IMPLEMENTAÇÃO DE UM CLUSTER HÍBRIDO EXPERIMENTAL PARA PROCESSAMENTO DE APLICAÇÕES DISTRIBUÍDAS.

por

Severino José de Barros Júnior

Trabalho de Graduação

Orientador: Manoel Eusebio de Lima

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CIN - CENTRO DE INFORMÁTICA

GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

graduacao@cin.ufpe.br

www.cin.ufpe.br/~graduacao

RECIFE, FEVEREIRO DE 2012.

Resumo

O projeto descrito neste Trabalho de Graduação consiste no desenvolvimento de um cluster experimental, onde cada nó é composto por uma *CPU* convencional (*PC*) e uma placa contendo um dispositivo coprocessador para processamento de alto desempenho. Este último baseado da tecnologia *Field Programmable Gate Array (FPGA)*, conectado ao host através de uma interface *PCI-e*.

Esse cluster deverá atender a uma demanda cada vez mais crescente nos últimos anos, que são aplicações complexas em áreas como bioinformática, processamento de imagens, modelagem computacional, etc., com processamento massivo de dados. Em geral, estas aplicações são resolvidas por algoritmos que realizam operações massivas com os dados, e cujo potencial de paralelismo contribui sensivelmente para seu desempenho em arquiteturas como *FPGAs*.

Para aproveitar o máximo de potencial fornecido pelo cluster, é preciso conhecer como a aplicação funciona, a fim de identificar as rotinas massivas da mesma. A partir daí, começa-se a realizar as adaptações e implementações necessárias para deixar o *FPGA* executar esses trechos. Essas adaptações devem ocorrer de forma transparente, ou seja, não deve mudar a forma como o usuário utiliza a aplicação. É preciso também definir como os dados serão distribuídos para cada nó, através de um protocolo de comunicação conhecido e largamente usado, o *Message Passing Interface* (*MPI*).

Para estudo de caso e testes dessa infraestrutura, foi escolhida uma aplicação de operação entre duas matrizes densas, cujo resultado também é uma matriz. O processamento será dividido entre os nós do cluster. O resultado final será ajustado em um único cluster onde a aplicação do usuário está implementada.

Agradecimentos

Gostaria de agradecer primeiramente à minha família, que me apoiou e me deu suporte com tudo o que podiam desde o começo de minha vida e foram os primeiros a me incentivar a fazer este curso que estou concluindo. Também agradeço ao Professor Manoel Eusébio por ter me dado essa e outras oportunidades dentro do Projeto HPCIn, por sua compreensão e incentivo a tocar este e outros projetos que virão pela frente. Aos professores de todos os centros os quais passei: CIn, Área II, e CTG. Não tem como falar de todos, até porque são muitos nomes e alguns até não lembro por agora, mas posso destacar alguns que podem representar muito bem a lista: Paulo Borba, Sílvio Melo, Marcília Campos, Eduardo Fontana, Hélio Oliveira e Edna Barros. Mesmo sem serem citados, todos sem exceção contribuíram direta ou indiretamente na minha formação acadêmica e me incentivaram, mesmo de maneira não muito convencional, a concluir o curso. Aos vários professores que também contribuíram em minha formação antes de entrar na universidade, desde minha professora de primeira série, passando por Ana Backer, Teresa, Tetsuo Usui, Gilberto, além dos professores Eduardo Alves e João Tavares, os quais foram os primeiros a apostarem e investirem no meu potencial. Aos amigos e companheiros de curso Bruno Pessôa, João Cleber e João Paulo, os quais me ajudaram a superar cada barreira que aparecia durante o curso, até o fim, além de me aceitarem com todas minhas qualidades e defeitos durante todo esse tempo. Aos meus outros colegas de curso e turma (2005.2), por todos os momentos que passamos e que me acompanharam nessa jornada, principalmente no começo do curso. A todos os amigos e amigas que permanecem em meus círculos, os quais não vou citar nomes, pelos incentivos, compreensão por minhas ausências, e alegrias nos momentos em que estivemos juntos. Aos colegas do HPCIn, que tanto me incentivaram e deram suporte para a conclusão deste trabalho. E por último, mas não menos importante, gostaria de agradecer às duas amigas e ex-chefes Margarete e Gilma, não só pela oportunidade de trabalho com elas, mas também por todo o apoio dado desde o primeiro dia de trabalho até hoje.

Sumário

Índice de figuras

1. Introdução

Ao longo dos anos o avanço tecnológico leva ao surgimento de aplicações cada vez mais complexas. Ao mesmo tempo, cresce a necessidade, por parte das organizações, por sistemas que produzam resultados com o menor tempo possível, ou seja, com o máximo de eficiência em seus processamentos.

Existem diversas áreas de pesquisa onde essas aplicações ocorrem, tais como Geofísica, Processamento de Imagens e Sons, Genética, finanças, entre outras [1][2][15]. Fazendo uma análise mais detalhada desses programas, verifica-se que eles possuem em seus processamentos a manipulação de grandes quantidades de dados, através de rotinas que influenciam sensivelmente o desempenho desses sistemas.

No processamento de tais sistemas, a arquitetura PC é hoje ainda a mais utilizada em todo o mundo . O primeiro dispositivo, ou microprocessador, que possui as características da família de processadores seguida até hoje, chamada de x86, foi o Intel 8086, em 1978 [24]. Essa arquitetura possui três elementos principais: a **CPU** (Unidade Central de Processamento), a qual realiza operações aritméticas e lógicas, tais como adição, multiplicação, lógica *AND*, *OR*, e deslocamento binário; a **memória**, responsável por armazenar tanto os dados a serem operados pela CPU, quanto às instruções do programa em execução; e a terceira parte são os **dispositivos de entrada e saída**, onde propicia ao chip uma comunicação com o meio externo. O modelo de processamento de instruções que incorpora estes três componentes, e que o PC segue até hoje, é o mesmo criado por *John Von Neumann* em 1945, conhecido desde então como Modelo *Von Neumann* [8]. A figura 1 ilustra de uma maneira geral como é estruturada a arquitetura e como seus elementos estão conectados [25].

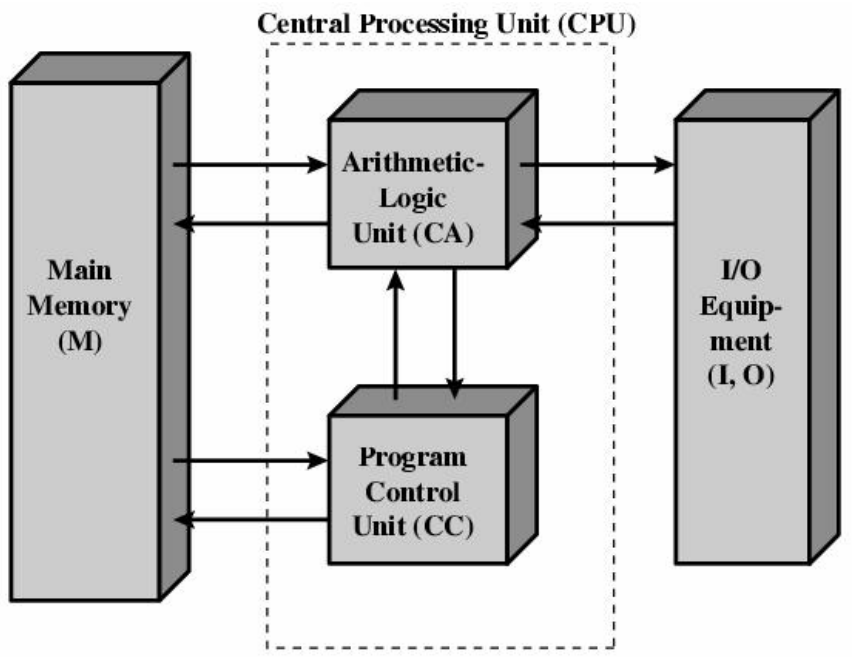

Figura 1- Arquitetura Von Neumann

O PC possui a característica de ter o acesso ao conteúdo na memória de forma sequencial. Isso faz com que as instruções tenham de esperar outras que estão à sua frente na fila de execução para serem processadas. Logo, tarefas as quais não possuem dependência de dados e poderiam ser executadas simultaneamente com as outras já em processamento, acabam ficando ociosas, diminuindo assim a eficiência dos programas. Esse problema é conhecido como o "Gargalo de *Von Neumann*" [8]. Aplicações massivas são as que mais sofrem com este gargalo, pois o alto grau de paralelismo de suas operações, ou seja, alto número de tarefas independentes acaba sendo inexplorado, e assim a eficiência dessa classe de programas fica abaixo do potencial que se poderia atingir.

Com o tempo e desenvolvimento de novas arquiteturas e técnicas de fabricação de circuitos integrados, começaram a surgir novos processadores mais robustos com o intuito de aumentar o desempenho do PC. Uma delas foi a arquitetura *multicore*, a qual é baseada na integração de mais de um processador numa mesma pastilha de silício [20]. Com isso, algumas instruções podem ser executadas de forma paralela. Esta solução, embora tenham permitido um salto significativo no desempenho destas máquinas, ainda apresentam um grau de paralelismo insuficiente em várias aplicações que requerem processamento massivo e paralelo de tarefas. Essas aplicações acabam assim, tendo seu desempenho prejudicado pela necessidade de tornar sequencial a execução de tarefas, por falta de núcleos de processamento paralelo do PC [1][2].

De forma a aumentar ainda mais o paralelismo, as organizações estão adotando a criação de uma associação de PC's, onde cada um pode trocar informações entre eles em uma rede de alta velocidade. Esse tipo de implementação é conhecida como *cluster* [1]. Nesta estrutura, cada PC participante é denominado nó, com funções bem estabelecidas dentro da rede. As aplicações capazes de serem executadas nesses sistemas são chamadas de aplicações distribuídas [1], onde seus dados e instruções são divididos em partes menores e distribuídos entre os nós do *cluster*, para que estes possam processar cada parte recebida, simultaneamente, dando assim, a ideia de um paralelismo explícito, da execução do algoritmo. Normalmente utiliza-se um nó, chamado de servidor que gerencia a massa de dados a ser processada no *cluster*. Em geral, para grandes problemas, estes dados são distribuídos, particionados em estruturas menores e distribuídos entre os outros nós, denominados clientes, para serem processados localmente, em cada um. Em geral, para cada aplicação, cada nó executa o mesmo algoritmo, com uma fatia dos dados do problema. Ao final do processamento, cada cliente envia seus resultados para o nó mestre, que monta o resultado final da aplicação, em função dos resultados parciais gerados em cada nó do cluster. Mesmo assim, a dependência sequencial dos nós acaba não atendendo de forma eficiente algumas dessas classes de programas, criticas em tempo.

Houve também, ao longo dos anos, o desenvolvimento de novos dispositivos, com arquiteturas diferentes das do PC, e com maior poder de processamento paralelo. Dois exemplos bastante significativos são a GPU (*Graphics Processing Unit*) [7] e o *FPGA* (*Field programmable Gate Array*)[3].

A *GPU* é composta por diversos elementos de processamento, em conjunto com uma grande quantidade de pequenas memórias próximas a esses elementos, com uma alta velocidade de acesso aos dados dentro de um barramento especializado. Inicialmente esse dispositivo foi criado para realizar processamento gráfico, mas logo se percebeu que poderiam também aumentar o desempenho dos programas, com razoável grau de paralelismo [7][15]. A figura 2 mostra a estrutura interna de uma *GPU* genérica, comunicando-se com a memória principal de um PC [26]. Estes dispositivos chegam a possuir hoje mais 500 núcleos de processamento independentes, gerenciados por um processador central (árbitro). Cada um destes núcleos de processamento utilizam hoje artimética ponto flutuante, e níveis de cache L1 e L2. No entanto, embora cada componente possa executar threads independente dos demais, resultando em sistema extremamente paralelo e, por conseguinte, de alto desempenho computacional, cada um destes processadores utilizam ainda mesmo paradigma de *Von Neumann* internamente e possuem arquiteturas fixas. Além disso, estas estruturas também

possuem dificuldades de acesso à memória externa, com pequena largura de banda de memória, o que dificulta sensivelmente seu desempenho, na leitura e escrita de dados da aplicação.

