partir de produtos já consolidados no mercado, surgiram associações

de computadores com processadores convencionais, os quais eram conectados em rede, de forma a processarem em conjunto e de forma paralela grandes aplicações. A essa associação foi dado o nome de *cluster*, e cada computador integrante desse *cluster* foi denominado de nó. Estes nós, por sua vez, não necessariamente precisavam ter uma configuração de *hardware* igual aos outros integrantes do conjunto (GORINO, 2006; HAWICK, 1999).

Uma das principais vantagens dos *clusters*, em comparação com os processadores vetoriais, é o baixo custo de implementação e manutenção, pois todos os componentes, como *PCs* e estruturas de rede, são altamente disponíveis no mercado. Outra vantagem relevante é a escalabilidade dos elementos de processamento, os quais podem alcançar desde dezenas a milhares de participantes processando de forma coordenada e simultânea (DONGARRA, 2005).

Existem duas categorias de clusters, considerando a capacidade computacional de seus nós. Uma delas é a *beowulf* (HAWICK, 1999), a qual se baseia na construção de um cluster a partir de nós formados por *PCs* de baixa capacidade, com vantagem de obter-se um alto poder de processamento coletivo e menores custos de aquisição dos componentes. Uma outra categoria abrange os workstation *clusters* (DONGARRA, 2005), onde cada nó é um *worktation*, e portanto possui um alto desempenho individual comparado com os *PCs* comuns.

Com relação a homogeneidade dos nós, podemos classificar os *clusters* como homogêneos (com nós possuindo os mesmos componentes) e heterogêneos (nós de diferentes componentes) (TSOI, 2010).

Com relação à finalidade dos *clusters*, podemos defini-los em três classificações: *clusters* de alta disponibilidade, *clusters* de balanceamento de carga, e *clusters* de alto desempenho (GORINO, 2006).

Os *clusters* de alta disponibilidade possuem a característica, de acordo com a própria definição, de manterem seus serviços em funcionamento de maneira contínua, ou seja, pelo maior tempo possível. Para garantir essa continuidade, a implantação de seus nós devem obedecer a garantias de redundância, tanto de *hardware* quanto de *software*, com o objetivo de, em caso de falha, o sistema poder substituir dinamicamente o componente ou programa por outro reserva, onde este deve estar sempre pronto para execução imediata.

A característica principal dos *clusters* para balanceamento de carga são a de prover o mesmo serviço para vários usuários. Neste caso os nós são projetados para proverem os mesmos serviços, e também são monitorados através de programas especiais chamados *daemons*, que possuem a função de monitorar a taxa de utilização dos recursos de cada nó, e

caso haja sobrecarga, redirecionar os serviços para outros nós menos carregados. Caso o sistema como um todo esteja completamente sofrendo sobrecargas, é possível aumentar a capacidade computacional, adicionando novos nós (GORINO, 2006).

Os *clusters* de alto desempenho são os que possuem mais relevância entre as organizações para processamento científico, pois essas associações de máquinas possuem como principal característica um grande poder de processamento em paralelo para classes especiais de aplicações. O principio de funcionamento é baseado na execução de uma grande aplicação complexa. A aplicação deve ser particionada em tarefas e dados, os quais são distribuídos entre os nós do *cluster*. Cada tarefa processa seus dados de forma independente, em cada nó, aumentando assim o paralelismo da execução, e portanto, o desempenho da aplicação como um todo. Esse tipo de *cluster* foi o principal responsável pela diminuição do uso dos processadores vetoriais, pois dependendo do tipo e da quantidade de componentes utilizados para sua constituição, podem ser mais poderosos computacionalmente e com melhor relação custo-desempenho, então aumentando sua presença, ao longo do tempo, na lista Top500 (DONGARRA, 2005, GORINO, 2006).

As aplicações capazes de aproveitar o máximo de recursos de um *cluster* de alto desempenho precisam sofrer algumas adaptações, para que suas tarefas massivas sejam divididas entre os nós. A essa aplicação é dada o nome de aplicação distribuída, a qual seus dados e instruções são divididos entre os membros do sistema, para poderem ser processados simultaneamente, e assim prover a ideia de um paralelismo explícito da execução da aplicação. Geralmente, em uma arquitetura como esta, é atribuído a um dos nós o controle do fluxo de execução principal do programa, o qual é responsável por dividir e transmitir cada parte dos dados e instruções para os outros membros, receber os resultados processados e reunir cada parte para formar a saída final do sistema (ESPENSHADE, 2009, ARAUJO, 2010).

O paralelismo dos *clusters* de alto desempenho geralmente é analisado sob dois níveis. O primeiro deles é o nível global, onde é relevante como o *cluster* realiza a distribuição das tarefas entre seus participantes (DONGARRA, 2005). A configuração desse nível de paralelismo é feita em nível de processos, com transferência de informações por meio de passagem de mensagem, onde há a opção de envio direto ou de forma coletiva dos dados de um nó para os outros participantes do sistema. O *MPI* (*Message Passing Interface*) (MPI, 2014) é um protocolo de comunicação amplamente usado para este propósito.

Com relação ao segundo nível de paralelismo, chamado por *Sterling* de nível constelação (DONGARRA, 2005), refere-se ao que é apresentado por cada componente do

cluster, ou seja, o paralelismo intrínseco de cada nó. Esse nível contribui sensivelmente com o desempenho em execução de um *cluster*, segundo a lei de Amidahl (SUN, 2010). Os *PCs* que possuem a capacidade de execução paralela, geralmente possuem *CPUs* do tipo *SMP* (*Symmetric MultiProcessor*) (NAKAJIMA, 2005), cuja principal característica de seu paralelismo é o uso compartilhado da memória, a nível de *threads*. O modelo de programação utilizando *OpenMP* (OPENMP, 2014) permite uma configuração desse nível, facultando ao usuário definir como os dados compartilhados interagirão entre essas *threads*.

É possível também classificarmos os *clusters* com relação a algumas propriedades, a fim de destacar os pontos fortes e fracos dos sistemas, em suas diversas maneiras de construção (DONGARRA, 2005). Uma dessas propriedades é o desempenho, o qual é mensurado em geral, pela taxa de dados processados por intervalo de tempo. As grandezas mais utilizadas para representar essa medida são os MIPS (Milhões de Instruções executadas Por Segundo) e os GFLPOS (bilhões de operações realizadas em ponto flutuante por segundo). A granularidade do paralelismo também é uma propriedade que merece destaque, pois com ela é possível determinar a quantidade de operações que são capazes de serem executadas simultaneamente no *cluster*. O tipo de paralelismo utilizado, tanto do *software* quanto do *hardware*, influencia diretamente nessa quantidade, podendo diferenciar um sistema dos outros de acordo com a quantidade de paralelismo que cada um oferece.

Outro aspecto de comparação está relacionado com o controle do sistema distribuído, isto é, como é apresentada a arquitetura de controle das operações massivas. Esse controle pode ser feito através de componentes de *hardware*, *software* ou ambos, podendo determinar a quantidade de *overhead* necessária para as execuções simultâneas, logo influenciando no desempenho do *cluster*.

O controle de latência também é importante para verificar o quanto um sistema é eficiente para diminuir atrasos na comunicação entre os nós. Neste caso é importante determinar a largura de banda e escalabilidade entre os sistemas. Recursos como vetores de *pipelining*, sistemas *multithread*, controle local de congestionamento da rede, controle de *cache* e direcionamento de mensagens, fazem parte da estrutura do controle de latência.

Outra característica diferencial são os sistemas de nomeação das tarefas e recursos de *hardware*. A maneira como são atribuídos identificadores, tanto aos elementos de *hardware*, tais como portas de entrada e saída, canais de comunicação ou até mesmo os próprios nós, como os elementos de *software*, de acordo com a identificação de *threads* e processos, constituem um diferencial para caracterização de um sistema. E por último, a dependência no fornecimento dos componentes, ou seja, se a construção de um *cluster* depende de materiais

com alta ou baixa disponibilidade no mercado, podem determinar o quanto um *cluster* é mais fácil de ser implantado do que os outros. Essa é uma das principais características que demonstram o quanto um *cluster* pode ser vantajoso em relação a sistemas que utilizam arquiteturas exclusivas e proprietárias, como por exemplo, processadores vetoriais, utilizados na máquinas *Cray X1* (DONGARRA, 2005).

Como podemos observar, o uso de *PCs* para construção de *clusters*, contribuiu para um aumento relevante no paralelismo das aplicações científicas, principalmente de forma global, ou seja, com um maior número de nós capazes de processar simultaneamente alguns trechos desses programas. No entanto, muitas aplicações possuem um potencial de paralelismo muito maior do que o aumento dos participantes num processamento coordenado. As partes de um grande programa de processamento científico, recebidas por cada nó, ainda podem possuir trechos massivos em processamento aritmético, em geral grandes *loops*, que acabam sendo processados localmente de maneira sequencial (BACKUS, 1978) pelos seus processadores de uso geral, que obedecem ao paradigma clássico de *Von Neumann*, prejudicando assim o potencial de desempenho dessa classe de programas. É preciso, portanto, recorrer a arquiteturas com uma alta granularidade de paralelismo para os nós, as quais permitam um aumento sensível no processamento simultâneo local e assim um melhor desempenho computacional por nó do *cluster* (ARAUJO, 2010).

2.2 Arquiteturas de Processamento Massivo

Com a evolução tecnológica, tanto em técnicas de integração nas pastilhas de silício, quanto em novas estruturas de computação, começaram a surgir novos dispositivos de processamento, cujas arquiteturas diferem do paradigma de *Von Neumann*, fornecendo um alto poder de execução paralela. Dois exemplos bastante utilizados e que se enquadram nessas características são a *GPU* (*Graphics Processing Unit*) e o *FPGA* (*Field Programable Gate Array*). Cada um desses dispositivos possuem características que podem ser determinantes na escolha de qual algoritmo executar em cada arquitetura (INTA, 2012; ROCHA, 2010).

2.2.1 *GPU*

A *GPU* surgiu inicialmente para uso em processamento gráfico, com o objetivo de atender à demanda das aplicações 3-D em computadores pessoais e servidores. Com o tempo, percebeu-se que seu poder de processamento, capaz de alcançar a ordem de centenas de Gflops, poderia ser usado para acelerar outros tipos de aplicações. Esse uso foi impulsionado com o desenvolvimento de ferramentas de desenvolvimento específicas para esse dispositivo, tais como *APIs* bem definidas e *IDEs* que facilitaram a programação para o usuário, aliado à fabricação de modelos com cada vez mais capacidade de processamento e velocidade de acesso aos dados (OWENS, 2008).

A arquitetura da *GPU* é composta por uma grande quantidade de elementos de processamento, cujo número supera bastante o total de *cores* de um processador *multicore*, chegando a cerca de milhares de núcleos independentes, em modelos como o *Nvidia Tesla* (TESLA, 2014). Na arquitetura da *Nvidia* há também um processador central (árbitro), o qual possui a tarefa de gerenciar o processamento dos demais elementos de processamento da *GPU*. A esses núcleos, estão ligados pequenos módulos de memória com alta taxa de transferência, provendo uma alta capacidade de acesso simultâneo dos dados.

O princípio de paralelismo de uma *GPU* é baseado em *threads*. Cada elemento possui a capacidade de operar com números em ponto flutuante de precisão simples ou dupla, e também contam com níveis de *cache* L1 e L2. A comunicação com o *host* é geralmente feita através de barramentos tipo *PCIe*, cujos *drivers* de reconhecimento são desenvolvidos pelo fabricante da placa.

A organização dos elementos de processamento e de memória é baseada no agrupamento de núcleos, cujo acesso aos níveis mais baixos de memória é compartilhado entre os membros pertencentes a esse grupo, como pode ser visto através da Figura 1.

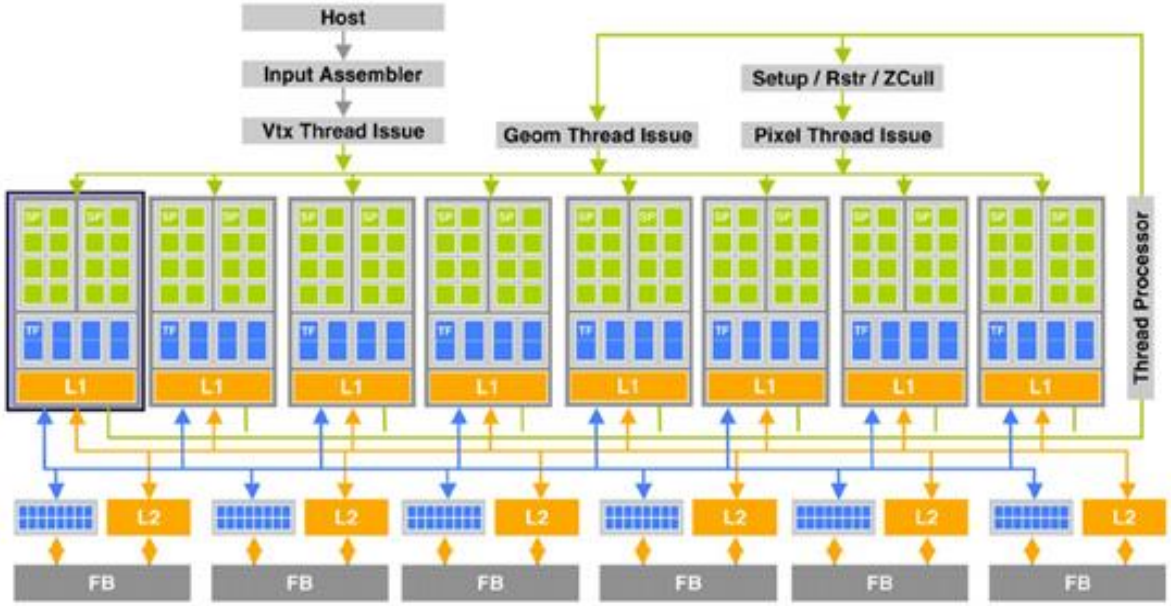

Figura 1: Visão geral da arquitetura de uma GPU (OWENS, 2008).

Como podemos observar na Figura 1, o *host* repassa o conjunto de instruções massivas para um módulo analisador inicial, cuja função é de distribuir as *threads* entre os grupos, de acordo com as configurações do usuário. Cada grupo de núcleos possui uma *cache* L1, a qual permite que *threads* pertencentes ao mesmo grupo compartilhem dados entre si. Os níveis de *cache* L2 permitem o compartilhamento de dados entre os outros grupos. O controle do fluxo geral de execução dos grupos é atribuído ao *Thread Processor*, o qual é responsável por manter a coerência da execução geral do algoritmo de acordo com as configurações passadas em nível de *software* (CUDA, 2014).

A *GPU* possui vantagens bastante atrativas para desenvolvimento de algoritmos massivos, tais como: o fornecimento, por parte de seus fabricantes, de ferramentas de compilação e programação padronizadas; uma arquitetura de maior granularidade de paralelismo em relação a processadores convencionais; além de uma boa eficiência em processamento em ponto flutuante (INTA, 2012). Isto tudo possibilitou uma grande adoção pelos desenvolvedores em *HPC* (*High Performance Computing*), deste tipo de arquitetura, resultando em numerosos trabalhos acadêmicos utilizando essas placas e em produtos comerciais. Em compensação, o potencial paralelismo de certas aplicações científicas podem conter tarefas independentes que ultrapassam a capacidade de processamento simultâneo desses dispositivos, degradando o desempenho das mesmas.

De fato, seus elementos de processamento seguem o mesmo paradigma "*Von Neumann*", o qual pode degradar o desempenho individual de cada trecho do programa. Há também um gargalo de acesso aos dados da *CPU*, pois apesar das memórias internas possuírem altas taxas de transmissão, estas são pequenas, além do barramento *PCIe* não acompanhar essa velocidade, gerando atrasos no processamento.

Um outro aspecto importante é que nos últimos anos, o consumo energético de placas aceleradoras com *GPUs* têm ganho grande relevância, desde que seus elementos processam dados a altas frequências, e consequentemente consomem mais energia, tornando-se um fator crucial para projetos com restrições de consumo de potência.

Também, assim como as *CPUs* convencionais, as *GPUs* possuem arquiteturas fixas para todos os tipos de aplicações.

2.2.2 *FPGA*

Os *FPGAs* são dispositivos que possuem uma enorme granularidade de paralelismo na sua estrutura, maior do que aquelas existentes em *CPUs* de propósito geral e *GPUs*. Seus elementos lógicos não seguem o paradigma de *Von Neumann*, embora alguns modelos desses dispositivos apresentem processadores embarcados como *hardcores* em sua estrutura, ou *softcores*, dependendo da demanda. Em sua área útil, são disponibilizados centenas de milhares de elementos básicos, os quais podem ser configurados e interligados para formar uma arquitetura computacional customizada, proporcionando assim uma alta flexibilidade de configuração do dispositivo (INTA, 2012; SADOGHI, 2010; ROCHA, 2010).

A menor unidade lógica do *FPGA* é o *CLB (Configurable Logic Block)*, o qual pode ser configurado para realizar operações lógicas simples ou fazer parte de blocos maiores de processamento. Os responsáveis internos pelas operações lógicas, são os *LUTs (Look up Tables)* e multiplexadores, onde ambos trabalham em conjunto.

Existe de fato uma ampla gama de famílias de *FPGAs* de vários fabricantes (ALTERA, 2014; XILINX, 2014), com milhões de componentes lógicos, blocos *DSP (Digital Signal Processor)*, *CPUs* embarcadas, *transceivers* para comunicação de alta velocidade, banco de memórias *RAM*, *FIFOs*, interface para barramentos *PCIe*, interface para bancos de memória DDR3, etc. Associado a este conjunto de blocos lógicos existe uma grande quantidade de matrizes de chaveamento *(switch matrix)*, e uma rede de conexões para roteamento, ambas configuráveis, que interligam os blocos lógicos, de acordo com a customização requerida no projeto.

Um *FPGA* possui também uma grande quantidade de blocos de *I/O*, associado às centenas de pinos normalmente disponíveis nestes dispositivos. Estes pinos são configuráveis e servem para interfacear a lógica contida no *FPGA* com barramentos de *I/O* externos e bancos de memória.

Os *FPGAs* apresentam três grandes vantagens sobre as arquiteturas fixas, como as encontradas nas *GPUs* e *CPU* de propósito geral, quais sejam: seus blocos lógicos e de *I/O* são reconfiguráveis de acordo com a demanda da aplicação, permitindo maior customização e desempenho; menor consumo de potência por trabalharem em frequências bem mais baixas que as arquiteturas concorrentes; disponibilizam uma grande quantidade de pinos de *I/O* para acesso a grandes bancos de memória. Assim, algoritmos que precisam acessar uma quantidade massiva de dados podem fazê-lo através da grande largura de banda disponibilizada nos *FPGAs*.

A Figura 2 destaca os elementos lógicos básicos característicos de um dispositivo *FPGA* (ROCHA, 2010; ARAUJO, 2010).

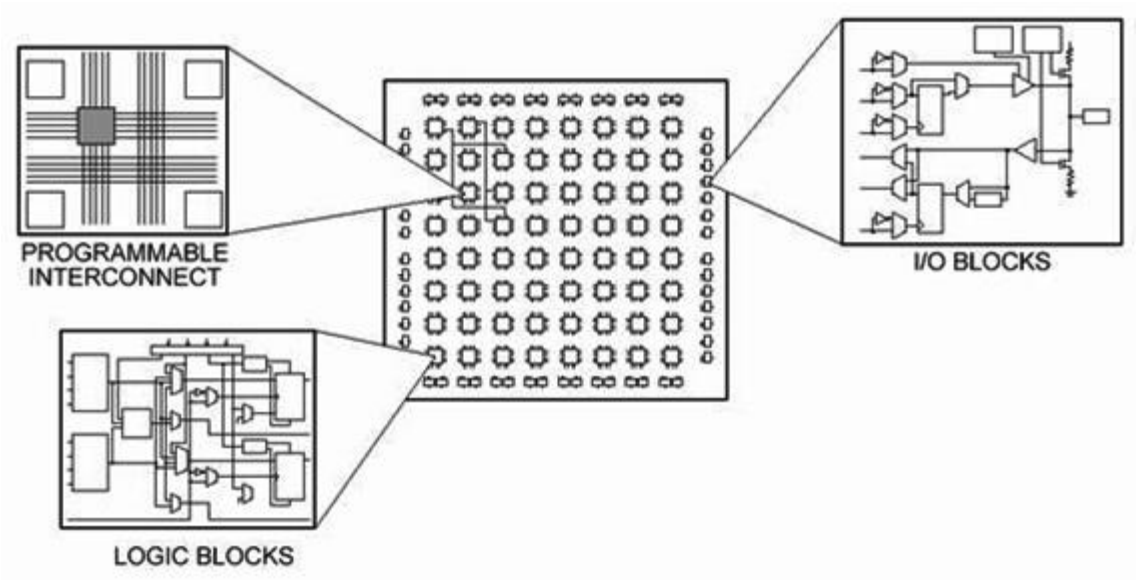

Figura 2: Elementos lógicos básicos de um *FPGA*[1].

Além disso, os elementos de processamento (*PEs*) podem ser projetados e customizados diretamente em *hardware*, com sua arquitetura específica e com uma estrutura totalmente customizada de arquitetura de memória, sem necessidade de etapas de busca e decodificação de instrução, característicos do paradigma seguido pelas *CPUs* convencionais. Ou seja, os blocos lógicos implementam elementos de processamento específicos para cada nova aplicação, a partir do uso direto de operações básicas (adição, subtração, multiplicação, etc...), necessárias para a computação dos dados recebidos (ULIANA, 2013; ROCHA, 2010).

[1] http://www.ni.com/white-paper/6983/pt/

Outra grande vantagem dos *FPGAs* em relação aos outros dispositivos é sua enorme granularidade de elementos lógicos, os quais permitem separar regiões físicas do *chip* para executarem tarefas de forma independente, inclusive com processamentos paralelos ao fluxo principal do algoritmo. Este alto grau de paralelismo e regularidade faz com que seja possível replicar os *PEs* várias, centenas de vezes, dependendo de seu consumo em blocos, possibilitando o aumento considerável do paralelismo de execução e, por conseguinte, desempenho das aplicações. Além disso, a execução de aplicações em *FPGAs* se dá em baixas frequências, quando comparadas àquelas empregadas nas *GPUs* ou *CPUs*, sendo um fator relevante para soluções que possuem como requisito um baixo custo energético por instrução, como dito acima.

Várias linguagens podem ser usadas para desenvolvimento de projeto de *hardware*, em geral denominadas de *Hardware Description Language*s (*HDLs*), tais como: *VHDL*, *Verilog*. *SystemVerilog*, etc. Nessas linguagens, é possível definir o comportamento de componentes lógicos, desde os mais simples, como portas lógicas, *flip-flops* e multiplexadores, até componentes mais complexos, como blocos de memória, *DSPs*, *CPUs*, sistemas inteiros.

Apesar dos fabricantes de *FPGAs* fornecerem ferramentas de desenvolvimento para projetos em *hardware* (ALTERA PRODUCTS, 2014), e existirem ferramentas de simulação para homologar os algoritmos desenvolvidos pelo usuário (MODELSIM, 2014), ainda não há ambientes padronizados de programação, integração com o *software* e *APIs*, como observamos para *GPUs*. Somando esse fator à alta complexidade de desenvolvimento algorítmico devido à flexibilidade intrínseca do *hardware*, o tempo para desenvolvimento de projetos envolvendo *hardware* e *software* costumam demorar um período de tempo bem maior do que os concorrentes. No entanto, iniciativas estão sendo propostas como solução para a redução deste *gap* de desenvolvimento, através de linguagens como a *OpenCL*, já bem difundidas em ambientes de projeto com arquiteturas *multi-cores* e *GPUs* (OPENCL, 2014).

Arquiteturas Híbridas

Em aplicações científicas, é notório que os maiores custos computacionais estão em seus trechos de processamento massivo de dados, mas também são dotadas de instruções cuja essência sequencial é latente. Isto se dá devido a paradigmas de programação bem estabelecidos e voltados para o desenvolvimento de programas para arquiteturas *Von Neumann*, além de alguns protocolos, como os de controle de *hardware* e de transferência serial de dados, que necessitam de um fluxo sequencial para o seu correto funcionamento (ROCHA, 2010). Para esses trechos, *CPUs* convencionais conseguem ser eficientes, pois sua arquitetura naturalmente sequencial, aliada às novas técnicas de processamento, como *pipeline*, previsão de desvios e *multithreading*, habilitam-as como a melhor opção para esse grupo de instruções.

Por outro lado, vimos também que mesmo com as novas técnicas e melhorias dos processadores tipo *SMP*, permitindo a execução de algumas *threads* simultâneas, a granularidade de paralelismo é extremamente insuficiente para atender à demanda massiva das aplicações científicas. A adoção de *clusters* para processar essas aplicações ameniza o problema. No entanto, o conjunto de tarefas recebidas por cada nó ainda contém um grande potencial paralelo, o que acarreta na espera de tarefas, de natureza independente, na fila de execução do processador. Em trechos de aplicações como estes, os dispositivos especializados em processamento paralelo massivo acabam obtendo uma maior eficiência no processamento. Por outro lado, o processamento de instruções sequenciais acaba sendo prejudicado com esta nova abordagem paralela, devido à frequência de operações mais baixa que a CPU e ausência de otimizações sequenciais (INTA, 2012; ESPENSHADE, 2009).

Para se ter um melhor aproveitamento dos mundos sequencial e paralelo das aplicações complexas, começaram a surgir soluções compostas por *CPUs* de propósito geral e dispositivos com alto poder de processamento paralelo, como os *FPGAs.* As arquiteturas de computação formada por esses dois elementos são chamadas de arquiteturas híbridas. As *CPUs* nessa nova estrutura são denominadas de *host*, o qual normalmente fica responsável por executar os trechos sequenciais da aplicação, além de controlar o fluxo geral de execução do programa, controlando o funcionamento e fornecimento de dados para os dispositivos massivos. Por sua vez, os dispositivos especializados em processamento paralelo, ficam responsáveis apenas por operar os algoritmos massivos, com os dados e instruções recebidos pela *CPU*, fazendo assim o papel de coprocessador (MITRION, 2014). Tipicamente o *host* e o coprocessador são conectados através de um barramento tipo *PCIe*, cuja velocidade de

transmissão pode chegar a até 32 GBps. A Figura 3 ilustra um exemplo de arquitetura híbrida que segue essa conexão, com um *PC* realizando o controle e comunicação com uma placa *FPGA* a partir de uma região de memória reservada em sua memória principal.

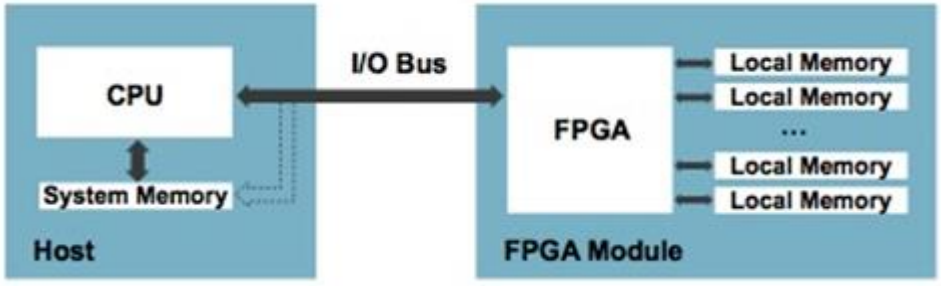

Figura 3: Arquitetura híbrida, com CPU e FPGA interconectados via barramento PCIe (MITRION, 2014).

Os *clusters* convencionais podem se beneficiar de soluções como esta, na qual os nós podem apresentar uma arquitetura híbrida, ou o próprio *cluster* apresentar nós formados por dispositivos heterogêneos. Assim, o sistema ganha um maior paralelismo nodal, e consequentemente uma capacidade maior de desempenho. *Clusters* que possuem essa característica são chamados de *clusters* híbridos, onde aplicações distribuídas precisam ter seus processamentos locais adaptados para otimizar o desempenho sequencial e massivo nesse novo ambiente. Esses *clusters* também apresentam uma vantagem na diminuição de recursos físicos, pois a granularidade atingida por dezenas de milhares de nós com apenas *PCs* convencionais, pode ser atingida com apenas alguns nós híbridos (ESPENSHADE, 2009).

Os *clusters* híbridos podem ser classificados com relação a como os seus nós são constituídos. Segundo o artigo a respeito do projeto Axel (TSOI, 2010) os *clusters* híbridos podem ser do tipo *UNNS (Uniform Node Nonuniform System)* ou *NNUS (Nonuniform Node Uniform System*), os quais representam as estruturas de *cluster* formadas por nós sendo *PCs* ou dispositivos aceleradores, ou os próprios nós possuindo arquiteturas híbridas, respectivamente.

Em sistemas *UNNS*, cada nó possui uma arquitetura homogênea, ou seja, cada participante possui uma característica única, sendo *CPU*, *GPU* ou *FPGA* integrado ao barramento. Nessa configuração, a instalação e gerenciamento dos participantes geralmente são simples, e os modelos de programação para cada componente são mais definidos. Cada nó pode ter comunicação entre os membros, ou a transferência de dados pode ser mediada pelo controlador da rede (INTA, 2012; TSOI, 2010). Cada uma dessas abordagens possuem vantagens e desvantagens, pois enquanto a comunicação direta proporciona uma maior eficiência no compartilhamento de dados, principalmente em aplicações com alta dependência

de reuso dos mesmos, é preciso implementar os *drivers* necessários para prover esse alcance. Já com a abordagem de mediação pelo controlador, geralmente um *PC*, os *drivers* são mais simples de serem desenvolvidos, no entanto, acrescentam *overhead* na comunicação entre dispositivos de processamento, desde que a transferência passa primeiro pelo controlador.

Numa configuração *NNUS*, o *cluster* possui uma formação mais homogênea, pois os nós possuem a mesma arquitetura híbrida. O controlador passa a ter menos atribuições, pois os nós podem utilizar protocolos de rede em nível de *software* para conversarem entre si. Geralmente cada participante é formado por um *PC* e um ou mais dispositivos coprocessadores, formando então um cluster *UNNS* interno, como em *Chimera* (INTA, 2012). O modelo de programação, apesar de abranger as instruções de *API* para os dispositivos pertencentes à estrutura, podem ser compilados e replicados de forma dinâmica, com a finalidade de serem executados em todos os participantes do *cluster*. A Figura 4 mostra exemplos ilustrativos dos dois tipos de *clusters* híbridos.

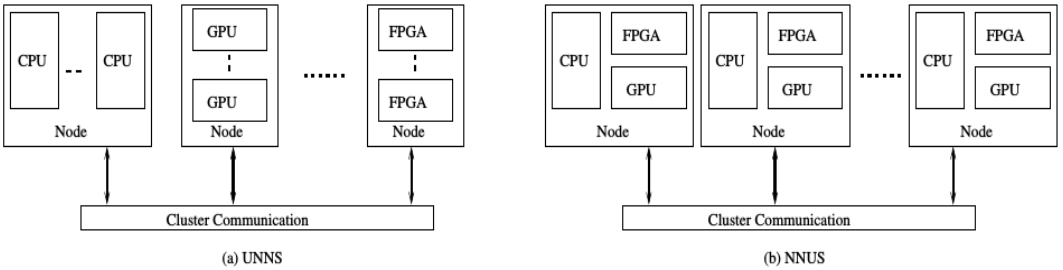

Figura 4: Tipos de clusters híbridos (TSOI, 2010). (a) UNNS e (b) NNUS

Como pode ser observado na Figura 4, as duas abordagens de *clusters* híbridos possuem diferenças estruturais importantes, as quais podem prover maior ou menor eficiência de processamento global, dependendo da natureza da aplicação distribuída. Em casos onde há pouca dependência de dados entre cada esquema de processamento, o *UNNS* pode atender satisfatoriamente essa classe de programas. Já em ambientes de execução com ata dependência de dados, principalmente com grande reuso de operandos nos trechos massivos, a abordagem *NNUS* é melhor aplicada (INTA, 2012).

2.3 Componentes de software

Nesta sessão serão abordados os padrões de configuração para execução paralela de uma aplicação num *cluster* convencional. Estas configurações são importantes para otimizar os trechos de software que serão integrados às arquiteturas de processamento durante o processo de desenvolvimento, de acordo metodologia proposta. Para o nível de constelação ou nível local, a configuração obedecendo ao princípio de memória compartilhada é a mais adequada, pois os processadores *multicores* são otimizados para esse nível. O *OpenMP* para esse cenário é um padrão amplamente adotado.

Para o nível de distribuição, o princípio de passagem de mensagem fornece melhor otimização para comunicação distribuída entre os nós participantes, obtendo melhor eficiência no controle e distribuição dos dados e instruções pela rede. O *MPI*, neste caso, é o padrão amplamente adotado pelas Organizações.