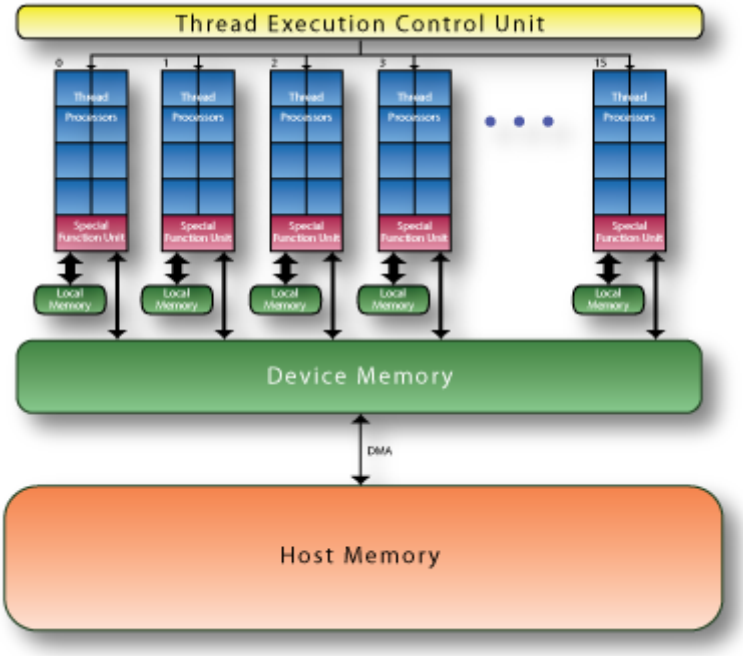

Figura 2- Estrutura de uma GPU

Os dispositivos lógicos *FPGA* por sua vez, não adotam o paradigma de *Von Neumann* para processar dados. De fato, embora alguns destes dispositivos possuam também em seu núcleo *hardcores* de processadores [3], em sua grande maioria a sua área de processamento possui elementos como *LUTs, DSP,* multiplexadores, e um grande malha de roteamento para confeccionar a lógica necessária para implementação do algoritmo da aplicação, a partir destes blocos lógicos. Estes blocos se multiplicam em cada nova versão de *FPGA*, permitindo a confecção de grandes sistemas digitais complexos em um único componente. Estes blocos lógicos são reproduzidos em sua estrutura paralela, onde cada uma delas pode ser customizada para executar uma função específica do algoritmo, diferentes ou iguais, umas das outras, dependendo da aplicação do usuário. Ou seja, estes blocos lógicos implementam os Elementos de Processamento (*PEs*) específicos, para cada nova aplicação, em hardware (*softcores*). Logo, o sistema é totalmente reconfigurável em hardware, permitindo que apenas as operações (mutiplicação, adição, etc.) realmente necessárias à execução do algoritmo sejam instancidas diretamente nos *PEs*. Neste caso não há necessidade de ciclos de captura e decodificação de instrução, etc clássicas do paradgima de *Von Neumann*. Este novo paradigma e dispositivo permite um alto poder de paralelismo, podendo atender à demanda

do processamento massivo de forma mais eficiente que os processadores convencionais e em muitas aplicações *GPUs*. Estes dispositivos também permitem uma largura de banda maior que as *GPUs*, pois possuem várias unidades de acesso direto a bancos de memória externo, além, de possuírem recursos de memória interna também. Detalhes sofre a estrutura do *FPGA* serão mostrados mais tarde.

Ao observarmos mais detalhadamente uma aplicação complexa, verificamos que a mesma possui tanto trechos essencialmente sequenciais quanto trechos massivos de processamento de dados, com característica de paralelismo implicíto. Como o PC tem seu desempenho otimizado para trechos sequenciais, começam a surgir implementações com arquiteturas híbridas, com PC e um dispositivo responsável apenas por operar os trechos massivos(coprocessadores) [2][15]. Assim, pode-se tirar proveito de desempenho nos trechos sequencial e paralelo dos programas, tirando o melhor proveito das arquiteturas existentes.

Este trabalho de graduação tem por objetivo a implementação de um cluster, onde cada nó possui uma arquitetura composta por um PC e um *FPGA* como dispositivo coprocessador. Cada nó deverá trocar informações através de um protocolo especializado em comunicação paralela, o MPI (*Message Passing Interface*) [6]. Como estudo de caso, será utilizada uma aplicação que realiza operação de adição de matrizes densas. Essa aplicação deverá passar por uma série de adaptações para ser executada de forma otimizada nesse cluster, de forma que as mudanças feitas fiquem transparentes ao usuário.

No Capítulo 2 serão abordados os conhecimentos utilizados para o desenvolvimento do projeto. No Capítulo 3, alguns trabalhos relacionados. No Capítulo 4 será mostrada e estrutura e a metodologia aplicada para desenvolver o projeto. No Capítulo 5 analisaremos a aplicação da metodologia num programa de adição de matrizes. E nos Capítulos 6 e 7, algumas conclusões e possíveis trabalhos futuros, respectivamente.

2. Fundamentação Teórica

2.1 MPI (*Message Passing Interface*)

O MPI é uma API que especifica um protocolo de comunicação entre *hosts*, nos quais os dados são passados da memória local para a memória da máquina remota [10][11][12]. Essa comunicação consiste basicamente em sincronização e movimentação de dados, os quais partem do espaço de endereços de um processo para o espaço de outro ou mais deles [11]. Com relação à sincronização, as instruções podem ser classificadas como bloqueantes (o envio de um dado só é concretizado, quando for confirmado seu recebimento) ou não-bloqueantes (o envio pode ser concretizado sem a confirmação do recebimento do dado). Os dois mecanismos são suportados pela API [11][12]. Já a movimentação dos dados é feita em paralelo (paralelismo explícito, ou seja, definido pelo usuário da biblioteca) [10].

Existem alguns tipos de instruções que são utilizados em computação paralela. Um deles é o SIMD (*Single Instruction Multiple Data*), no qual uma instrução pode enviar vários dados de forma simultânea para as unidades de processamento. Outro tipo é o MIMD (*Multiple Instruction Multiple Data*), correspondente ao envio de vários dados a serem executados por diferentes instruções. Também há o SPMD (*Single Program Multiple Data*), que é equivalente ao MIMD, pois cada programa MIMD pode ser, de fato, ioplementado por vários núcleos SPMD (similar ao SIMD, mas não no senso prático). O MPI possui suporte para MIMD/SPMD [11].

O MPI possui implementações para C/C++ e Fortran; possui também *binds* para outras linguagens, como Java e *Python* [13]. Esse protocolo é hoje considerado um padrão de comunicação para aplicações distribuídas, onde a execução de tarefas e operações de controle passam a ser sincronizadas entre os *hosts* participantes de um cluster [11][12].

O MPI tem os seguintes objetivos: fornecer uma topologia virtual (grafo ou cartesiana); sincronizar as mensagens transmitidas; prover mapeamento e garantir funcionalidade dos processos responsáveis pela comunicação entre *hosts*; transparência de linguagem utilizada para desenvolver os aplicativos que irão se comunicar (contanto que sejam suportadas pela API) sem mudanças significativas na comunicação; comunicação síncrona e assíncrona; comunicação ponto-a-ponto ou coletiva (*broadcast*); suporte a paralelismo por meio de *threads* [12].

Existem alguns conceitos que fazem parte do jargão do MPI. São eles [10][12]:

Processo: constitui um programa distribuído, que pode estar num host ou em vários.
Rank: é um número inteiro, usado como identificador único de um processo, atribuído a ele quando o mesmo é inicializado.
Grupo: é um conjunto ordenado de processos, atrelados a um comunicador (cujo padrão é o MPI_COMM_WORLD).
Comunicador: representa um domínio de uma comunicação. São objetos que conectam grupos de processos com identificadores diferentes numa sessão MPI, arranjados numa determinada topologia.
Comunicação ponto-a-ponto: conjunto de operações as quais implementam a comunicação entre dois elementos, podendo ser de forma bloqueante ou não bloqueante.
Comunicação coletiva: reúne funções para comunicação entre vários processos, de um mesmo grupo, simultaneamente.

Existem também os **tipos derivados**. Para realizar comunicação paralela, o MPI define tipos próprios para especificar as mensagens a serem transmitidas. Os tipos pré-definidos são MPI_INT, MPI_CHAR, MPI_DOUBLE. Para transmitir outro tipo fora de um pré-definido, é preciso defini-lo com a função *define_MPI_datatype ()* [12]. A tabela abaixo ilustra os tipos suportados nativamente pelo MPI e seus equivalentes em C [13].

Tabela 1: Tipos definidos em MPI

C Data Types	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

Uma das primeiras implementações em MPI foi o MPICH, desenvolvida inicialmente por Argonne National Laboratory (ANL) e Mississippi State University [12]. Há também a OpenMPI, uma junção de outras implementeções, como LAM/MPI [6]. Outras implementações comerciais são derivadas do MPICH ou do LAM/MPI, como as feitas pela HP, Intel e Microsoft. *Binds* para outras linguagens realizam normalmente chamadas às bibliotecas de C/C++ [12]. As implementações MPICH e OpenMPI possuem suporte para os principais Sistemas Operacionais do mercado, como o Windows e Linux [1][6].

Há ainda duas versões do protocolo MPI: o MPI-1 e o MPI-2. O MPI-1 envolve os conceitos clássicos para comunicação em cluster. Já o MPI-2 envolve esses e outros conceitos, como de gerenciamento paralelo de entrada e saída (MPI-IO), o qual é baseado num conjunto abstrato de funções que auxiliam nas operações de entrada e saída em sistemas distribuídos via MPI, além de facilitar a manipulação de arquivos usando os tipos pré-definidos da biblioteca [12]. Também há o gerenciamento dinâmico de processos, cuja principal funcionalidade é a capacidade de um processo MPI criar novos processos ou estabelecer comunicação com outro processo que foi iniciado em separado [12].

Outro conceito novo é o de *One-side Communication*, responsável por desacoplar a sincronização das funções de envio e recebimento de dados, incluindo sub-processos entre elas, como os de *put, get e accumulate*, de forma a aumentar o controle na manipulação dos dados na memória compartilhada [11][12].

Existe um conjunto de funções básicas para uma aplicação MPI. Elas servem tanto para troca de dados quanto para controle de um processo durante a execução de um programa distribuído, podendo ser bloqueante ou não-bloqueante [11] [13]. São eles:

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** é 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 memoria 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, **processId**). O primeiro argumento, **comm**, indica o grupo 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, as funções utilizadas são [10][11]:

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 o 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** (padrão MPI_COMM_WORLD) 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** (padrão MPI_COMM_WORLD) é o comunicador do grupo de processos a qual eles pertencem e **status** representa a situação da mensagem (para efeitos de controle).

Para comunicações coletivas, as funções utilizadas são [10][11]:

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) – cada processo possui um operador, e todos os dados serão reunidos utilizando esse operador binário, a partir de aplicações sucessivas. 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.

2.2 Dispositivos Lógicos Programáveis (*FPGA*)

Dispositivos lógicos programáveis são caracterizados por terem estruturas já definidas dentro de um chip, que podem ser configuradas para exercerem as mais variadas funcionalidades [14]. Possuem aspectos intermediários entre os processadores de propósito geral e os ASIC's, podendo oferecer o melhor dos dois lados, ou seja, a flexibilidade dos processadores e o alto desempenho dos ASIC's [1]. Sua flexibilidade relecionada a possibilidade destes poderem ser reconfigurados várias vezes, customizado, de acordo com aplicação do usuário [14].

O *FPGA* é um dispositivo lógico programável que possui três estruturas básicas: o CLB (*Configurable Logic Block*), interconexões e blocos de I/O. Os CLB's são encontrados em enorme quantidade dentro da pastilha de silício, são compostos por módulos que implementam lógicas combinacionais, multiplexadores e conexões internas [1][3][15]. Os módulos de lógica combinacional são chamados de LUT's (*Look-up Tables*) e podem implementar, em conjunto com os multiplexadores, qualquer lógica combinacional, dependendo apenas do número de bits de entrada, os quais correspondem ao número de bits de endereçamento. A figura 3 a seguir ilustra os componentes de um CLB [1][3]:

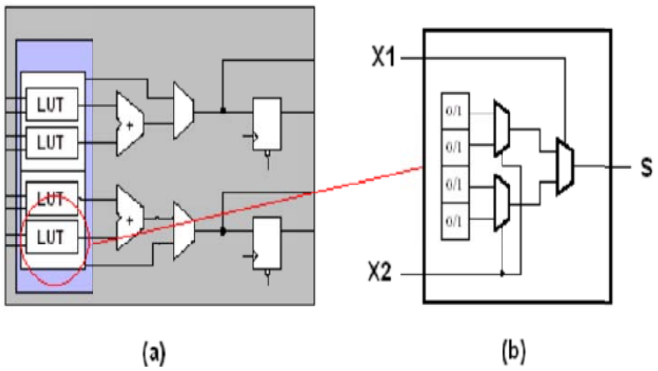

Figura 3: a) CLB b) LUT com duas entradas e uma saída

Os CLB's realizam comunicação entre si através das interconexões, baseadas em estruturas reconfiguráveis chamadas de *switch-matrix*, as quais são compostas por trilhas verticais e horizontais, com chaves que servem para permitir ou não a ligação dessas trilhas.
A figura 4 ilustra um *FPGA* com sua estrutura básica, mostrando com maiores detalhes suas interconexões [1].

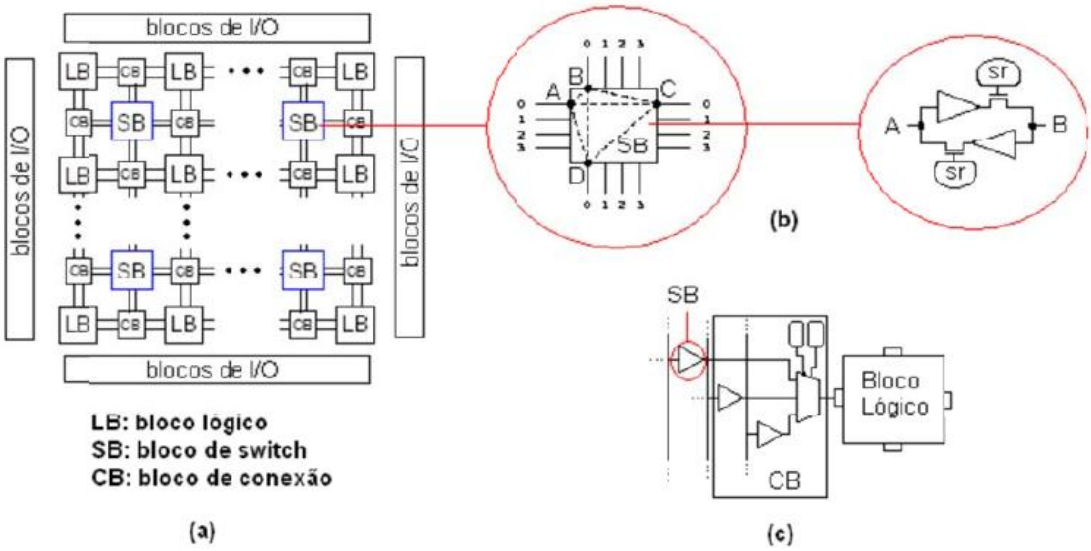

Figura 4: a) FPGA e seus componentes b) Switch-matrix c) bloco de conexão

Uma das vantagens mais importantes do *FPGA* é que todos os componentes são configuráveis, e podem ser reconfigurados de forma total ou parcial [14]. Assim, o *FPGA* é tido como um dispositivo de alta granularidade, podendo ser dividido em mais de uma região, de forma que cada região da pastilha possa realizar uma tarefa distinta, inclusive de forma paralela com relação às outras regiões. Com isso, tarefas que podem ficar ociosas durante a operação sequencial dos processadores, no *FPGA* podem ser executadas paralelamente ao fluxo principal de um algoritmo [15].

O *FPGA* ten sido utilizado em diversas áreas, como processamento digital de sinais, automação industrial, indústrias de automóvel e aeroespacial, telecomunicações, criptografia, medicina, entre outras [1].

Dispositivos *FPGA*s de última geração permitem implementar sistemas digitais de alta complexidade em um único chip. Hoje, fabricantes como Altera [3] e Xilinx [16] fornecem componentes com núcleos internos (*hardcores*) de *CPUs*, blocos *DSP,* memória RAM de alta velocidade (*BRAMs*), *transceivers* para comunicação de alta velocidade (Gbps), controladores de memória (DDR3), alem de disponibilizar *softcores* de interfaces tradicionais como: PCI-e, Ethernet, I^2C, etc. Uma vasta gama de ferramentas para síntese, simulação e testes são fornecidos pelos fabricantes e terceiros para desenvolvimento de sistemas nestes dispositivos.

2.3 Arquitetutas Híbridas

As CPUs de propósito geral levam ainda bastante vantagem em relação a outros dispositivos, em aplicações de natureza sequencial [8][15], devido a paradigmas de programação tradicionais. Nestas arquiteturas, programas com manipulação massiva de dados ou com alto grau de paralelismo acabam tendo seu desempenho degradado devido à ociosidade provocada pela espera na fila de processamento da arquitetura. Apesar de prover um aumento sensível no processamento de grandes quantidades de dados em estruturas *multicore*, a granularidade de operações ainda é muito baixa com relação aos outros dispositivos, como *GPU's* e *FPGA's*. Além disso, o conjunto de operações designadas para cada *core* ainda são executados de maneira sequencial, ou seja, as tarefas ainda perdem eficiência esperando a execução de outras, até o *core* estar disponível para as mesmas [8][15][20].

Em aplicações grandes, há também a implementação de *clusters* de PC's, onde uma aplicação pode ser quebrada em várias menores e distribuída entre os integrantes deste *cluster*. Essa técnica aumenta o grau de paralelismo do processamento nas aplicações, mas ainda com uma granularidade inferior às dos dispositivos reconfiguráveis. Além do mais, o gargalo sequencial ainda permanece presente em cada nó [1][15].

De forma a aproveitar o melhor dos mundos sequencial e paralelo explícito, surgiram as arquiteturas híbridas, as quais são compostas basicamente por *CPUs* convencionais e por

elementos de computação paralela, acoplados a estas, como coprocessador [15][17][21]. O *FPGA* é um exemplo de coprocessador, responsável pelo processamento essencialmente paralelo da aplicação. Nesse tipo de implementação, as *CPUs* ficam responsáveis pelas operações sequenciais do *software*, além de realizar o controle de envio e recebimento de dados para o *FPGA*. Devido a essas funcionalidades, a *CPU* é também chamada de *host* [21]. Já o *FPGA* possui a responsabilidade de executar a parte massiva de uma aplicação, com os dados recebidos do *host*, e devolver o resultado do processamento para o mesmo. O *host* e o coprocessador normalmente são interconectados por um tipo de barramento, como a *PCI*, com *device drivers* específicos [17][21]. A figura 5 ilustra uma arquitetura híbrida típica com uma *CPU* e um *FPGA*, conectados por um barramento PCI-e [21]:

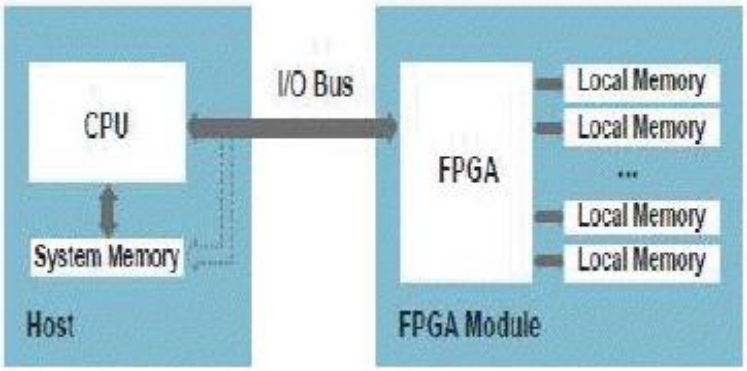

Figura 5: Arquitetura PC + FPGA

2.4 Kit de Desenvolvimento da *Gidel*

O Kit de desenvolvimento da *Gidel* abrange duas partes: uma delas compreende a estrutura física da plataforma, a outra parte engloba as ferramentas em software fornecidas para desenvolvimento [17].

2.4.1 PROCeIII

A PROCeIII é uma placa PCI-e que contém uma *FPGA* e vários outros componentes de *hardware*, que incrementam os recursos fornecidos pelo dispositivo reconfigurável. O *FPGA* integrado à placa é um Stratix III, da Altera [17][18], cuja frequência de operação pode chegar a 300MHz [15][18]. Como periféricos, um dos recursos relevantes é um módulo de memória RAM DDR2 de 512 MB, com taxa de transmissão de até 4GB/s e 64 bits de barramento [15][17]. Possui também dois slots de memória DDR2 que suportam até 8GB de capacidade [17]. Além disso, seu barramento PCI-e faz parte do padrão 4x, suportado por grande parte das placas-mãe atuais, e também possui 32 canais DMA, o que permite a placa acessar o barramento PCI em modo *master* [15]. A figura 6 mostra a placa PROCe III [17]:

Figura 6: Placa PROCeIII da GIDEL

2.4.2 *ProcWizard*

O conjunto das ferramentas em *software* fornecida pela *Gidel* fazem parte do *ProcWizard*. As ferramentas englobam: *driver* para comunicação entre a placa *PCI* e o PC, e

o ambiente de desenvolvimento para configuração, este por sua vez possibilita a configuração do *FPGA* da placa, integrando os módulos com os fornecidos pelo fabricante, além de integrá-lo à parte de *software* da aplicação [5][15].

O *ProcWizard* trabalha em conjunto com a placa, realizando algumas configurações automáticas, e gerando automaticamente código HDL e C/C++, de forma a agilizar o desenvolvimento *hardware*/ *software co-design* [5]. A ferramenta permite que o desenvolvimento de hardware e o de software ocorra paralelamente, de forma que as duas equipes responsáveis por cada um possam trocar informações durante esse processo [5].

As principais características do *ProcWizard* são:

Integração automática de *software* e *hardware*:

A ferramenta *ProcWizard* realiza geração automática de código HDL e C/C++ (e também sua documentação), permitindo que as equipes de *hardware* e *software* compartilhem da mesma informação e interface [5]. A geração de código HDL tem o objetivo de realizar a integração do módulo desenvolvido pela equipe de hardware, dos *IP-cores* fornecidos pela Gidel, e toda a periferia necessária para comunicação com o host [5]. Já a parte do *software* é responsável por gerar um driver em C/C++, que será usado pela aplicação executada no PC, através de uma API, chamada *ProcAPI*, cujas funções realizam o controle de inicialização e fluxo de dados da placa *PCI* [5]. A figura 7 seguinte retrata uma visão geral da integração hardware/ software proporcionada pelo *ProcWizard* [5].

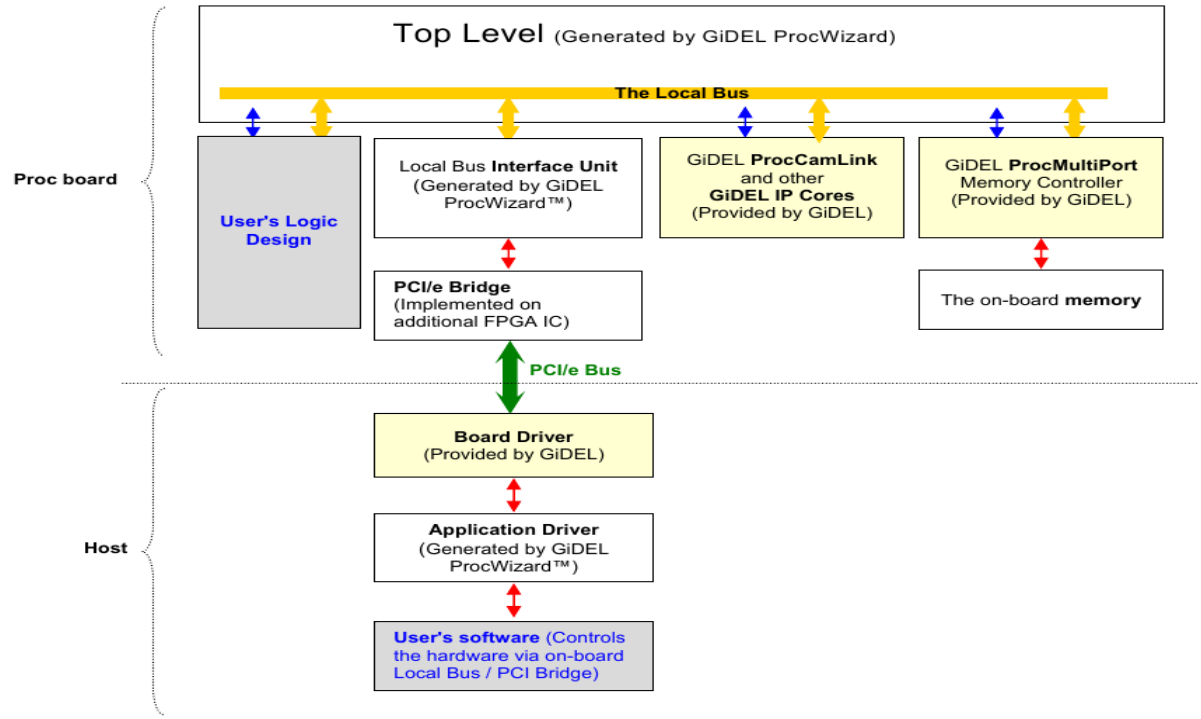

Figura 7: visão geral do sistema integrado na placa

Geração de código HDL:

Como descrito anteriormente, seu objetivo é proporcionar a integração do módulo criado pela equipe de *hardware* com os *IP-cores* fornecidos pelo fabricante, e toda a periferia da placa responsável pela comunicação externa [5]. Essa integração se dá através da geração de um *top-level* em uma linguagem de descrição de hardware como AHDL, VHDL ou Verilog [5].

Um dos *IP-cores* mais importantes do sistema é o ***ProcMultiport***, fornecido pela *Gidel*, o qual implementa uma interface simples entre a memória DDR 2 (também outros tipos de memória) e o módulo *HDL* criado pela equipe de desenvolvimento. Ele também elimina a possibilidade do módulo estar obsoleto devido ao uso de outro tipo de memória, além de diminuir os custos de área, caso ocorra a adição de mais módulos de memória externa [15][23].

O *ProcMultiport* realiza a conversão de uma memória *on-board* para uma memória *multiport*, podendo ser acessada por até 16 portas simultaneamente. Cada porta trabalha com seu domínio de *clock*, dependendo do barramento de dados trafegados por ela [23]. Os modos de acesso aos dados fornecidos pelo *IP-core* são: randômico, sequencial e segmentado. Em

geral, em aplicação do tipo *stream processing*, por questões de desempenho, o seqüencial é o mais recomendado, desde que ele permita a transferência dos dados da memória através de rajadas (blocos de palavras) ou *bursts* [15][23]. A figura 8 ilustra como é a estrutura interna desse *IP-core* em modo sequencial. Observa-se que cada porta (leitura ou escrita) possui *FIFOs* de sincronização, com seus respectivos sinais de controle para o correto gerenciamento dos dados entre o módulo criado e a memória [15][23].