Tanto o *OpenMP* quanto o *MPI* serão apresentados, com suas principais características, a seguir.

2.3.1 *OpenMP*

OpenMP, acrônimo para *Open Multi-Processing*, é uma *API* que permite configuração explícita pelo usuário para que certos trechos de um programa possam ser executados com paralelismo. Essa *API* possui suporte para C/C++ e Fortran, podendo-se desenvolver código para diversas arquiteturas de computadores e Sistemas Operacionais, como *Windows* e *Linux* (OPENMP, 2014).

Seu princípio de paralelismo é baseado em compartilhamento de memória, com execução simultânea dos trechos de código em forma de *threads*. Essas *threads* podem ser executadas de forma independente nos núcleos (*cores*) dos processadores e também podem compartilhar informações entre eles, armazenadas na memória.

O *OpenMP* surgiu no começo dos anos 90, quando fabricantes de máquinas com arquiteturas de memória compartilhada criaram algumas extensões de compilação para Fortran. Os objetivos eram semelhantes, tais como paralelização de *loops*, os quais eram desenrolados e divididos para execução entre os processadores *SMP*, mas as implementações dessas extensões eram incompatíveis entre fabricantes diferentes.

A partir de 1994, foi criado o primeiro padrão para compartilhamento de memória, o *ANSI X3H5*. Esse padrão não foi utilizado em larga escala na época devido à diminuição de

interesse em máquinas com memória compartilhada, com o aumento das vendas de sistemas com memória distribuída (*clusters*). Após o surgimento de novas arquiteturas que utilizam o princípio de compartilhamento de memória, houve uma necessidade de definir um novo conjunto de especificações para elas, e então foi criado o *OpenMP* por volta de 1997.

O *OpenMP* foi inicialmente lançado para Fortran, com a implementação em C/C++ sendo definida pela primeira vez em 1998. A *API* teve duas versões distintas para as linguagens até a versão 2.5, lançada em 2005, onde ocorreu a unificação das implementações, mantidas até hoje, no decorrer das versões. Atualmente o *OpenMP* possui o apoio de várias organizações, além de fabricantes de *hardware* (OPENMP HOMEPAGE, 2014).

O gerenciamento das *threads* nesta *API* obedece o princípio *fork-join*, o qual consiste em um processo de inicialização (*fork*), execução, e finalização das mesmas (*join*). As *threads* são reunidas em grupos e são inicializadas simultaneamente. A execução de um próximo grupo só poderá ocorrer quando todas as *threads* do grupo anterior forem concluídas. O gerenciamento desses grupos é feito a partir de uma *thread* especial, chamada *master thread*. Esta *thread* segue sua execução de acordo com a ordem de grupos definida em tempo de compilação, realizando sincronização apenas para garantir o termino das *threads* de um mesmo grupo. O comportamento de cada *thread* de um mesmo grupo é definido previamente, de acordo com as regras determinadas pelo usuário. Logo, é responsabilidade do desenvolvedor do programa garantir que trechos do código poderão ou não serem executados de forma paralela, além da corretude da operação dos dados. A Figura 5 retrata um exemplo de fluxo típico de um programa com *OpenMP*, e seus principais componentes.

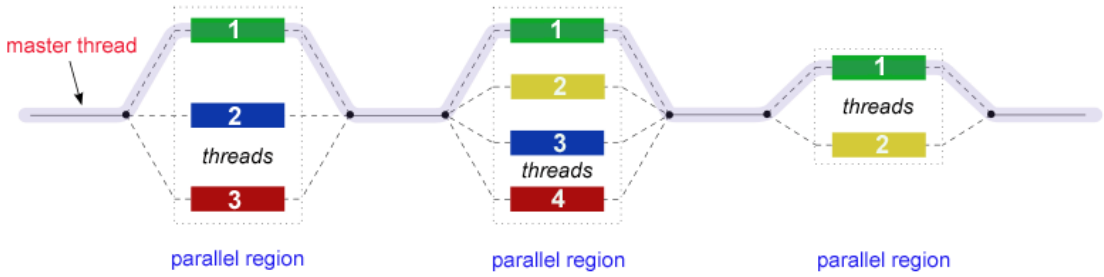

Figura 5: Comportamento de um programa com *OpenMP*, e seus principais componentes (OPENMP, 2014)

A configuração das *threads* pode ser feita no código fonte, em forma de diretivas de compilação e de funções específicas da *API*, ou através de variáveis de ambiente, podendo um mesmo programa ser executado de maneiras diferentes sem precisar recompilar o código. O objetivo é que uma mesma configuração possa ser compatível para qualquer compilador que

possua implementação desse protocolo. No entanto, alguns não possuem suporte a determinadas configurações, como por exemplo, paralelismo aninhado.

Com relação às diretivas, elas se apresentam de forma semelhante a comentários dentro do código, onde só os compiladores com a implementação da *API* conseguem interpretar. Ou seja, elas são apresentadas como *pragmas*, de forma a informar ao compilador que o trecho compreendido deve ter a configuração descrita. As diretivas possuem a seguinte estrutura em C/C++:

```
#pragma omp nome_da_diretiva cláusulas
    {
        (…)
    }
```

Cada uma das partes dessa estrutura possuem funções específicas. O **pragma** indica que aquela linha possui informação adicional de compilação. O **omp** informa que a informação faz parte do *OpenMP*. O **nome_da_diretiva** é referente a que tipo de configuração será feita naquele trecho, e quanto às **cláusulas** são as condições referentes a diretiva informada. Para indicar qual o trecho abrange a diretiva, coloca-se o mesmo entre chaves, onde a abertura da chave deve ficar abaixo da linha de configuração, da mesma forma que a mostrada na estrutura anterior.

Diretivas de configuração

Diretiva *Parallel*

Esta diretiva indica ao compilador que o trecho compreendido deve ser executado por todas as *threads* disponíveis, ou seja, será criado um grupo com todas as *threads* possíveis e com o mesmo trecho de código a ser executado. Dinamicamente, as *threads* recebem uma numeração, onde é definido que uma delas terá o valor zero e será considerada a mestre do grupo, enquanto as outras terão numeração entre um e (número_de_threads - 1). É considerada fundamental para a construção de trechos com *OpenMP*, visto que ela determina que o conteúdo dentro do escopo fará execuções simultâneas. As *threads* dentro do mesmo grupo são inicializadas, e a *master thread* só permite a inicialização de um novo grupo, com

outra configuração, quando todas as *threads* desse grupo são concluídas, obedecendo a barreira de sincronização implícita descrita anteriormente (OPENMP, 2014).

As cláusulas para a diretiva *parallel* definem basicamente o número de *threads* que serão criadas e como as variáveis dentro do escopo irão se comportar. Detalhes de funcionamento de algumas delas serão mostrados a seguir:

- ***if***: esta cláusula define uma condição para a região ser ou não executada com paralelismo, caso a variável relacionada tenha o valor verdadeiro (valor não zero) ou falso (zero), respectivamente.
- ***private***: define que em cada *thread*, a variável recebida como parâmetro terá um valor diferente. Muito útil para iterações.
- ***shared***: neste caso, a variável torna-se pública, ou seja, todas as *threads* terão acesso à mesma região de memória e qualquer alteração ocorrida em uma das *threads* pode ser vista por todas as outras.
- ***default***: dá liberdade ao usuário definir o comportamento da variável em cada *thread*, podendo ser de qualquer tipo.
- ***reduction***: realiza uma operação em determinada variável, ou seja, seu valor final passa a ser o resultado de uma operação feita entre valores finais dela nas *threads*.
- ***num_threads***: determina a quantidade de *threads* que serão criadas dentro do grupo.

A diretiva *parallel* deve ser inserida antes das outras, caso seja desejado o comportamento *multithread*. Caso o bloco a ser configurado contenham *branchs* (*if* ou *goto*), não será possível o uso dessa diretiva, e consequentemente de explorar o paralelismo com o *OpenMP*, pois execuções condicionais podem levar a regiões de memória fora do escopo definido estaticamente.

Diretiva *For*

A diretiva *for* é usada para configuração de trechos com *loop* do tipo *for*. Com ela, podemos trabalhar como cada iteração dentro do *loop* será executada dentro das *threads*. Muito útil para definir como o *for* será desenrolado, ou seja, como extrair paralelismo das iterações. Para tanto, é preciso que a diretiva *parallel* seja inserida antes.

As cláusulas aplicáveis nessa diretiva podem determinar o comportamento das iterações. Também podem ser usadas as mesmas da *parallel*, caso elas não tenham sido incluídas na diretiva principal. Seguem as descrições de algumas cláusulas do *for*:

- ***schedule***: realiza divisão das iterações entre as *threads* do grupo, podendo ser:
    - *static*: onde a divisão é definida estaticamente de acordo com um número, chamado de *chunk*, ou seja, cada *thread* executará um número de iterações definido pelo *chunk*. Caso o *chunk* não seja definido, as iterações são divididas igualmente entre as *threads*.
    - *dynamic*: realiza a divisão das iterações de forma dinâmica, onde cada *thread* executa *chunk* iterações. Caso a *thread* termine seu conjunto atribuído, é alocado um novo conjunto de iterações. Caso o *chunk* não seja inserido, o valor padrão passa a ser um.
    - *guided*: semelhante ao *dynamic*, mas com ajuste dinâmico de divisões. Inicialmente a divisão entre as *threads* é definida pelo número de iterações dividido pelo número de *threads*, caso ainda sobre iterações a serem executadas, a divisão passa a ser o número de iterações restantes dividido pelo número de *threads*.
    - *runtime*: a decisão da divisão passa a depender dinamicamente do valor da variável de ambiente *OMP_SCHEDULE*.
    - *auto*: é delegado ao compilador ou ao ambiente de execução a divisão das iterações.
- ***nowait***: esta cláusula determina que as *threads* não precisam ter sincronização após o término das execuções.
- ***ordered***: determina que a execução das iterações entre as *threads* obedeçam a mesma ordem, caso o *loop* fosse executado serialmente.

Diretiva *Sections*

A diretiva *sections* permite definir que trechos diferentes podem ser executados simultaneamente. Assim como a *for*, é preciso que a *parallel* seja definida previamente. O grupo criado com essa diretiva também possui sincronização implícita para garantir o término das *threads*, a qual pode ser desativada com o uso da cláusula *nowait*. As outras cláusulas de variáveis, caso não estejam previamente declaradas em *parallel*, também podem ser usadas.

Se o trecho possui algum *branch*, não poderá ser utlizado como uma sessão independente.**iretiva *Single***

Essa diretiva, ao contrário das anteriores, define que o trecho definido seja executado de maneira serial, em outras palavras, apenas uma *thread* é escolhida para executar o escopo de código englobado por essa configuração. As outras *threads* ociosas, por padrão, ficam esperando o término do trecho, podendo ser contornado com o uso da cláusula *nowait*. Também é possível utilizar as cláusulas de dados, e não pode ser utilizado para englobar trechos que contenham *if* ou *goto*.

Diretivas de sincronização

As diretivas de sincronização são úteis para determinar como certos trechos de código, os quais precisam ter garantia de exclusividade de execução, irão se comportar no grupo de *threads*. Uma dessas características é o bloqueio ou sessão crítica, a qual uma vez qualquer uma das *threads* começa a executar dentro dessa região, nenhuma outra pode executar. Outra situação é quando precisa-se garantir a mesma ordem de execução de um programa serial. Essas e outras situações são descritas a seguir.

Diretiva *Master*

Quando se utiliza esta diretiva, o trecho compreendido só pode ser executado pelo mestre do grupo, ou seja, a *thread* que recebe o identificador zero.

Diretiva *Critical*

O trecho o qual é atribuído essa diretiva pode ser replicado para todas as *threads*, mas a execução só pode ser feita por uma delas de cada vez. Operações como funções de *I/O* ou manipulação de variáveis globais podem ser exemplos de utilização dessa configuração. No *OpenMP* é possível nomear as regiões críticas, de forma a permitir trechos diferentes de código serem tratados da mesma forma, ou seja, não poderão ser executados simultaneamente. Por padrão, regiões as quais não possuem nome são consideradas como se tivessem o mesmo nome (*unnamed*).

Diretiva *Barrier*

Essa diretiva permite ao programador criar uma barreira de sincronização explícita, a qual as *threads* só poderão seguir seus fluxos de execução após todas do mesmo grupo passarem por ela.

Funções do *OpenMP*

As funções do *OpenMP* possuem a finalidade de realizar configuração das *threads* durante a execução do código, podendo definir o comportamento das mesmas dinamicamente. Podem ser usadas para habilitar ou desabilitar certas características de funcionamento dos grupos, como execuções com paralelismo aninhado. A seguir, temos algumas funções do *OpenMP*:

- ***omp_set_num_threads (int n)**:* define o número de threads que serão executadas na região. Recebe como entrada um inteiro, referente ao número desejado de threads.
- ***int omp_get_num_threads ()**:* função responsável por indicar o número de *threads* em sua variável de retorno. Esse número pode ser configurado, tanto estaticamente, através da cláusula *num_threads*, quanto dinamicamente, pela função *omp_set_num_threads* explicada anteriormente, ou então através da variável de ambiente *OMP_NUM_THREADS*.
- ***int omp_get_thread_num ()**:* retorna o identificador da *thread*, único dentro de um grupo. Esse identificador é um número inteiro que varia entre zero e (*num_threads* - 1), e é atribuído a *thread* durante toda sua execução dentro do grupo.
- ***int omp_in_parallel ()**:* indica, através de sua variável de retorno, se aquela região onde a função foi chamada está sendo executada ou não em paralelo. O retorno pode ter um valor zero (falso), indicando que a região não é executada de forma paralela, ou não zero (verdadeiro) para indicar que a execução concorrente ocorre.
- ***omp_set_dynamic (int n)**:* permite configurar o número de *threads* que poderão ter ajustes em tempo de execução. Esse número é um inteiro passado como parâmetro dessa função.
- ***int omp_get_dynamic ()**:* indica se a *thread* está apta a receber ajustes dinâmicos, retornando um valor não nulo (verdadeiro) caso esteja, ou zero (falso) caso o recurso esteja indisponível para ela.

Variáveis de ambiente

São variáveis que são inicializadas ou alteradas dentro do ambiente de execução, antes do programa ser executado. Podem modificar o comportamento dos programas, dependendo da configuração estática e das funções aplicadas ao código fonte. Sua ativação no Linux se dá através dos comandos *setenv* ou *export*.

- ***OMP_SCHEDULE***: permite configurar o escalonamento dos trechos paralelos que contam loops for. Os tipos de configuração são os mesmos da diretiva correspondente.
- ***OMP_NUM_THREADS***: determina o máximo número de threads que serão criadas durante a execução do programa.
- ***OMP_DYNAMIC***: pode ou não habilitar configurações dinâmicas nas threads.
- ***OMP_PROC_BIND***: permite nomear as *threads*, de forma que o Sistema Operacional escalone-as da mesma forma como os processos.
- ***OMP_NESTED***: permite habilitar ou não o paralelismo aninhado em um programa.

2.3.2 MPI (Message Passing Interface)

O *MPI* é um protocolo de transmissão paralela de dados, a qual possui uma *API* bastante conhecida e considerada padrão de comunicação em clusters. O princípio básico de seu funcionamento, o *message passing*, consiste em sincronização e movimentação dos dados, onde os mesmos partem da memória do emissor para a memória do receptor, mantendo o mesmo espaço de endereços. Logo, o protocolo é indicado para configuração de paralelismo a nível de processo, com modelos de memória distribuída, podendo inclusive dividir uma grande porção de memória de um nó em regiões menores, com as mesmas propriedades de uma memória nodal (OPENMPI, 2014).

A especificação do *MPI* foi concebida em meados de 1992, num encontro envolvendo cerca de 40 instituições, e surgiu a partir da unificação de diversos outros protocolos de comunicação de memória distribuída. A padronização de fato começou em 1993, sendo concluída em 1994 com a versão MPI-1.0. Houveram diversas atualizações, como o MPI-1.1 (1995), MPI-1.2 (1997) e o MPI-1.3 (2008). Surgiram também versões extendidas do protocolo, uma delas é o MPI-2, o qual envolve todos os conceitos da versão 1, mais novas funcionalidades. Uma dessas funcionalidades é o gerenciamento das funções de *I/O* (*MPI-IO*), capazes de gerenciar funções abstratas de gerenciamento de dispositivos de

entrada e saída pela biblioteca. Outro novo conceito é o gerenciamento dinâmico dos processos, onde processos *MPI* podem criar novos processos ou estabelecer comunicação com outro processo iniciado separadamente. Já o *MPI-3*, aprovado em meados de 2012, é uma especificação melhorada dos conceitos presentes nas versões anteriores (MPI-3, 2014).

Uma das primeiras implementações da *API* é o *MPICH*, uma biblioteca inicialmente desenvolvida pelo *ANL* (*Argone National Laboratory*) e *Mississipi State University*. Uma outra implementação é o *OpenMPI* (OPENMPI, 2014), a qual surgiu a partir da união de outras implementações, como o *LAM/MPI*. Também há implementações comerciais, baseadas geralmente no *MPICH* ou no *LAM/MPI*. Tanto o *MPICH* quanto o *OpenMPI* possuem suporte para os principais Sistemas Operacionais utilizados, como Windows e Linux. Essas implementações dão suporte às linguagens C/C+ e *fortran*, podendo ser utilizada por outras linguagens, como *Java* e *Python*, através de *binds*, ou simplesmente chamadas de funções implementadas em C (MPI, 2014).

O protocolo pretende atingir os seguintes objetivos: fornecer uma topologia lógica (em forma de grafo ou cartesiana), sincronizar as mensagens transmitidas, prover um mapeamento dos elementos de comunicação para garantir sua funcionalidade entre os hosts, permite portabilidade de linguagem e Sistema Operacional, possibilita comunicação síncrona e assíncrona, comunicação ponto-a-ponto e coletiva, e suporte a paralelismo a nível de processos.

De acordo com os conceitos de *Flynn* (FLYNN, 1972), o *MPI* suporta os fluxos de execução *MIMD* (*Multiple Instruction Multiple Data*), com o processamento de vários dados por diferentes instruções, e o *SPMD* (*Single Process Multiple Data*), o qual é similar ao *SIMD*, considerando que um processo é um conjunto de instruções *SIMD*, e este pode transmitir os mesmos dados para diferentes processos.

Existem alguns conceitos que definem elementos do ambiente de comunicação e fazem parte do jargão do *MPI*. São eles:

- **Processo:** é um programa distribuído, o qual possui uma ou mais réplicas em cada um dos nós participantes.
- **Rank:** é um inteiro, cuja função é ser o identificador único de um processo, recebendo esse número assim que ele é iniciado. A numeração inicia com zero, ou seja, os processos recebem valores entre zero e (total_de_processos - 1).
- **Grupo:** conjunto de processos que podem trocar mensagens entre si.

- **Comunicador:** é um identificador de grupo, o qual representa um domínio de uma comunicação. O padrão é *MPI_COMM_WORLD*.
- **Comunicação ponto-a-ponto:** conjunto de funções que suportam comunicação entre dois processos.
- **Comunicação coletiva:** conjunto de funções que suportam comunicação de um para vários processos de um mesmo grupo, de forma simultânea.

Com relação à sincronização dos dados transmitidos em comunicações ponto-a-ponto, o *MPI* possui suporte aos tipos bloqueante ou não-bloqueante. A comunicação bloqueante tem por base apenas permitir a próxima instrução quando o receptor envia uma confirmação para o emissor (*ACK*). Caso a comunicação seja não-bloqueante, não há confirmação de recebimento, e a próxima instrução será executada independente do término da transferência.

Caso a comunicação seja do tipo coletiva, as funções que realizam essa funcionalidade são todas não bloqueantes, cabendo ao usuário utilizar funções de barreira para forçar sincronização entre os processos.

O protocolo suporta ainda diversos paradigmas de controle do fluxo de processamento distribuído, e entre eles, o mais simples de implementar e mais utilizado é o mestre-escravo, onde geralmente atribui-se o *rank* zero para ser o mestre (ESPENSHADE, 2009).

Com relação à topologia virtual de rede, o *MPI* suporta diversos tipos de configuração, sendo as mais simples os tipos anel e estrela (RAMOS, 2004).

A configuração em anel funciona em forma de comunicação cumulativa, ou seja, cada nó recebe um resultado parcial do seu vizinho e incrementa esse resultado com o seu processado internamente, passando essa soma para o próximo vizinho. O tipo de comunicação *MPI* utilizado geralmente é o ponto-a-ponto. O mestre processa sua parte e transmite para o primeiro nó escravo, responsável por passar seu resultado para o próximo escravo vizinho e assim por diante, até a chegada do último escravo, o qual passará o resultado final para o mestre. Essa configuração pode ser vista na Figura 6.

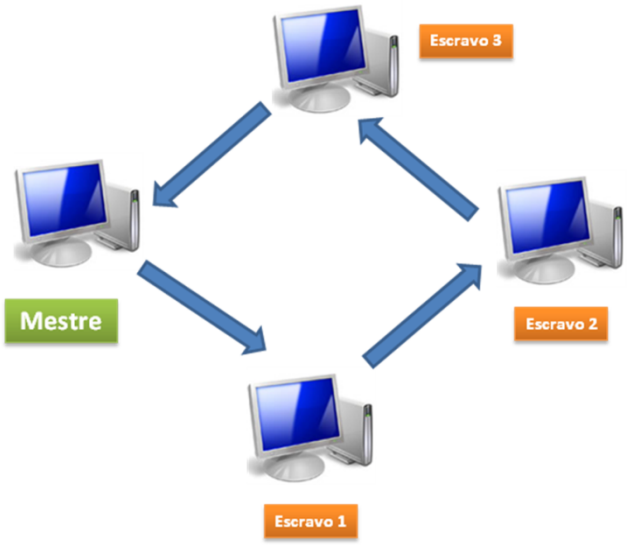

Figura 6: Topologia em anel, com 4 *hosts*.

Já a configuração estrela consiste em um processo realizar comunicação com todos os participantes do grupo. O mestre normalmente fica responsável por transmitir para todos os escravos os dados necessários para a computação local, e recebe os resultados parciais de cada um, a fim de reuni-los e formar a saída do sistema. O tipo de comunicação pode ser ponto-a-ponto ou coletiva. A Figura 7 ilustra como essa topologia é estruturada.

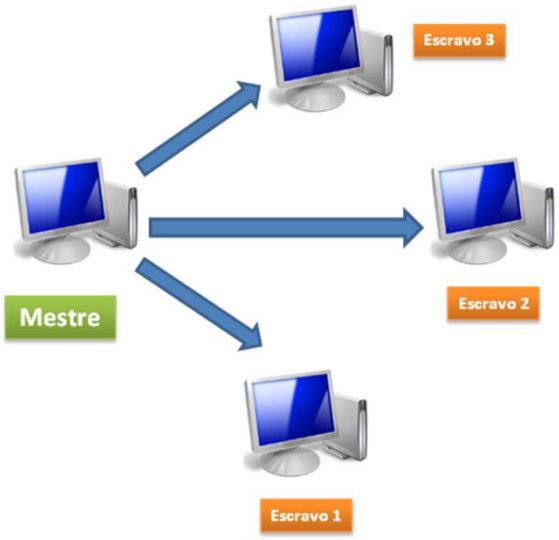

Figura 7: Topologia em estrela, com 4 hosts.

O modelo de programação do *MPI* consiste em configuração explicita do paralelismo, ou seja, o programador tem liberdade de configurar como e quando os processos serão inicializados e executados, além de definir como a transmissão de dados se comportará. Com relação à transmissão, devido a sua natureza paralela, as funções de envio e recebimento utilizam tipos próprios de dados que suportem paralelismo de comunicação. É possível também especificar um tipo fora dos pré-definidos, e para tanto, utiliza-se a função

define_MPI_datatype(), cabendo ao usuário adaptar esse novo tipo, de acordo com as características de paralelismo especificadas pelas funções. Os tipos padrão do *MPI* possuem equivalência com os tipos de C, conforme pode ser visto na Tabela 1.

Tabela 1: Tipos utilizados pelo MPI, e seus equivalentes em C (MPI, 2014).

C Data Types	
MPI_CHAR	signed char
MPI_WCHAR	wchar_t - wide character
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT MPI_LONG_LONG	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double

O *MPI* possui um conjunto de funções básicas, cujo objetivo é prover o controle do fluxo de execução e sincronização dos processos (MPI, 2014). As funções com essa característica são:

- ***MPI_Init (&argc, &argv)*** - inicia um processo *MPI*. Primeira função de uma aplicação que utiliza esse protocolo. Serve também para sincronizar os processos durante a inicialização. Os parâmetros devem ser os mesmos da função *main*, ou seja, *argc* é referente ao número de parâmetros passados pelo processo e *argv*, um *array* desses parâmetros.
- ***MPI_Finalize ()*** - termina um processo *MPI*, logo deve ser a última função do código. Assim que é chamada, ela libera a memória utilizada pelo protocolo. Não possui parâmetros.
- ***MPI_Comm_size (comm, &noProcesses)*** - retorna o número de processos (no endereço do segundo argumento, *noProcesses*) contidos em um grupo (indicado no parâmetro *comm*, cujo padrão é *MPI_COMM_WORLD*), passado como primeiro argumento representado por seu comunicador.

- ***MPI_Comm_rank (comm, &processId)*** - retorna o id do processo que chama a função (no endereço do segundo argumento). O primeiro argumento, *comm*, indica o grupo de que o programa participa cujo padrão é *MPI_COMM_WORLD*.
- ***MPI_Get_processor_name (computerName, &nameSize)*** - retorna o nome (em *computerName*) da máquina que efetua o processo chamador da função. O segundo parâmetro, *nameSize*, indica o tamanho do nome a ser retornado.

Para comunicações ponto-a-ponto e bloqueante, as seguintes funções estão disponíveis:

- ***MPI_Send (&src, n, MPI_TYPE, dest, tag, comm)*** - função de envio de dados para um processo específico. Nos parâmetros, *src* é o dado a ser enviado, o *n* é o número de itens a ser enviado, o *MPI_TYPE* é o tipo *MPI* ao qual o dado pertence, o *dest* é o processo (inteiro) que receberá o dado enviado, a *tag* é um identificador da mensagem enviada (se é de envio ou recebimento) e o *comm* é o comunicador do grupo de processos a qual eles pertencem.
- ***MPI_Recv (&dest, n, MPI_TYPE, src, tag, comm, &status)*** - função de recebimento de dados de um processo específico. Nos parâmetros, *dest* é a variável que receberá o dado, *n* é o número de itens a ser recebido, o *MPI_TYPE* é o tipo *MPI* o qual o dado pertence, o *src* é o processo (inteiro) de origem do dado, a *tag* é um identificador da mensagem enviada, *comm* é o comunicador do grupo de processos a qual eles pertencem e status representa a situação da mensagem (para efeitos de controle).

Para comunicações ponto-a-ponto e não-bloqueantes, as funções ***MPI_Isend*** e ***MPI_Irecv*** são disponibilizadas. Suas características e lista de parâmetros são os mesmos de seus equivalentes bloqueantes.

Para comunicações de natureza coletiva, as funções disponibilizadas pela *API* são:

- ***MPI_Bcast (&numberRects, n, MPI_TYPE, rank, comm)*** – função em que um único processo envia os mesmos dados para todos os participantes do grupo. O primeiro parâmetro, *numberRects*, representa o buffer de dados a serem enviados, já o *n* é o número de itens do buffer, *MPI_TYPE* é o tipo de dados do *buffer*, *rank* é o identificador do processo emissor, e *comm* é o comunicador.
- ***int MPI_Reduce (*operand, *result, count, MPI_TYPE, op, root, comm)*** – função onde todos os dados sofrerão uma operação, a partir de aplicações sucessivas, com o

resultado final sendo recebido por um nó escolhido. O *operand* é *buffer* que contém o valor a ser reunido de cada processo, *result* é o buffer que deve receber o resultado após a união dos dados, *count* é o número de elementos a serem reunidos, *MPI_TYPE* é o tipo *MPI* do dado no *buffer*, *op* é o tipo de operação de redução, *root* é o *rank* do processo a receber o resultado, e *comm* é o comunicador que representa o grupo. O retorno dessa função (inteiro) representa o status da operação, ou seja, indica se a função terminou de maneira bem-sucedida ou não.

- ***MPI_Barrier (comm)*** - esta função é capaz de criar uma barreira de sincronização explícita, ou seja, permite ao usuário forçar a sincronização naquele trecho, e todos os processos só seguirão para a próxima instrução quando todos executarem essa função. Nos parâmetros, temos o *comm* como comunicador.
- ***MPI_Scatter (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)*** - realiza uma transmissão simultânea de um *buffer* de dados para todos os membros do grupo, de forma que o receptor recebe uma fatia desse *buffer* inicial, isto é, a função quebra um grande *buffer* de dados em um número de pedaços igual ao número de processos envolvidos, depois transmite cada divisão para os membros, cuja regra é entregar a primeira fatia de endereços para o *rank* mais baixo, a segunda fatia para o segundo *rank* mais baixo, e assim por diante até esgotarem as fatias. Como parâmetros, temos *sendbuf* como o buffer de envio, *sendcnt* é o tamanho de cada parte da divisão do buffer, *sendtype* o tipo *MPI* dos elementos do *buffer*, *recvbuf* o buffer de recebimento, *revccnt* o tamanho desse buffer, *recvtype* o tipo *MPI* que será cada integrante do buffer de recebimento, *root* o *rank* do processo emissor e *comm* é o comunicador.
- ***MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)*** - esta função possui comportamento simétrico ao da função *MPI_Scatter*, quer dizer, aqui ocorre uma transmissão simultânea de *buffers* menores e de mesmo tamanho localizados em todos os processos do grupo, depois eles são reunidos de forma ordenada para formar um grande *buffer* de dados, no processo receptor. A regra de concatenação dos *buffers* é semelhante a de divisão, onde o *buffer* do primeiro processo preenche os primeiros endereços do *buffer* de recepção, o segundo preenche o segundo espaço de endereços e assim por diante. Com relação aos parâmetros são praticamente os mesmos do *Scatter*, ou seja, temos *sendbuf* como o buffer de envio, *sendcnt* é o tamanho desse *buffer*, *sendtype* o tipo *MPI* dos elementos do *buffer*, *recvbuf* o *buffer* de recebimento, *recvcount* o tamanho de cada parte que será

concatenada a esse *buffer*, *recvtype* o tipo *MPI* que será cada integrante do *buffer* de recebimento, *root* o *rank* do processo receptor e *comm* é o comunicador.

2.4 *RTM (Reverse Time Migration)*

O conceito de *RTM* inserido neste capítulo é importante para compreender como foi o desenvolvimento aplicado a um dos estudos de caso descritos neste trabalho. Com a introdução do algoritmo, é possível verificar toda complexidade do mesmo, e o motivo dele ser escolhido para ser acelerado num cluster híbrido.

O *RTM* é um método cuja principal funcionalidade é processar informações extraídas de um terreno, comparando com modelos matemáticos, e gerar imagens que permitam uma análise mais precisa das características do mesmo.

Aplicações na área de geofísica têm uma importância substancial para Organizações que exploram as riquezas de terrenos, em especial os subaquáticos, pois são mais difíceis de obterem-se informações visuais precisas. A exploração de minérios, combustíveis fósseis, ou até mesmo a busca por novos sítios arqueológicos, geram riquezas e benefícios para as Instituições, e consequentemente para o país. Os resultados mais notórios estão no setor de exploração do petróleo marítimo, com o aumento da extração em diversas áreas do território nacional, e a descoberta de uma nova e difícil região de extração, a camada Pré-sal, onde empresas, como a Petrobras, investem bilhões de dólares para explorar parte dessa nova região (ROCHA, 2010).

Essas aplicações de análise de terreno, geralmente baseiam-se em métodos de exploração sísmica, o qual possui um princípio de funcionamento simples. Primeiro, é preciso captar os dados de entrada do sistema. Ondas sonoras são disparadas de uma fonte em direção ao fundo do mar, refletem no fundo e retornam à superfície, sendo então captadas por receptores. Depois, são registrados a amplitude do sinal e o tempo que a onda gasta para percorrer desde o início até o destino final. Esse processo é repetido várias vezes, em diversos pontos, gerando um grande conjunto de dados. A partir daí, as empresas processam esses dados para estimar como se apresenta visualmente o relevo da região. Com essa estimativa visual, geólogos são capazes de verificar regiões características, com possibilidade de exploração (SANTOS, 2011). Esse método é bastante utilizado por empresas petrolíferas para descobrir reservas de petróleo e gás mineral. A Figura 8 nos dá um exemplo de como a captação de dados é feita nos oceanos e mares.