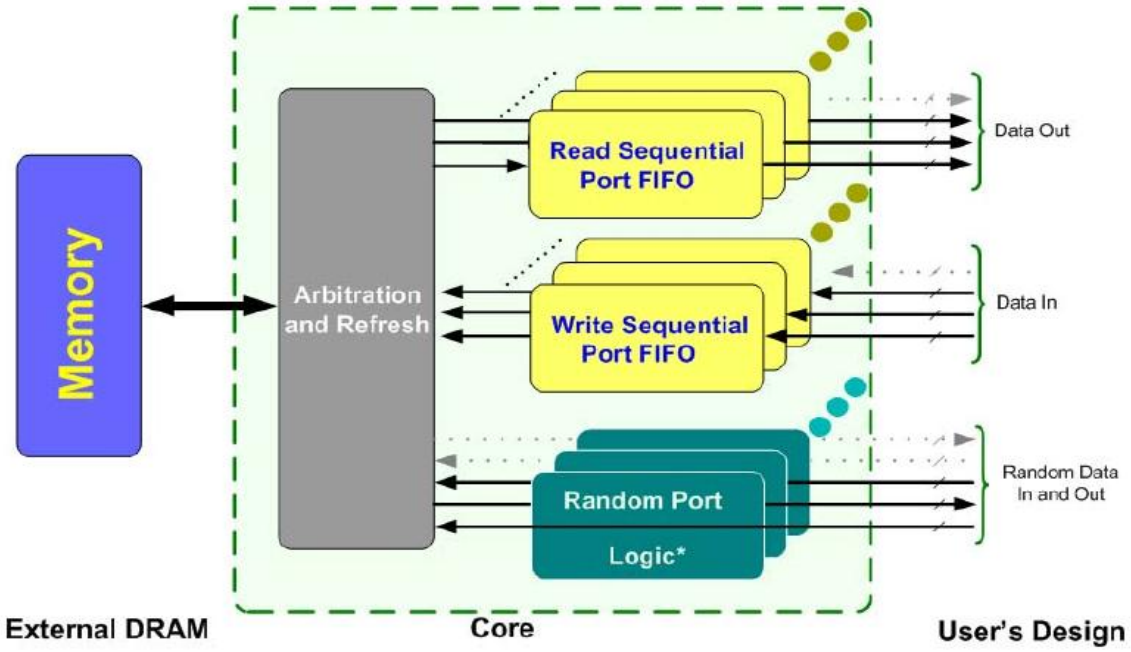

Figura 8: estrutura de blocos do ProcMultiport

Geração de projetos externos:

Nesta ferramenta, tanto o código C/C++, quanto o *HDL*, uma vez gerados, são referenciados e tratados incialmente em ferramentas especiais e posteriormente integrados na plataforma. Os módulos de *software* são agregados a um projeto para o Visual Studio [22], com as configurações adequadas para o compilador. Já os módulos de *hardware* são agregados a um projeto no Quartus II [3], com todas as configurações e restrições necessárias para a síntese do *top-level* e dos submódulos [5].

Geração de driver C/C++:

O *driver* gerado pela ferramenta é basicamente uma classe, com todos os métodos necessários para gerenciamento da placa e comunicação com o *FPGA* (consequentemente com o módulo *HDL* integrado contido nele) [5]. Uma de suas principais funcionalidades é a inicialização da placa, através do carregamento do *bitstream* sintetizado a partir do projeto no Quartus, e configuração dos *clocks* da placa *PCI*. *Bitstream* é código binário que configura todos os blocos lógicos da *FPGA*. O *device driver* também provê manipulação dos dados utilizados pela plataforma, além de fornecer interfaces para diversos serviços, tais como gerenciamento de DMA, interrupções, entre outros. Todas essas funcionalidades possuem interface com o *software* através da API *ProcAPI* [5].

Geração de documentação:

O *ProcWizard* gera um *template*, em forma de documento de texto ou em HTML, com uma descrição dos módulos, *clocks*, entidades desenvolvidas, configurações feitas, etc.. O objetivo é facilitar a escrita da documentação do projeto [5].

Depuração de hardware:

O ambiente do *ProcWizard* possui algumas características as quais facilitam na depuração do projeto da placa, tais como: carregamento rápido do projeto na placa, permitindo maior rapidez nos testes; configuração das frequências de operação dos blocos lógicos (*FPGA*) da plataforma; acesso à memória e aos registradores, além de acesso aos dados do aplicativo. Por fim, é possível ainda se criar macros ou programa especiais de teste para pequenas tarefas durante a depuração [5].

3. Trabalhos Relacionados

3.1 "Um Cluster De Pc's Usando Nós Baseados Em Módulos Aceleradores De Hardware (*FPGA*) Como Coprocessadores." Rodrigo Araújo, 2010

Nesta dissertação de Mestrado, Rodrigo Araújo [1], desenvolveu um *cluster* híbrido experimental, formado por 2PC's, cada um contendo uma placa com um *FPGA* para acelerar operações massivas de dados. A comunicação entre os *hosts* foi feita através do protocolo MPI, o qual possui a função de distribuir os dados particionados, recuperar e retornar os resultados parciais para o *host* responsável por fornecer o resultado final para o usuário. Essas operações devem se comportar de forma semelhante àquela composta com os *hosts* contendo apenas PC's, ou seja, as mudanças devem ocorrer de forma transparente para o usuário [1].

A interface de rede da aplicação é composta por funções que implementam o protocolo *MPI*, utilizado como padrão de comunicação em *clusters* comerciais. Uma das vantagens de utilização desse protocolo é seu fácil uso, visto que sua *API* é relativamente simples e direta. Outra vantagem é com relação à portabilidade da biblioteca, como já mencionado, podendo o código ser compilado tanto em *Windows* quanto em *Linux* [1][6]. A implementação escolhida para o projeto foi a *MPICH2* [31].

O conjunto de funções *MPI* que foram utilizadas nesse projeto caracterizam o uso de um protocolo ponto-a-ponto e bloqueante, ou seja, como já mencionado, significam que o envio e recebimento de dados são feitos de forma direta e a função só é liberada de sua execução quando recebe uma resposta de recebimento, por parte do seu outro par de comunicação. Sendo assim, as funções utilizadas foram: *MPI_Init, MPI_Send, MPI_Recv, MPI_Comm_size, MPI_Comm_rank* e *MPI_Finalize* [1]. As explicações de como cada uma dessas funções trabalham estão na sessão sobre *MPI* (sessão 2.1).

Quanto à estrutura de cada nó, a mesma é composta por uma CPU convencional e um *FPGA* como coprocessador, para aceleração de algoritmos massivos em dados. Nesse projeto, a placa com *FPGA* escolhida foi a plataforma ML555, da *Xilinx*, a qual possui as seguintes características [1]:

- Modelo do *FPGA*: Virtex 5 XC5VLX50T da *Xilinx*;
- conectores *PCI, PCI-X* e *PCI-Express* 8x (o qual foi utilizado no projeto);

- *socket SODIMM* de 200 pinos com uma memória DDR2 SDRAM 256MB;
- três fontes de *clock* na placa, duas entradas diferenciais SMA de *clock* e dois sintetizadores de *clock* programáveis;
- permite programar o *FPGA* via plataforma *Flash* (contida na placa) ou via cabo USB;
- *switches* e *LED's* que podem ser controlados pelo usuário.

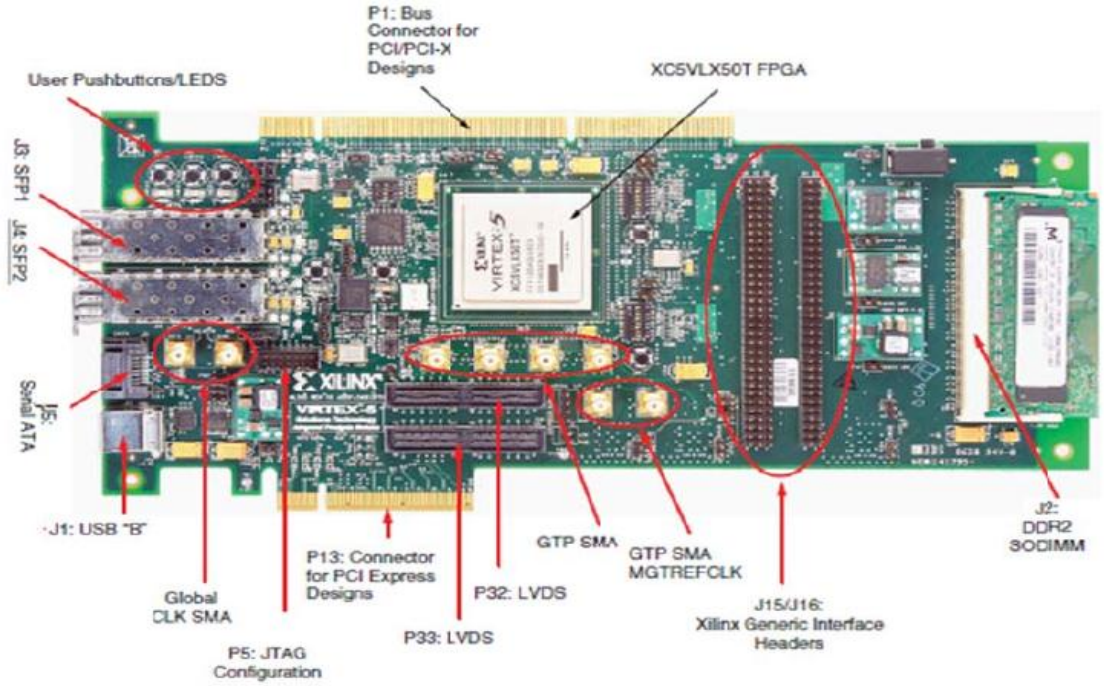

Figura 9: Plataforma ML555, da Xilinx

Essas características possibilitam a transmissão de dados em formato *burst* [1]. A figura 9 ilustra a placa utilizada.

De forma a servir como interface entre o *host* e o algoritmo implementado na *FPGA*, foi preciso utilizar um *IP-core* fornecido pela *Xilinx*, chamado de ***Core PCIe***. Este *core* foi gerado pela ferramenta *Core Generator*, da própria fabricante da placa . O ***Core PCIe*** implementa todo o protocolo do barramento *PCI-e*, com a característica de transmissão rápida de dados entre o *host* e o *FPGA*, baseado no uso de canais de *DMA*. Todos os registradores de controle e status do barramento *PCI-e* são mapeados no controlador de *DMA*, onde podem ser usados pelas chamadas de sistema da aplicação. O processador é responsável por iniciar e verificar o término das operações de *DMA*, através dos registradores de controle e status, correspondentes no controlador de *DMA* [1].

Também foi desenvolvido um *device driver* para o Sistema Operacional Linux, cujo objetivo foi possibilitar o fluxo de dados, em modo *burst*, entre o *PC* e a placa com o *FPGA*

[1]. Foram utilizadas duas estruturas padrão do Linux para o *driver*: o "*char driver*" e o "*pci driver*". Através delas, o *driver* realiza leituras e escritas de dados de forma simples, por meio das funções *read* e *wirte*, respectivamente [1].

Como estudo de caso, foi escolhido implementar uma das funções do *BLAS* (*Basic Linear Algebra Subprograms*), uma biblioteca conhecida que realiza cálculos entre vetores e matrizes e é utilizada em aplicações científicas. Essa biblioteca possui três níveis de abstração: o primeiro engloba operações entre vetores; o segundo, entre vetor e matriz; e o terceiro é responsável pelas operações entre matrizes. O nível escolhido para o projeto foi o terceiro, com foco na multiplicação de matrizes densas [1]. O ambiente físico de testes foi composto por dois PC's com processadores *Intel Core* 2 Quad de 2.66 Ghz e 4GB de RAM. Um deles foi escolhido para ser o Mestre da rede, enquanto o outro seria o escravo. Ambos os *hosts* possuíam uma placa *ML555*, já descrita anteriormente. A operação do *BLAS* escolhida para multiplicação de matrizes é do tipo:

$$\mathbf{C = \alpha AB + \beta C}$$

Onde A, B e C são matrizes, α e β são inteiros. Para esse estudo de caso, foi considerado que α = 1 e β = 0. O Algoritmo para divisão dos dados utilizado, para futura distribuição entre os nós, foi o *SUMMA* (*Scalable Universal Matrix Multiplication Algorithm*) [1]. A aplicação desenvolvida apresentou-se de forma que todas as mudanças da biblioteca ficaram transparentes para o usuário. A comunicação dos dados na rede se deu através do protocolo MPI, utilizando a biblioteca MPICH2, já referida anteriormente. O algoritmo implementado para realizar as operações densas de multiplicação dentro da *FPGA* foi resultado de um outro projeto de mestrado, que fazia parte do grupo HPCIn assim como o projeto desenvolvido por Rodrigo. Ele implementa a operação C = A . B, com A, B e C sendo matrizes densas [1].

Como resultados, temos que a arquitetura proposta pode ser usada de forma simples, através do comando *mpirun* [1][6]. Com relação ao desempenho, foram realizados testes com diversos tamanhos de matrizes, utilizando três plataformas distintas: um *cluster* com apenas PC's, um único nó com uma *FPGA* e um *cluster* com *FPGA* em cada nó. Os resultados mostraram que houve um aumento de desempenho de um cluster híbrido, em relação a um *cluster* só de PC's, a partir da utilização de matrizes de 220x220 como entradas.

Conclusões:

Este projeto se assemelha à proposta deste Trabalho de Graduação (TG). No entanto, as arquiteturas dos componentes reconfiguráveis (*FPGAs*), o ambiente de desenvolvimento e o estudo de caso são diferentes. Nesta dissertação o elemento acelerador é uma *FPGA* da *Xilinx*, enquanto que neste TG é utilizada uma plataforma fornecida pela empresa *Gidel* [4], com uma *FPGA* da Altera. Neste novo ambiente todos os *device drivers* para reconhecimento do *PC* e chamadas do sistema foram incorporadas automaticmante por ferramentas de desenvolvimento fornecidas pela *Gidel* (*ProcWizard* e *ProcAPI*), o que acelera bastante Todo o fluxo de desenvolvimento da aplicação.

3.2 "*Reconfigurable Supercomputing With Scalable Systolic Arrays And In-Stream Control For Wavefront Genomics Processing.*" C. Pascoe, A. Lawande, *et al*

Este trabalho trata do desenvolvimento de algoritmos utilizados na área biológica para serem executados num *cluster* híbrido, batizado de **novo-g**, localizado na Fundação *CHREC*, na Flórida [2]. Esse *cluster* é baseado num conjunto de 24 *PC's* com processadores *Opteron* de 2,4 GHz. Cada nó do *cluster* possui duas placas *Gidel ProcStar III* [4], com quatro *FPGA's Stratix III* da Altera em cada uma [2].

O algoritmo implementado em hardware reconfigurável é do tipo *array* sistólico escalável, o qual é baseado em um fluxo de dados com um elemento de controle, vários elementos de controle menores, ou uma mistura dos dois [2]. Para alinhamento de *DNA*, o trabalho descreve que essa estrutura é a melhor possível para resolver o problema, mas requer uma unidade de controle complexa, com pequenas máquinas de estado por elemento de processamento (PE), além de mais registradores, contadores e sinais de controle, consumindo uma grande área do *FPGA* [2]. Para reduzir a complexidade, é proposto um método de controle *in-stream*, o qual se constitui em substituir máquinas de estado complexas e sinais de controle, com lógica especial de controle inserido diretamente no conjunto de dados. Então palavras de controle que se encontram misturadas aos dados vindos da aplicação são usadas para controlar a transição de estados nessa máquina de estados, ou de coordenar outras ações durante o processamento do fluxo [2]. Com isso, ocorre uma aceleração na execução dos algoritmos de alinhamento genético, ao custo de ocupar maior área no dispositivo reconfigurável.

Os *arrays* sistólicos são normalmente implementados em *pipeline*, formando uma fila de *PE's*, onde cada um é responsável por uma coluna de uma matriz de escore, gerada por algum modelo de programação dinâmica [2]. Neste trabalho foram implementados três algoritmos, *Needleman-Wunsch* (NW), o *Smith-Waterman* (SW) e o *Needle-Distance* (ND). A figura 10 mostra a arquitetura do algoritmo em *FPGA* para a implementação do *Smith-Waterman*.

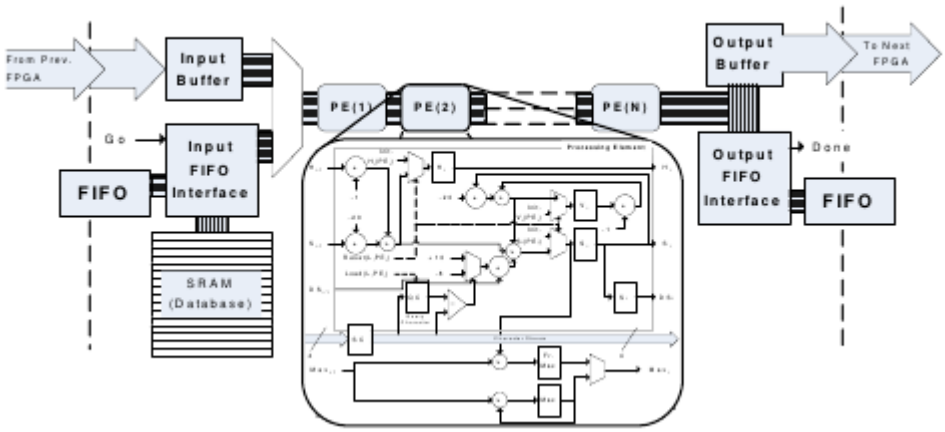

Figura 10: estrutura de *array* sistólico para o *Smith-Waterman*

Para avaliar o desempenho em sistemas distribuídos, dos algoritmos descritos anteriormente, foram implementados na arquitetura do *novo-g*. Cada um dos algoritmos foram implementados de forma a ocupar a melhor área possível no *FPGA*, que permitisse o maior desempenho no processamento dos dados. Os testes foram realizados em um único nó do *novo-g*, em uma única placa *Stratix III*. Os algoritmos foram também implementados e executados em software, em um nó do cluster. Comparações foram feitas quanto ao desempenho das versões implementadas, puramente em software e no modelo híbrido. Os resultados obtidos, considerando as aplicação em um único *host*, mostraram que a versão híbrida alcançou desempenho superior (*speedup*) da ordem de 3000 em relação a versão em software.,

Conclusões:

O projeto descrito nesse artigo possui algumas semelhanças com a arquitetura proposta neste Trabalho de Graduação. Uma delas é o uso do ambiente da GIDEL, com a diferença na placa PCI que foi usada por eles. Outra diferença é que no estudo de caso foi

utilizado apenas um nó do cluster, logo não foi implementado de fato um sistema distribuído com protocolo MPI, suportado *a priori* pelo novo-g.

4. Arquitetura Proposta

4.1 Estruturas de *Hardware* e *Sofwtare*

O cluster implementado é composto por nós híbridos, cada um composto por um *PC* e uma placa *PROCeIII* da *Gidel* [17], já mencionada anteriormente. Os nós são ligados em uma rede *ethernet* cabeada, de forma a aproveitar a infraestrutura de rede já existente no laboratório HPCIn, onde foi realizado o experimento. As máquinas realizam a troca de dados através do protocolo *MPI,* logo a aplicação distribuída deve implementar as chamadas do *MPI*, a fim de ser compatível com os padrões já estabelecidos em aplicações comerciais e científicas. A estrutura do *cluster* e da aplicação em detalhes será mostrada mais adiante.

A aplicação distribuída é composta por dois tipos de processo: o processo servidor e os processos clientes. O servidor é responsável por carregar os operandos que estão em arquivos de texto, particionar esses operandos transformando-os em blocos parciais, envia-los aos clientes via protocolo *MPI*, aguardar o processamento de cada bloco, receber os resultados parciais e reunir esses dados para gerar o arquivo de saída do sistema. Os clientes, por sua vez, possuem as tarefas de receber os operandos parciais vindos do servidor, processar esses dados e enviar o resultado parcial para o servidor. A figura 11 apresenta uma visão geral do *cluster* e de uma aplicação distribuída.

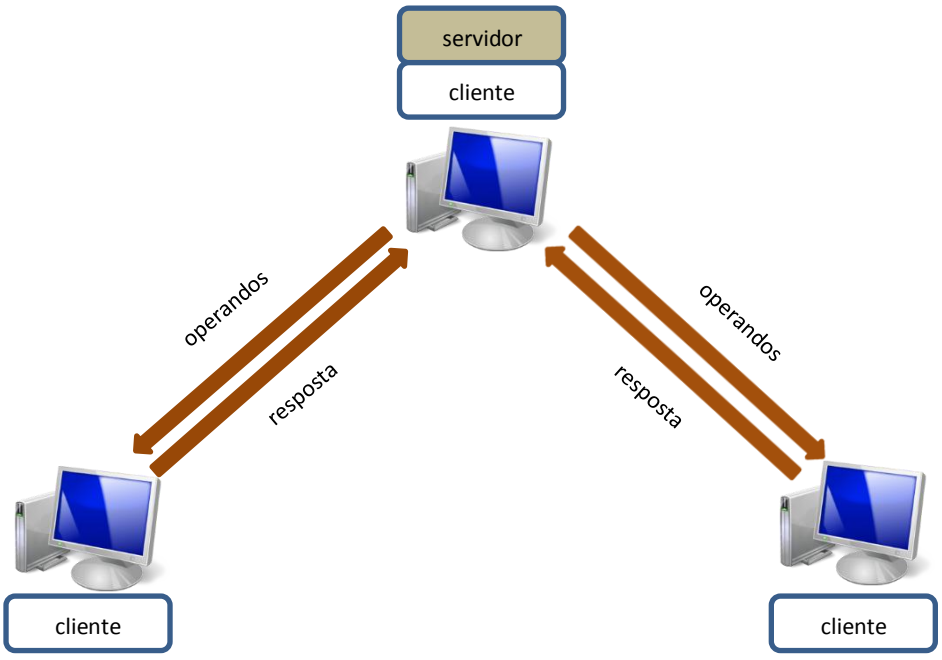

Figura 11: estrutura geral do cluster híbrido

Como podemos ver na figura 11, os nós possuem funções bastante definidas. Um deles é responsável por executar o processo servidor e um cliente, logo também processa uma parte dos dados; os outros são apenas incumbidos de executarem os clientes. O processo servidor é definido com um identificador (*rank*) de valor zero, os clientes possuem *ranks* de números a partir do um. Então, o servidor é o responsável por inicializar e terminar a aplicação distribuída.

Cada nó deve executar um processo local, o qual é composto pelas rotinas de *software* e com chamadas de rotinas implementadas em *hardware*. Esta última para o processamento massivo de dados. Como o processo servidor deve conter apenas instruções de controle e manipulação de arquivos, suas tarefas devem ser executadas apenas em *software*. Então, o processamento híbrido está apenas no processo cliente, ou seja, é no cliente que se encontram as rotinas em hardware.

No processo cliente, os trechos os quais devem ser executados pelo *FPGA* são aqueles que possuem um alto grau de paralelismo, de forma a aproveitar melhor o potencial do dispositivo reconfigurável. Assim, espera-se obter uma aceleração na execução da aplicação local como um todo, visto que a parte sequencial da mesma é executada no processador local, otimizado para essa tarefa, e a parte massiva em uma estrutura desenvolvida especialmente para a aplicação e implementada no *FPGA*. A figura 12 retrata a estrutura de um processo cliente, indicando o processamento das duas rotinas.

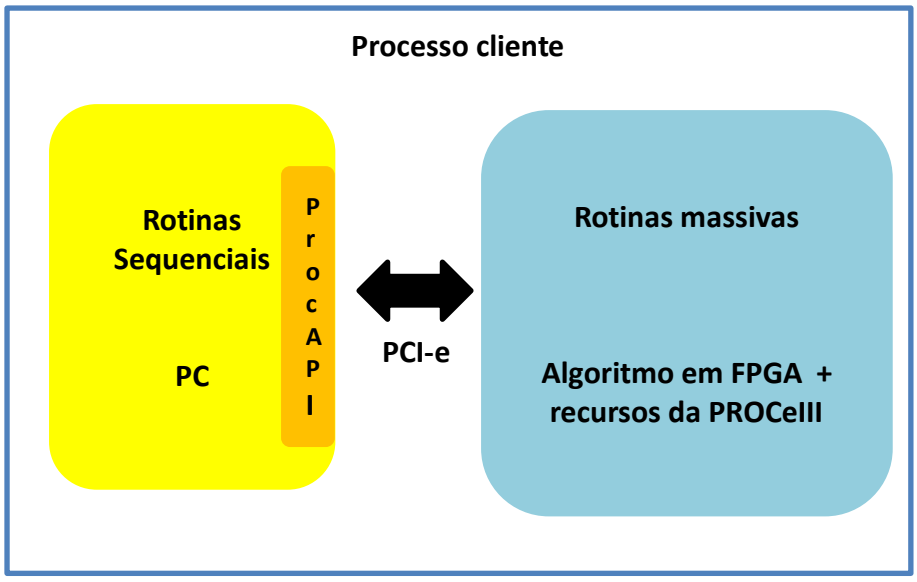

Figura 12: estrutura de um processo cliente

Na figura 12 percebemos claramente onde cada trecho do processo será executado. As rotinas do *software* são formadas pela parte sequencial da aplicação, a qual não sofre alteração com relação à aplicação original, além das funções que fazem parte da *ProcAPI*, onde estão contidas as instruções de controle da placa e de troca de dados através de chamadas *DMA* para o barramento *PCI-e*, já explicadas anteriormente. Essas funções deverão atuar durante a execução da função massiva da aplicação local, substituindo as rotinas internas anteriores, pois estas agora fazem parte do algoritmo implementado no *FPGA*.

Ao mesmo tempo, a função adaptada deve manter seu cabeçalho e funcionalidade inalteradas, de forma a deixar transparentes as mudanças internas para o usuário.

O algoritmo massivo desenvolvido para a *FPGA* deve ser projetado de forma a aproveitar ao máximo o paralelismo das rotinas e reuso de dados, sendo então um *hardware* criado unicamente para uma dada aplicação. Normalmente a estrutura desse *hardware* é composta por várias unidades de processamento, responsáveis por realizar o processamento do dado e com funcionamento paralelo entre elas, além de uma unidade de controle, cujas funcionalidades são de controlar todo fluxo de dados necessário para alimentar o algoritmo, recepção e transmissão do resultado do algoritmo para o PC através de uma interface apropriada. Essa unidade de controle geralmente funciona como uma máquina de estados, que também gerencia todos os atrasos provenientes do fluxo que os dados farão dentro do algoritmo, possíveis reusos de dados para prover processamento contínuo e otimizado, além de controlar as unidades de processamento, fazendo com que elas funcionem em sincronia com os dados.

O algoritmo desenvolvido, núcleo da aplicação, não é a única estrutura presente no dispositivo reconfigurável. De fato, ele faz parte de um conjunto de outros módulos cuja geração e integração com o algoritmo são feitos no ambiente *ProcWizard* da Gidel. Assim, além do algoritmo, é sempre incorporado ao código executável do *FPGA (bitstream),* os códigos (*cores*) da interface *PCI-e* e o do controlador de memória *DDR2*. A interface com essa memória é implementada através do *IP-core ProcMultiport*, cuja explicação pode ser vista na sessão 2.4.

A Figura 7, também contida nessa sessão, ilustra como o algoritmo é integrado dentro da *FPGA*. Além disso, a integração do algoritmo com o software também é feita no *ProcWizard* através da geração de uma classe derivada da classe *Proc*, sendo todas as configurações de inicialização e de interface com a placa geradas de acordo com as configurações de projeto feitas pelo desenvolvedor [5].