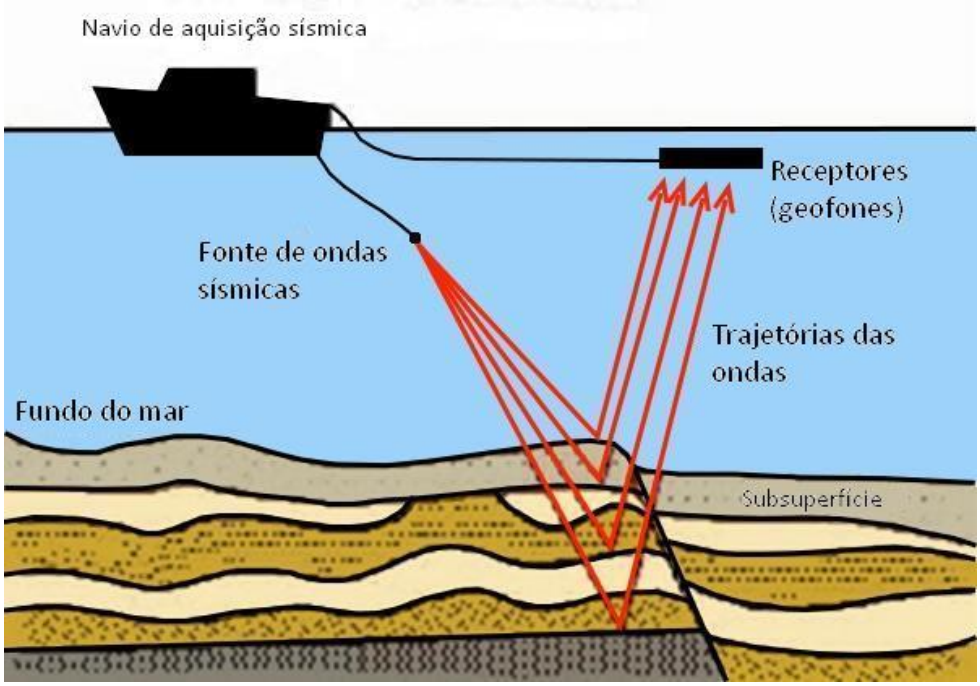

Figura 8: Método de exploração sísmica por reflexão (ROCHA, 2010).

Podemos observar no exemplo da Figura 8, um navio contendo uma fonte de pulsos e um compartimento com receptores, normalmente utilizados hidrofones, movimentando-se pela superfície sobre a região que se deseja analisar. Durante seu movimento, a fonte emite os pulsos sísmicos, e os receptores captam essas ondas refletidas dos pontos abaixo do trajeto percorrido pela embarcação. No final, podemos ter uma grande região percorrida por esse navio, gerando uma enorme quantidade de dados, em forma de sismograma (SANTOS, 2011).

Um dos objetivos do processamento sísmico é filtrar os possíveis ruídos existentes nesse sismograma, como reflexões de embarcações vizinhas, animais marinhos, submarinos, ou outros objetos que não façam parte do terreno subaquático. Assim, as reflexões reais formam a saída do sistema, gerando então um resultado melhor interpretável por um geólogo. Para esse processamento, um dos métodos mais utilizados pela indústria é o de *Kirchhoff*, o qual não resolve diretamente a equação da onda, e aplica diversas aproximações para calcular em áreas complexas. Embora mais simples de ser processado, em áreas com inclinações acima de 70°, o resultado final apresenta menor precisão (ROCHA, 2010, SANTOS, 2011).

O *RTM* possui a característica de resolver diretamente a equação da onda, de forma a gerar resultados mais precisos para os geólogos, mesmo em terrenos muito acidentados. Embora seja mais exato, seu custo computacional é muito alto, e apresenta um potencial paralelismo, caracterizando uma aplicação científica com trechos massivos, e portanto uma boa candidata para projetos em *HPC* (ROCHA, 2010). A Figura 9 mostra a diferença de imagem entre os dois métodos.

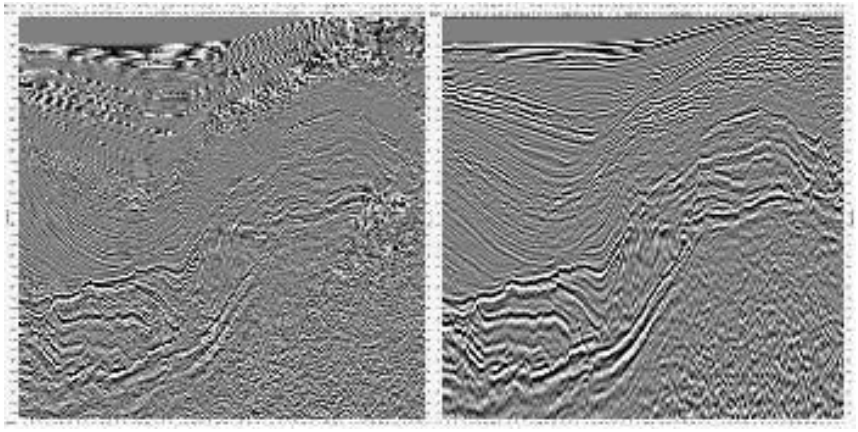

Figura 9: Imagem processada pelos métodos de *Kirchhoff* (esquerda) e *RTM* (direita)[2]

É possível perceber na Figura 9 que a imagem processada pelo RTM (direita) possui traços mais nítidos do terreno, provendo para o geólogo uma imagem que pode ser melhor analisada.

As principais etapas do processamento sísmico são a modelagem e a migração. Na modelagem, o processamento é feito a partir da fonte até os hidrofones, indicando o processamento direto da propagação da ondas. Na migração, o processamento é inverso, ou seja, do hidrofone até a fonte geradora. O resultado é considerado correto quando o resultado do processo direto é o mesmo do processo inverso. Logo, a filtragem de um terreno real é feita a partir de um modelo matemático processado diretamente, cujo resultado é utilizado como entrada para o processo de migração com o modelo de sismograma captado com ruído (ROCHA, 2010).

Com relação a modelagem, inicialmente é utilizado um modelo de velocidades do terreno, o qual fornece a velocidade instantânea de cada ponto do caminho de propagação entre a fonte e o hidrofone, ou seja, os dados da emissão das ondas são gerados de forma sintética, para servirem de entrada. Então é feito o processamento direto, de forma a resolver o calculo da equação de onda, cuja fórmula é escrita assim (1):

$$\nabla^2 u(x;\ y;\ z;\ t) - \frac{1}{V(x;y;z)^2}\frac{\partial^2 u(x;y;z;t)}{\partial t^2} = f(t)\delta(x - x_s)\delta(y - y_s)\delta(z - z_s) \quad (1)$$

Onde:

f(t) - função que representa a fonte

(x_s, y_s, z_s) - são as coordenadas da fonte com relação ao referencial de observação

u (x, y, z, t)- são o campos de pressão nos pontos (x, y, z) com relação ao tempo t.

[2] http://petex.pesgb.org.uk/cgi-bin/somsid.cgi?page=html/abstracts/abstractid37

Para que a equação torne-se computável, é preciso discretizá-la, de acordo com as regras de diferenças finitas. Logo, a equação pode ser calculada da seguinte maneira (2):

$$C_{i,j,k} = 2 * B_{i,j,k} - A_{i,j,k} - \{Vel^2_{i,j,k} * fat * [16 * (B_{i,j+1,k} + B_{i,j-1,k} + B_{i+1,j,k} + B_{i-1,j,k} + B_{i,j,k+1} + B_{i,j,k-1}) - 1 * (B_{i,j+2,k} + B_{i,j-2,k} + B_{i+2,j,k} + B_{i-2,j,k} + B_{i,j,k+2} + B_{i,j,k-2}) - 90 * B_{i,j,k}]\} \quad (2)$$

Onde:

$C_{i,j,k}$- matriz do campo de pressão e ser processada

$B_{i,j,k}$ - matriz do campo de pressão no tempo atual

$A_{i,j,k}$- matriz do campo de pressão do tempo anterior

$Vel_{i,j,k}$- matriz de velocidades

A computação se dá com um conjunto de iterações aninhadas, as quais possuem o papel de varrer de forma ordenada as matrizes participantes da fórmula. No final de cada iteração no tempo, a matriz do campo de pressão processada passa a ser a matriz de pressão atual, e a matriz de pressão atual passa a ser a matriz de pressão anterior. Cada elemento necessário para calcular um ponto da matriz de C forma uma janela de processamento, a qual realiza a varredura em toda a matriz de B. A Figura 10 mostra graficamente essa janela.

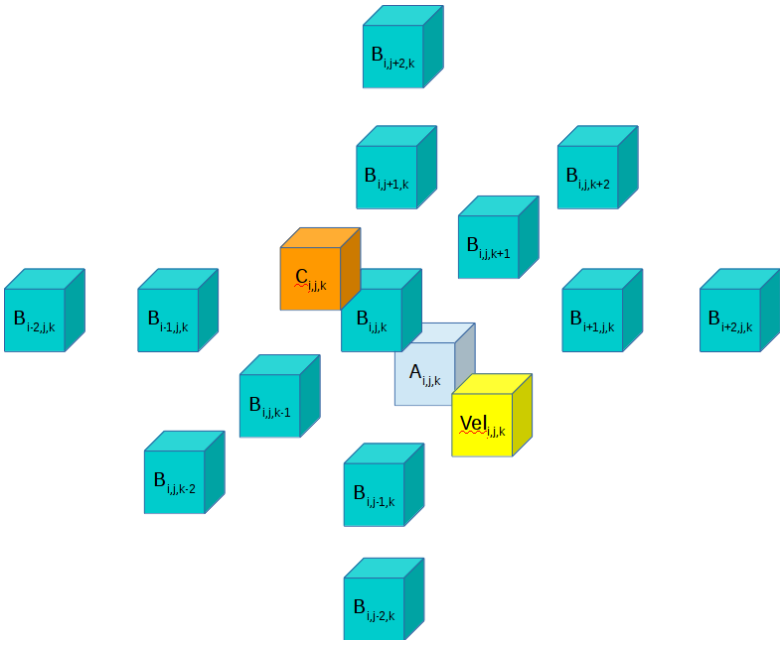

Figura 10: Janela de processamento do *RTM* 3D

Depois de processada toda a matriz de tempo futuro C, utilizando a matriz de velocidades e iterando em todos os intervalos de tempo, os quais chamaremos a partir de agora de passos temporais ou *time steps*, a matriz de sismograma é gerada. Essa matriz é composta pelos dados capturados ao longo do tempo pelos hidrofones, ou seja, as colunas representam os receptores das ondas, e as linhas os intervalos de tempo. Um exemplo dessa matriz pode ser visto na Tabela 2.

Tabela 2: Exemplo de uma matriz de sismograma (ROCHA, 2010).

		Receptor			
		rec_1	rec_2	rec_3	rec_4
Tempo de Trânsito	tt_1	0.06	0	0	0
	tt_2	0.08	0.05	0	0
	tt_3	0.07	0.07	0.04	0
	tt_4	0	0.04	0.06	0.04
	tt_5	0	0	0.04	0.05
	tt_6	0	0	0	0.03

Na Tabela 2, observamos que em cada coluna há um conjunto de valores numéricos. Esses valores são as amplitudes do sinal recebido em cada intervalo de tempo. Na verdade, o eixo x representa as medidas de afastamento entre a fonte e cada receptor, onde começa com zero e termina com o valor da distância do receptor mais longe. A Figura 11 ilustra como seria essa representação.

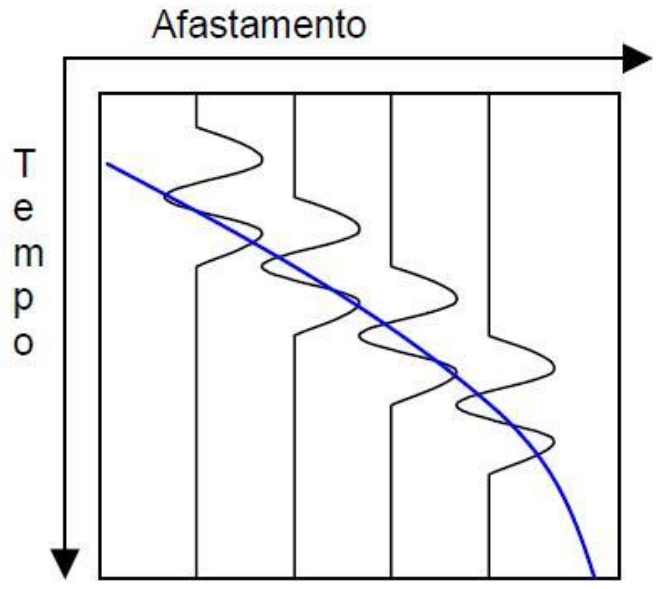

Figura 11: Representação gráfica do sismograma (ROCHA, 2010).

As linhas verticais apresentadas na Figura 11 são representações do traço sísmico de cada fonte, com uma semi-hipérpole ligando os pontos de amplitude, e esta por sua vez representa uma estimativa do traçado do terreno a ser investigado.

Depois de gerado o sismograma, a partir de dados sintéticos da fonte, é feita uma análise de comparação dos dados do sismograma, com o objetivo de encontrar os intervalos de tempo em que ocorrem as maiores amplitudes, pois isso é indicativo do tempo em que ocorreu a reflexão, e consequentemente o tempo que a onda tocou no terreno. Então é formada a matriz de tempo de trânsito, com os tempos das amplitudes máximas de todos os receptores (ROCHA, 2010).

Por fim, para comparar o sismograma real com o sintético, é feita a migração do sismograma real, que recebe como entrada também a matriz de tempo de trânsito gerada, a fim de realizar mesmo cálculo da modelagem direta. O final desse processamento resulta na matriz de sessão migrada, a qual está pronta para ser analisada pelo geólogo.

3. Trabalhos relacionados

Este capítulo apresenta um levantamento de trabalhos acadêmicos e soluções comerciais relacionadas ao desenvolvimento de aplicações de alto desempenho em cluster que utilizam coprocessadores aceleradores baseados em *FPGAs*. O objetivo é posicionar este trabalho com os projetos já existentes, comparando as funcionalidades e deficiências oferecidas por eles, e quais funcionalidades serão oferecidas pela metodologia empregada na arquitetura desenvolvida.

No trabalho de *Inta et al.* (INTA, 2012) é descrito o desenvolvimento de um nó de um *cluster* híbrido, o qual é composto por uma *CPU* convencional e uma arquitetura de coprocessamento, equivalente a um *cluster* interno. Esse *cluster* interno é formado por placas contendo *GPUs* e outras equipadas com *FPGAs*, interligadas por um barramento de alta velocidade, que por sua vez é parte constituinte do *PC*. A inspiração do nome "Chimera" veio da semelhança do sistema com a figura descrita na mitologia grega, formada por três animais: uma cabra, um leão e uma cobra.

A área de atuação do sistema é a computação científica, cujos algoritmos possuem uma gigantesca quantidade de dados. Esses dados são processados com algoritmos que possuem um alto potencial paralelo, tais como operações com matrizes esparsas, matrizes densas, em domínio da frequência e cálculos geométricos.

A escolha de uma arquitetura, integrando *CPU, GPUs* e *FPGAs*, para resolver algoritmos científicos, deu-se pelo fato de possibilitar o aproveitamento das melhores características de cada um de seus componentes. O *FPGA* possui um conjunto de componentes que podem ser configurados de forma bastante flexível, podendo instanciar mais elementos de processamento e com arquitetura customizada de acesso aos dados. Possui também alta eficiência de paralelismo em operações envolvendo números inteiros e de ponto fixo. Já a *GPU* possui arquitetura fixa e especializada em operar eficientemente vetores e matrizes, com elementos de processamento de quantidade pré-estabelecida, os quais possuem acesso a elementos de memória de alta velocidade. Além disso, a *GPU* possui unidades de operação em ponto flutuante mais eficientes que o *FPGA* e a *CPU*, além de um ambiente de desenvolvimento de seus algoritmos bem definido e mais fácil de utilizar que o *FPGA*.

O tipo de *cluster* híbrido que poderia ser implementado com o nó desenvolvido, seria do tipo *NNUS*, onde cada participante teria a mesma arquitetura híbrida. Ou seja, um nó, com

sua arquitetura interna heterogênea (ou um *UNNS*), representaria o sistema não uniforme, enquanto a associação de nós com essa mesma configuração resultariam num conjunto uniforme. O texto não demonstra uma implementação desse *cluster* externo, só descreve que seria possível desenvolvê-lo, pois o foco escolhido foi analisar as características e o desempenho apenas de um nó, isto é, a análise dos resultados do cluster interno.

Com relação ao *cluster* interno, cada tipo de participante possui funções bem definidas. O PC realiza o papel do *host*, controlando o fluxo de execução dos dispositivos internos, além de servir de interface para o usuário poder usufruir do sistema. As placas aceleradoras apenas executam as operações massivas sobre os dados da aplicação, de acordo com as requisições determinadas pelo *host*.

Em *Chimera*, a *GPU* escolhida como nó foi a *Nvidia Tesla C2070*, com interface *PCIe 16x*. A placa com *FPGA* foi a *DE-530*, contendo uma *FPGA Stratix IV* e uma interface *PCIe 8x*. O meio de comunicação escolhido foi o barramento *PCIe*, cuja velocidade de transmissão de dados pode chegar a 32 GB por segundo. A escolha dos componentes foi motivada por eles serem produtos disponíveis no mercado, evitando a necessidade de se encomendar um componente específico, o que aumentaria os custos de produção do sistema. A *CPU* escolhida foi uma *Intel Core i7*, e uma placa mãe *Intel DX58SO2*. Essa placa possui duas portas *PCIe* de *16x*, onde uma das portas pode ser usada para placas com suporte a 16x, e a outra é dividida em duas entradas as quais suportam conexões 8x. De acordo com a mesma lógica de minimização de custos, o barramento escolhido para montar a arquitetura heterogênea foi o próprio *PCIe* disponível no *PC*. Logo, o *Chimera* possui um cluster interno, com a capacidade máxima de alocação para uma placa *Tesla 2070* e duas placas *DE-530*. A arquitetura do sistema pode ser visualizada na Figura 12.

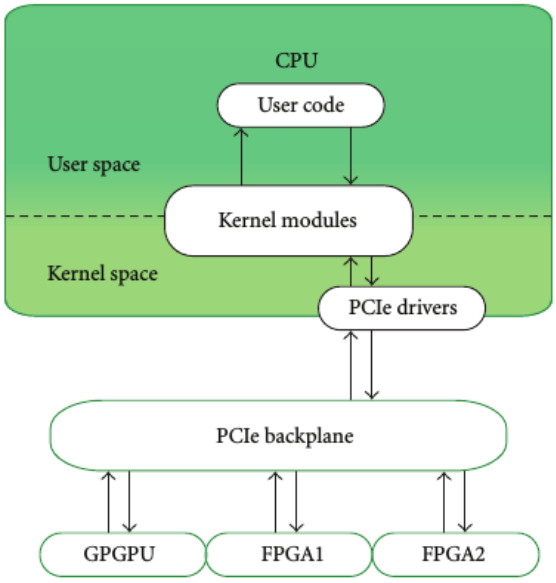

Figura 12: Arquitetura implementada do Chimera (INTA, 2012).

Na Figura 12 temos uma visão geral da estrutura do *Chimera*. Podemos observar o canal de comunicação *PCIe* com ligações entre os dispositivos atuantes. Embora a imagem passe a impressão de todos os participantes poderem transmitir dados entre si, este recurso não foi implementado, nem os fabricantes dos dispositivos aceleradores o disponibilizaram. Então, tanto o *FPGA* quanto a *GPU*, só realizam comunicação com o *PC*, o controlador do barramento. Esse controle é feito através dos *drivers* instalados no Sistema Operacional. O Sistema Operacional utilizado no projeto foi o *kernel Linux*, devido à facilidade de desenvolvimento e suporte. Recursos como memória virtual para cada periférico, inclusão e exclusão dos módulos controladores de dispositivo (através dos comandos *insmod* e *rmmod*) e prevenção de acesso indevido de programas do usuário à região de memória reservada ao dispositivo, são bons exemplos de estímulo ao uso do *kernel* de código aberto.

Para que o usuário pudesse aproveitar os recursos disponíveis nesse sistema de forma transparente, foi preciso adaptar os programas utilizados por eles, para que estes fossem executados de forma híbrida. A saída encontrada foi desenvolver uma biblioteca de alto nível para um ambiente bastante utilizado em aplicações científicas, como por exemplo o *MATLAB*. As funções dessa biblioteca teriam chamadas de sistema para enviar os algoritmos a serem executados pelas placas *PCIe*. Essas chamadas de sistema foram desenvolvidas de acordo com a característica de cada dispositivo, utilizando tanto as funções da *API* incluída nos *drivers* da *Nvidia*, como referenciando as desenvolvidas para o *driver* da *DE-530*. Com relação ao algoritmo a ser executado em cada placa, temos duas situações distintas de ambiente de desenvolvimento. Enquanto no lado do *Tesla*, o fabricante oferece a arquitetura

CUDA, e uma linguagem de programação a qual é um subconjunto de *C/C++*, no lado do *FPGA* foi utilizado a linguagem *Verilog* e a ferramenta *Modelsim* para simulação dos algoritmos.

Percebe-se que uma arquitetura definida de elementos de processamento e memória, como a *GPU*, facilita o fluxo de desenvolvimento de algoritmos para ela. Enquanto que a *FPGA*, por ter uma maior flexibilidade de seus elementos, tende a ter uma maior otimização, customização de recursos para executar um algoritmo, mas com um tempo maior de desenvolvimento.

Com relação aos estudos de caso, foram considerados exemplos que poderiam explorar as principais características dos dois tipos de placas aceleradoras, ou seja, operações vetoriais envolvendo ponto flutuante na *GPU*, e operações de alta escalabilidade com números inteiros e de ponto fixo na *FPGA*. Um desses exemplos foi o conhecido algoritmo de Integração de *Monte Carlo* para cálculo do número π. Esse algoritmo é baseado no cálculo da divisão entre a área de um quadrado e a área de um círculo inscrito nesse quadrado, como está ilustrado na Figura 13.

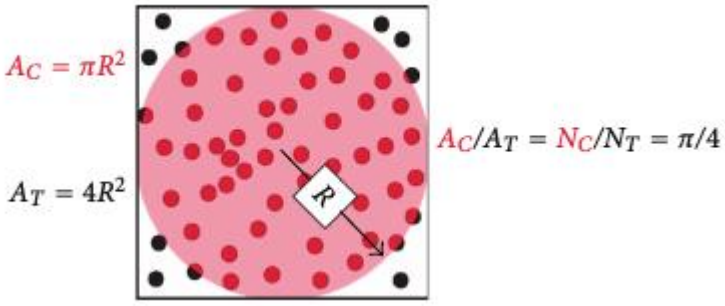

Figura 13: Visão geométrica do algoritmo de Monte Carlo. (INTA, 2012).

Para adaptar esse cálculo para a plataforma heterogênea desenvolvida, houve uma separação das etapas do fluxo do algoritmo. No *FPGA*, foi implementado com melhor desempenho, árias ordens de grandeza maior que a geração implementada em *GPU*, o algoritmo de geração de pontos que resulta na geração de números randômicos uniformemente distribuídos. Já para a *GPU*, constatou-se que cálculo da inequação obteve mais resultados em menos tempo, visto que a resolução das potências quadradas, envolvendo números de ponto flutuante, são mais eficientes. Uma visão geral do fluxo de processamento final do algoritmo é mostrada na Figura 14.

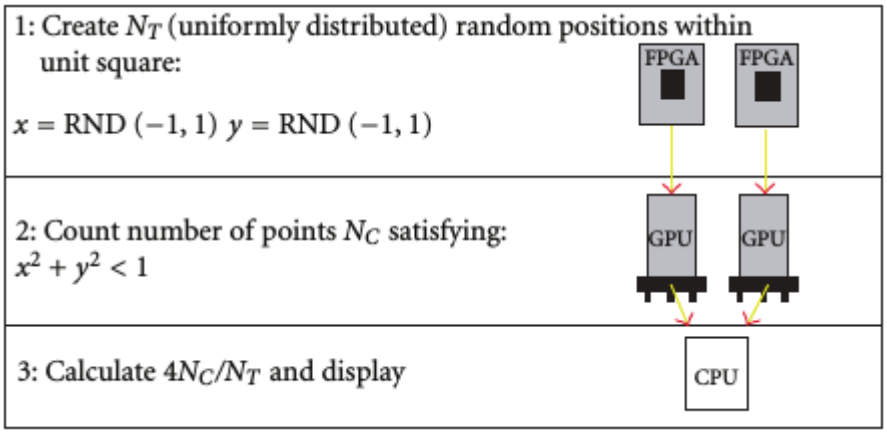

Figura 14: fluxo de execução do Monte Carlo na plataforma heterogênea (INTA, 2012).

Como se pode observar na Figura 14, cada uma das classes de participantes do *Chimera* possuem tarefas definidas. Para a *FPGA*, a geração aleatória dos pontos com números entre -1 e 1. Para a *GPU*, o cálculo da inequação do círculo $x^2 + y^2 < 1$. E por fim, a *CPU* do sistema é encarregada de calcular a divisão entre a área aproximada do círculo e do quadrado, multiplicar por 4 e disponibilizar o resultado para o usuário.

O Artigo apresenta a arquitetura de um *cluster* interno heterogêneo, com uma boa diversidade de recursos oferecidos tanto pelo *FPGA* quanto pela *GPU* instaladas. É mostrado que os dispositivos aceleradores possuem qualidades intrínsecas, que podem agir de forma complementar para resolver diversos tipos de problemas, com aritmética distinta, na área científica. A transparência para o usuário através de adaptações ao *MATLAB* também merece destaque, pois torna a curva de aprendizado de utilização do sistema mais suave.

Embora o texto mostre que é possível a implantação de um *cluster* com nós híbridos heterogêneos, não há um estudo de caso a respeito, dado o foco do mesmo em realizar estudos com apenas um nó. Com relação ao *cluster* interno, nota-se uma limitação de escalabilidade, pois depende fortemente da oferta de portas *PCIe* por parte da placa mãe. Além disso, há um gargalo do tráfego de dados, devido ao controle exclusivo do barramento por parte da *CPU*, onde é desperdiçado um maior aproveitamento das características de processamento das diferentes placas, como por exemplo, realizando comunicação direta entre o término do processamento de uma e início da próxima.

Em *Espenshade et al* (ESPENSHADE, 2009) foi descrito o desenvolvimento de um *framework* para um *cluster* híbrido comportando um ou mais *FPGAs* com processador embarcado. O *framework* é baseado na implementação de uma estrutura de serviços que interage de maneira padronizada com os algoritmos desenvolvidos pelo usuário. Assim, há

um encapsulamento do conjunto "*algoritmo + interface de dados*" em uma estrutura de abstração denominada *Hardware Abstraction Unit* (*HAU*). Essa estrutura é integrada a um sistema de *core services*, com diversos recursos adicionais, também há uma estrutura em software capaz de fornecer acesso direto à rede via *MPI*, e uma *API* simples para realizar comunicação e controle da arquitetura do usuário.

Para implementar o *cluster* os autores escolheram trabalhar com placas contendo *FPGAs* da *Xilinx* (XILINX, 2014) desde que seus modelos *Virtex-4 FX* e *Virtex-5 FXT* possuiam um processador embarcado baseado na arquitetura *PowerPC*. Assim, com a adição dos módulos de rede, elas estão habilitadas para a implementação das *HAUs*.

Com o objetivo de tornar o *framework* uma plataforma mais confortável para adaptação do usuário, foram selecionadas as principais características que seu ambiente de desenvolvimento deveria ter, considerando o alto poder de flexibilidade do *FPGA* e sua variedade de modelos e placas, além de certos requisitos considerados atraentes pelos usuários. Um desses requisitos foi a facilidade de programação, pois é preciso que o desenvolvimento, ou a adaptação dos programas para a plataforma, utilize uma linguagem mais próxima possível das conhecidas pelos usuários. Para comunicação entre os nós, foi adotado o *MPI*, devido a sua padronização e grande adoção no segmento de comunicação interprocessos. Com relação ao controle e comunicação com o algoritmo em hardware, as funções *fopen, fread*, *fwrite* e *fclose* de *C* foram escolhidas para serem adaptadas como chamadas de sistema. Outro requisito relevante para os autores foi a metodologia *hardware/software co-design* para desenvolvimento das aplicações, utilizando o *framework*.

Uma outra funcionalidade abordada como importante foi a minimização de *overhead* entre os componentes do *HAU*, pois é preciso encontrar um equilíbrio entre flexibilidade e integração com a pilha de implementações utilizadas no ambiente. Para tanto, foi decidido limitar um pouco a flexibilidade de implementação na *FPGA*, de forma a certas estratégias de implementação possam ser aplicadas a um conjunto maior de algoritmos. E por último, o texto retrata como requisito a escalabilidade, onde a utilização do *MPI* embarcado satisfaz essa necessidade. A implementação *MPI* escolhida para o ambiente foi o *OpenMPI*.

A arquitetura de *software* desenvolvida para o *framework* possui camadas bem definidas e estão instaladas no Sistema Operacional com *kernel linux* embarcado. O *kernel* em questão, versão 2.6, foi configurado e compilado com um reconhecimento mínimo de *hardware*, apenas garantindo o de placas *ethernet* e cartões *flash*, necessários para embutir o *SO* na placa com *FPGA*. Para a instalação do *OpenMPI*, foi preciso compilar seu código fonte, já que o sistema alvo de instalação é bastante específico. Além disso, o *OpenMPI*

utiliza outros *softwares* para garantir comunicação segura e acesso remoto, como o *OpenSSL* e *OpenSSH* respectivamente. Esses programas também precisaram ser compilados, assim como foi necessária a instalação dos *drivers* de reconhecimento dos algoritmos em *hardware*. A estrutura em camadas da arquitetura de *software* está ilustrada na Figura 15.

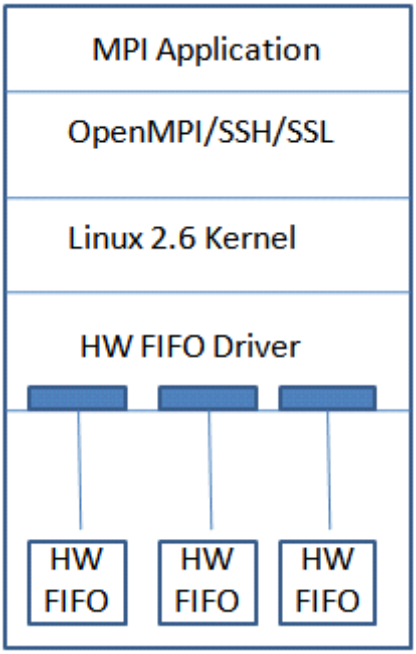

Figura 15: Pilha de software do framework (ESPENSHADE, 2009).

A Figura 15 dá uma visão geral da pilha de software que um nó do *cluster* possui. Vemos na primeira camada o protocolo *MPI*, responsável por gerenciar as instruções e dados a serem enviados e recebidos entre os nós. Mais abaixo, estão os protocolos de segurança *SSH* e *SSL*, utilizados pelo *MPI*. Na terceira camada, o sistema operacional embarcado, com *kernel linux 2.6*, o qual estão instalados todos os softwares e realiza o gerenciamento tanto dos protocolos de segurança e comunicação quanto dos *drivers* controladores do *HAU*. E na camada mais baixa, temos os *drivers*, responsáveis por permitir o controle e gerenciamento de dados dos algoritmos em hardware.

Quanto à estrutura de hardware, primeiramente foi preciso definir qual protocolo de comunicação entre o algoritmo e o software seria implementado. De forma a facilitar a portabilidade e simplificar a estrutura, foi adotado o esquema de *FIFOs*, onde cada algoritmo teria uma *FIFO* de entrada de dados e uma de saída, correspondendo, respectivamente, a *FIFO* de escrita e de leitura de dados. Como descrito anteriormente, os modelos *FX* e *FXT* de *FPGAs* da *Xilinx* possuem um processador *PowerPC* embarcado, onde será executado o *Linux* com os *drivers* para gerenciamento do algoritmo, da placa *ethernet* e do cartão *flash*. Esses dispositivos fazem interconexão através de um barramento chamado *Processor Local Bus*

(PLB), através do qual o processador pode realizar seu gerenciamento. Uma visão geral da estrutura de *hardware* do projeto está ilustrada na Figura 16.

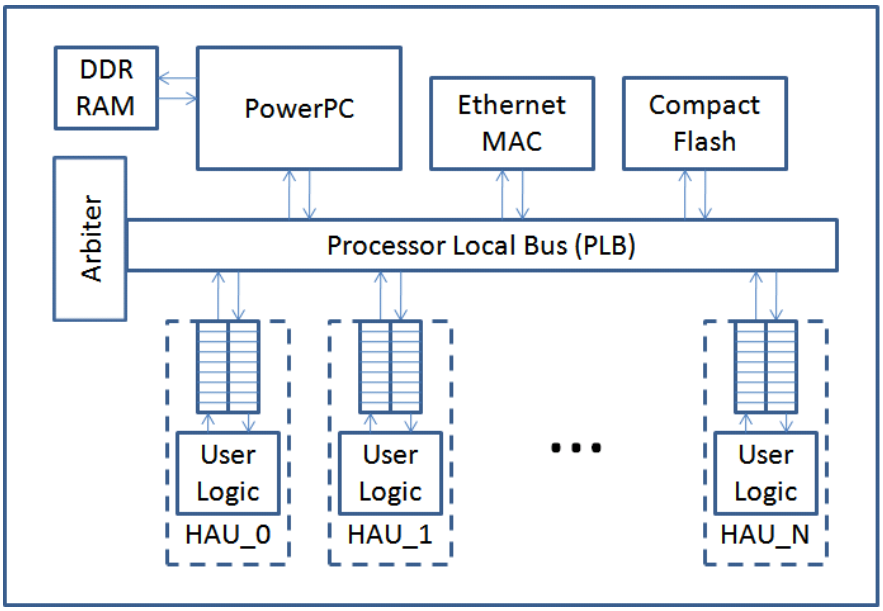

Figura 16: Visão geral da estrutura de hardware implementada (ESPENSHADE, 2009).

Como pode ser observado na Figura 16, o barramento *PLB* permite comunicação entre o processador, os periféricos e as *HAUs*, as quais possuem uma interface *FIFO* que recebe dados do barramento, chamada *Write FIFO*, e outra *FIFO* a qual disponibiliza os resultados processados de volta ao processador, a *Read FIFO*.

Os recursos disponíveis para o usuário desenvolver suas soluções são baseados em *APIs* já existentes, com o objetivo de deixar transparentes as mudanças estruturais da arquitetura. Para comunicação entre os nós, utiliza-se o *OpenMPI* embarcado, cujas funcionalidades originais foram mantidas. Para inicialização das *HAUs*, foi criada uma função especial, a *AllocateResource*, cujo retorno é um ponteiro de arquivo, o qual permitirá a manipulação do mesmo através das funções de arquivo modificadas de C. Por convenção, cada processo *MPI* é responsável por inicializar e controlar uma *HAU* do sistema, possibilitando a comunicação de cada elemento de *hardware* pela rede.

O estudo de caso escolhido foi o *DES (Data Encryption Standard)*, um conhecido algoritmo da área de criptografia, cujo princípio é baseado em encriptação e desencriptação utilizando um vetor de 56 bits chamado de chave. O objetivo dos pesquisadores era conseguir executar, na plataforma proposta, uma aplicação capaz de decifrar, via força bruta, qual a chave foi utilizada para encriptar um texto conhecido, dado o texto já encriptado, resultado da utilização dessa chave a ser encontrada. No pior caso, o número de chaves que seriam processadas pelo programa é de 2^56, uma enorme quantidade de dados de entrada.

Devido à flexibilidade da arquitetura dos *FPGAs*, enquanto dispositivos convencionais, aceitam apenas tipos definidos como inteiros (32 bits) ou *longs* (64 bits), estes aceitam implementações com tamanhos variados de bits, como 56 bits por exemplo. Além disso, a área de criptografia possui muitos algoritmos de comparação ou de deslocamento, cuja implementação pode resultar em simples comparadores binários ou *shift-registers*, permitindo um grande número de operadores desse tipo dentro de uma pequena área do *chip*. Também é possível criar arquiteturas de *pipeline* e de reuso próprias para cada algoritmo, de forma a otimizar e aumentar o paralelismo das operações.

Para demonstrar e obter resultados, foi utilizado uma estrutura física formada por duas placas com *Virtex-5 FX70T*, duas com *Virtex-5 FX130T* e uma com *Virtex-4 FX60*, todas conectadas em rede. Essas placas contém, cada uma, controlador *ethernet*, até 512 MB de capacidade de *RAM DDR2*, e um cartão f*lash* de 512 MB. Além disso, elas possuem um *PowerPC* embutido em cada *FPGA*, onde nos modelos *Virtex-5* o modelo apresentado é o *PowerPC 440*, e na *Virtex-4* há um *PowerPC 405*. Os processadores embarcados são conectados através do *PLB*, com frequências de 400 MHz (*Virtex-5*) e 300 MHz (*Virtex-4*). Os *HAUs* implementados rodam a 100 MHz, distribuídos entre as placas de acordo com a seguinte organização: 1 unidade para a *Virtex-4*, máximo duas para a *Virtex-5 FX70T* e um máximo de três unidades para a *Virtex-5 FX130T*. Logo, temos uma estrutura física formando um cluster com no máximo 11 *HAUs*, pois temos duas *Virtex-5 FX70T* (com máximo de 4 *HAUs*), duas *Virtex-5 FX130T* (máximo de 6 *HAUs*), além da unidade embutida na *Virtex-4*. Cada uma dessas unidades possuem um desempenho ideal de 8.8 bilhões de chaves testadas por segundo.

O texto compara o desempenho de sua estrutura de hardware e software com relação a algumas plataformas de mercado, como o *cluster* com *Intel Xeon GPP* executando apenas *software* distribuído, e os supercomputadores *Cray* e *SRC*. A tabela da Tabela 3 mostra o resultado de desempenho da solução proposta, baseada em *FPGAs* disponíveis comercialmente, ou *Commodity FPGAs*, comparado com as soluções em *CPU*, *Cray* e *SRC*.

Tabela 3: Comparativo de desempenho do cluster *HAU* com as soluções de mercado (ESPENSHADE, 2009).

Platform	Key Throughput (keys/second)	Speedup Over Software Solution
Commodity FPGAs	8,548 Million	1107 x
SRC-6	4,000 Million	518 x
Cray XD1	7,200 Million	930 x
2.33 GHz Xeon (11 cores)	7.722 Million	1 x

Podemos observar na Tabela 3 que a solução proposta no artigo possui um bom desempenho quando comparada a outras soluções de mercado. Em relação ao *Cray*, o desempenho pode ser considerado ligeiramente maior, enquanto com relação ao *SRC* percebemos um aumento de 2x em desempenho utilizando o projeto desenvolvido.

O texto comparou também a relação desempenho/ custo da solução em *FPGA* e o do cluster de *CPUs*. Considerando que o custo para montar um cluster com 11 processadores *Xeon* custariam *U$4000*, e o custo para implementar o projeto com *FPGA* custou aproximadamente *U$8000*, a relação desempenho / custo do FPGA é 1107/8000 = 0,138375, enquanto que a relação da solução em *CPUs* é 1 / 4000 = 0,00025. Com isso, vemos que a referida relação envolvendo *FPGA* é em torno de 550x maior que a relação encontrada se considerarmos as *CPUs*.

O projeto apresentado no artigo mostra um ambiente de *hardware* e *software* bastante estruturado e cumprindo bem o objetivo da transparência. A estrutura de *software* permite a utilização de *APIs* e funções conhecidas, onde a programação do usuário acaba gerando uma curva de aprendizagem menor, restando apenas adquirir o conhecimento das funções especiais de gerenciamento do algoritmo. Já a interface entre o algoritmo em *hardware* e o *software*, baseada em uma *FIFO* de entrada e uma de saída, facilita tanto o conhecimento da estrutura de dados que o algoritmo precisa trabalhar internamente, como na implementação das funções de leitura e escrita dos mesmos na parte do *software*. O ambiente que possibilita essa integração entre *hardware* e *software*, através de um *Linux* modificado para executar no processador embarcado na *Virtex*, além de toda uma estrutura disponível na placa com *FPGA*, permitiu que o cluster não precisasse de uma *CPU* convencional. O estudo de caso mostrou o quanto esse *cluster* pôde ser eficiente para a classe de aplicação selecionada.

Embora tenha sido bastante positiva a utilização de um processador embarcado e já disponível na placa da *Xilinx*, esse recurso demandou bastante trabalho por parte dos desenvolvedores, pois foi preciso configurar e compilar um *kernel* padrão *Linux* para poder executar num *PowerPC*, uma arquitetura menos usual que o x86, com maior tempo para

estudo e pesquisa, principalmente para gerar o conjunto distribuição + *kernel* de maneira que ocupasse o menor espaço possível no cartão *flash*. Outra limitação é o fato da portabilidade do f*ramework* ser fortemente dependente da placa com *FPGA* utilizada, ou seja, se ela atende a todos os requisitos mínimos (processador embarcado, controlador *ethernet* e controlador de cartão *flash*). Com relação ao próprio *HAU*, o texto retrata uma interface composta por apenas duas *FIFOs*, mas algoritmos que necessitam de mais *FIFOs* de interface precisariam ser separados em estruturas menores, para serem compatíveis com o *cluster*, gerando um aumento significativo da troca de dados entre cada estrutura menor.

Knodel et al. (KNODEL, 2013) descreve uma proposta de arquitetura na qual o cluster híbrido possui uma arquitetura peculiar com os nós compostos por um *PC* e uma placa com um *FPGA*, organizados de maneira a permitir uma comunicação direta entre os dispositivos reconfiguráveis. Além disso, a comunicação entre os *hosts* precisa passar pela interconexão de seus respectivos *FPGAs*. Foram implementados ambientes de programação para o usuário utilizando os padrões do *OpenMP* e *OpenCL*.

A arquitetura proposta neste trabalho atende mais especificamente à comunicação interdispositivos, onde a arquitetura de um *cluster* híbrido, baseada no conceito de ilhas de nós híbridos, passa a ter um barramento de comunicação dedicada e de alta velocidade para seus coprocessadores. A transferência de dados entre os *hosts* se dá também através desse barramento. Outras ilhas de nós terão comunicação entre si. O objetivo desse arranjo computacional é simplificar o acesso e controle dos membros, além de reduzir os custos de implantação da interface dedicada de comunicação. A Figura 17 mostra uma visão geral desse *cluster* proposto e como essas ilhas realizam comunicação entre seus membros.

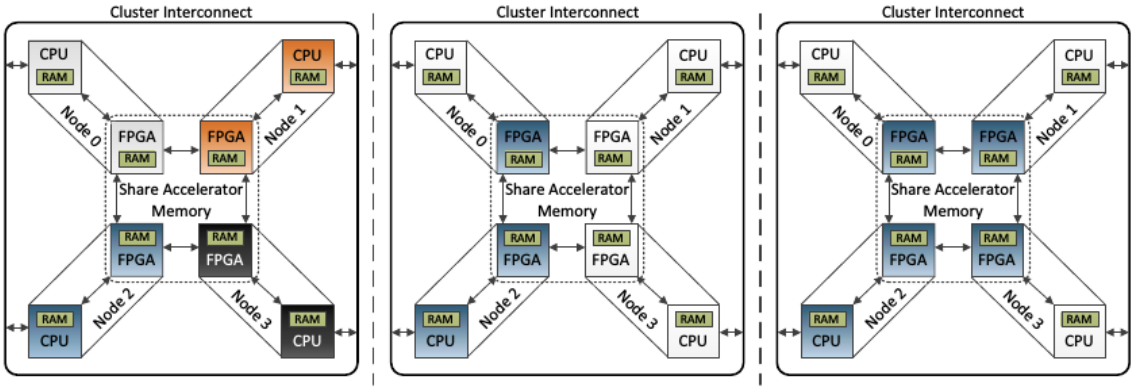

Figura 17: Visão geral do *cluster* proposto (KNODEL, 2013).

Como podemos observar na Figura 17, o *cluster* é formado por ilhas, cuja construção é baseada na interconexão dos dispositivos reconfiguráveis, utilizando um barramento rápido. É possível perceber também que cada *FPGA* pode realizar comunicação apenas com dispositivos reconfiguráveis específicos, ou seja, ocorre comunicação inter-*FPGA* apenas nos sentidos horizontal e vertical, não havendo, portanto, a transmissão de dados no sentido diagonal. Essa forma de arranjo foi inspirada em soluções já existentes utilizando *GPUs* (DUATO, 2010).

O gerenciamento dos recursos por parte dos usuários de um *cluster* híbrido, também foi um dos requisitos analisados pelos autores. Foi preciso idealizar um ambiente que seja o mais próximo possível de ferramentas já conhecidas no mercado. Então, foi desenvolvido um sistema em *batch* que mapeia os recursos necessários para gerenciamento, tais como número de *cores CPU*, memória e ambiente de execução. Assim, é possível localizar e gerenciar de forma simplificada os *jobs* ativos no sistema, de acordo com os recursos disponíveis no *cluster*, como por exemplo, controlando a fila de execução desses *jobs*. Além disso, esse sistema deve ser multiusuário, para que diferentes usuários possam executar e gerenciar suas tarefas, além de vincular informações da conta do usuário no registro de cada *job* iniciado por ele. Com relação às placas aceleradoras, foi preciso incluir suas informações de disponibilidade no *batch*, pois os *jobs* também precisam acessar esses recursos, e essas tarefas precisam ser gerenciadas de acordo com a capacidade dos dispositivos coprocessadores.

Cada *FPGA*, dentro do esquema proposto no texto, possui características funcionais tanto para execução das implementações desenvolvidas pelos usuários, quanto para comunicação e disponibilidade de seus recursos, com controle atribuído ao *host* associado. Com a finalidade de integrar aos sistemas de gerenciamento de *jobs*, os algoritmos em *hardware* são alocados de forma dinâmica, pois como as tarefas podem conter execuções massivas distintas, a placa precisa ser preparada para reconfiguração dinâmica dessas estruturas de processamento. Para atender a todos esses requisitos, foi desenvolvida uma arquitetura de *core services*, cujos componentes principais estão ilustrados na Figura 18.

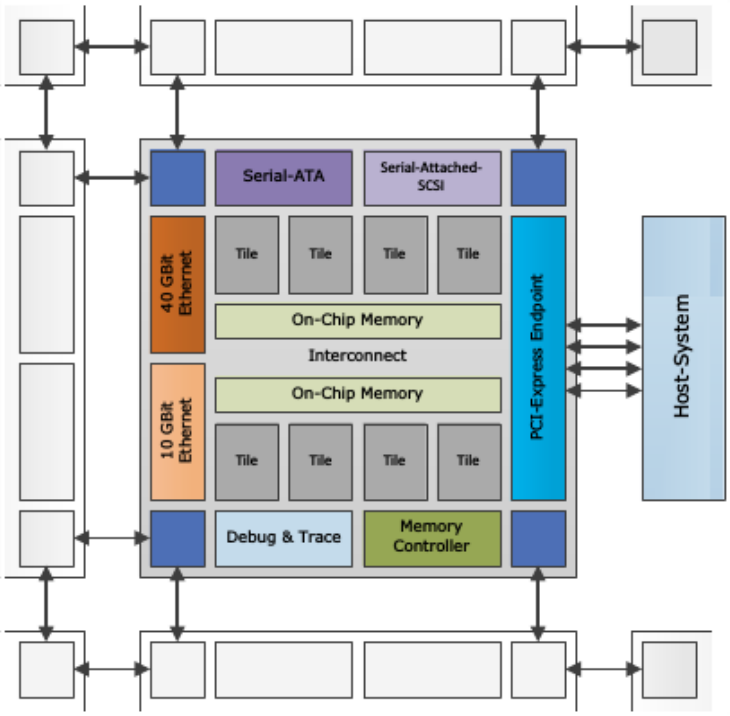

Figura 18: Arquitetura do *core services* implementada no *FPGA* (KNODEL, 2013).

A Figura 18 mostra uma visão geral da estrutura de serviços desenvolvida para cada *FPGA*. Podemos observar que os módulos estão em localidades fixas, pois facilita a otimização dos mesmos e o gerenciamento dos recursos. Há dois tipos de componentes envolvidos nessa arquitetura. Um deles corresponde aos módulos cuja configuração é total ou estável, ou seja, uma vez carregados no *FPGA*, não sofrerão mudanças durante o ambiente de execução. O outro tipo abrange os componentes de reconfiguração parcial ou dinâmica, os quais podem ter sua lógica substituída de acordo com as necessidades durante o ambiente de execução. Grande parte dos componentes de configuração total estão nas regiões periféricas, como forma de deixar mais próximo possível dos pinos de *I/O* reservados para cada componente. Eles são responsáveis por prover os recursos necessários, tanto para comunicação entre o *host*, placas vizinhas e entre a placa e a próxima ilha, quanto para os algoritmos a serem carregados na placa.

Os algoritmos desenvolvidos pelo usuário devem ser encaixados em regiões pré-determinadas do dispositivo. Essas áreas são chamadas de *tiles*, as quais estão configuradas para receber as lógicas de forma dinâmica, através de reconfiguração parcial do *FPGA*. Assim, o *host* pode definir como a placa irá processar os dados das aplicações. Na Figura 18 temos um exemplo de uma estrutura capaz de receber até oito algoritmos dinamicamente.

Para realizar a análise e otimização das aplicações híbridas para o cluster desenvolvido, foram feitas algumas adaptações, a fim de permitir que ferramentas

automatizadas já existentes no mercado possam trabalhar e gerar suas indicações para melhoria do sistema. Os autores escolheram utilizar um programa, chamado *Vampir* (KNÜPFER, 2008), que realiza análise semelhante em ambientes com coprocessamento em *GPU*. Para que esse programa pudesse interagir com o *FPGA*, foi preciso estender uma *API CUDA* para o dispositivo reconfigurável, provendo assim que informações de desempenho fornecidas pela placa pudessem ser analisadas pela ferramenta. A partir dos resultados obtidos foi possível identificar gargalos de regiões ineficientes de memória, aumentando assim a eficiência das transferências de dados.

A *API* que o usuário poderá utilizar para desenvolver suas soluções massivas é o *OpenCL*, um *framework* largamente utilizado para desenvolvimento de programas em plataformas heterogêneas, como *CPUs*, *GPUs* e *FPGAs*. A implementação desse protocolo em *FPGA* foi baseada no princípio de que o *FPGA* é um dispositivo *OpenCL*, com os objetos do *framework* simplesmente mapeados para o *hardware*. A Figura 19 mostra de forma ilustrada como foi feito o mapeamento desses recursos da *API*.

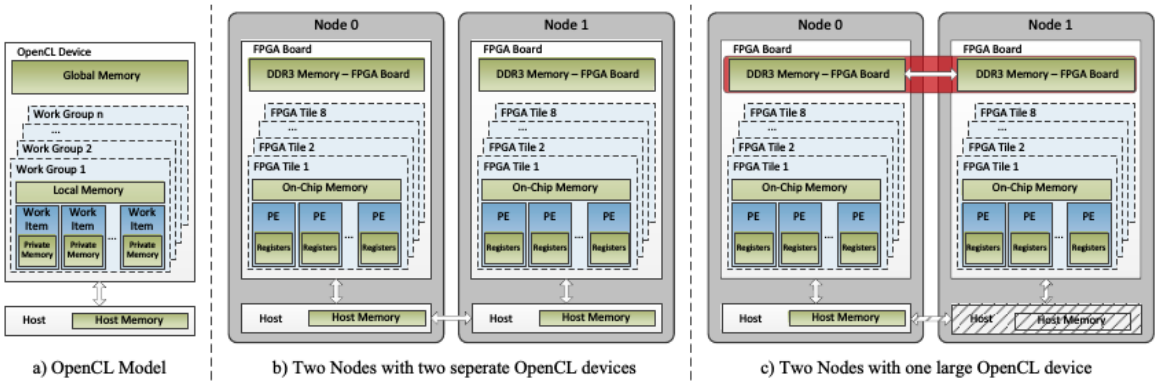

Figura 19: Mapeamento de um OpenCL Device para o FPGA (KNODEL, 2013).

A Figura 19 mostra como é realizado o mapeamento direto das estruturas *OpenCL* para o dispositivo reconfigurável. A Figura 19a apresenta uma estrutura genérica de um dispositivo *OpenCL*, composta por grupos contendo pequenas estruturas de processamento, as quais possuem elementos de memória local e compartilhada. A implementação em *FPGA*, mostrada na Figura 19b realiza uma equivalência direta dos elementos, onde cada grupo é implementado em uma região de *tile*, e cada estrutura de processamento é implementada por um *PE*, cujo algoritmo é gerado a partir de código *VHDL* ou *Verilog*, cujos elementos de memória internos são atribuídos a registradores e as memórias compartilhadas são representadas pelos módulos de memória incluídos no dispositivo reconfigurável. Na Figura 19c é mostrada que as regiões de memória de cada modelo *OpenCL* implementado em *FPGA* também podem ser compartilhadas entre placas, utilizando os recursos de conexão fornecidos

pela estrutura desenvolvida, formando então um dispositivo *OpenCL* distribuído, composto por duas placas participantes de uma mesma ilha.

Esta estrutura do *cluster* permite um processamento híbrido, tanto entre *host* e *FPGA* quanto entre *FPGAs*, contando com um barramento dedicado de comunicação bastante rápido e seguro. A metodologia de adaptação para um *cluster* comum, baseado em ilhas de nós híbridos, facilitou a implantação física dessa estrutura com conexão interplacas. A comunicação entre os *hosts* também passa pelo dispositivo reconfigurável, sendo controlado por regiões específicas do coprocessador. Outros recursos, como módulos de memória, controladores para memória adicional, barramentos opcionais de transmissão, estruturas de depuração e áreas reservadas para configuração do usuário, contribuem para que a estrutura interna do *FPGA* tenha um ambiente com padrão de configuração e facilidade no gerenciamento de recursos. A transparência para o usuário foi resolvida em duas frentes. Com relação ao gerenciamento dos recursos de *cluster*, foram desenvolvidas estruturas de *hardware* e *software* que são compatíveis com os modelos existentes no mercado, e permitem ao usuário gerenciar as tarefas que estão sendo executadas no sistema, otimizando os recursos, inclusive da placa *PCIe*. Já no desenvolvimento de programas, a implementação dos modelos *OpenCL* e *OpenMP* permitem ao usuário utilizar bibliotecas já consolidadas e bem documentadas, resultando numa menor curva de aprendizado.

Apesar de possuir uma arquitetura interna com vários recursos, a perda de flexibilidade de implementação influencia no desenvolvimento dos algoritmos. Aplicações híbridas que possuem partes essencialmente sequenciais e com transferência de dados nos trechos executados no PC, podem perder desempenho com o maior *overhead* de transmissão, devido a obrigatoriamente seus dados passarem pelo *FPGA*. Além disso, as áreas reservadas dos algoritmos desenvolvidos pelo usuário restringem o tamanho das implementações, forçando projetos já consolidados a repensar em como adaptar seu algoritmo a essa nova estrutura. O uso de *OpenCL* pode ajudar nessa adaptação, mas é preciso analisar se a possível perda de desempenho, com relação ao projeto original, será satisfatória. O aumento de complexidade na estrutura do *core services* é também um fator a se considerar, pois acarreta em maior tempo de desenvolvimento da integração *host-algoritmo* e, consequentemente, do cluster híbrido.

3.1 Conclusões

Os trabalhos apresentados tinham objetivos semelhantes, tais como: a criação de um ambiente com transparência para o usuário; padronização na integração entre componentes de *hardware* e *software*, reduzindo o *gap* no desempenho de artefatos tanto da parte do host quanto da parte do *FPGA*, visando também diminuir *overheads* de comunicação.

No trabalho de *Inta et al.* (INTA, 2012) permitiu-se a utilização de rotinas híbridas prontas para usuários a partir do *MATLAB*. O *cluster* interno, formado por *CPU, FPG*A e *GPU*, permitiu às aplicações extraírem as características positivas de cada elemento de processamento, mas a escalabilidade do sistema é fortemente atrelada à disponibilidade do barramento *PCI* do *host*. Não há um suporte para comunicação direta entre os dispositivos internos, o que acarreta um maior *overhead* na troca de dados entre *GPU* e *FPGA* ou vice versa. Há também um limite implícito da quantidade de dados que o sistema é capaz de processar massivamente, devido à restrição do número de elementos aceleradores.

A segunda arquitetura proposta, *Espenshade et al* (ESPENSHADE, 2009), introduziu o conceito de um nó abstrato, onde sua implementação é dividida em duas estruturas, uma de *software*, a qual é representada por um processo executado na *CPU* embarcada, e outra de *hardware*, chamada de *HAU*, como sendo uma implementação do conjunto "algoritmo + 2 *FIFOs*" em um *FPGA*. Esse encapsulamento de um elemento do *cluster* permitiu que mais de uma instância do mesmo pudesse ser implantada numa mesma estrutura física, cuja placa precisa atender a um conjunto rígido de requisitos. Um destes requisitos é a existência de um *core PowerPC* embarcado na *FPGA*, cuja função é executar um sistema operacional embarcado, capaz de controlar os dispositivos necessários para acesso à memória e comunicação entre outras placas, além de gerenciar as estruturas de *software* de cada participante do *cluster*.

Comparado com o *Chimera*, o ambiente de programação para o usuário demonstra uma dificuldade de aprendizagem um pouco maior, pois além de necessitar saber a funcionalidade de uma função, que não pertence a um padrão de mercado, também é necessário definir a distribuição dos dados e de processamento entre os membros do sistema. Com relação ao desenvolvimento das estruturas do *cluster*, as dificuldades de embarcar um sistema operacional otimizado para um processador pouco comum e as limitações de desenvolvimento, tanto do limite de entradas e saídas, quanto do tamanho da lógica reservada aos algoritmos integrados à *HAU*, quanto de restringir essa construção para placas específicas da *Xilinx*, afeta a portabilidade do *framework* proposto.

O trabalho de *Knodel et al.* (KNODEL, 2013) tinha o objetivo de abranger um conjunto maior de problemas, com uma arquitetura que permite tanto transferência de dados entre o *host* e o algoritmo, como comunicação dedicada entre algoritmos localizados em placas vizinhas. O arranjo de nós em formato ilha foi uma boa maneira de desenvolver um ambiente de comunicação *inter-FPGA* e *FPGA-host*. Além desses módulos, outras implementações, como estruturas de depuração de *hardware*, módulos controladores de rede, de barramentos *SATA* e *SAS* e recursos internos para implementação de memória compartilhada entre os membros da ilha, contribuíram para uma rica estrutura de *core services*, onde os algoritmos, localizados em locais definidos e estratégicos, podem usufruir dos recursos de *hardware*, com pouca ou nenhuma interferência do *host*. O ambiente de programação baseado em *OpenCL* também proporcionou uma transparência para o usuário, embora um pouco mais complexo do que as outras duas soluções anteriores, mas com uma fidelidade maior à *API* consolidada, ou seja, a implementação das funções seguem o mesmo padrão de estrutura em outros dispositivos, como *GPUs*. No entanto, o desenvolvimento desse *cluster* possui uma alta complexidade, desde a adaptação de um grande cluster para o formato de ilhas, passando pela comunicação entre *hosts* ser realizada em parte pelo FPGA, até o desenvolvimento de todos os recursos de controle, gerenciamento, e infra-estrutura do *core services* para o algoritmo.

A arquitetura posposta neste trabalho tem o objetivo principal de definir uma metodologia de desenvolvimento de uma aplicação para um cluster híbrido, provendo transparência e uma padronização da integração *hardware-software*. O arranjo dos nós híbridos irá seguir uma arquitetura mais simples, onde todos os membros podem transmitir dados entre si ao mesmo tempo. As ferramentas de desenvolvimento, *APIs*, estrutura de *core services* e *drivers* de reconhecimento da placa com *FPGA*, são disponibilizados pelo fabricante da placa. O Sistema Operacional utilizado será um padrão de *Desktop*, adicionando apenas as *APIs* necessárias para desenvolvimento, tanto de hardware quanto do paralelismo de software, com implementações do *OpenMP* e do *MPI*.

A *API* de integração *hardware-software* foi definida por um dos fabricantes da placa. Portanto, possui funções que não pertencem às bibliotecas padrão, adicionando então uma curva de aprendizado para esse conjunto de chamadas de sistema. E finalmente, a estrutura de *core services* implementada também pelo fabricante, é baseada em *FIFOs* e possui uma flexibilidade maior do que os trabalhos apresentados, podendo ser definida uma quantidade maior de interfaces para leitura e escrita de dados, além de prover configuração de certos registradores para receber sinais de controle diretamente da aplicação do *host*.

De forma a comparar e posicionar o trabalho proposto nesta dissertação com os trabalhos apresentados neste capítulo, uma tabela comparativa foi criada, com as principais funcionalidades encontradas nos projetos envolvidos. Os trabalhos relacionados e o proposto estão destacados na primeira coluna. Nas outras colunas estão quatro características que foram escolhidas por estarem presentes em maior ou menor grau em todos os trabalhos. A primeira diz respeito à implementação, ou seja, se os elementos de *hardware* e *software* necessários para implementação do ambiente de desenvolvimento e execução das aplicações do usuário, são mais simples ou mais complexos de serem integrados. A segunda funcionalidade destacada é quanto a sua utilização, se o sistema como um todo possui recursos os quais o usuário terá maior facilidade para executar seus programas, desenvolver, ou ambos. A terceira retrata o quanto o sistema pode ser escalável, com relação a seus elementos de processamento, sejam eles físicos ou lógicos. E a quarta característica verifica o quanto o algoritmo desenvolvido em hardware pode ser flexível quanto a sua construção, ou seja, quanto menos restrições quanto a tamanho, limitação de interfaces com o meio externo ou de arquitetura, melhor. Foi feita uma análise de cada projeto e definido um critério de comparação utilizando estrelas, onde quanto mais estrelas a característica de um projeto possui, melhor essa característica. A Tabela 4 resume a análise feita, de acordo com as características já apresentadas de cada trabalho.

Tabela 4: Comparativo entre os trabalhos relacionados.

Projetos	Implementação	Utilização	Escalabilidade	Flexibilidade do algoritmo
Chimera	★ ★	★ ★ ★	★	★ ★ ★
HAU	★	★ ★	★ ★ ★	★
Inter-FPGA	★	★ ★	★ ★	★ ★
Proposto	★ ★ ★	★ ★	★ ★ ★	★ ★ ★

Com relação à Tabela 4, podemos observar que o projeto *Chimera* possui uma boa implementação, pois seus componentes são integrados de maneira simples, necessitando apenas desenvolver o *driver* para a placa com *FPGA* e as funções híbridas do *MATLAB*. Sua utilização é considerada excelente, pois utiliza um ambiente bastante conhecido por grande parte dos usuários. Ele também possui baixa escalabilidade, pois seu cluster interno depende

da quantidade de *slots* da placa mãe. Por último, o algoritmo implementado tanto na *FPGA* quanto na *GPU* possui grande flexibilidade de arquitetura, pois não há descrição de restrições para desenvolvimento do mesmo.

Para o trabalho envolvendo o desenvolvimento de um *HAU*, sua implementação possui diversas restrições de hardware, como modelo de *FPGA* e dispositivos auxiliares para a placa *PCIe*, além do desenvolvimento de um ambiente embarcado, com uma infraestrutura de software bastante específica. Sua utilização é boa, bastando apenas conhecer poucas funções de manipulação do algoritmo. Possui também uma ótima escalabilidade, pois mais de um nó pode ser implantado numa mesma placa, mas sua flexibilidade é bastante prejudicada devido à restrição de tamanho do algoritmo e por permitir apenas duas interfaces de dados.

Com relação ao trabalho com comunicação Inter-*FPGA*, temos uma alta dificuldade de implementação, devido à necessidade de desenvolvimento de uma infraestrutura complexa de *hardware* e *software*. Possui uma boa utilização, com a adoção de ferramentas de gerenciamento de recursos e do uso de *OpenCL* para desenvolvimento de programas. Sua formação baseada em ilhas de clusters contribui para uma boa escalabilidade, apesar de não haver comunicação direta entre os nós. E sua flexibilidade é boa, pois há apenas uma restrição de tamanho e posicionamento do algoritmo a ser desenvolvido.

Já para o projeto proposto nesta dissertação, a implementação é considerada ótima devido a sua simplicidade de construção do cluster de uso de ferramentas fornecidas pelo fabricante da placa. Possui boa utilização, através de uma *API* de gerenciamento de *hardware* específica, mas bastante documentada. Sua escalabilidade é considerada ótima pela simplicidade da topologia dos nós, e o algoritmo possui alta flexibilidade, com uma estrutura de *core services* que permite a configuração de interfaces simples de dados e não possuir restrição explícita do tamanho de sua arquitetura. Mais detalhes sobre a arquitetura e a metodologia de desenvolvimento para a mesma estão presentes no próximo capítulo.

4. Arquitetura Proposta

Neste capítulo serão descritos os componentes de *hardware* e *software* utilizados neste projeto, e suas importâncias para a construção do *cluster* híbrido. Também é descrito o fluxo geral de execução a qual uma aplicação distribuída deverá seguir na estrutura implantada. Além disso, também será mostrada a metodologia de desenvolvimento para que uma aplicação possa executar de forma otimizada no sistema, aproveitando ao máximo os recursos de processamento paralelo que o *cluster* oferece.