Portanto, o *cluster* desenvolvido permite acelerar as aplicações complexas, principalmente com relação às suas rotinas massivas, pois estas últimas são processadas diretamente em hardware. Para aproveitar todo o potencial desse *cluster*, a aplicação precisa passar por algumas adaptações, preservando o mesmo comportamento do programa original. O ambiente fornecido pela *Gidel*, o *ProcWizard*, auxilia nesse processo de adaptação, facilitando a integração *hardware/ software codesign*, gerando automaticamente as interfaces necessárias para que o software possa trocar informações com o algoritmo em *hardware*, utilizando os recursos existentes na placa *PCI* escolhida para o projeto.

4.2 Metodologia Aplicada

O projeto apresentado neste Trabalho de Graduação foi realizado com a aplicação de um fluxo de desenvolvimento compatível com a estrutura oferecida e com a classe de aplicações almejada. As etapas adotadas são compatíveis com uma metodologia *hadware/ software codesign,* com módulos sendo executados em software (*CPUs* convencionais), módulos implementados, como chamada de sub-rotinas, em *hardware (FPGA)*, conectados via um a interface de comunicação rápida *PCI-e*. O uso do ambiente proporcionado pela ferramenta *ProcWizard* foi importante para agilizar o trabalho de integração entre *hardware* e *software*, reduzindo significantemente o tempo de desenvolvimento deste projeto. A figura 13 mostra o fluxo de desenvolvimento adotado, cuja ordem foi definida ao longo do tempo de desenvolvimento.

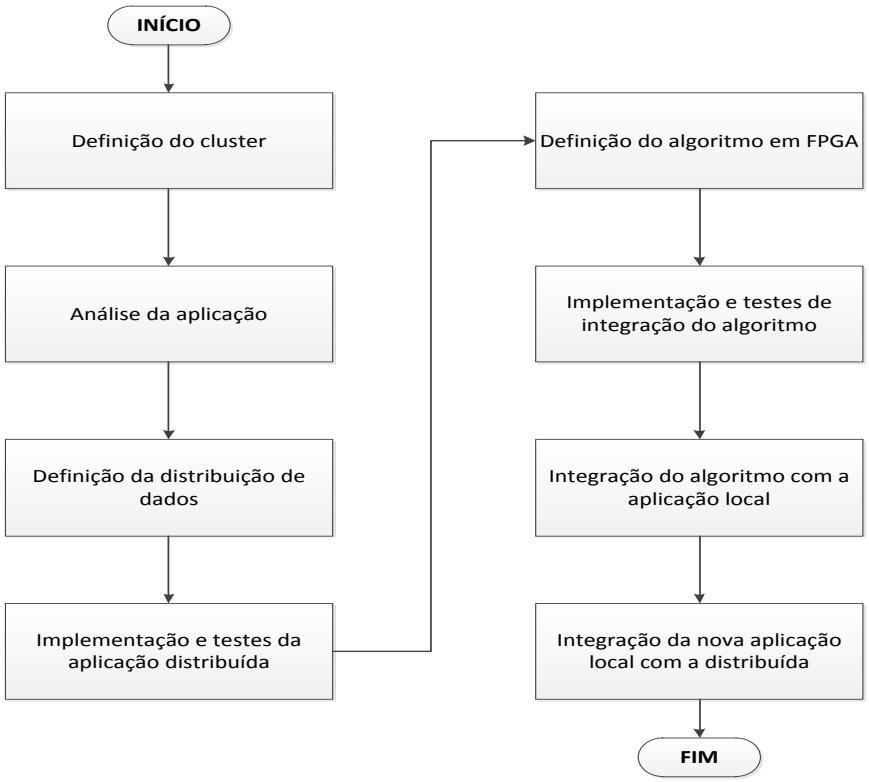

Figura 13: fluxo de desenvolvimento do projeto

Observa-se na figura 13 a divisão do fluxo em dois blocos. O primeiro (o da esquerda) retrata as etapas de desenvolvimento que incluem desde a definição do *cluster* em conjunto com a definição da aplicação distribuída, de forma a desenvolver a interface de comunicação entre nós da mesma, caso este trecho não esteja implementado de forma satisfatória para o *cluster* em questão. O segundo bloco (o da direita) engloba as etapas de desenvolvimento da aplicação local, onde o foco principal é no algoritmo que passará a ser executado no *hardware* reconfigurável, e sua integração com o trecho do *software*. além da implantação física do *cluster* já definido. A seguir, serão explicados os detalhes dos procedimentos envolvidos em cada etapa, incluindo as metas a serem alcançadas por cada uma.

Definição do cluster:

Inicialmente o desenvolvimento foi feito em apenas um nó. As características da rede e dos outros nós foram apenas definidas. A conexão física escolhida para o *cluster* foi via cabo *ethernet*, devido à disponibilidade da mesma no laboratório onde foi feito o experimento, o HPCIn. Para que haja comunicação entre as máquinas através do *MPI*, elas precisam estar num mesmo segmento de rede *TCP/IP*. Foi definida uma configuração manual em cada nó para garantir isso. As máquinas, as quais farão o papel dos nós, podem ter

configurações de *PC* variadas, contanto que tenham em comum uma placa *PROCeIII* como coprocessador.

O Sistema Operacional definido para cada nó foi o Linux, cuja distribuição é a *Debian*, versão 6 (*Wheezy*) [29], uma distribuição comunitária onde seus principais diferenciais atrativos ao uso em servidores e *workstations* são: estabilidade de seus *softwares*, ferramentas e configurações boas e simples de segurança, facilidade na manutenção do sistema com atualizações fáceis de administrar, além de boa compatibilidade com programas e *drivers* certificados para outras distribuições comerciais, como a *Red Hat* [30]. O driver *PCI* fornecido pela *Gidel* é fundamental para reconhecimento da *PROCeIII* pelo computador e um dos exemplos de *drivers* certificados, mas sua instalação é compatível com o *kernel* distribuído pelo *Debian*. Logo, a placa trabalha normalmente na distribuição comunitária mencionada.

A implementação de *MPI* escolhida para o projeto foi a do *OpenMPI*, cuja instalação, configuração e testes foram baseados num tutorial [27]. O *MPI* usa *SSH* nas conexões de rede, sendo necessário, portanto, a instalação e configuração de um servidor de *SSH* em cada nó participante. A configuração deve permitir que o servidor possa iniciar o processo cliente em cada integrante do *cluster*. Uma das características desta configuração é que não há necessidade de se usar senha para o usuário executar a aplicação distribuída.

Análise da aplicação:

Para que se possa tirar proveito de todo o potencial de desempenho que o *cluster* criado pode proporcionar, é necessário realizar uma análise prévia das aplicações complexas, candidatas a serem executadas nessa plataforma. Nesta fase torna-se necessário identificar claramente todas as rotinas de processamento massivo de dados presentes no programa, de forma a se poder definir que trechos serão implementados em hardware. Essas rotinas normalmente possuem um alto poder de paralelismo, devido ao grandioso número de operações independentes existentes nesses trechos. Então esses blocos podem ser isolados numa função com uma assinatura fixa, ou seja, com um conjunto fixo de parâmetros e retorno, cujo objetivo é de deixar transparentes ao usuário todas as adaptações que ocorreram internamente a essa função. Neste trabalho não será dado ênfase a metodologia de particionamento hardware/software.

Na Figura 14 temos uma amostra de como seria uma adaptação de função a ser implementada em hardware. O trecho principal da operação é substituído por chamadas de

sistema para a placa *PCI-e*, neste caso, funções que implementam a *API ProcAPI*, além da inicialização e fechamento da classe derivada de *Proc*. Assim, toda a parte sequencial que ficará a cargo do *PC* não sofrerá alterações significativas.

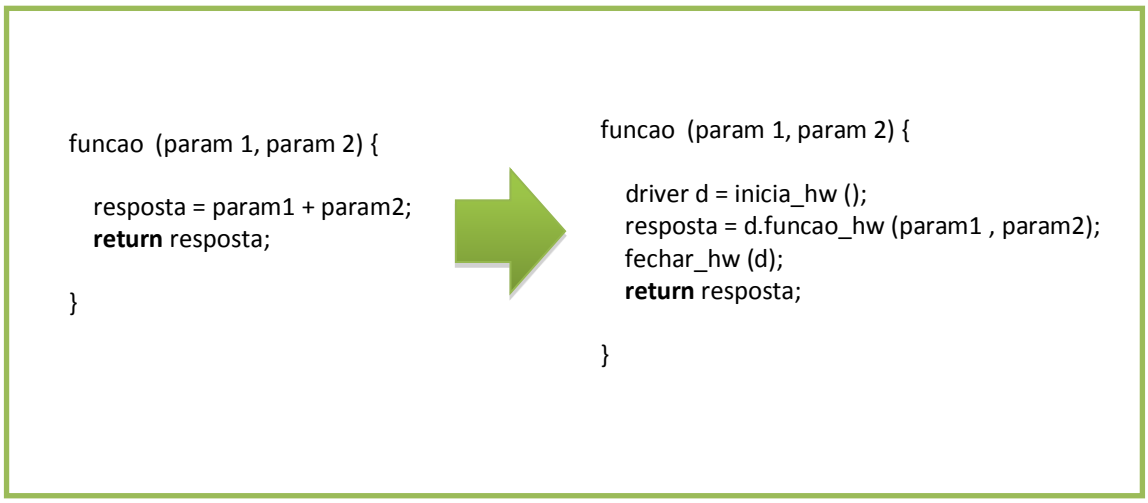

Figura 14: transformação interna de uma função massiva

Definição da distribuição dos dados:

Nesta etapa é preciso definir como os dados serão divididos entre os nós, para que cada um tenha uma formatação otimizada dos operandos parciais a ser enviada para o algoritmo em *FPGA*. Como a maneira mais simples de envio de dados através do *MPI* é em *arrays* de dados, é preciso enviar os dados de entrada para cada participante em ordem linear. Além disso, a forma de acesso mais recomendada para o *FPGA*, através do *ProcMultiport*, é do tipo sequencial. Logo, os operandos recebidos pelo *host*, em cada nó, já estão no formato ideal, podendo ser enviados diretamente para o *FPGA* sem qualquer processamento adicional. Os dados de retorno do processamento no algoritmo em hardware, do mesmo modo, serão recebidos de forma sequencial pelo *host* do nó e enviados para o servidor, da mesma maneira que os operandos.

Aqui também é definido qual o subconjunto do protocolo *MPI* será usado, ou seja, qual o tipo de comunicação a aplicação usará: ponto-a-ponto ou distribuída. Caso a escolha seja a primeira, deverá também ser considerada a escolha de um protocolo, se bloqueante ou não-bloqueante. Dependendo do comportamento apresentado no *software*, uma das opções pode ser mais eficiente que outras, entretanto, os protocolos ponto-a-ponto e bloqueantes são os mais simples de se implementar, pois o controle de envio e recebimento ficam sob responsabilidade do *MPI*.

Escolhidos a forma de distribuição e o subconjunto do protocolo de comunicação, pode-se agora realizar as adaptações necessárias na aplicação, caso a mesma não a possua previamente ou ela seja uma aplicação local. É preferível que as adaptações não alterem o comportamento geral do programa, mantendo a mesma funcionalidade na visão do usuário.

Implementação e testes da aplicação distribuída:

O comando que inicia uma aplicação distribuída em MPI é o ***mpirun***, ou seja, ele é responsável por iniciar o processo servidor e os clientes. Essa inicialização é feita no nó que executará o processo servidor. Os parâmetros que mais chamam atenção desse comando são: ***-np*** (especifica o número de processos que serão executados), ***-hosts*** (listam todos os hosts, ou os nomes das máquinas, que serão envolvidos na execução), ***-hostfile*** (define um arquivo de configuração onde se explicita cada host participante e sua configuração específica), ***--path*** (permite carregar diretórios especiais no *PATH* de cada host remoto) e ***-rf*** (define um arquivo de configuração, fixando o *rank* de cada host, assim como outras configurações particulares de cada um). Ele só termina quando todos os processos executam o *MPI_Finalize ()*.

O *MPI* é essencialmente um gerenciador de processos, podendo estes serem executados numa mesma máquina ou numa rede com vários nós. Para verificar se os dados estão sendo processados de forma correta, primeiro os processos foram executados em único nó. Assim, tornou-se mais fácil os testes iniciais e a adoção das correções dos problemas encontrados. Passando pelos testes em um nó, pode-se testar a comunicação em rede, para avaliar se a execução remota do processo local esta funcionando satisfatoriamente.

Definição do algoritmo em *FPGA*:

Após o particionamento *hardware/software*, feito de forma manual, passamos para a fase de análise do componente de *hardware*, o qual ser implementado na *FPGA* da *PROCeIII*. A arquitetura em *hardware* deve considerar também como será sua comunicação com os bancos de memória e interface *PCI-e*, como dito anteriormente. Em particular também se faz necessário definir quais portas do *multiport* da *Gidel* (acesso aos bancos de memória) serão usadas para integração com o algoritmo em hardware. Os bancos de memória podem ser divididos em até 16 blocos (de A a P)[23]. Os sinais de controle gerados pelo *multiport* são assim distribuídos:

Sinais de entrada:

***clrn*:** representa o *reset* global do módulo. O *reset* é ativo em nível lógico baixo.

***clk*:** é o *clock* do módulo.

***port_addr*:** endereço requisitado para acesso randômico, ou endereço inicial do acesso sequencial.

***addr_base*:** endereço para escolha dinâmica de uma parte física da memória.

***port_start*:** quando em nível baixo, força o módulo a ler um novo endereço de port_addr, quando alto, reseta as FIFO's internas e inicia a leitura sequencial.