4.1 Visão Geral

O *cluster* híbrido proposto neste trabalho é composto por nós formados por um *PC* convencional e um coprocessador contendo uma ou mais *FPGAs*. Os nós possuem uma arquitetura de conexão do tipo *ethernet*, a qual permite aos seus nós uma comunicação direta entre si. O controle do fluxo de operações é centralizado e atua apenas na execução distribuída do *software*, de forma a aumentar a eficiência da distribuição das instruções e dados. O protocolo de comunicação escolhido entre os nós do *cluster* foi o *MPI*, por ser um padrão em comunicação paralela em *clusters*, além de permitir, de forma bastante flexível, configurar a execução de programas em memória distribuída. Com isso, a implantação de um novo cluster ou a adaptação de um já consolidado torna-se bastante simples, devido ao fato de, em nível de estrutura distribuída, o projeto trabalhar com ferramentas já utilizadas em larga escala.

Cada nó do *cluster* possui uma arquitetura híbrida bem definida, composta por um *PC* (*host*) e uma placa com *FPGAs* e memória local, como dispositivos coprocessadores, especializada em acelerar os trechos que requeiram processamento massivo e paralelo de dados. O barramento escolhido para comunicação entre estes componentes foi o *PCIe*, versão 2.0, devido à boa disponibilidade no mercado de placas com o dispositivo reconfigurável e com essa interface de conexão.

O *PC* será responsável pela execução dos elementos de *software* da aplicação distribuída e pela comunicação com o *cluster*, necessárias para integração do nó com o sistema. Além disso, ele também será responsável pelo controle e gerenciamento de dados e instruções que serão executadas pela placa aceleradora.

O fluxo de desenvolvimento de uma aplicação, a fim de que a mesma possa aproveitar todos os recursos desse novo cluster, requer uma sequencia bem definida de etapas, pois a complexidade inerente à estrutura de distribuição, gerenciamento nodal e execução massiva em hardware, requer cuidados específicos. Um dos desafios é garantir a corretude da execução, com relação aos resultados obtidos pela aplicação original. Também é necessária a otimização de instruções independentes, para atender o objetivo de aceleração das aplicações. As etapas de desenvolvimento de uma aplicação híbrida distribuída visam, entre outras, o aumento de eficiência nos trechos que requeiram processamento massivo de dados, e manutenção da transparência de comportamento da aplicação, com relação à aplicação original, para os usuários das mesmas.

O desenvolvimento do algoritmo a ser executado em *FPGA* é uma etapa bastante sensível do projeto, pois é onde se concentra a maior carga de trabalho. Depois de definidos os trechos a serem desenvolvidos em *hardware*, é preciso projetar uma estrutura de processamento e controle capaz de proporcionar o máximo de paralelismo possível, considerando todos os recursos disponíveis da placa aceleradora. Para que seja possível uma alta quantidade de elementos de processamento, consideram-se tanto as características de execução do trecho de software escolhido, como o tipo de dados operados por eles, até chegar-se à análise da estrutura de elementos de memória disponíveis na placa aceleradora. Para este último aspecto, a placa selecionada precisa atender a certos requisitos, como alta disponibilidade de canais para escrita e leitura dos dados, suporte a transferência de dados em rajada ou *burst*, e protocolo de comunicação simples para o algoritmo desenvolvido.

Apesar do desenvolvimento desse algoritmo ainda ter certo grau de dificuldade, uma estrutura que provê um acesso simples e com alta multiplicidade de canais para os módulos de memória diminui bastante o trabalho de desenvolver interface para gerenciamento dos dados.

A linguagem usada para esse desenvolvimento de módulos de *hardware*, nesta versão do *cluster* é a linguagem *Verilog*, por ser uma *HDL* compatível com desenvolvimento em *FPGA* e com a qual o autor possui familiaridade. O fato da escolha por *Verilog*, em detrimento de outras linguagens da mesma família, como *VHDL*, não influencia no desempenho dos algoritmos desenvolvidos. As ferramentas de desenvolvimento em *hardware* são todas fornecidas pelos fabricantes da placa e do *FPGA*, cuja finalidade é de prover a integração e implantação do algoritmo de forma compatível com a placa escolhida.

4.2 Estruturas de *Hardware* e *Software*

O *cluster* híbrido proposto é formado atualmente por três *workstations*, sendo então considerado um *workstation cluster*. Cada nó é um modelo *Supermicro X8DTG-QF* (X8DTG-QF, 2014), com uma placa mãe de modelo com mesmo nome, a qual sua carcaça possui um padrão de *rack* 4U e comportam dois processadores *Intel Xeon E5645* (E5645, 2014). Cada *CPU*, por sua vez, possui seis núcleos x86, capazes de processar até 12 *threads* simultaneamente. O *clock* pode atingir uma frequência entre 2.4 até 2.67 GHz caso esteja ativo o modo *turbo boost*. O processador também possui uma memória *cache* de 12 MB, o que auxilia bastante no armazenamento maior de instruções e dados a serem reutilizados. A tecnologia do processador é de 32 nm, com consumo máximo *TDP* de 80 W. A placa mãe também conta com dois *chipsets* Intel 5520 que realizam comunicação com cada processador, de forma a gerenciar os periféricos disponíveis. A placa inclui ainda duas placas *ethernet* com *chipset* Intel 82574L *Gigabit ethernet*, e sete *slots PCIe*, dos quais quatro são do tipo x16 e três do tipo x4, além de um *HD* de 2TB e 48 GB de memória *RAM ECC*.

Para comunicação entre os nós do *cluster*, foi escolhida a rede *ethernet*, pois é o padrão disponível no laboratório HPCIn, onde estão localizadas as *workstations*.

Essa rede pode atingir uma banda passante de até 1 Gbps. Os nós estão conectados à rede por meio de um *switch*, o qual permite que cada participante possa comunicar-se com qualquer outro integrante do *cluster*. Essa estrutura de conexão está ilustrada na Figura 20.

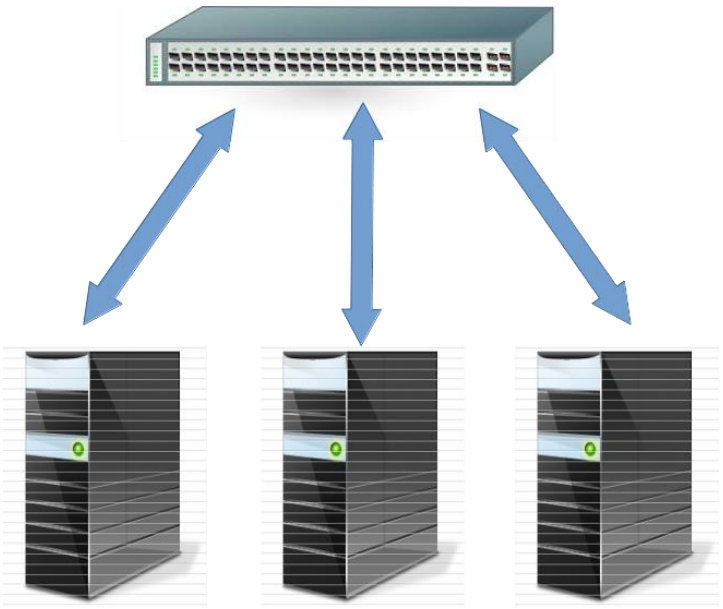

Figura 20: Conexão do cluster à rede Gigabit Ethernet.

Com relação aos *softwares* instalados em cada *workstation*, foi escolhido o *Debian* (DEBIAN, 2014), versão 7.0, um sistema operacional baseado em *kernel Linux*, o qual permite uma ótima configuração dos componentes de *software* nele instalados, graças a sua filosofia de código aberto. Além disso, o *Debian* possui repositórios de *software* contendo milhares de programas, provendo segurança de instalação e manutenção centralizada das atualizações, funcionalidades que facilitam bastante para o desenvolvedor configurar um ambiente de desenvolvimento. Para que as máquinas pudessem realizar comunicação via *MPI* foi escolhida e instalada a implementação *OpenMPI*, visto que é uma das implementações bastante utilizadas no meio científico, com uma comunidade de desenvolvedores ativa e vasta documentação.

Um dos nós foi escolhido para ser a máquina com finalidade desenvolvedora da aplicação híbrida distribuída. Neste nó foram instaladas alguns programas especiais como o compilador *gcc*, capaz de compilar programas em C/C++ e outras linguagens, além de possuir suporte ao protocolo *OpenMP*, necessário para prover o paralelismo do software local a nível de *thread*. A *IDE* escolhida foi o Eclipse com extensão *CDT* (ECLIPSE, 2014), cuja interface é de fácil utilização, além de possuir extensões que incrementam funcionalidades e auxiliam no desenvolvimento dos artefatos de *software*. Para reconhecimento da placa com *FPGA*, foi instalado o *driver* fornecido pelo fabricante, o qual provê também uma *API* de chamadas de sistema que permitem o controle e transferência de dados e sinais de controle para o algoritmo desenvolvido. Nesse nó foi preciso também instalar o sistema operacional *Windows*, pois a ferramenta de integração *hardware-software* do fabricante (cujo detalhamento de suas características será apresentado posteriormente) possui suporte apenas nesse sistema. Além dessa ferramenta, também foi instalado o *Altera Quartus II* (QUARTUS, 2014), versão 13.0 sp1, o qual permite o desenvolvimento dos algoritmos a serem implantados no *FPGA*. O *Quartus* também é considerada uma *IDE*, com editor de código *HDL*, configurações de otimização e síntese do projeto, e geração de um arquivo *bitstream* para configuração dos elementos básicos de processamento no *FPGA*. Para simulação e verificação do comportamento do algoritmo em *hardware*, foi instalada a ferramenta *Modelsim* (MODELSIM, 2014), capaz de realizar simulações funcionais e com *timing*, com o objetivo de verificar a corretude do funcionamento dos projetos em *HDL*.

O *cluster* está configurado para prover um ambiente de execução distribuída, no qual seus nós podem realizar comunicação paralela de seus dados e instruções a nível global, e de acelerar a execução local através de nós híbridos. Cada nó suporta paralelismo em nível de *thread*, onde seus processadores podem prover execução simultânea de 24 delas, e em nível

de *hardware*, graças a seu coprocessador baseado em *FPGA*. Além disso, o cluster pode ser usado como ferramenta de desenvolvimento para adaptar as aplicações de forma a aproveitar da melhor maneira possível seus recursos.

4.3 Organização do processamento no cluster

Como um *cluster* de alto desempenho, os participantes precisam estar configurados para executar as aplicações híbridas e distribuídas com maior eficiência possível. Devido à configuração dos nós e a necessidade do sistema ser escalável, então a configuração em estrela é a mais adequada. Logo, os integrantes podem realizar comunicação ponto-a-ponto, coletiva, ou ambas, dependendo da natureza das aplicações a serem executadas.

O paradigma de controle escolhido para o *cluster* foi o mestre-escravo, suportado pelo *MPI*, e de fácil implementação. Nesta arquitetura o mestre é responsável pelo controle do fluxo principal da aplicação distribuída e gerenciamento dos dados entre os nós. Uma de suas principais tarefas é de inicializar o processamento distribuído, dando início à execução dele mesmo e dos outros processos escravos. Depois de iniciar a execução, o mestre deve distribuir os dados e parâmetros a serem processados localmente, inclusive para ele próprio, de acordo com a estratégia a qual o programador considera melhor para seu algoritmo no nó. Então, o mestre processará seu trecho de instruções locais, e após seu término, aguardará o final da execução local dos outros processos. Após todos os escravos terminarem suas execuções locais, os mesmos geram, cada um, um conjunto de dados, o qual é resultado parcial da aplicação, e os envia para o nó mestre, que deverá então concatená-los com o seu próprio resultado gerado, formando a saída completa da aplicação.

Devido à natureza da proposta do projeto, ficou definido que cada nó deverá processar uma parte igual do trecho de processamento massivo de dados da aplicação, sendo que não deverão ter outros algoritmos embutidos no *FPGA*. Logo, cada nó executará um processo, o qual poderá ser escravo ou mestre. O nó escolhido para ser o mestre será aquele que possuir as configurações adicionais, como as ferramentas de desenvolvimento.

O fluxo de execução distribuída pode ser visto na Figura 21. Nesta figura podemos verificar de forma resumida como se comporta a aplicação no cluster proposto.

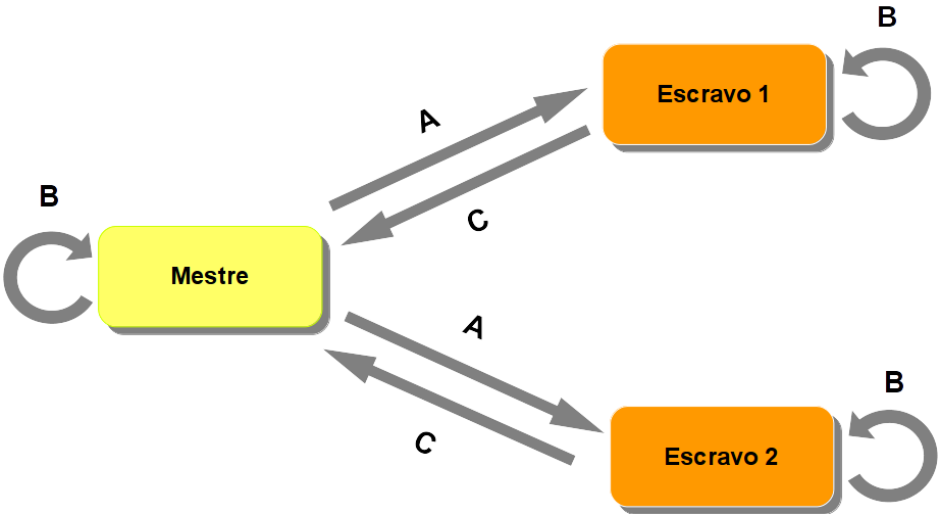

Figura 21: Fluxo de dados da aplicação distribuída.

Como podemos ver na Figura 21, a seta A representa a etapa de divisão da grande massa de dados e distribuição de suas partes entre os escravos, com parte ficando também com o emissor, ou seja, o mestre. A seta B indica o processamento local de cada parte, e a C representa envio dos resultados parciais para que o mestre possa formar a saída do sistema.

O processo escravo possui as funções de receber os dados parciais vindos do mestre, executar o processamento local e enviar os dados processados de volta ao mestre. O envio dos dados pelo escravo pode ser feito por comunicação ponto-a-ponto ou coletiva, dependendo da natureza da aplicação.

O processamento local, o qual também está presente no mestre, é o principal responsável pelo ganho de desempenho paralelo do *cluster*, pois é responsável pelo gerenciamento do processamento massivo, o qual é realizado pelo algoritmo embutido na placa aceleradora. Com a finalidade de realizar esse gerenciamento, os trechos executados em cada nó possuem instruções especiais, as quais fazem chamadas de sistema para que a placa possa realizar corretamente sua execução em *hardware*. O fluxo do processo escravo pode ser visto na Figura 22.

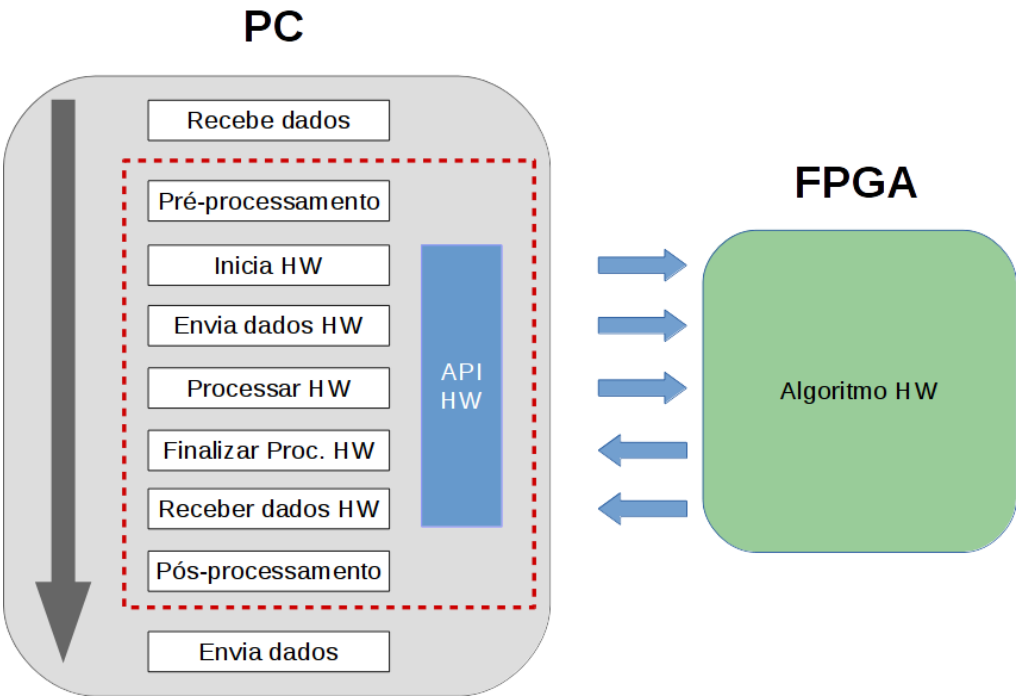

Figura 22: Fluxo de execução do processo escravo.

Verificamos na Figura 22 que as instruções do processo escravo seguem um fluxo bem definido, com o destaque em pontilhado indicando o conjunto de instruções que fazem parte do processamento local. As instruções locais que terminam em "HW" são as responsáveis pelo gerenciamento da placa aceleradora, e utilizam funções específicas da *API* do *hardware*, disponibilizadas pelo fabricante da placa. O fluxo de processamento local começa após a recepção dos dados provenientes do mestre, os quais podem ser compostos por parâmetros de execução em *hardware* e por dados de entrada.

Inicialmente é feito um pré-processamento da entrada, que é uma formatação do conjunto de dados a fim de que sejam compatíveis com o que o *hardware* espera, ou seja, caso o algoritmo trabalhe com um fluxo de dados específico e diferente do utilizado em *software*. Então é feita a inicialização da placa com *FPGA*, a qual passa a ter representação no *software* através de um ponteiro, geralmente uma instância de classe em C/C++. Essa inicialização pode ser configurada a partir dos parâmetros recebidos. Em seguida são executadas as instruções de carregamento dos parâmetros necessários para execução do algoritmo, os quais normalmente representam endereços da memória dedicada da placa em que serão carregados os dados, ou informações de processamento dedicadas à arquitetura do algoritmo implementado em *hardware*.

Após o envio dos parâmetros, os dados de entrada são enviados para os endereços da memória dedicada na placa *PCIe*. O envio de dados utiliza funções que recorrem às chamadas de *DMA (Direct Memory Access)*, como forma de aumentar a eficiência na transferência dos

dados para a memória. Após o envio dos dados de entrada, o *software* sinaliza para o algoritmo em *hardware* iniciar o processamento. Ao final do processamento, uma *flag* de finalização, ativado pelo módulo em *hardware*, informa ao componente de *software* que os resultados estão prontos. Os resultados são em geral armazenados nos bancos de memória existentes na placa aceleradora. Assim, após a sinalização de finalização de processamento, o componente de *software* executa o processo de captura dos dados processados na memória da placa aceleradora, através de chamadas de *DMA*. Em seguida é realizado o pós-processamento desses dados, cuja funcionalidade é inversa ao pré-processamento, ou seja, converte-se os dados processados pelo *hardware* para uma saída compatível com a aplicação, sendo este também o último trecho de execução do processamento local. Ao final do fluxo de processamento local, cada nó escravo envia os resultados parciais para o processo mestre.

O processamento local deve ser feito com o máximo de eficiência paralela possível, deixando qualquer mudança de execução interna transparente para o usuário. Em outras palavras, o processamento local possui uma transparência de comportamento, na qual os trechos executados em *hardware* possuem funcionalidades gerais equivalentes com os trechos originais processados na *CPU*. Essa transparência é obtida com o programador da função substituindo o conteúdo das instruções em *software* por chamadas da *API* de *hardware*, mas mantendo sua assinatura. Assim, o usuário poderá usar as funções da mesma maneira a qual já estava acostumado. A Figura 23 ilustra como seria essa transparência aplicada numa função genérica, com uma *API* também genérica.

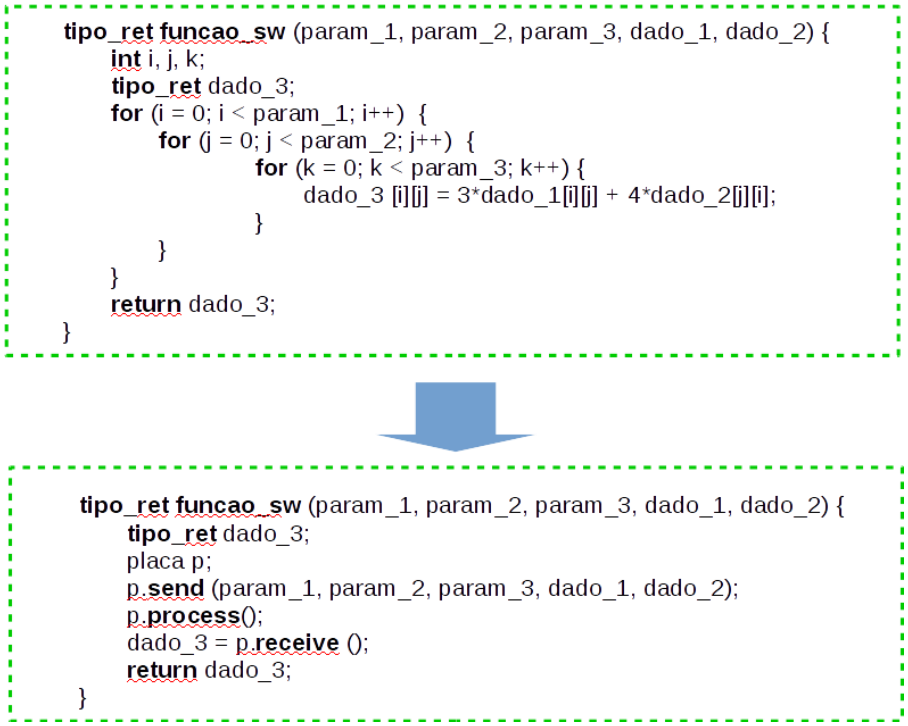

Figura 23: Aplicação de transparência numa função, com chamadas da *API* de *hardware*.

De acordo com o código exposto na Figura 23, é possível perceber que uma função, inicialmente composta por trechos complexos e com alto custo computacional, quando adaptada para a execução na plataforma híbrida torna-se mais simples, contendo apenas instruções de gerenciamento do algoritmo em *FPGA*, o qual fica agora encarregado de processar todas as ações complexas. Geralmente os trechos complexos são compostos por *loops*, onde ocorrem varreduras aninhadas dos conjuntos de dados. Eles são bastante custosos e com alto potencial de paralelismo, além de possuírem um grande reuso de dados. Na parte mais interna dos *loops*, ocorre a operação principal, onde é processado cada elemento de entrada a fim de preencher o vetor de dados de saída.

O desenvolvimento do algoritmo em *FPGA* é específico para cada aplicação, a fim de otimizar ao máximo a execução massiva da mesma, considerando os recursos de *hardware* disponíveis na placa utilizada. Logo, é preciso identificar como extrair paralelismo dos trechos massivos. *Loops* limitados, como os formados pelo *for*, podem ter certa independência de dados para o processamento mais interno, permitindo que várias iterações possam ser feitas simultaneamente. Para tanto, geralmente é desenvolvido uma arquitetura de controle que fornece continuamente os dados para os *PEs*, que por sua vez implementam a operação interna dos *loops* de *software*. Essa arquitetura possui interface com a memória dedicada da placa, tanto para ler como para escrever dados. Para prover fornecimento contínuo, geralmente essa interface trabalha com frequências de relógio (*clock*) mais baixas que a memória, pois o protocolo de acesso à memória não suporta nativamente transferências em rajada, e cada nova informação (dados) demora certa quantidade de ciclos para estar disponível. A arquitetura também possui comunicação direta com o *software* através de regiões de memória mapeada, a qual permite a passagem de parâmetros e sinais de controle entre o *PC* e o dispositivo reconfigurável, funcionando então como registradores.

Essas e outras interfaces implementadas para acesso aos outros recursos da placa formam o chamado *core services*, cuja finalidade é prover acesso a esses recursos com protocolos de gerenciamento mais simples do que os originais.

4.4 Kit de desenvolvimento da placa aceleradora

A fabricante escolhida como fornecedora das placas para este projeto foi a *Gidel* (GIDEL, 2014), cujas placas possuem características bem semelhantes, de forma a serem compatíveis com sua ferramenta de geração automática de código. Essa ferramenta é o

ProcWizard (PROCWIZARD, 2014), a qual permite a integração do algoritmo a ser processado no *FPGA*, aos recursos existentes na placa e ao *software* no *host*.

As placas com *FPGAs* possuem cominicação com o *PC* através de uma interface *PCIe*. Elas possuem também alguns periféricos que podem ser acessados pelo algoritmo via interfaces bem definidas. Um dos periféricos mais importantes são as memórias dedicadas padrão *DDR2*, as quais podem ser um *chip* soldado na placa, com capacidade de 512 Mb, ou módulos padrão *notebook*, encaixados em *slots*, com capacidades dependentes da especificação de cada placa.

A interface utilizada para essas memórias é o *ProcMultiport* (PROCMULTIPORT, 2014), um *IP-core* que provê um protocolo simples de acesso aos dados na memória. Esse protocolo pode ser configurado para que os dados sejam escritos ou lidos de forma randômica ou em rajada, sendo a segunda forma mais interessante devido à necessidade do algoritmo realizar processamento contínuo. Cada módulo de memória possui seu próprio *multiport*, possibilitando assim o uso simultâneo de todos os módulos e, por conseguinte aumentando a quantidade de canais de acesso aos dados.

Esses canais de acesso às memórias são baseados em *FIFOs*, as quais possuem um protocolo específico de acesso para leitura ou escrita, e podem ser instanciadas pela quantidade máxima de dezesseis vezes, ou seja, cada módulo de memória pode ter até dezesseis *FIFOs* gerenciadas pelo seu *multiport*. A Figura 24 mostra a estrutura do *multiport*.

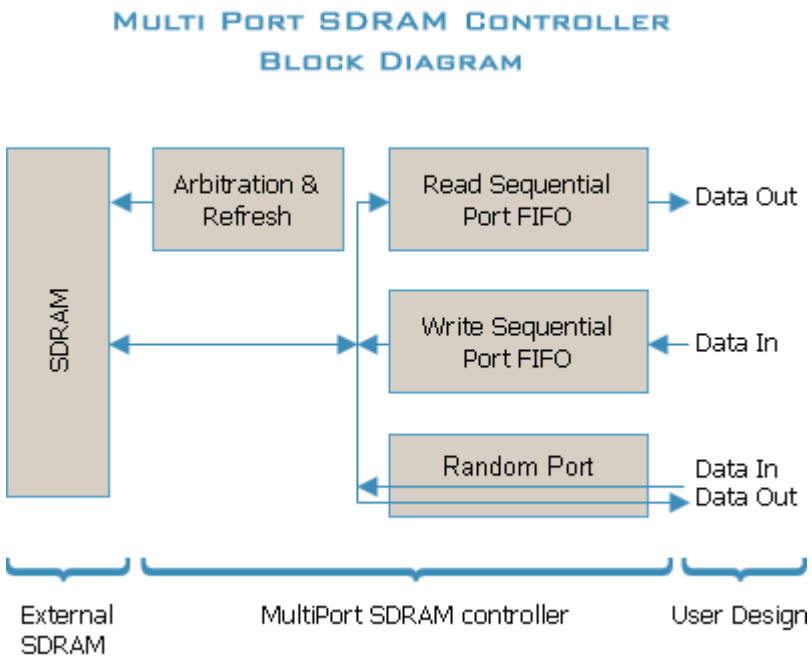

Figura 24: Diagrama de blocos do *Multiport* (PROCMULTIPORT, 2014).

Na figura 24 percebemos que o *multiport* suporta *FIFOs* de leitura, escrita, e de acesso randômico, sendo esta última opção preterida pelos motivos já explicados. As *FIFOs* de leitura e de escrita possuem protocolos que permitem ao algoritmo do usuário controlar a inicialização e o fornecimento em rajada dos dados.

Os principais sinais para as *FIFOs* de leitura são:

- ***start:*** sinal de saída, cuja funcionalidade é inicializar a *FIFO*, fazendo ela apontar para o endereço indicado no *addr*. Ambos os sinais precisam ser mantidos por dois ciclos de relógio para garantir a correta inicialização.
- ***select:*** sinal de saída que, quando ativado, habilita a rajada de dados para serem lidos pelo algoritmo. Para parar a rajada, basta colocar este sinal para nível baixo.
- ***almost_empty:*** sinal de entrada indicador de que a FIFO está com uma quantidade abaixo do limiar especificado para ela, e esse número mínimo depende da placa escolhida, correspondendo a 1/8 do número de elementos da *FIFO* suportada pela mesma.
- ***data_in [N:0]:*** sinal de entrada que representa o dado recebido. É um vetor de *bits* cujo tamanho N pode chegar a 256 *bits*.
- ***addr [N:0]:*** sinal de saída correspondente ao endereço inicial de leitura da memória reservada para a *FIFO*, cujo N pode chegar a 32 *bits* de tamanho.

Os principais sinais das *FIFOs* de escrita são:

- ***start:*** sinal de saída, cuja funcionalidade e característica é semelhante ao de leitura.
- ***select:*** sinal de saída que, quando ativado, habilita a rajada de dados, fornecidos pelo algoritmo, a serem escritos. Para parar a rajada, basta colocar este sinal para nível baixo.
- ***almost_full:*** sinal de entrada indicador de que a *FIFO* de escrita está com uma quantidade acima do limiar especificado para ela, e esse número máximo depende da placa escolhida, correspondendo a 7/8 do número de elementos da *FIFO* suportada pela mesma.
- ***empty:*** sinal de entrada, o qual indica que a *FIFO* está vazia.
- ***data_out [N:0]:*** sinal de saída que representa o dado enviado. É um vetor de *bits* cujo tamanho N pode chegar a 256 *bits*.
- ***addr [N:0]:*** sinal de saída correspondente ao endereço inicial de escrita da memória reservada para a *FIFO*, cujo N pode indicar até 32 *bits*.

A geração automática de código provida pelo *ProcWizard* consiste em produzir as interfaces essenciais para a integração das partes em *software* e *hardware* da aplicação híbrida. Para tanto, o programa oferece uma *GUI*, a qual possibilita ao usuário configurar o projeto a ser embarcado na placa, instanciando o módulo *top* de seu algoritmo, as memórias, através de seus *multiports*, com o número e nome das *FIFOs* desejadas, e os registradores de controle e parâmetros da aplicação. Após o término da configuração do projeto, a ferramenta está apta a gerar tanto a parte de *hardware* quanto a de *software*. O código automático do *hardware* corresponde a um projeto para o Quartus II, com os arquivos de configuração de projeto e os códigos *Verilog* de todos os componentes especificados no *design* criado pelo usuário, inclusive o *top*, sendo esse nada mais que um *template* contendo apenas os sinais de entrada e saída especificados no *design*. Com relação à geração do *software*, o resultado é um arquivo ”.h”, o qual contém as especificações de uma classe em C++, na qual estão contidos os sinais de controle e de parâmetros definidos na *GUI*. Além disso, essa classe referencia a *API* da *Gidel*, chamada *ProcAPI*, permitindo que o *software* possa usar as chamadas da *API* a partir de objetos dessa classe. A classe em C++ é considerada uma representação em *software* da placa, pois permite que o *software* possa realizar seu gerenciamento e transferência de dados para ela. A *Gidel* também fornece os *drivers* do barramento *PCIe* que permitem o reconhecimento do dispositivo pelo *PC*, além de instalar a *ProcAPI* no computador.

O kit de desenvolvimento da *Gidel* é bastante útil, pois o *driver* e a geração automática das interfaces entre *hardware* e *software* poupa bastante trabalho. Ele também permite integrar projetos de *hardware* e *software* cujas equipes estão separadas, bastando apenas seguir o padrão definido pelo design no *ProcWizard* (PROCWIZARD, 2014).

4.5 Metodologia Utilizada para a Aplicação Híbrida e Distribuída

O projeto proposto nesta Dissertação de Mestrado foi desenvolvido de acordo com as especificações de *hardware* e *software* disponíveis, além de definições de configuração as quais se espera atender aos objetivos de eficiência e transparência da aplicação distribuída no *cluster* híbrido. O fluxo de projeto dessa aplicação seguiu uma abordagem baseada no modelo *hardware-software codesign* (ARAÚJO, 2001), partindo da programação em *software*, adaptação para a aplicação distribuída, arquitetura em *hardware* e adaptação da aplicação híbrida no cluster. As etapas do desenvolvimento podem ser visualizadas na Figura 25.