***port_[X]_data_in*:** contém o dado a ser escrito via porta X.

***port_write*:** indica o sentido do fluxo do acesso randômico. Alto para escrita, baixo para leitura.

***port_select*:** quando alto, habilita a leitura ou escrita sequencial da porta, sendo levado a nível baixo quando o sinal *almost_full* (para escrita) ou *almost_empty* (para leitura) estiver em zero.

***port_flush*:** quando alto, escreve o último valor, armazenado na FIFO, na memória. É um procedimento para se ter certeza que todos os valores armazenados na FIFO (em modo sequencial) foram escritos na memória.

***port_1d_size*:** tamanho da coluna em modo segmentado de acesso.

***port_2d_size*:** tamanho da linha (número de linhas) em modo segmentado de acesso.

***port_skip_size*:** tamanho do intervalo entre linhas em modo segmentado de acesso.

Sinais de saída:

***port_[X]_data_out*:** contém o dado a ser lido via porta X.

***port_ready*:** indica que a porta está pronta para uma próxima requisição, ou seja, quando o dado torna disponível (randômico) ou quando o processamento dos dados sequenciais termina.

***empty*:** quando alto, indica que a FIFO está vazia.

***almost_empty*:** quando ativo, indica que a FIFO interna está com 1/8 de dados armazenados.

***almost_full*:** quando ativo, indica que a FIFO interna está com 7/8 de dados armazenados.

***port_internal_addr*:** o endereço atual em que o dado disponível na FIFO está na memória, sendo incrementado a cada acesso sequencial realizado.

***ptr_enable*:** serve para sincronizar o endereço interno com o *clock* do módulo do usuário, tornando alto quando o sinal *port_internal_addr* estiver disponível. É usado apenas para acesso sequencial.

***port_error*:** indica que o dado está corrompido na FIFO interna durante o acesso sequencial.

Uma informação importante para o projeto de hardware é com relação ao atraso necessário para leitura dos primeiros dados sequenciais o qual possui o seguinte cálculo:

***delay* = 16 + (profundidade_da_FIFO * 1/8)** (em ciclos de *clock*)

A profundidade da *FIFO* varia de acordo com o tamanho das *FPGA*s na placa. No caso da *PROCe III*, o valor corresponde a 256 palavras, de tamanho 256 bits [23]. Essa informação é necessária para o controle dos atrasos a serem implementados pela unidade de controle.

Implementação e testes de integração do algoritmo:

Para testes do *hardware* desenvolvido, foi adotada inicialmente uma simulação funcional em formas de onda, ou *waveforms*, que são descrições gráficas do comportamento dos sinais ao longo dos ciclos de *clock*. Para simular desta maneira, utiliza-se ferramentas de simulação para *FPGA*, presentes em vários produtos no mercado atual. Não é possível testar diretamente o *ProcMultiport*, pois o *Ip-core* está disponível em formato não suportado por esse tipo de ferramenta. A simulação fornecida pelo *ProcWizard* também não rendeu um aprendizado suficiente, de modo que sua utilização não foi possível. Então, foi adotada a utilização de um *driver* para simular o comportamento do *multiport*, com todas as portas utilizadas pelo algoritmo para comunicação com o *Ip-core*, além dos atrasos semelhantes com os do *ProcMultiport*.

Integração do algoritmo com a aplicação local:

Com o sucesso dos testes de integração em *hardware*, agora o algoritmo está pronto para integrar à aplicação local. Para gerar as interfaces necessárias, o algoritmo é então adicionado a um projeto no *ProcWizard*. A partir daí, realizam-se as configurações

necessárias para a posterior geração de código em *HDL* para ser implantado na placa, e de C/C++ para interface da placa com o *software*. Uma das configurações necessárias para comunicação com o *PC* são as configurações dos *multiports*, onde devem ser um para cada módulo de memória, que será utilizada, da placa. Essas configurações são correspondentes às portas de envio ou recebimento de dados pela *CPU*, e de envio ou recebimento de dados pelo algoritmo (*FPGA*). Outra configuração importante é com relação às portas do algoritmo que irão se comunicar com o *host*, de forma a permitir o controle do processamento em *hardware* por parte do PC. Feitos todos os ajustes necessários, o ambiente *ProcWizard* está pronto para gerar os códigos de integração *hardware-software*. Observou-se que na geração em *HDL*, o *top-level* possui ligação do algoritmo desenvolvido com o *ProcMultiport*. Mas, os sinais gerados são genéricos, precisando-se fazer o mapeamento correto desses sinais antes da fase de síntese.

Quanto à parte do *software* local, ela pode ser trabalhada a partir da definição dos sinais de interface da placa com o *software* e da geração de código em C/C++. Aqui ocorre a implementação das funções necessárias para a inicialização, envio e recebimento de dados entre o *host* e ao algoritmo, utilizando as chamadas da *ProcAPI*. No código gerado, é preciso verificar se o local do *bitstream* está correto, de forma permitir o carregamento do projeto de hardware na *PROCeIII*.

Integração da nova aplicação local com a distribuída:

Se o trecho de execução local estiver funcionando corretamente, o próximo passo é apenas adicionar as rotinas à aplicação distribuída puramente *software*, substituindo as linhas correspondentes à função massiva pelas chamadas de hardware, e adicionando ao projeto os outros códigos-fonte necessários para o funcionamento desse novo corpo da função (como mostrado na figura 14). Para compilar os novos códigos implementados, utiliza-se o **mpicxx**, de forma a otimizar o código para o ambiente *MPI*. Então, para testar a comunicação dessa nova aplicação distribuída, faz-se necessário a implantação física do cluster híbrido já definido inicialmente, copiando o executável resultante da compilação, ou compilando em cada máquina. Para que o *mpirun* funcione corretamente, é preciso ter um executável de mesmo nome em cada nó de forma a ser acionado remotamente pelo protocolo.

5. Estudo de Caso

Para validar o funcionamento do *cluster* híbrido desenvolvido para este Trabalho de Graduação, foi escolhida uma aplicação que realiza a operação de adição de matrizes densas quadradas. Ela foi escrita inicialmente para ser executada num único nó, para que pudesse passar por todas as etapas da metodologia até ser adaptada para executar no *cluster*. A aplicação em si, escrita em linguagem C/C++, possui as características de operar duas matrizes de inteiros como entradas, cujos dados são extraídos de dois arquivos de entrada, e escrever os resultados da matriz resultante num arquivo de saída.

A definição do *cluster* foi a mesma descrita anteriormente, utilizado como nó para testes locais um servidor *HP Proliant ML350* (quinta geração), com processador *Xeon quad core*, cada *core* com 1,6 GHz, placa *PROCeIII*, rodando o S.O *Debian 6*. As configurações do *OpenMPI* e do *SSH* foram as mesmas.

Na etapa da análise da aplicação, foi detectada a função a qual teria seu principal processamento sendo executado em hardware. A mesma é mostrada a seguir:

int * soma_matrizes (int *matrizA, int *matrizB)

Essa função realiza a soma de duas matrizes A e B, somando elemento a elemento cada dado de mesma posição nas duas matrizes. O resultado de retorno é a matriz resultante dessa operação. Como a função possui o principal trecho de processamento massivo da aplicação (pois poderá realizar a operação com matrizes densas), ela foi selecionada para ter seu corpo de execução ser adaptado para as chamadas da placa.

Para a etapa de definição da distribuição dos dados, foi definido um algoritmo de particionamento dos operandos, com a divisão da matriz para cada cliente baseada no número de linhas dividido pela quantidade de nós, ou seja, cada processo cliente ficará com uma quantidade igual de linhas das matrizes de entrada. O número de clientes está restrito para apenas uma quantidade que seja divisora do número total de linhas das matrizes operandos. Por exemplo: em matrizes 10x10, os números de clientes permitidos são: 1,2,5,10. As matrizes parciais de resposta seguem um particionamento análogo às de entrada. O subconjunto do *MPI* escolhido para a aplicação foi o de comunicação ponto-a-ponto e bloqueante. Logo, as linhas dos operandos as quais serão transmitidas, foram postas num

array linear, com cada linha posta em sequência, uma após a outra. A figura 15 ilustra graficamente um exemplo de como foi esse particionamento.

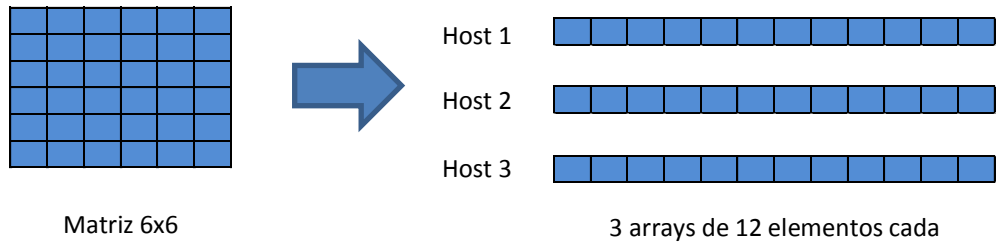

Figura 15: um exemplo do particionamento adotado

O particionamento descrito na figura 15 é relativamente simples, e seu resultado fica armazenado num array duplo de **n** x **tam_fatia**, onde n é o número de hosts que possuem um cliente e tam_fatia é o resultado do número de linhas que cada nó receberá vezes o número de colunas. Para esse novo particionamento, a função massiva local teve que sofrer uma pequena adaptação, ganhando um novo parâmetro, o **tam_fatia**. Então a nova função massiva local ficou assim:

int * soma_matrizes (int *div_matrizA, int *div_matrizB, int tam_fatia)

Concluída a aplicação distribuída, a mesma ficou com as seguintes características:

- Arquivo executável único, replicado para os *hosts* antes da execução do *mpirun*
- O processo servidor está num trecho definido do código, sendo responsável por ler as matrizes de entrada, dividir as mesmas em blocos de linhas dependendo da quantidade de clientes, enviar as matrizes parciais aos clientes, receber os resultados parciais e juntá-los para formar a saída.
- O processo cliente também está num trecho definido do código, o qual é responsável por receber as matrizes parciais de entrada, operar essas matrizes e enviar a matriz parcial de resposta ao servidor.
- Os processos clientes devem ser em um número que seja divisor do número de linhas das matrizes de entrada.
- Realiza a leitura de dois arquivos de entrada, referentes às matrizes a serem operadas e as grava num arquivo de saída.
- Os números de linhas e de colunas são definidos num arquivo de parâmetros.

- O processo servidor, não processa matriz parcial, sendo esta tarefa apenas dos clientes.

Os testes iniciais foram feitos na própria máquina (*Proliant*), com números variados de matrizes de entrada e de processos. Testes em rede foram feitos com o auxílio de duas máquinas virtuais, com configuração de *software* semelhante a do *cluster* a ser implantado. A configuração de rede testada nas *VM's* (*Virtual Machines*) foi a de rede local, com valor da rede como *192.168.0.0* e máscara *255.255.255.0*. Uma observação importante é com relação à execução correta da aplicação utilizando o *mpirun*, pois devemos executar o servidor e os clientes, isto é, o número de processos deve ser igual a um (servidor) mais o número de clientes ou nós envolvidos.

Com relação à definição do algoritmo em *FPGA*, foi preciso separar o processamento de adição propriamente dito do controle de fluxo de dados. Assim, o algoritmo foi dividido em duas partes: a unidade processamento e a unidade de controle. Inicialmente foi definido que a unidade de adição seria o núcleo de processamento, o qual seria responsável pela adição de dois elementos, dos sinais de entrada, fornecendo o resultado num sinal de saída. Como a operação de soma em *hardware* é relativamente simples e combinacional, as unidades de processamento puderam ser feitas sem a necessidade de máquinas de estado, isto é, elas simplesmente processam os dados que recebem. Com a finalidade da unidade de controle poder controlar a operação desses núcleos, alguns sinais, além dos operandos e resposta, foram adicionados ao módulo. Os sinais de entrada e de saída do núcleo desenvolvido estão ilustrados na figura 16.

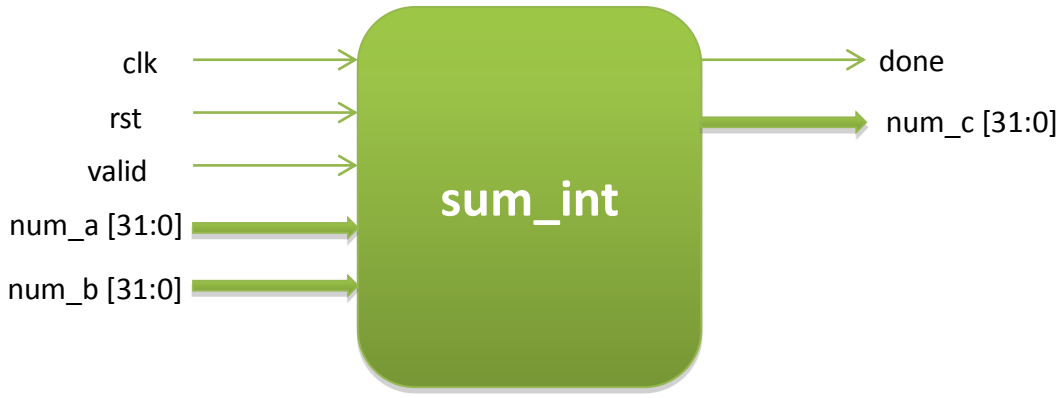

Figura 16: sinais de entrada e saída do sum_int

O ***sum_int*** é o nome dado à unidade de processamento do algoritmo em *FPGA*. Cada sinal de entrada e saída dessa unidade possui uma funcionalidade definida da seguinte forma:

Sinais de entrada:

***clk*:** é o *clock* do módulo, recebido da unidade de controle.
***rst*:** sinal de *reset*, responsável por limpar os sinais de saída
***valid*:** indica que os operandos de entrada possuem valores válidos
***num_a*:** primeiro operando inteiro (32 bits), integrante da matriz A.
***num_b*:** segundo operando inteiro (32 bits), integrante da matriz B.