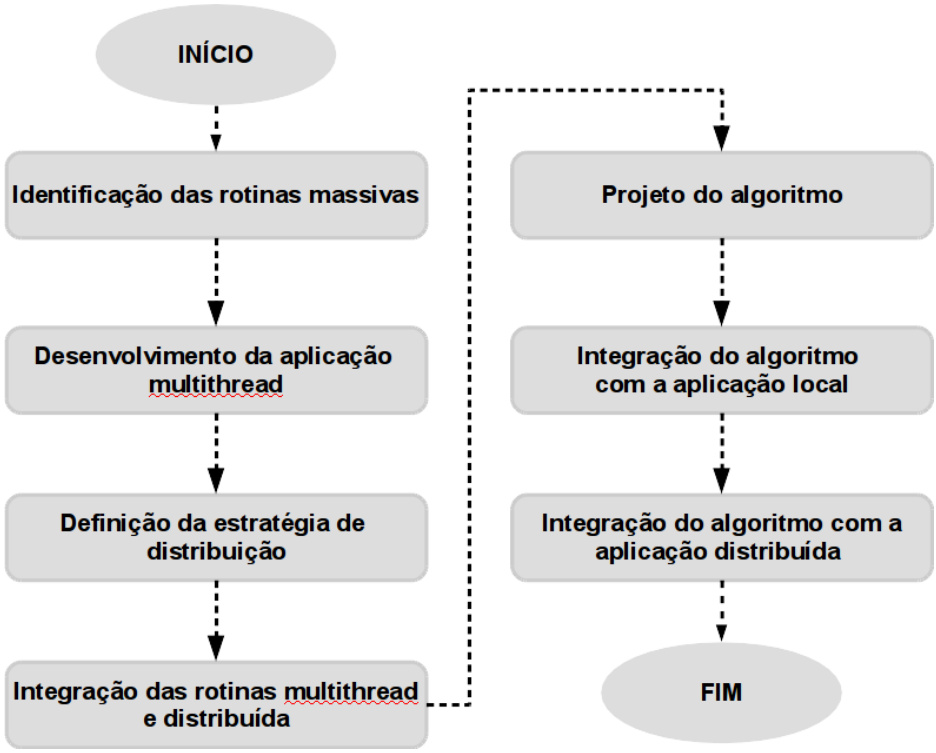

Figura 25: Fluxo de desenvolvimento de uma aplicação distribuída para o *cluster*.

De acordo com a Figura 25, podemos dividir as etapas do desenvolvimento de uma aplicação híbrida e distribuída em dois grupos. O primeiro grupo, da esquerda, é voltado para as etapas de desenvolvimento de artefatos de *software*, os quais são adicionadas as bibliotecas que suportam paralelismo a nível de *thread* e de processo, além do isolamento das rotinas de processamento massivo de dados, com o objetivo de facilitar a sua futura substituição pelo algoritmo em *FPGA*. Outra função importante do primeiro grupo é desenvolver uma aplicação distribuída apenas em *software*, para fins de comparação de desempenho com a aplicação híbrida.

O segundo grupo, da direita, agrupa as etapas de desenvolvimento do algoritmo em *hardware*, e sua integração com as rotinas de *software*, resultando assim numa aplicação capaz de aproveitar ao máximo os recursos oferecidos pelo *cluster* híbrido. Cada etapa de desenvolvimento pode ser descrita assim:

4.5.1 Identificação das rotinas massivas

Nesta etapa, ocorre uma análise da aplicação, a fim de descobrir quais rotinas estão consumindo maior tempo para serem processadas em *CPU*. Então, é feito um estudo para verificar o potencial paralelismo que essas rotinas possuem. Geralmente instruções com vários *loops* aninhados apresentam uma alta independência de suas instruções iteradas, ou seja, ocorre um alto poder de paralelismo entre as instruções internas ao *loop* e que sofrem várias replicações de operação. Outro ponto importante a se verificar é como os dados são reusados nessa operação interna, a fim de ser considerada durante a definição da unidade de controle do algoritmo em *hardware*.

É preciso verificar também a possível dependência de dados presente nessas iterações, pois ela influencia no potencial paralelismo que pode ser explorado tanto em nível de *software* como em nível de *hardware*. Ao final da análise, pode-se verificar a necessidade de adaptação das rotinas, seja com agrupamento em uma única função, ou inclusão de algum parâmetro de distribuição, tomando a precaução de manter a transparência das alterações para o usuário, como mostrado anteriormente (Figura 23).

4.5.2 Desenvolvimento da aplicação *multithread*

Nesta etapa começa a ser definido o primeiro estágio do processamento paralelo da aplicação. A partir dos estudos do potencial paralelismo, reuso e dependência de dados feitos na etapa anterior, é possível definir como se comportarão as *threads*, ou seja, quais instruções elas poderão executar simultaneamente e quais conjuntos parciais de dados poderão trabalhar. Então são definidas as configurações do *OpenMP* que serão aplicadas às funções. O objetivo dessa fase é criar um modelo de execução em *software* capaz de aproveitar ao máximo todos os recursos disponíveis nos processadores *multicore*, pois assim será possível afirmar que, mesmo com todas as otimizações possíveis de execução em processador, a execução massiva em *hardware* tem um desempenho superior.

4.5.3 Definição da estratégia de distribuição

Nesta etapa são definidos como os dados serão distribuídos entre os processos, modelando assim o paralelismo global da aplicação. É preciso definir também se há necessidade de criação de parâmetros de distribuição, de forma a facilitar o processamento de cada participante. Logo, são definidas quais funções do *MPI* serão utilizadas para realizar esse processamento distribuído, dependendo do tipo de comunicação que trará maior ganho de desempenho para a aplicação: ponto-a-ponto bloqueante, não-bloqueante ou coletiva. Por fim, é preciso implementar também as rotinas de mestre-escravo, observando as peculiaridades da aplicação, para então ter um eficiente fluxo de execução global.

4.5.4 Projeto do algoritmo em *hardware*

Esta é a parte que demanda mais tempo de desenvolvimento, pois é a elaboração, implementação e testes de uma arquitetura de processamento para *FPGA*, capaz de explorar de forma otimizada os trechos massivos da aplicação. Como já foi mencionado, o algoritmo deve interagir com os módulos de memória interna da placa, através das *FIFOs* do *multiport*, e com a aplicação por meio dos registradores mapeados. Essas interações devem fazer parte da máquina de estados principal de unidade de controle, a qual implementa uma arquitetura de transmissão e reuso de dados para os *PEs*, que por sua vez implementam a operação equivalente à instrução do *loop* mais interno da aplicação. Para realizar as simulações de corretude do módulo, é preciso simular as interfaces da arquitetura com o meio externo. A *Gidel* não fornece código de seus *IP-cores*, de forma a proteger a propriedade intelectual deles. Então é preciso desenvolver módulos que simulem o comportamento dos *multiports*, para dar uma visão mais fiel de como o algoritmo deve se comportar para processar corretamente. Após aprovado na simulação, o algoritmo está pronto para ser sintetizado junto com o *core services* que será gerado.

4.5.5 Integração do algoritmo com a aplicação local

Uma vez que o algoritmo está concluído, é preciso gerar as interfaces necessárias para que o *software* possa realizar seu gerenciamento. Para tanto, cria-se um projeto na ferramenta *ProcWizard*, onde o algoritmo é instanciado, incluindo seus sinais e parâmetros que comunicam-se com o *host*. Além disso, são configurados os *multiports* de acordo com o número de *FIFOs* de leitura e escrita dos quais o algoritmo necessita. Após definido o projeto,

realiza-se a geração automática das interfaces de *hardware* e *software*. Para o *hardware*, é preciso mapear corretamente o *top* (nível mais alto da hierarquia de *hardware*) da arquitetura desenvolvida, além de adicionar os módulos integrantes da arquitetura para o projeto gerado do *Quartus II*, e então é feita a síntese para geração do *bitstream*.

Para o *software*, a inclusão da classe da placa, instanciação da mesma, e adição das chamadas da *API* para substituir as rotinas massivas são suficientes para sua integração.

Concluídas as integrações, a aplicação local híbrida está pronta para ser testada quanto à sua corretude, em relação à versão original sem aceleração, desenvolvida em *software*. Esta versão totalmente em *software* é chamada de modelo de referência.

4.5.6 Integração do algoritmo com a aplicação distribuída

Depois de passar com sucesso nos testes, a aplicação híbrida está pronta para substituir os trechos de processamento equivalentes do processo escravo, formando então a aplicação final para o cluster. Faz-se necessário realizar testes para verificar se a aplicação está bem integrada e funcionando corretamente. Após o sucesso nesses testes, a aplicação estará pronta para ser avaliada com relação ao seu desempenho, frente à sua equivalente apenas em *software*.

4.6 Conclusões

Neste capítulo vimos os componentes necessários para a construção do *cluster* híbrido utilizado neste trabalho. Também vimos a estrutura lógica de cada participante do sistema, incluindo a descrição do controle de fluxo de uma aplicação distribuída, desde sua inicialização, distribuição dos dados, seu processamento massivo local, distribuição dos resultados parciais e terminando com a formação da saída do sistema.

Também foi mostrado que um dos nós do *cluster* pode ser utilizado para desenvolver os artefatos necessários para aceleração das rotinas massivas de uma aplicação. De forma a aproveitar tanto os elementos de paralelismo em *software* e de *hardware* na placa com FPGA, ferramentas de desenvolvimento foram instaladas nesse nó. Assim, o desenvolvedor de uma aplicação a ser acelerada no *cluster* pode produzir os artefatos e testar no mesmo ambiente, ganhando maior produtividade.

Por fim, vimos a metodologia desenvolvida para definir o fluxo de desenvolvimento da aplicação distribuída, com o objetivo de acelerar as rotinas massivas da mesma,

aumentando a eficiência da aplicação como um todo. As etapas abrangem a análise dessas rotinas de processamento massivo de dados, definição do paralelismo em nível de *thread* e de distribuição, definição do paralelismo em *hardware*, sua integração com o *PC* e com o *cluster* como um todo. Durante cada etapa, testes de corretude são necessários para uma integração correta, garantindo assim as mesmas funcionalidades gerais da aplicação original, mantendo a transparência de comportamento para o usuário da mesma.

Os detalhes dessa metodologia de desenvolvimento dependem fortemente da natureza da aplicação que sofrerá as adaptações para execução acelerada no cluster. No próximo capítulo mostraremos dois exemplos onde, apesar de seguirem a mesma metodologia, suas etapas foram aplicadas de forma específica para cada aplicação escolhida.

5. Estudos de Caso

Neste capítulo veremos dois exemplos de aplicações e seus fluxos de desenvolvimento, com o objetivo de executar de forma eficiente no *cluster* híbrido. Essas aplicações foram escolhidas por serem importantes nas suas áreas de conhecimento.

Durante o detalhamento do fluxo, será mostrado como a metodologia aplicada nos dois exemplos influenciou num melhor aproveitamento dos recursos paralelos do *cluster*, tanto no ambiente de software (*threads* e distribuição dos dados) como no ambiente do *FPGA* (arquitetura customizada do algoritmo). Para cada exemplo serão mostrados os resultados, onde podemos verificar o ganho de desempenho de cada nova aplicação híbrida e distribuída, a qual foi resultante do processo de desenvolvimento.

5.1 Multiplicação de Matrizes Densas

Uma das operações mais utilizadas em cálculos científicos é a multiplicação entre matrizes. Essa operação faz parte de diversas ferramentas e bibliotecas que realizam cálculos matriciais e vetoriais, como o *MATLAB*, o *PETSc*, além de servir como exemplo para tutoriais de ambientes de programação. Logo, essa operação é considerada um exemplo clássico, com diversos projetos e linhas de pesquisa produzidas em torno desse tema.

Matriz densa nada mais é do que uma matriz onde a grande maioria de seus elementos é diferente de zero. Logo, praticamente todos os elementos da matriz são considerados úteis para processamento, e à medida que o tamanho da matriz cresce, o custo computacional para gerar a matriz resultante passa a ser maior.

Uma operação de multiplicação entre matrizes pode ser representada da seguinte forma (3):

$$C_{(ixj)} = A_{(ixk)} x\, B_{(kxj)} \qquad (3)$$

Onde A, B e C são matrizes com a representação de suas linhas e colunas sendo feitas através das variáveis *i*,*j* e *k.* Como regra dessa operação, a matriz A deve ter o número de colunas do mesmo tamanho que as linhas da matriz B, gerando uma matriz C cujo número

de linhas equivale ao de A e o de colunas ao de B. E a geração de cada elemento dessa matriz C é feita de acordo com a seguinte fórmula (4):

$$c_{i,j} = \sum_{n=1}^{k} a_{i,n} * b_{n,j} \qquad (4)$$

Onde *a*,*b* e *c* são os elementos de cada matriz, e *i*,*j*,*n* são suas posições. Podemos perceber que a geração de cada elemento da matriz C pode ser feita de forma independente, com um alto reuso dos dados das matrizes operandos. Por isso que esse é considerado um problema de *HPC*, pois a geração dos elementos dessa matriz é um processamento massivo de dados com alto potencial paralelo. Computacionalmente, diversos algoritmos foram desenvolvidos para combinar da melhor forma possível as características de reuso e paralelismo dessa operação, um deles é o *SUMMA* (GEIJIN, 1996), cujo princípio visa fixar uma coluna de A e multiplicá-la por cada linha de B, gerando uma matriz parcial C, depois passar para a próxima coluna de A, realizar a mesma operação e somar com a matriz parcial anterior, e assim por diante até atingir a última coluna de A.

A análise da aplicação foi concebida a partir do desenvolvimento de um programa capaz de multiplicar duas matrizes. Para o nosso exemplo, vamos utilizar apenas matrizes quadradas, ou seja, as matrizes A, B e C possuem linhas e colunas de mesmo tamanho. Esse tipo de matriz é mais simples de trabalhar, pois não são necessárias verificações ou adaptações para que a operação de multiplicação seja efetuada corretamente. Cada matriz operando está armazenada num arquivo de entrada, e a matriz resultado C será carregada num arquivo de saída. O tamanho das matrizes foi alocado num arquivo de parâmetros. A Figura 26 demonstra as estrutura de entradas e saída do *software*.

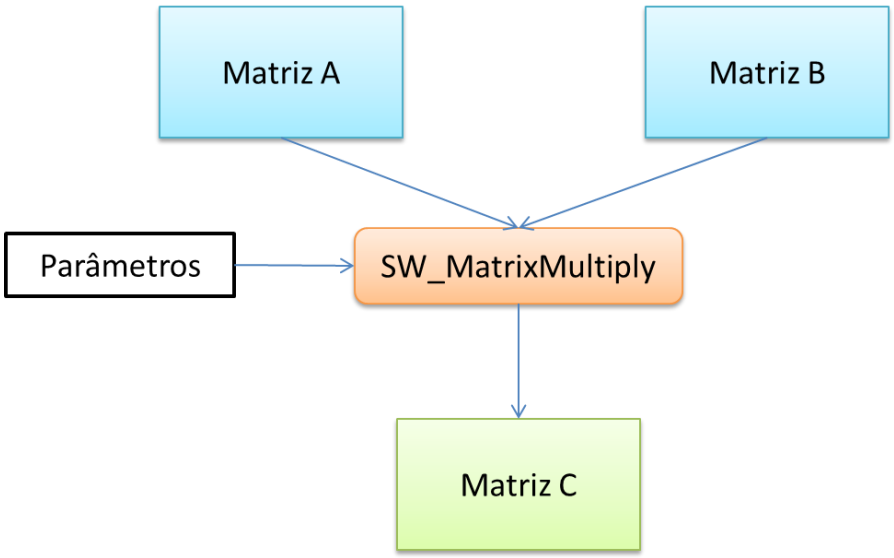

Figura 26: Estrutura da aplicação desenvolvida inicialmente para multiplicação de marizes.

O programa inicialmente desenvolvido opera com matrizes quadradas, onde em cada arquivo de entrada os números estão dispostos numa sequência vertical de linhas, e os elementos pertencentes a uma dessas linhas estão separados por um espaço para indicar que são de colunas distintas. Esses números estão em formato de ponto flutuante com precisão simples. O arquivo de parâmetros armazena o número de linhas e colunas, sendo para nosso caso esses números são iguais. A matriz C é armazenada com o mesmo formato das entradas. A rotina massiva, que é a própria operação de multiplicação de matrizes, foi isolada numa função, cuja assinatura é mostrada a seguir:

int * multiplica_matrizes (**float** *matrizA, **float** *matrizB, **int** num_linhas, **int** num_colunas)

A função **multiplica_matrizes** recebe como parâmetros os *buffers* das matrizes de entrada A e B e o número de linhas e de colunas, os quais para nosso escopo são iguais. Os *buffers* das matrizes operandos são estruturados a partir da linearização dos arquivos de entrada, ou seja, cada *buffer* de entrada é um grande vetor de linhas concatenadas da matriz, como exemplificado na Figura 27.

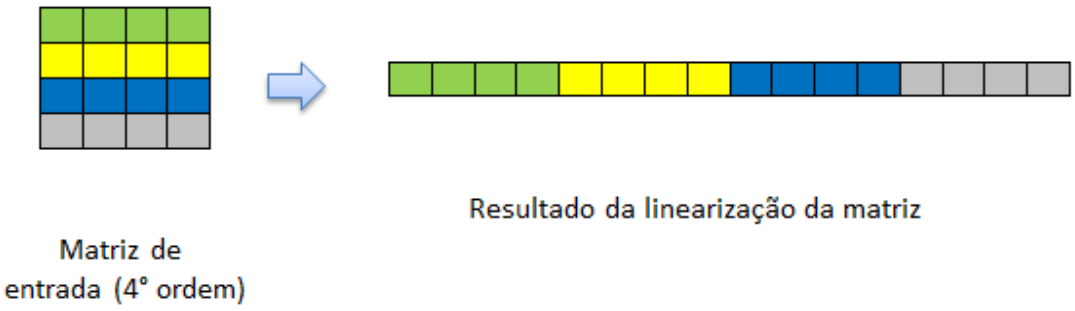

Figura 27: Exemplo de linearização de uma matriz 4x4.

Como podemos verificar na Figura 27, para um *buffer* de uma matriz 4x4, os dezesseis elementos são dispostos em quatro linhas concatenadas em sequência. Vale salientar que as matrizes de entrada utilizadas nesse experimento são bem maiores do que o exemplo da Figura 27, utilizado apenas como uma forma mais fácil de visualizar a linearização. O motivo de linearizar os operandos e a saída é preparar os dados para as funções de distribuição, as quais serão definidas na etapa do desenvolvimento correspondente.

Ainda sobre a função de multiplicação de matrizes, sua operação massiva é composta por *loops* aninhados, onde no primeiro nível percorre as linhas da primeira matriz, o segundo varre as colunas da segunda e o terceiro nível percorre ao mesmo tempo a coluna da primeira e a linha da segunda matriz, para então realizar a operação principal de multiplicação e acumulação de um elemento da matriz resposta. Este trecho de instruções foi selecionado para ser substituído por um algoritmo de processamento em *FPGA*, pois seus dois *loops* mais externos demonstram uma independência de dados, isto é, podem ser "*desenrolados*" e processados paralelamente. Além disso, o *loop* mais interno, o qual implementa o somatório para calcular o elemento $c_{i,j}$ já mencionado (4), também podem ter suas operações paralelizadas, pois as multiplicações elemento a elemento são independentes entre si.

Para desenvolvimento da estrutura de execução paralela baseada em *threads*, ficou definido que o *loop* mais interno da função de multiplicação seria paralelizado, possibilitando a multiplicação e acumulação de partes do conjunto linha-coluna ser realizada de forma simultânea. Logo, cada *thread* ficou responsável por realizar multiplicação e acumulação de uma parte da linha de A com uma parte correspondente da coluna de B, gerando um resultado temporário, como pode ser visto na Figura 28. Ao término da execução paralela do *loop* interno, os resultados parciais são somados para formar um elemento da matriz C.

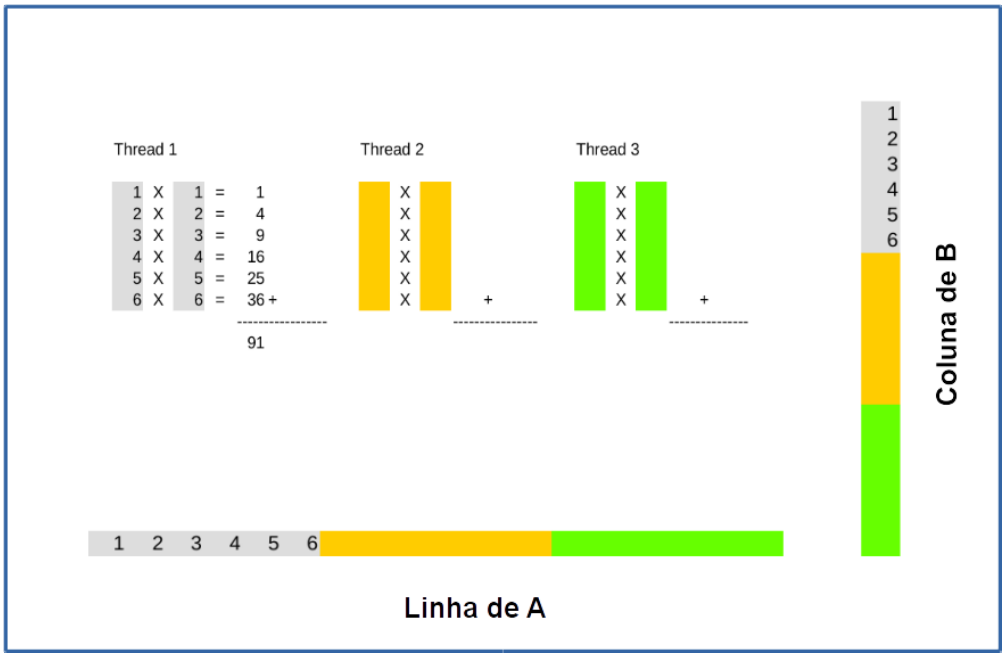

Figura 28: Abordagem em *threads* para geração de um elemento da matriz C.

Para implementar a abordagem apresentada na Figura 28, foi utilizada a diretiva *for* do *OpenMP*, pelo fato da iteração ser deste tipo. A divisão de dados para cada thread foi feita segundo a regra (num_linhas / num_threads), ou seja, há uma divisão em partes iguais para cada fluxo escalonável de processamento. A implementação dessa configuração pode ser vista no trecho do código a seguir (Figura 29).

```
float * multiplica_matrizes (float *fatiaA, float *matrizB, int num_linhas, int num_colunas, int tam_fatia) {
    float *resultado = alocar_buffer_linear (tam_fatia);
    int i,j,k;
    int cont = 0;
    int linhas_por_fatia = tam_fatia / num_colunas;
    int chunk;

    for (i = 0; i < linhas_por_fatia; i++) {
        for (j = 0; j < num_colunas; j++) {

            float tmp = 0;

            #pragma omp parallel shared (fatiaA, matrizB, num_colunas, chunk) \
            reduction (+:tmp) private (k)
            {
                chunk = num_linhas / omp_get_num_threads();

                #pragma omp for schedule (dynamic, chunk) nowait
                for (k = 0; k < num_linhas; k++) {
                    tmp = tmp + (fatiaA [i*num_colunas + k] * matrizB [j + k*num_colunas]);
                }
            }
            resultado [cont] = tmp;
            cont++;
        }
    }
    return resultado;
}
```

Figura 29: Função de multiplicação de matrizes com as diretivas do OpenMP

A estratégia adotada para a distribuição dos dados no cluster foi concebida de acordo com a topologia e estrutura de controle definidas para este projeto. Após a leitura das entradas, os parâmetros e a matriz B são compartilhados por todos os nós, ou seja, são transferidos através de *broadcast* para todos os nós. Por outro lado, a matriz A é dividida com relação ao número de linhas, e cada parte da mesma é entregue para cada nó participante do *cluster*. Um exemplo simplificado dessa divisão é mostrado na Figura 30.

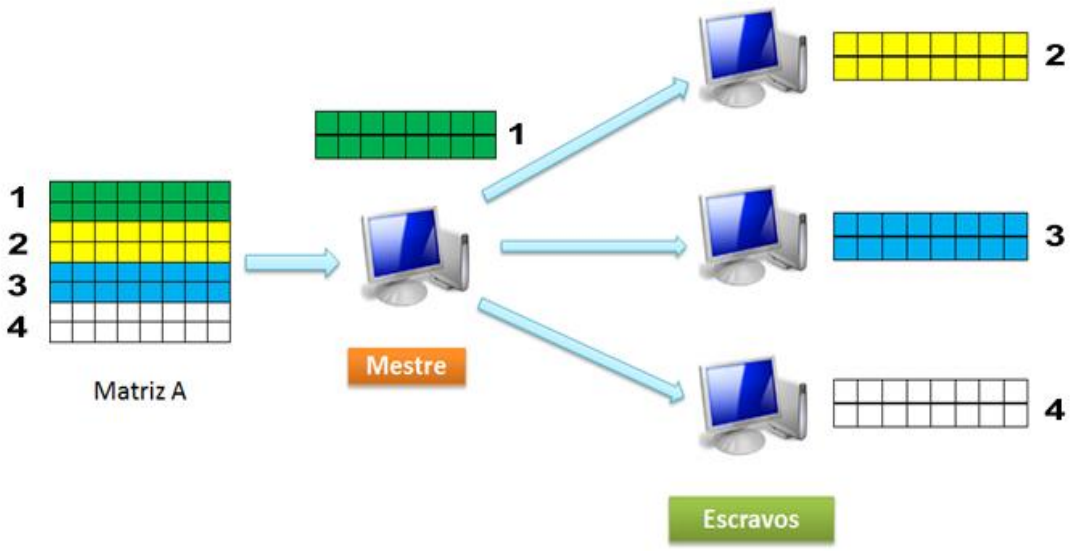

Figura 30: Processo de divisão da matriz A para cada nó.

O exemplo mostrado na Figura 30 mostra a divisão de uma matriz 8x8 em quatro conjuntos de duas linhas cada, distribuídos para cada participante do sistema. A função do *MPI* utilizada para esse recurso foi a *MPI_Scatter*, a qual suporta comunicação coletiva e faz simultaneamente a divisão de um grande buffer. O trecho do código onde ocorre a distribuição dos parâmetros e das matrizes de entrada pode ser visualizado na Figura 31.

```
.
if (id_processo == MESTRE) {
    FILE *param = fopen ("parametros.txt", "r");
    ler_parametros (param, &parametro [0], &parametro [1]);
    fclose (param);
}
MPI_Bcast (parametro, 2, MPI_INT, MESTRE, MPI_COMM_WORLD);

tam_total_matriz = (parametro [0] * parametro [1]);
tam_fatia = tam_total_matriz / num_processos;
bufferA = alocar_buffer_linear (tam_fatia);

matrizC = alocar_buffer_linear (tam_total_matriz);

if (id_processo == MESTRE) {

    FILE *entradaA = fopen ("matriz_a.txt", "r");
    FILE *entradaB = fopen ("matriz_b.txt", "r");

    matrizA = lerArquivoMatrizes (entradaA, parametro [0], parametro [1]);
    matrizB = lerArquivoMatrizes (entradaB, parametro [0], parametro [1]);
}
else {
    matrizB = alocar_buffer_linear (tam_total_matriz);
}

MPI_Bcast (matrizB, tam_total_matriz, MPI_INT, MESTRE, MPI_COMM_WORLD);
MPI_Scatter (matrizA, tam_fatia, MPI_INT, bufferA, tam_fatia, MPI_INT, MESTRE, MPI_COMM_WORLD);
MPI_Barrier (MPI_COMM_WORLD);
```

Figura 31: Trecho da aplicação responsável pela distribuição dos parâmetros e das matrizes de entrada

A partir dessa divisão, cada nó irá operar as linhas recebidas de A com todas as colunas de B, resultando numa matriz parcial C. Essa operação pode ser visualizada na Figura 32:

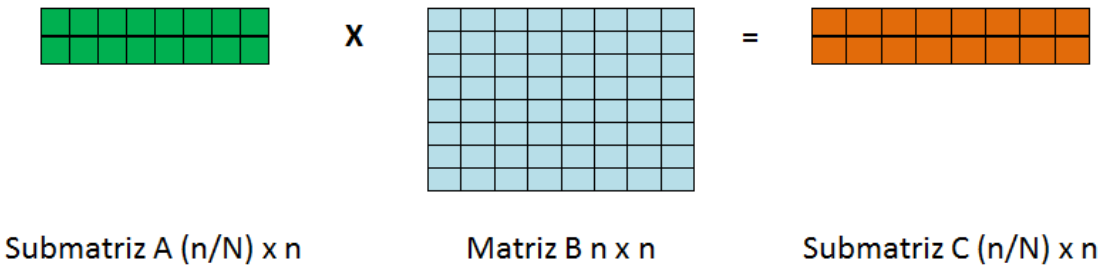

Figura 32: Operação de multiplicação em cada nó.

Na Figura 32, é possível verificar que uma parte da matriz A, de tamanho ((n/N) x n), onde n é ordem da matriz e N o número de nós do sistema, resulta numa submatriz C, e por via da regra de multiplicação, essa submatriz possui o mesmo tamanho da fatia de A. Assim, tanto a definição da distribuição como a do paralelismo local da operação massiva de multiplicação, são reunidas em uma única aplicação distribuída, reunindo as funções e configurações tanto do *OpenMP* quanto do *MPI*, e cujo objetivo é servir de parâmetro de

desempenho para a placa aceleradora, pois o paralelismo nodal baseado em threads representa o máximo desempenho simultâneo que um processador *SMP* pode oferecer.

Para o mestre receber os dados processados dos escravos, utilizou-se a função *MPI_Gather*, para reunir cada resultado parcial num único *buffer* de saída, como pode ser visto no trecho de código da Figura 33.

```
MPI_Gather (bufferC, tam_fatia, MPI_INT, matrizC, tam_fatia, MPI_INT, MESTRE, MPI_COMM_WORLD);
MPI_Barrier (MPI_COMM_WORLD);
```

Figura 33: instrução MPI_Gather, utilizada para recebimento dos resultados parciais dos escravos.

Para o projeto do algoritmo em *FPGA*, o algoritmo de multiplicação utilizado foi o clássico, o qual realiza a multiplicação de cada linha de A com cada coluna de B. O projeto desenvolvido fez parte do trabalho desenvolvido por *Neves, B.* (NEVES, 2012), onde possui maiores detalhes das funcionalidades internas da arquitetura. Foi definido que a matriz A seria o alvo do reuso de seus dados, isto é, uma linha de A é utilizada para multiplicação com todas as colunas de B, para então a arquitetura descartá-la e carregar a próxima linha.

Para permitir o reaproveitamento de uma linha da matriz A, a mesma é armazenada em 24 *FIFOs* de tamanho (número_de_colunas / 24). A coluna de B é acessada diretamente da memória dedicada da placa por 24 canais, sendo consumida até o final e depois carregada a próxima coluna. Ao final de todas as colunas de B terem sido processadas, a próxima linha de A é carregada, e um novo ciclo de operações começa. A estrutura de acesso aos dados das duas matrizes está ilustrada na Figura 34.

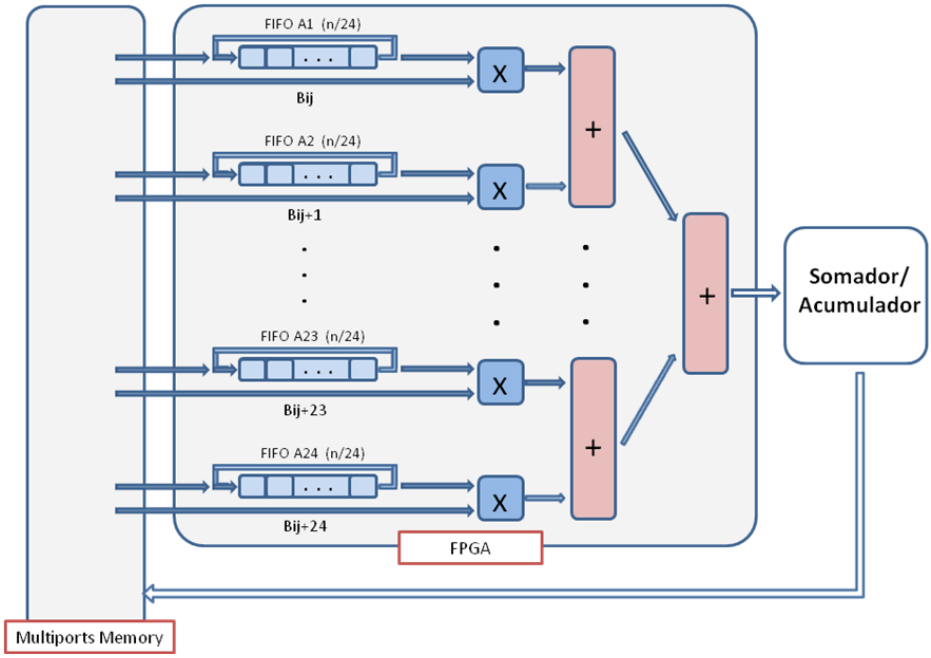

Figura 34: Visão geral do algoritmo de multiplicação, com as formas de acesso aos dados de cada matriz (NEVES, 2012).

A placa *PCI* utilizada para implementar o algoritmo foi a *PROCeIII* (PROCEIII, 2014), da *Gidel*. Ela possui um *FPGA Altera Stratix III*, com frequência de operação que pode chegar a até 300 MHz. Ela possui 32 canais *DMA*, permitindo que ela possa trabalhar em modo *master* no barramento. Como elementos de memória, temos a memória soldada na placa, com 512 MB e taxa de transferência em torno de 4 GB/s, além de dois *slots* que suportam módulos de até 4 GB. Para este trabalho, foram utilizados dois módulos de 2GB cada. Logo, podemos instanciar três *multiports* para o algoritmo, com largura de canal e 256 bits cada. Essa é a justificativa da limitação da arquitetura em trabalhar com apenas 24 números de A e 24 de B por vez, pois o *multiport* do fabricante suporta a configuração de *FIFOs* com no máximo 256 bits, comportando assim 24 números de 32 bits.

Para fornecer um fluxo contínuo dos dados para os elementos de processamento, foi desenvolvida uma estrutura chamada de Árvore de Soma, a qual consiste numa entrada de 24 multiplicadores, responsáveis por realizar a multiplicação elemento a elemento da linha de A e coluna de B, e nos estágios seguintes há cadeias de somadores, os quais vão diminuindo a cada estágio, até chegar a um no estágio final. Os módulos de multiplicação e soma utilizados, ou seja, os elementos de processamento da arquitetura, foram os *IP-cores* fornecidos pela *Altera*, os quais realizam operações com números em ponto flutuante precisão simples padrão *IEEE 754*, tipo representado pelos 32 bits reservados para um número, como descrito anteriormente. A Árvore de Soma implementada está ilustrada na Figura 35.

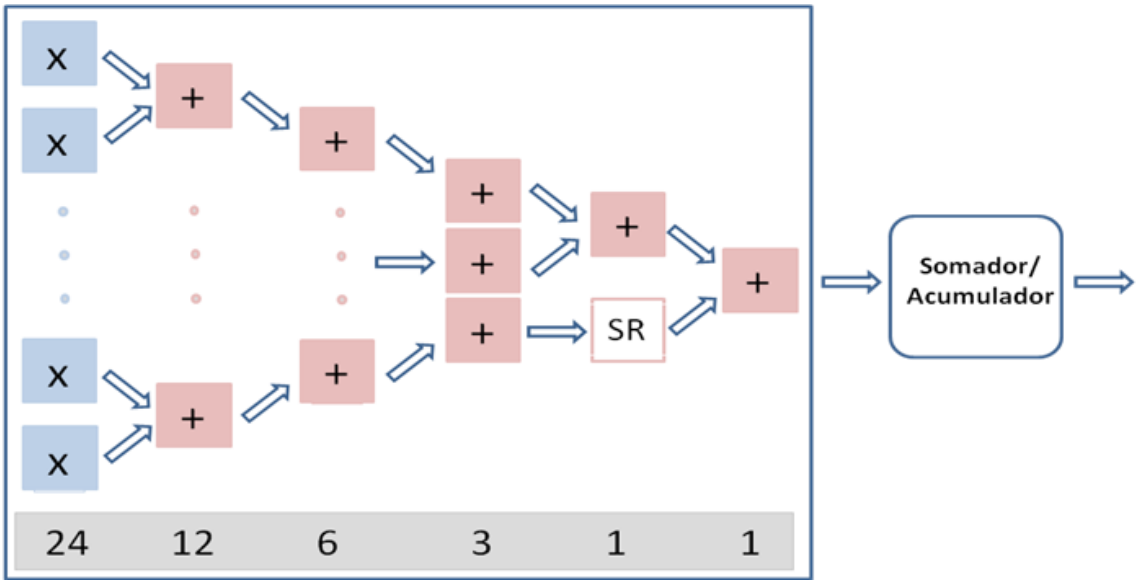

Figura 35: Arvore de soma implementada (NEVES, 2012).

De acordo com a Figura 35, a árvore realiza 24 multiplicações simultâneas entre cada elemento da linha de A e coluna de B, e depois vai somando-os no decorrer dos estágios somatórios, até chegar ao acumulador, resultando na soma de todas as 24 multiplicações feitas na entrada. A escolha de estágios somatórios foi feita para sincronizar o atraso do *pipeline* dos módulos de ponto flutuante da *Altera*. Com isso, o fornecimento contínuo dos dados durante toda a execução do algoritmo é assegurado. O somador / acumulador mostrado nas Figuras 34 e 35 possui uma função especial. Como já foi mencionado, a arquitetura suporta por vez a operação entre 24 números da linha de A e 24 da coluna de B. Como matrizes densas possuem bem mais do que essa quantidade, é preciso guardar o resultado de cada iteração de *hardware* realizada, somar com a próxima iteração, e assim por diante até o término da coluna, formando assim um elemento da matriz de resposta. Essa tarefa é feita no somador/ acumulador, o qual possui um controlador que regula os quatorze estágios de *pipeline*, os quais realizam a soma e acumulação do resultados advindos da árvore de soma, até terminar o processamento de uma coluna inteira de B.

A integração do algoritmo em *FPGA* com a camada de software já desenvolvida é feita a partir da geração automática do *ProcWizard*, como já foi descrito anteriormente. Para o *hardware*, o Projeto integra as *FIFOs* do *multiport* ao algoritmo, e aloca seus sinais de controle para registradores especiais. Na parte do *software*, ocorrem algumas adaptações.

O *software* obrigatoriamente deve ser escrito em C++, de forma a ser compatível com a classe da placa gerada. Além disso, é preciso implementar as chamadas de sistema

responsáveis pelo controle da placa, a partir dos registradores, e transmissão de dados entre as memórias principal e dedicada.

Todas essas funções responsáveis pelo gerenciamento do algoritmo terão funções da *API* do fabricante, habilitadas a partir da inclusão da classe gerada no projeto de software.

O algoritmo de processamento massivo trabalha com um fornecimento diferenciado de dados, pois sua entrada comporta no máximo 24 números da matriz A e da matriz B por vez. Então, foi preciso criar uma rotina de pré-processamento dos dados, de forma a eles serem alocados da maneira que a arquitetura em *hardware* espera. Cada módulo de memória da placa aloca conjuntos de 8 números de 32 bits, sendo que o próximo conjunto sempre deve estar 16 posições após o conjunto anterior, ou seja, dividindo uma linha ou coluna de uma matriz em conjuntos de 24 elementos, os primeiros oito serão alocados para o primeiro módulo de memória, os segundos conjuntos de oito para a segunda os terceiros para a terceira memória, repetindo-se esse ciclo entre os próximos 24 grupos. A organização das memórias da placa é feita por uma função de pré-processamento e um exemplo de funcionamento da mesma é ilustrado na Figura 36.

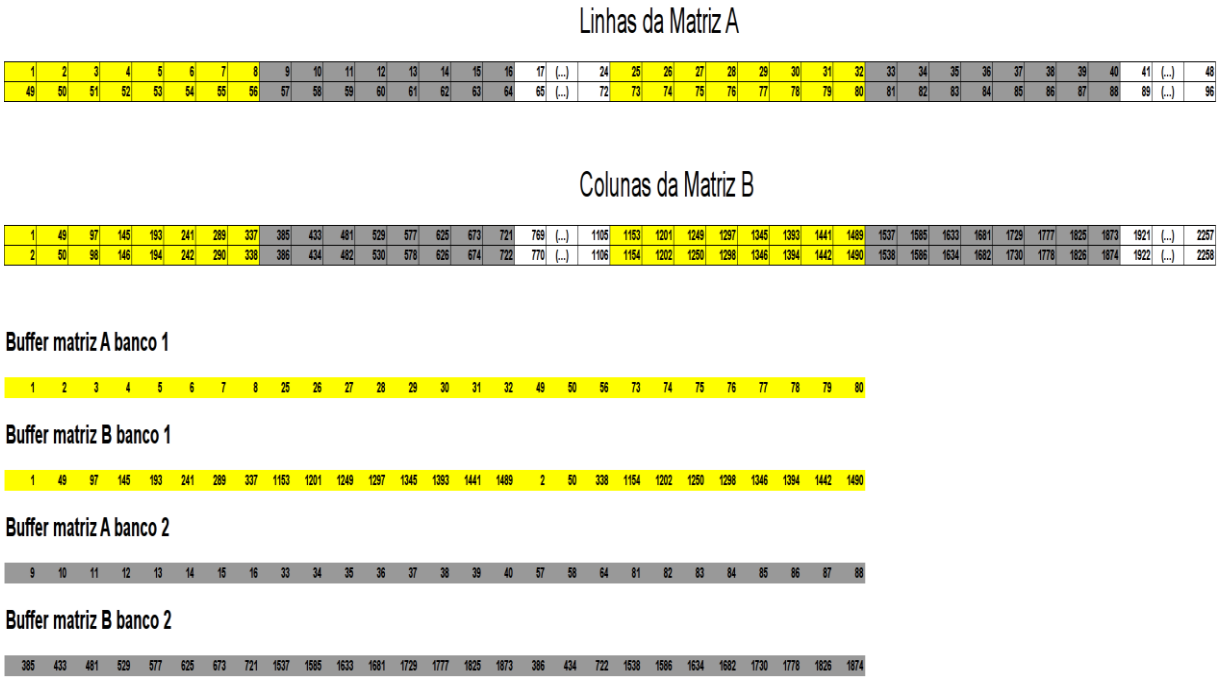

Figura 36: Particionamento das matrizes de entrada.

A Figura 36 retrata um exemplo com as duas primeiras linhas e as duas primeiras colunas de uma matriz de 48x48. Os elementos de cada uma dessas matrizes possuem valores ordenados, onde a primeira linha recebe os valores de 1 a 48, a segunda de 49 a 96, e assim por diante. Abaixo das linhas e colunas estão os buffers que irão carregar as memórias da

placa. Eles realizam um carregamento serial, então a ordem em que estão dispostos os números no *buffer*, são alocados nos módulos de memória dedicados. A sequência dos números segue as seguintes condições de alocação:

buffer_A1 = ai,j, onde 1 < i < n, 1 < j < n e j mod 24 < 8
buffer_A2 = ai,j, onde 1 < i < n, 1 < j < n e 8 <= j mod 24 < 16
buffer_A3 = ai,j, onde 1 < i < n, 1 < j < n e j mod 24 >= 16
buffer_B1 = bi,j, onde 1 < i < n, 1 < j < n e i mod 24 < 8
buffer_B2 = bi,j, onde 1 < i < n, 1 < j < n e 8 <= i mod 24 < 16
buffer_B3 = bi,j, onde 1 < i < n, 1 < j < n e i mod 24 >= 16

A integração da aplicação local híbrida com as instruções de distribuição já implementadas, ocorreu de forma suave, com apenas a junção os trechos de código. Com isso, a aplicação distribuída híbrida foi concluída com sucesso, estando apta a ser avaliada quanto a seu desempenho no *cluster* desenvolvido.

Resultados

A análise de desempenho da aplicação híbrida e distribuída foi feita considerando dois cenários. Um deles demonstra o desempenho entre o *cluster* convencional e o híbrido. Devido à configuração das entradas na implementação feita em *hardware*, as matrizes operandos devem ter um tamanho que seja múltiplo de 24, que é o número de operações feitas a cada ciclo pelo algoritmo. A outra restrição possui causa a construção do módulo somador / acumulador, o qual só desempenha corretamente seu papel se seu *pipeline* de 14 estágios estiver cheio. Então para garantir essa característica, é preciso 16 grupos de 24 números, ou seja, o tamanho mínimo de matriz que a arquitetura aceita é de 16x24=384. Como cada nó híbrido aceita no mínimo uma matriz de tamanho 384, então o mínimo para análise é de 3x384 = 1152. Os outros exemplos seguem a regra:

$N = 1152 + n*24*3$, onde N é o tamanho das matrizes, e n é um fator de crescimento da matriz.

A Figura 37 mostra o gráfico de desempenho do cluster híbrido frente ao cluster convencional.

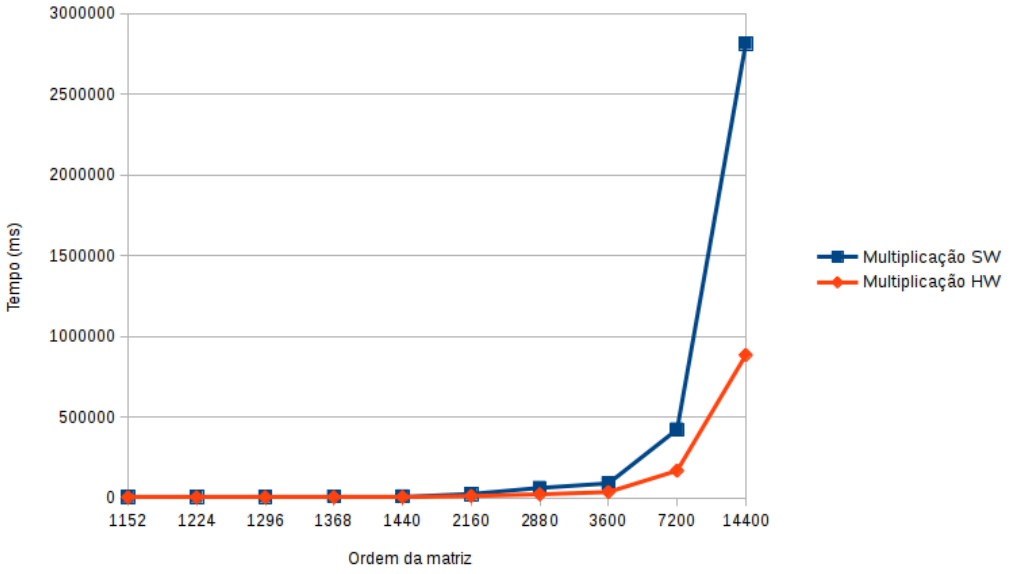

Figura 37: Desempenho do cluster híbrido comparado com o convencional.

Podemos notar na Figura 37 que o desempenho do cluster híbrido teve uma certa inclinação. Isso é causado pelos custos dos trechos de *software* da aplicação híbrida, como a latência, taxa de distribuição dos dados, pré-processamento, entre outros fatores, os quais não foram inseridos neste gráfico. De qualquer forma, a aplicação híbrida, para matrizes densas, mostra obter bons resultados, com média de ganho de desempenho em torno de 3x, e tendência a ter esse fator aumentado, mesmo com o *software* sendo executado rodando num conjunto de máquinas com alto poder de processamento.

Se considerarmos a divisão das principais etapas de execução da aplicação distribuída, sendo elas o carregamento dos dados, distribuição, processamento em si, recebimento dos dados e gravação da saída, é possível verificar a influência de cada uma delas na formação do tempo total do *cluster*. A Figura 38 ilustra graficamente a influência das etapas na execução apenas em *software* no *cluster*.

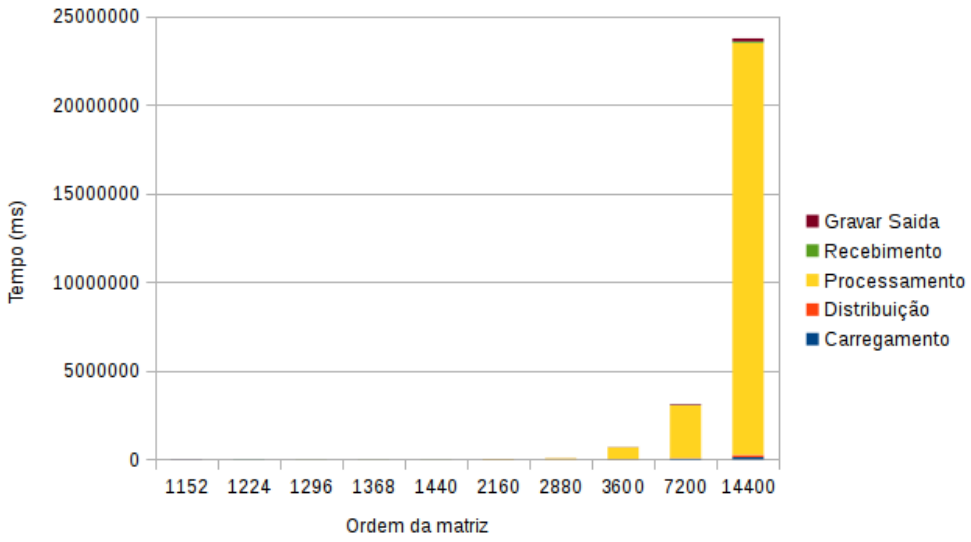

Figura 38: Influência das etapas de execução distribuída no cluster SW

Com o auxílio da Figura 38, percebemos que, com o aumento do tamanho da matriz, o processamento principal em *SW*, em cada nó, ocupa a maior parte do tempo do processamento geral, indicando a deficiência do processador no trecho massivo da aplicação.

A Figura 39 retrata a mesma influência das etapas, considerando o processamento principal em cada nó sendo feito na placa aceleradora.

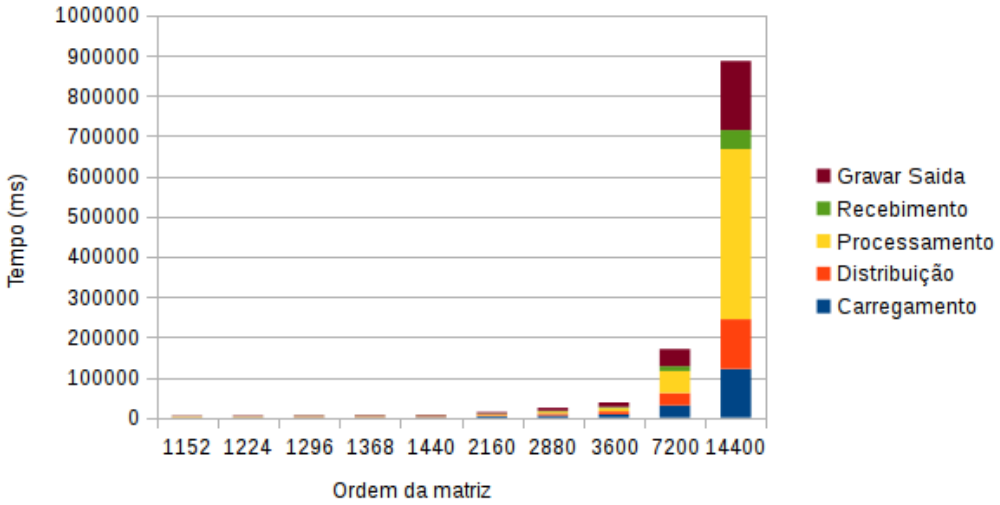

Figura 39: Influência das etapas de execução distribuída no cluster híbrido

Na Figura 39, percebemos que há uma maior influência das outras etapas da execução no processamento híbrido distribuído. Isso é devido a maior eficiência do processamento principal em *hardware*, enquanto que as outras etapas executadas em *software* não sofreram alterações. Portanto, demonstra que a maior influência no desempenho do

cluster híbrido está de fato no seu processamento massivo de dados realizado na placa aceleradora.

O gráfico da Figura 40 destaca o processamento da multiplicação no processador quando comparado com a placa com *FPGA*.

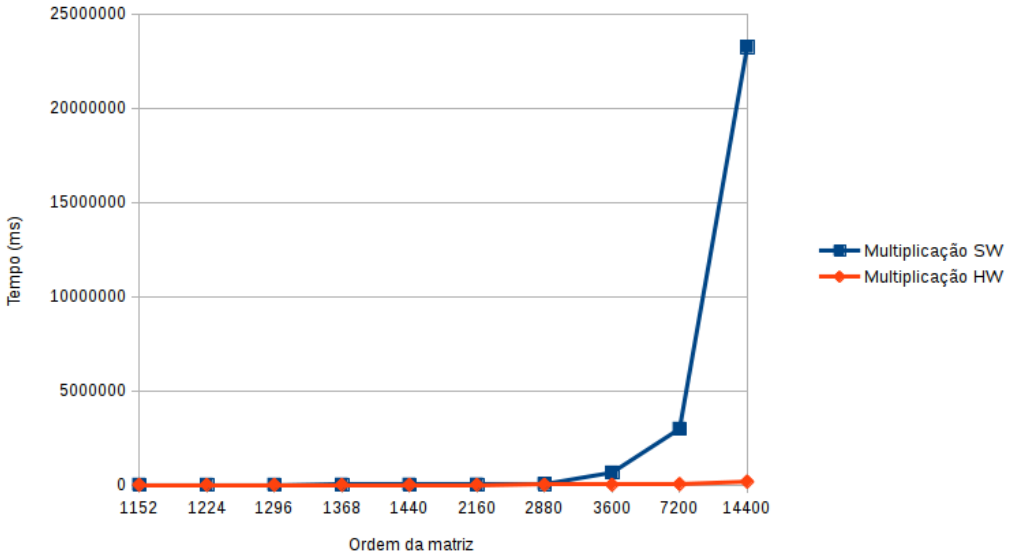

Figura 40: Desempenho do hardware e do software no processamento principal

Na Figura 40, percebemos que, considerando apenas a etapa do processamento principal de multiplicação, tanto da CPU de cada nó quanto da placa com FPGA, verificamos um desempenho bastante próximo nas matrizes até a ordem 2880, quando, em ordens superiores de matrizes, a curva da CPU começa a comportar-se de maneira exponencial. Isso acontece porque o tamanho dessas matrizes passa a não caber mais na cache de 12 MB do processador, como vemos em 3600, cujo tamanho em bytes da matriz é 3600x3600x32/8 = 51,84 MB. Portanto, podemos ver o quão eficiente é a arquitetura desenvolvida em *hardware*, gerando um *speedup* médio em torno de 22x.

5.2 RTM 3D (Etapa Modelagem)

O *RTM*, como já foi explicado no Capítulo 2, possui duas etapas de processamento. A etapa escolhida como estudo de caso para este trabalho foi a de modelagem, a qual consiste em processar diretamente a equação da onda, usando como entrada um modelo de velocidades

do terreno a ser explorado, para no final gerar o sismograma do mesmo. Este projeto foi realizado em parceria da Petrobras com o Projeto HPCIn / CIn-UFPE.

Inicialmente houve uma análise da aplicação original, na qual foram identificados diversos trechos de processamento massivo de dados, devido aos dados das matrizes serem vetores tridimensionais. Instruções de inicialização dos vetores na memória, estruturas de pré-processamento da imagem e de variáveis, além é claro da iteração em *loops* aninhados que processam a janela da equação da onda ao longo do tempo, são alguns exemplos. Inclusive o processamento da janela foi escolhido para ser substituído por uma arquitetura de processamento em *FPGA*, pois é o trecho do programa de maior custo computacional e com grande potencial paralelo.

A inclusão de processamento *multithread* no código ocorreu em vários de seus trechos. A começar, no *loop* de inicialização da matriz de velocidades, na qual há como operação no *loop* mais interno a multiplicação *vel*vel*fat*, onde *fat* é um fator constante na equação. Para esse trecho, o escalonamento proporcionado pela diretiva *for*, permite que o grande bloco de iterações seja dividido em grupos de processamento independentes entre as *threads*, cujo princípio é bem parecido com do processamento de multiplicação de matrizes. Em trechos onde ocorrem algumas operações que calculam o amortecimento de bordas na imagem, foram separados por s*ections*, e cada processador pode executá-los independentemente. Na inicialização de vetores tridimensionais e de troca de dados, como cada iteração possui independência total de dados, foi inserida apenas a diretiva *parallel* para que fosse executado paralelamente, independente da ordem. Já para o processamento da equação da onda acústica propriamente dita, cuja fórmula está na sessão 2.4, foi feito, além da configuração de diretiva *for* para o *loop* mais interno, configurações adicionais para algumas variáveis que são utilizadas no processamento principal, como as matrizes que representam os campos de pressão atual, passado e futuro, além da matriz de velocidades, precisaram ser declaradas como compartilhadas entre as *threads*. Assim, evita-se que *threads* possam arbitrariamente operar sobre dados que ainda estão sendo utilizados em outras *threads*. O funcionamento das iterações com o uso de *threads* pode ser visto a partir da ilustração da Figura 41.

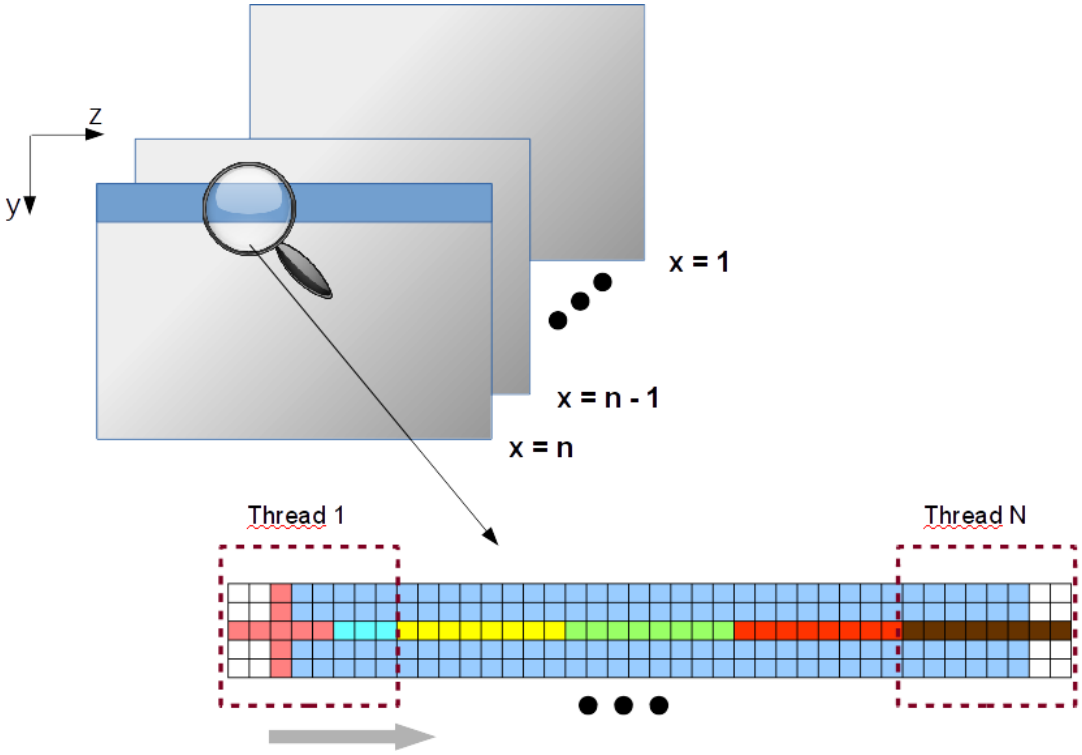

Figura 41: Divisão das regiões de processamento da janela por *threads*.

Como pode ser visto na Figura 41, o algoritmo de processamento em *software* para a modelagem do *RTM* utiliza uma lógica de orientação espacial pouco comum, apesar de obedecer corretamente à regra da mão direita. O *loop* mais interno itera com elementos do eixo z da matriz tridimensional, depois itera com y, para depois iterar com x, seguindo para o próximo plano. Durante a iteração em z, cada conjunto de pontos do processamento central da janela é atribuído a uma *thread*, a qual vai processar sequencialmente essa região. Percebemos também que as regiões entre as *threads* são também utilizadas para processamento das janelas vizinhas. Caso a *thread* processasse antes essa região, seriam gerados resultados inconsistentes. Considerando que cada *thread* foi configurada para processar primeiro os dados de endereços menores, então o cálculo das primeiras *threads* utilizaram os dados necessários antes das janelas anteriores chegarem nos últimos dados.

Com relação à estratégia de distribuição, o esquema de execução num *cluster* foi mais simples. O paralelismo foi baseado na quantidade de tiros ou pulsos a serem processados. A posição de cada tiro seque a seguinte lógica: um ponto de z é fixado, enquanto que nos eixos x e y, as coordenadas dos tiros são uniformemente espalhadas, ou seja, cada valor de coordenada do tiro, tanto no eixo x quanto no y, possui igual distância entre seus pontos vizinhos. Um exemplo de divisão num plano de z pode ser visto na Figura 42.

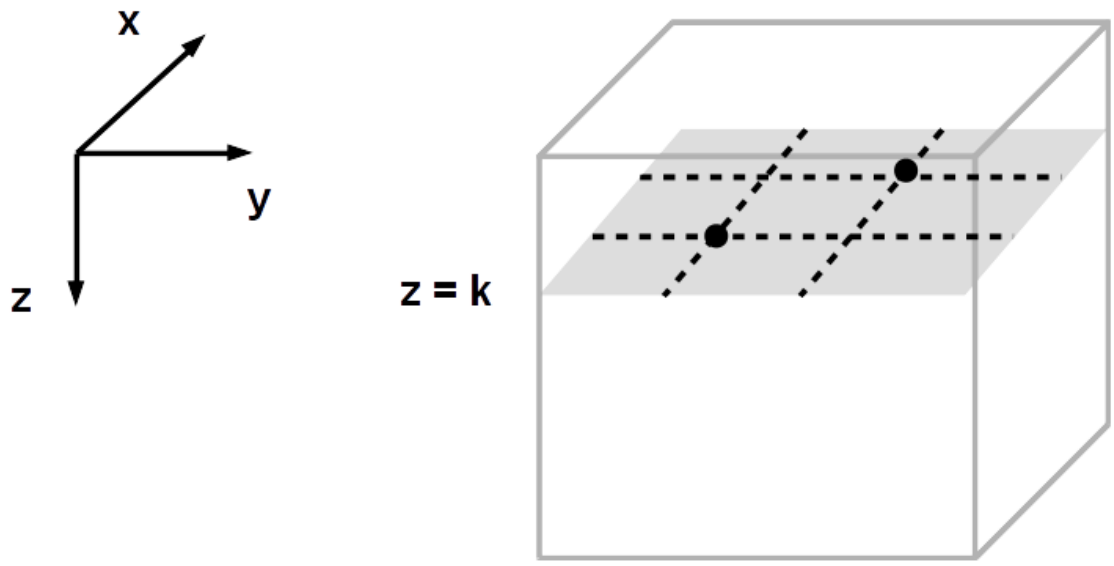

Figura 42: Um plano de z, com divisão por dois tiros.

Na Figura 42, percebemos que o exemplo realiza uma divisão por dois pontos. Cada projeção desses pontos nos eixos x ou y apresentam-se uniformemente distanciados.

Então, o mestre da aplicação distribuída inicialmente transmite a matriz de velocidades para todos os escravos, via *broadcast*. Depois os tiros são divididos igualmente entre os membros. Para fins de testes a quantidade de tiros foi proporcional à quantidade de nós do cluster.

O desenvolvimento do algoritmo em *hardware*, cuja autoria foi do grupo de pesquisa HPCIn/ Sísmica, foi baseado na substituição das rotinas de resolução da equação da onda, onde foi constatado o maior custo computacional do *software*. O paralelismo explorado pela arquitetura em *hardware* foi concebido em duas frentes. A primeira diz respeito o número de operações simultâneas, ou seja, quantas janelas de processamento podem ser executadas em paralelo. Então, a janela foi implementada como um *PE*, e o algoritmo foi desenvolvido para fornecer dados para 4 desses *PEs*, realizando assim quatro operações por ciclo de relógio. A janela de operações representada pelos 4 *PEs* pode ser vista na Figura 43.

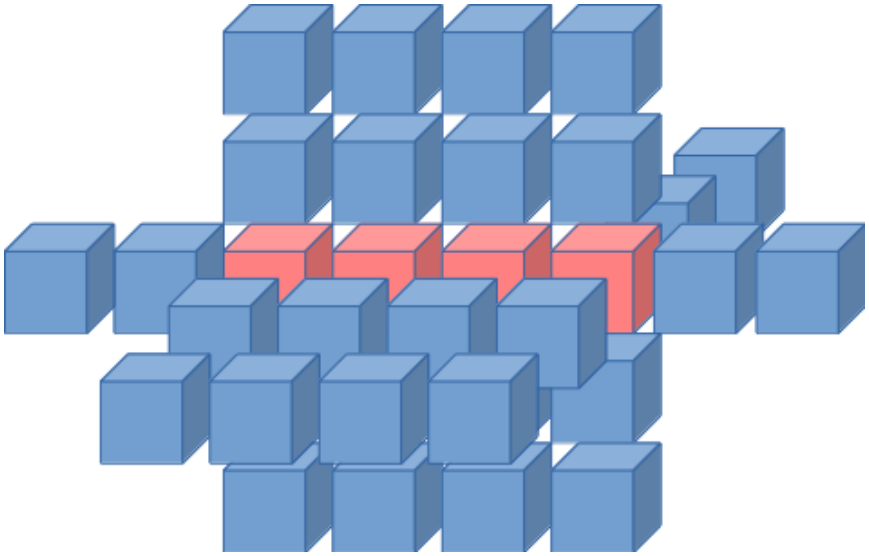

Figura 43: Janela de resolução em hardware composta por 4 PEs,

Os 4 *PEs* visualizados na Figura 43, percorrem uma matriz tridimensional do campo de pressão atual, primeiramente no sentido de z, depois no sentido do eixo y, e por último no sentido de x.

A outra frente de paralelismo atacada pelo algoritmo foi a de execução temporal. Foi desenvolvida uma arquitetura de fluxo contínuo de dados, capaz de executar quatro *time steps* simultaneamente. Para tanto, um grupo de quatro *PEs*, como o mostrado na Figura 53, é responsável pelo processamento de um *time step*. O primeiro grupo de *PEs* processa o primeiro passo temporal e a unidade de controle guarda esses dados em *FIFOs* internas para servir de entrada para o segundo grupo de *PEs*, cuja saída também é direcionada para um conjunto de *FIFOs* e assim por diante, até chegar ao quarto grupo, onde sua saída passa a ser escrita na memória. Logo, temos um algoritmo que possui 16 *PEs* que podem processar simultaneamente 4 *time steps*, e em cada passo, processa-se 4 janelas de processamento.

A placa utilizada para o algoritmo foi a *ProcStarIV*, da *Gidel*. Ela é equipada com 4 *FPGAs Stratix IV*, da *Altera*. Cada uma dessas *FPGAs* contam com um módulo de memória soldado na placa de 512 MB *DDR2*, e dois *slots* para memórias *DDR2* padrão *notebook*, que podem chegar a até 4GB. Logo, a placa possui oito *slots* para módulos de memória. Para este estudo de caso, foram utilizados dois módulos de 4GB, para uma *FPGA*. O barramento *PCIe* é do tipo x8. Esta placa, assim como a *ProceIII*, possui suporte a geração automática de artefatos para integração *hardware-software* através do *ProcWizard*, cujo design para o algoritmo de modelagem 3d foi desenvolvido, de maneira semelhante ao primeiro estudo de caso, com a diferença de haver um número bem maior de sinais de controle e parâmetros,

além de uma maior quantidade de *FIFOs* de leitura e escrita nos *multiports*, cujo tamanho máximo do barramento de dados ficou definido em 128 bits.

Para geração do sismograma, objetivo final do algoritmo escolhido para este estudo de caso, as matrizes podem ter um tamanho limite de 2048x2048x2048, gerando uma massa de dados de 32 GB em números de ponto flutuante precisão simples, uma enorme quantidade que não pode ser embutida nas memórias da placa com *FPGA*. Então foi preciso dividir essa grande matriz em partições, cujo tamanho do eixo x é fixado com o máximo possível, e as outras dimensões foram definidas com o tamanho 48, ou seja, a grande matriz de entrada foi particionada em conjuntos menores de matrizes com tamanho 2048X48X48. Esse particionamento está ilustrado na Figura 44 e faz parte do trecho de pré-processamento realizado no *software* da aplicação híbrida local, onde as partições são geradas e gravadas em arquivos temporários, sendo abertos quando for preciso carregá-los no *FPGA*. A ordem de acesso às partições também está indicada na Figura 44.

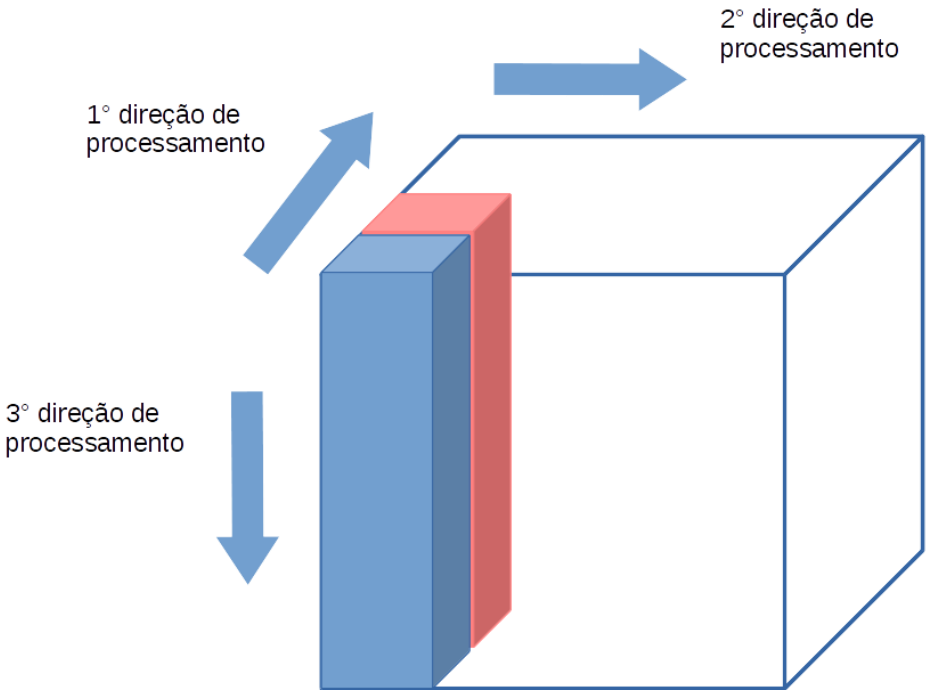

Figura 44: Particionamento da matriz de entrada e direção do processamento.

A integração da aplicação local com os trechos distribuídos foi realizada sem muitos problemas. A única adaptação feita foi a escrita da matriz de velocidades, recebida por cada nó, em um arquivo, de forma a possibilitar o pré-processamento do mesmo.

Resultados

Os resultados analisados seguem os cenários abordados no primeiro exemplo, ou seja, primeiramente comparamos o desempenho geral do *cluster* híbrido em relação ao convencional, e depois analisamos as etapas de processamento de cada sistema. O *RTM* possui uma quantidade muito grande de parâmetros e configurações de entrada, tais como número de *time steps*, tamanho da matriz de velocidades, quantidade de partições, número de tiros, entre outros. Então, para seguir um critério de análise semelhante com o de multiplicação de matrizes, foi fixado um número de *time steps*, cuja influência no aumento de eficiência temporal contribuísse para o desempenho geral da geração de sismograma. De maneira heurística, foi determinada a quantidade de 500 *time steps*, onde é possível verificar essa eficiência temporal, ou seja, comparando a execução simultânea de quatro passos feito pelo *hardware* com a mesma execução de apenas um passo por iteração realizado pelo *software*. O tamanho da partição também foi fixado para o suportado pela arquitetura em *FPGA*, com valor 48. O critério de distribuição consiste em cada nó do *cluster* processando um tiro, então o *cluster* desenvolvido é capaz de executar até três tiros simultaneamente, já que ele possui três nós.

Com relação ao primeiro cenário, o gráfico da Figura 45 mostra o desempenho geral da aplicação híbrida distribuída, comparada com a aplicação executada apenas pelo processador de cada nó. No eixo das abcissas, temos a variação crescente do tamanho da matriz de velocidades, a qual é a entrada do programa. O tamanho dessa matriz deve ser proporcional ao tamanho da partição processada pelo algoritmo no *FPGA*.

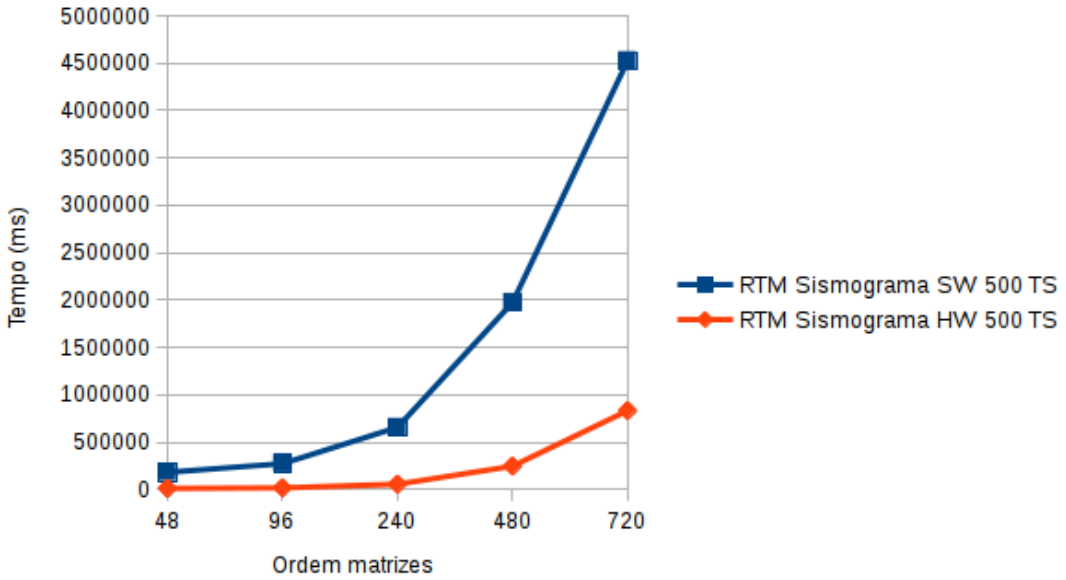

Figura 45: Desempenho do *cluster* híbrido, comparado com o convencional, executando o *RTM*.

Como pode ser visto na Figura 45, o tamanho da matriz de velocidades segue a proporção do tamanho da partição, isto é, o tamanho da matriz é do tipo 2048 x **n***48 x **n***48, onde **n** é uma variável que determina a ordem da matriz. Por exemplo, para **n** igual a 2, a ordem da matriz é 96, e seu tamanho é 2048x96x96.

Podemos observar que a partir da ordem 96, a aplicação híbrida passa a ter maior eficiência em seu processamento, apresentando uma inclinação menor que a de seu equivalente em *software*. A inclinação ainda apresentada na execução híbrida, onde é um pouco mais evidente com o aumento do tamanho da entrada, pode ser analisada de forma semelhante ao primeiro estudo de caso, pois as rotinas de *software* contidas no processamento híbrido influenciam nesse comportamento. Esse aspecto pode ser observado se analisarmos ambas as aplicações de acordo com as mesmas etapas de processamento utilizadas na análise da aplicação de multiplicação matricial: carregamento da entrada, distribuição dos dados, processamento principal, recebimento dos dados processados e formatação da saída. A Figura 46 mostra o desempenho da aplicação em *software*, em cada uma dessas etapas.

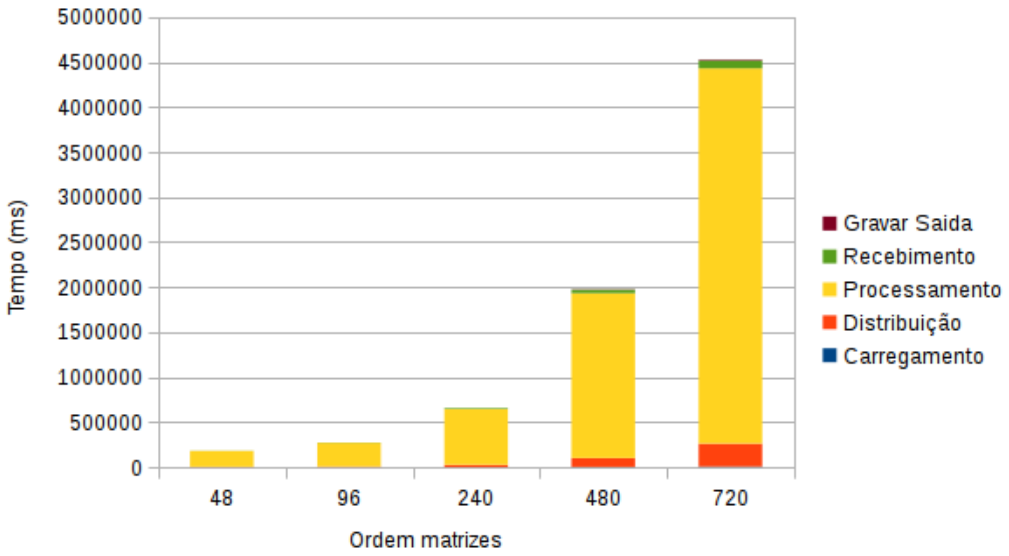

Figura 46: Influência das etapas de processamento do RTM em software.

Semelhante ao primeiro estudo de caso, podemos observar na Figura 46 uma influência maior do processamento principal para o desempenho geral da aplicação. Isso se deve ao fato de a resolução direta da equação da onda a qual resulta na geração do sismograma, constitui numa operação massiva de dados com um grande número de instruções independentes. Logo, o aumento da eficiência do processamento massivo contribui sensivelmente para o ganho geral de desempenho da execução no cluster. A Figura 47 demonstra esse ganho, onde o aumento de desempenho do processamento principal da aplicação híbrida, atribuído ao algoritmo em *FPGA*, implica numa execução geral realizada numa ordem de tempo menor.

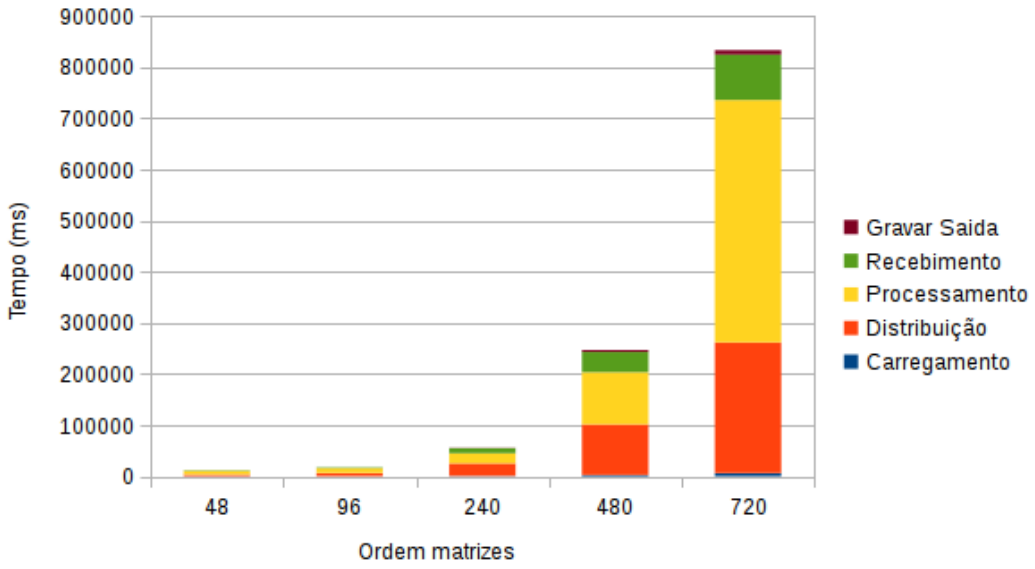

Figura 47: Influência das etapas de processamento do *RTM* híbrido.

Na Figura 47, é possível verificar que, com o ganho de eficiência no processamento principal, proporcionado pela placa aceleradora, as outras etapas de processamento da aplicação híbrida ganham maior influência no cálculo de tempo de execução. Isso se dá devido a essas rotinas, pelo mesmo motivo do primeiro estudo de caso, ainda serem executadas pelo processador. Logo, o processamento em *FPGA* contribui diretamente no ganho de desempenho da aplicação hibrida distribuída. Considerando apenas o processamento principal, a Figura 48 ilustra o desempenho das aplicações convencional e híbrida, como forma de comparar diretamente a capacidade de paralelismo da arquitetura desenvolvida para o coprocessador.

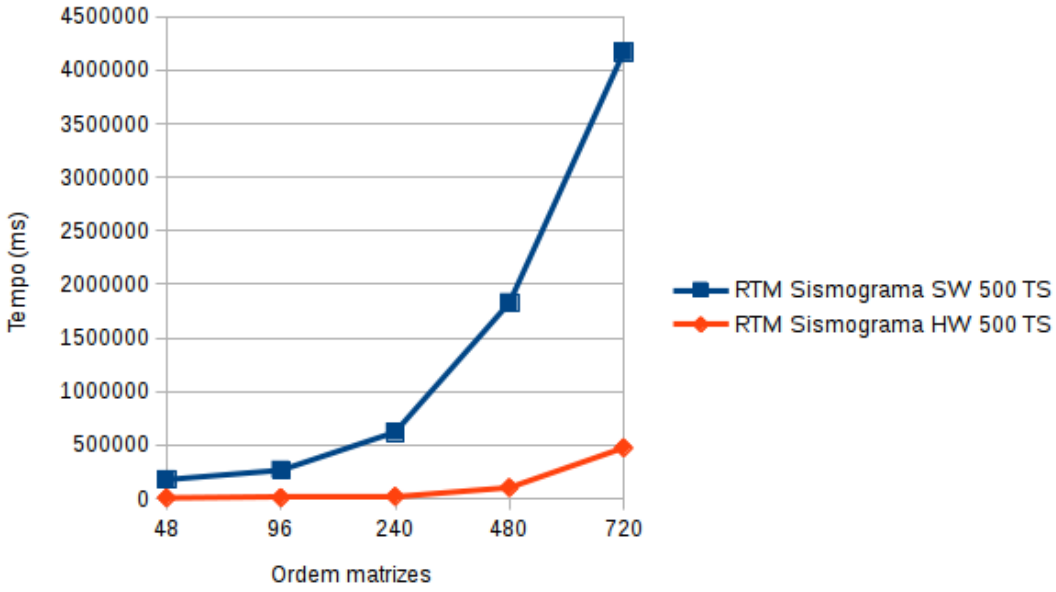

Figura 48: Gráfico comparativo das aplicações em *hardware* e *software* do *RTM*, considerando apenas o processamento principal.

Conforme verificamos na Figura 48, o processamento massivo realizado pelo algoritmo em *FPGA* passa a ter um aumento sensível de desempenho a partir de matrizes de ordem maior que 96, onde a curva do processamento em *software* passa a ser mais íngreme, indicando sua degradação de execução devido à ineficiência de aproveitamento do alto potencial paralelo do algoritmo. O grande número de elementos de processamento massivo do *FPGA* contribui para uma execução mais eficiente, de forma a atender melhor o potencial massivo do processamento *RTM*. A leve inclinação apresentada pela execução em *hardware* é atribuída às instruções essencialmente sequenciais ou das funções auxiliares da aplicação, as quais fazem parte das outras etapas de processamento e são de responsabilidade do processador, onde apresentam uma alta manipulação de dados.

6. Conclusões e Trabalhos Futuros

A metodologia proposta para o desenvolvimento do *cluster* híbrido apresentado nesta dissertação serviu para demonstrar a eficiência de uma arquitetura híbrida, na qual *CPUs* de propósito geral e *FPGAs* aproveitam suas melhores características arquiteturais para aumentar o desempenho computacional em aplicações que requerem um processamento massivo de dados.

Clusters com nós contendo placas aceleradoras, e um método de desenvolvimento hardware/software adequado, certamente permitem um melhor desempenho computacional para grandes sistemas que requeiram um processamento massivo de dados.

Vimos também que os trabalhos semelhantes apresentaram algumas características semelhantes, visando propósitos específicos para seus sistemas. De acordo com essas funcionalidades, o projeto proposto mostrou melhorias nos aspectos de escalabilidade, implantação da estrutura e flexibilidade da arquitetura em *hardware*, deixando apenas de apresentar uma melhor usabilidade de seus recursos, apesar de utilizar tecnologias conhecidas e outras bem documentadas.

Nos exemplos apresentados pode-se observar as características intrínsecas de paralelismo dos *FPGAs* e seu poder de customização de funções complexas. A metodologia aplicada para essa customização paralela de funções aritméticas em *hardware* demonstrou o quão o fluxo de desenvolvimento aumentou a eficiência das aplicações escolhidas como estudos de caso, alcançando um desempenho computacional da ordem de 22x quando comparado a *CPUs* de propósito geral.

Para trabalhos futuros pode-se sugerir o desenvolvimento de uma aplicação híbrida distribuída com o *RTM* completo, adicionando o desenvolvimento da etapa de migração do algoritmo. Também pode-se realizar o experimento com outras placas aceleradoras, como o protótipo criado em parceria com a *DHW e Cenpes/Petrobras*, batizado de Saturno, cuja estrutura é composta por um *FPGA Stratix V* e seis módulos de memória *DDR3* de 8GB, como forma de analisar suas vantagens e desvantagens com relação às placas utilizadas neste trabalho.

Outro ramo de pesquisa bastante interessante seria o uso de linguagens como *OpenCL* para desenvolver os algoritmos massivos nessas ou em outras placas compatíveis, como forma de saber o quanto a linguagem pode ser ou não eficiente em comparação a uma

arquitetura customizada. Também é possível realizar melhorias nos exemplos demonstrados, como o desenvolvimento do algoritmo de multiplicação em placas com mais canais de acesso à memória, ou com *FPGAs* maiores.

Pode-se também investigar o desenvolvimento híbrido de outras aplicações que foram mencionadas nos trabalhos relacionados, as quais pode-se realizar comparações de desempenho e adquirir novos conhecimentos em outras áreas científicas.

7. Referências

ALTERA. Altera Site. Disponível em <http://www.aletra.com>. Acessado em: 31 de Agosto de 2014.

ALTERA PRODUCTS. Ferramentas de desenvolvimento da Altera. Disponível em: <http://www.altera.com/products/ software/ sfw-index.jsp>. Acessado em: 31 de Agosto de 2014.

ARAÚJO, Cristiano. InterfPISH – Uma ferramenta para geração automática de interfaces em hardware/software Codesign. In: CIn, UFPE, 2001. Dissertação de Mestrado... UFPE, 2001. Disponível em: < http://www.cin.ufpe.br/~cca2/phd_bak/documentos/msc_thesis_cca2.pdf>. Acessado em: 31 de Agosto de 2014.

ARAUJO, Rodrigo. Um Cluster De Pc's Usando Nós Baseados Em Módulos Aceleradores De Hardware (FPGA) Como Co-Processadores. In: CIn, UFPE, 2010. Dissertação de Mestrado... UFPE, 2010. Disponível em: <http://www.repositorio.ufpe.br/jspui/handle/123456789/2446>. Acessado em: 31 de Agosto de 2014.

ASANOVIC, K.; BODIK, R.; CATANZARO, B. et al. The landscape of parallel computing research: a view from Berkeley. In: UC Berkeley, 2006. Technical Report no. UCB/EECS-2006-183. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>. Acessado em: 31 de Agosto 2014.

BACKUS, J. Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs. In: Communications of the ACM, 1978. New York, NY, USA. Anais… ACM, 1978. Vol. 21, number 8. p. 613-641.

BELLETTI, F.; GUIDETTI, M.; MAIORANO, A. et al. Janus: an FPGA-based system for high-performance scientific computing. In: Computing in Science and Engineering, 2009. IEEE, 2009. Vol. 11, no. 1, Article ID 4720223. p. 48–58.

CRAY. Cray X2. Disponível em: <http://www.cray.com/Assets/PDF/products/xt/CrayX2Blade.pdf>. Acessado em: 31 de Agosto de 2014.

CUDA. Nvidia CUDA. Disponível em: <https://developer.nvidia.com/cuda-toolkit>. Acessado em: 31 de Agosto de 2014.

DEBIAN. Sistema Operacional Debian. Disponível em: <http://www.debian.org>. Acessado em: 31 de Agosto de 2014.

DONGARRA, J.; STERLING, T.; SIMON, Horst et al. High-Performance Computing: Clusters, Constellations, MPP's, And Future Directions. In: Computing in Science & Engineering, 2005. Anais... IEEE, 2005. Volume 7, number 2. p. 51-59.

DUATO, J.; PENA, A. J.; SILLA, F. et al. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In: International Conference on High Performance Computing and Simulation (HPCS), 2010, France. Anais… IEEE, 2010. p. 224 – 231.

E5645. Intel Xeon E5645. Disponível em: <http://ark.intel.com/pt-br/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI>. Acessado em: 31 de Agosto de 2014.

ECLIPSE. Eclipse IDE. Disponível em: <http://www.eclipse.org>. Acessado em: 31 de Agosto de 2014.

EL-AYAT, K. A.; AGARWAL R. K., The Intel 80386- Architecture and Implementation. In: IEEE Micro, 1985. Anais… IEEE, 1985. Vol. 5, number 6. p. 4-22.

ESPENSHADE, J.; LUKOWIAK, M.; SHAABAN, M. et al. Flexible Framework for Commodity FPGA Cluster Computing. In: FIELD-PROGRAMMABLE TECHNOLOGY, 2009. FPT 2009. INTERNATIONAL CONFERENCE ON, 2009, Sydney, NSW. Anais... IEEE, 2009. p. 465-471.

FLYNN, M. Some Computer Organizations and Their Effectiveness. In: IEEE Transactions on Computers, 1972. IEEE, 1972. Vol. C-21, issue 9. p. 948 – 960.

GEIJIN, R. A. "SUMMA: Scalable Universal Matrix Multiplication Algorithm", In: NASA High Performance Computing and Communications Program's Earth and Space Sciences Project, 1996. Disponível em: < http://www.cs.utexas.edu/ftp/techreports/tr95-13.pdf >. Acessado em: 31 de Agosto de 2014.

GIDEL. Gidel Site. Disponível em: <http://www.gidel.com>. Acessado em: 31 de Agosto de 2014.

GORINO, F., Balanceamento de Carga em Clusters de Alto Desempenho: Uma Extensão Para a Lam/Mpi. Dissertação de Mestrado... Universidade Estadual de Maringa, Paraná, 2006. Disponível em: <http://www.din.uem.br/~mestrado/diss/2006/gorino.pdf>. Acessado em 31 de Agosto de 2014.

HAWICK, K. A.; GROVE, D. A.; VAUGHAN, F. A. Beowulf – A New Hope for Parallel Computing? In: 6th IDEA Workshop, 1999, Australia. pp. 23-30.

INTA, Ra; BOWMAN, D. J.; SCOTT, S. N. The 'Chimera': An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform. In: INTERNATIONAL JOURNAL OF RECONFIGURABLE COMPUTING, 2012, Australia. Anais… ACM, 2012. Article ID 241439, 10 pages.

INTEL. Core i7. Disponível em: <http://www.intel.com.br/content/www/br/pt/processors/core/core-i7-processor.html>. Acessado em: 31 de Agosto de 2014.

KNODEL, O.; GEORGI, A.; LEHMANN, P. et al. Integration of a Highly Scalable, Multi-FPGA-Based Hardware Accelerator in Common Cluster Infrastructures. In: International Conference on Parallel Processing, 42., Lyon, France. Anais... IEEE, 2013. p. 893 – 900.

KNÜPFER, A.; BRUNST, H.; DOLESCHAL, J. et al. The vampir performance analysis tool-set. In: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, 2008, Stuttgard, Germany. Anais… Springer, 2008. p. 139-155.

MARKS, M.; JANTURA, J.; NIEWIADOMSKA-SZYNKIEWIC, E. et al., Heterogeneous GPU&CPU Cluster For High Performance Computing In Cryptography. In: Computer Science, 2012. pp. 63-79.

MITRION. Mitrion User Guide. Disponível em: <http://www.mitrionics.com/index627f.html?page= developers>. Acessado em: 31 de Agosto de 2014.

MODELSIM. Modelsim Site. Disponível em: <http://www.mentor.com/products/fpga/simulation/modelsim>. Acessado em: 31 de Agosto de 2014.

MODELSIM. Modelsim Site. Disponível em: <http://www.mentor.com/products/fv/modelsim/>. Acessado em: 31 de Agosto de 2014.

MPI. MPI Tutorial. Disponível em: <https://computing.llnl.gov/tutorials/mpi/>. Acessado em: 31 de Agosto de 2014.

MPI-3. MPI-3 Reference. Disponível em: <http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>. Acessado em: 31 de Agosto de 2014.

NAKAJIMA, K. Parallel iterative solvers for finite-element methods using an OpenMP/MPI hybrid programming model on the Earth Simulator. In: Parallel Computing, 31., 2005. Jornal... Elsevier, 2005. p. 1048–1065.

NEVES, B. Implementação de um núcleo aritmético para operação de matrizes em sistemas de computação reconfigurável. In: CIn, UFPE, 2012. Trabalho de Graduação... UFPE, 2012.

OPENCL. Altera OpenCL. Disponível em: <http://www.altera.com/products/software/opencl/opencl-index.html?utm_source=altera&utm_medium=press_release&utm_campaign=opencl&utm_content=>. Acessado em: 31 de Agosto de 2014.

OPENMP HOMEPAGE. Site OpenMP. Disponível em: <http://openmp.org/wp/>. Acessado em: 31 de Agosto de 2014.

OPENMP. OpenMP Tutorial. Disponível em: <https://computing.llnl.gov/tutorials/openMP/>. Acessado em: 31 de Agosto de 2014.

OPENMPI. OpenMPI Tutorial. Disponível em: <http://www.open-mpi.org/>. Acessado em: 31 de Agosto de 2014.

OWENS, J. D.; HOUSTON, M.; GREEN, S et al. Graphics Processing Units- powerful, programmable, and highly parallel- are increasingly targeting general-purpose computing applications. In: Proceedings of the IEEE, 2008. Journal… IEEE, 2008. Volume 96, number 5. p. 879 – 899.

PETSC. Biblioteca PETSc. Disponível em: <http://www.mcs.anl.gov/petsc/>. Aceessado em: 31 de Agosto de 2014.

PROCEIII. Gidel PROCeIII. Disponível em: <http://www.gidel.com/PROCe%20III.htm>. Acessado em: 31 de Agosto de 2014.

PROCMULTIPORT. Gidel ProcMultiport. Disponível em: <http://gidel.com/procMultiPort.htm> acessado em 31 de Agosto de 2014.

PROCWIZARD. Gidel ProcWizard. Disponível em: <http://www.gidel.com/ProcWizard.htm>. Acessado em: 31 de Agosto de 2014.

QUARTUS. Quartus II. Disponível em: <https://www.altera.com/download/sw/dnl-sw-index.jsp>. Acessado em: 31 de Agosto de 2014.

RAMOS, Luiz. Proposta e Desenvolvimento de Funções MPI de Comunicação Coletiva Reconfiguráveis. In: PUC-MG, 2004, Belo Horizonte. Dissertação de Mestrado... PUC-MG, 2004. Disponível em: <http://www.biblioteca.pucminas.br/teses/EngEletrica_RamosLE_1.pdf>. Acessado em: 31 de Agosto 2014.

ROCHA, Rodrigo. Desenvolvimento De Uma Plataforma Reconfigurável Para Modelagem 2d, Em Sísmica, Utilizando FPGA's. In: CIn, UFPE, 2010. Dissertação de Mestrado... UFPE, 2010. Disponível em: < http://repositorio.ufpe.br:8080/xmlui/handle/123456789/2826>. Acessado em: 31 de Agosto de 2014.

SADOGHI, M.; LABRECQUE, M.; SINGH, Harsh et al. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. In: PROCEEDINGS OF THE VLDB ENDOWMENT, 2010, Singapore. Anais… ACM, 2010. Vol. 3, number 2. p. 1525-1528.

SANTOS, J. Aplicação do Método de Migração Reversa do Tempo de Dados Sintéticos Utilizando Como Condição de Imagem de Correlação Cruzada. In: Universidade Federal Fluminense, 2011, Rio de Janeiro. Trabalho de Conclusão de Curso… Universidade Federal Fluminense, 2011. Disponível em: <http://www.geofisica.uff.br/sites/default/files/projetofinal/monografia_2-jorge_guida_versao_final.pdf>. Acessado em: 31 de Agosto 2014.

SUN, X., CHEN Y. Reevaluating Amdahl's law in the multicore era. In: Journal of Parallel Distributed Computing, 2010, Orlando, USA. Anais... ACM, 2010. Volume 70, number 2.pp. 183–188.

TESLA. Nvidia Tesla. Disponível em <http://www.nvidia.com.br/object/tesla-servers-br.html>. Acessado em: 31 de Agosto de 2014.

TSOI, K. H.; LUK, W. Axel: A Heterogeneous Cluster with FPGAs and GPUs. In: FPGA'10, 18., 2010, New York, NY. Anais... ACM, 2010. p. 115-124.

ULIANA, D.; KEPA, K.; ATHANAS, P. FPGA-Based HPC Application Design for Non-Experts. In: APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS (ASAP), 2013 IEEE 24TH INTERNATIONAL CONFERENCE ON, 2013, Washington, DC. Anais... IEEE, 2013. p. 261-264.

X8DTG-QF. Supermicro X8DTG-QF. Disponível em: <http://www.supermicro.com/products/motherboard /QPI/5500/ X8DTG-QF.cfm>. Acessado em: 31 de Agosto de 2014.

XILINX. Xilinx Site. Disponível em: <http://www.xilinx.com>. Acessado em: 31 de Agosto de 2014.

XILINX EDK. Site Xilinx EDK. Disponível em: <http://www.xilinx.com/tools/platform.htm> Acessado em: 31 de Agosto de 2014.