Sinais de saída:

***done*:** indica que a saída gerada possui um valor válido e está pronto pra ser lido.
***num_c*:** resultado inteiro (32 bits), integrante da matriz de resposta C.

A unidade de controle, batizada de ***control_adders***, é responsável por realizar a troca de dados com o *ProcMultiport,* ou seja, possui a funcionalidade de receber os dados sequencias das matrizes de entrada, lidos da memória, e fornecer a saída para ser escrita sequencialmente na memória. De forma a aumentar o desempenho do algoritmo em *hardware*, ficou definido que um dos módulos de memória deve armazenar os dados de entrada e o outro módulo será de escrita da saída. Assim, o módulo ***control_adders*** deverá se relacionar com duas instâncias de *multiport*, as quais fazem interface para cada memória existente na *PROCeIII*. Ele também realiza o controle e fornecimento dos dados para os núcleos de processamento. Portanto, o ***control_adders*** em conjunto com o ***sum_int*** fazem parte do algoritmo desenvolvido em *hardware* para adição de matrizes, onde as unidades de processamento são os submódulos da unidade de controle. A figura 17 mostra uma visão geral do algoritmo desenvolvido e suas ligações com as memórias existentes na *PROCeIII*.

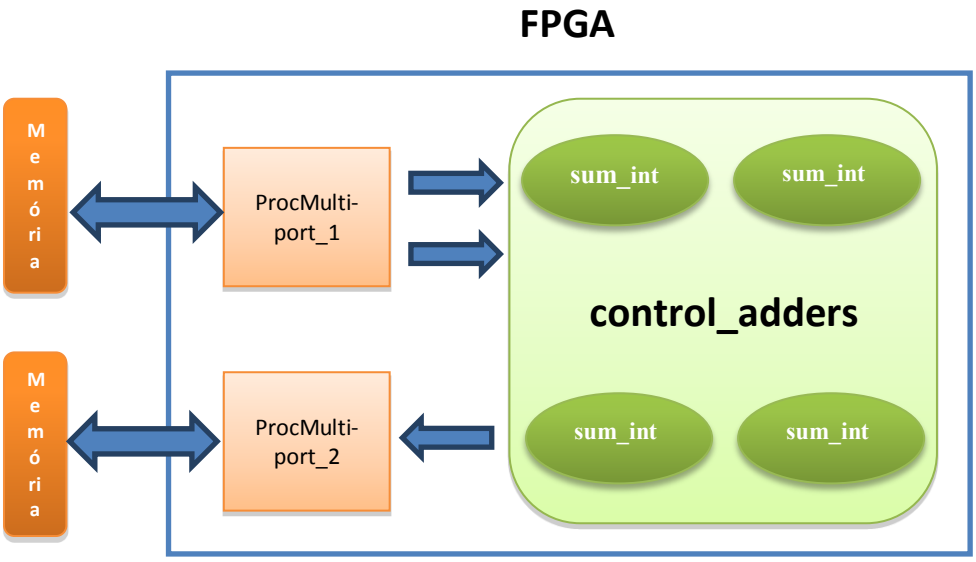

Figura 17: visão geral do algoritmo e sua comunicação com a memória

Como é mostrado na figura 17, um dos *multiports* é responsável pela leitura e entrega dos operandos ao ***control_adders***, e o outro *multiport* é encarregado de receber a matriz serializada de resposta do algoritmo e escreve-la na memória apropriada.

A unidade de controle pode conter mais de uma unidade de processamento. Esta quantidade depende da largura das portas de dados especificadas para os *multiports*. Na *PROCeIII*, a largura máxima de cada porta de dados é de 256 bits, podendo fornecer assim, até 8 números inteiros de 32 bits. Neste projeto, ficou definida uma largura de porta de 128 bits, pois o uso de mais de uma porta com a largura máxima proporciona uma degradação na frequência de operação do *multiport*. Então cada porta, tanto as duas de entrada como a de saída, podem transportar, a cada ciclo de *clock*, 4 números de 32 bits. Logo, quatro pares de dados, cada um contendo um dado da matriz do primeiro operando e outro da matriz do segundo, ficam disponíveis por ciclo de relógio, sendo necessário instanciar quatro módulos ***sum_int*** internamente ao ***control_adders***. A figura 18 apresenta os sinais de entrada e saída do módulo ***control_adders***.

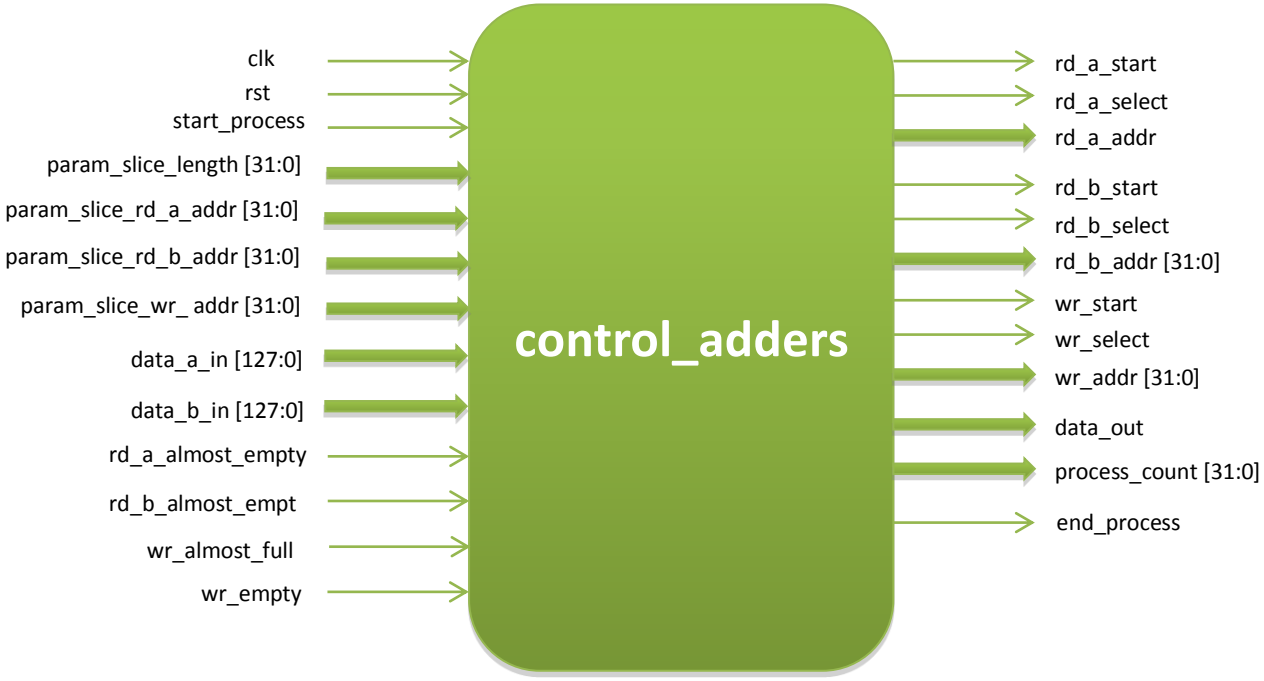

Figura 18: sinais de entrada e saída do control_adders

Sinais de entrada:

***clk*:** sinal de *clock* do módulo.

***rst*:** é o *reset*, para zerar os sinais de saída quando em nível alto.

***start*_process:** indica para a máquina de estados que ela pode iniciar o seu processamento.

***data_a_in*:** porta que recebe sequencialmente os operandos da matriz A.

***data_b_in*:** porta que recebe sequencialmente os operandos da matriz B.

***rd_a_almost_empty*:** recebe sinal de almost_empty da porta A (mais detalhes na sessão 4.2).

***rd_b_almost_empty*:** recebe sinal de almost_empty da porta B.

***wr_almost_full*:** recebe o sinal de almost_full da porta de escrita.

***wr_empty*:** recebe o sinal de empty da porta de escrita.

***param_slice_length*:** recebe a quantidade total de números de uma matriz (os operandos possuem mesmo tamanho).

***param_slice_rd_a_addr*:** endereço inicial dos dados da matriz A.

***param_slice_rd_b_addr*:** endereço inicial dos dados da matriz B.

***param_slice_wr_addr*:** endereço inicial dos dados de escrita.

Sinais de saída:

***rd_a_start*:** sinal que, quando ativo, inicia o envio sequencial dos dados da matriz A.

***rd_a_select*:** sinal que habilita ou não a leitura de um valor válido da porta da matriz A.

***rd_a_addr*:** endereço inicial de leitura dos dados da matriz A, a serem enviados para o *multiport*.

***rd_b_start*:** sinal que, quando ativo, inicia o envio sequencial dos dados da matriz B.

***rd_b_select*:** sinal que habilita ou não a leitura de um valor válido da porta da matriz B.

***rd_b_addr*:** endereço inicial de leitura dos dados da matriz B, a serem enviados para o *multiport*.

***wr_start*:** sinal que, quando ativo, inicia o envio sequencial dos dados da matriz de resposta.

***wr_select*:** sinal que habilita ou não a escrita de um valor válido da porta da matriz de resposta.

***wr_addr*:** endereço inicial de leitura dos dados da matriz de resposta, a serem enviados para o *multiport*.

***data_out*:** dados da matriz de resposta, que faz conexão com a porta de escrita.

***process_count*:** contador que indica quantos cálculos válidos foram feitos pelas unidades de processamento.

***end_process*:** indica que terminou o processamento das matrizes.

Para o controle dos atrasos dos dados provenientes das FIFO's do *multiport*, tanto de leitura quanto de escrita, a unidade de controle possui uma máquina de estados especial, cujo fluxo de tarefas, ao longo de sua atividade, é basicamente inicializar a leitura, controlar o processamento e escrita dos dados e avisar para o *host* o término da escrita da matriz de resposta.

Para simular o comportamento desta unidade de controle, foi utilizado um *driver* que simula o comportamento do *multiport*, de forma a visualizar o comportamento do algoritmo ao longo do tempo na ferramenta de simulação *Modelsim* [32]. Com a verificação satisfatória do comportamento da unidade de controle, o algoritmo está pronto para ser integrado ao *software*.

Inicia-se então a criação de um novo projeto no *ProcWizard*, para que se possa modelar toda a infraestrutura que será utilizada na placa. A estrutura adotada para o algoritmo está ilustrada na figura 19.

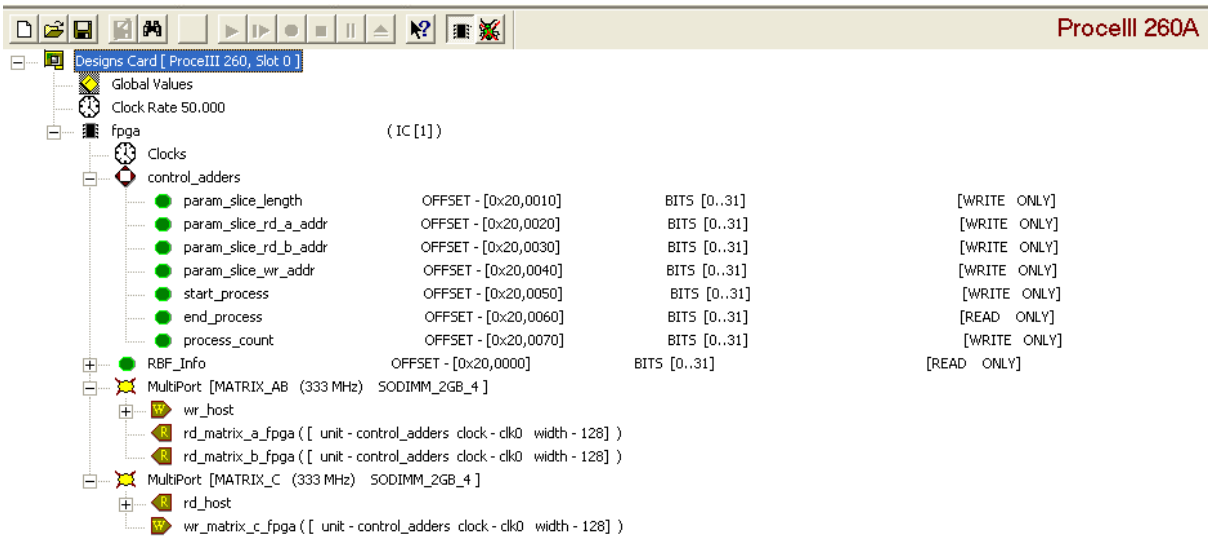

Figura 19: configuração do ProcWizard para o algoritmo *FPGA*

A figure 19 mostra a configuração feita no ambiente *ProcWizard* para a integração do algoritmo com a *PROCeIII*. Logo acima observamos a configuração do *clock* do sistema, com valor adotado no projeto de 50MHz, o qual poderá ser utilizado por todos os módulos a serem implantados na *FPGA*. Podemos observar também que alguns sinais do ***control_adders*** estão em evidência, isso significa que eles podem ser vistos ou alterados pelo PC, dependendo da configuração feita (*read only* ou *write only*). Assim, é permitido ao *software* realizar o controle de inicialização (***start_process***), configuração dos parâmetros da FSM (os sinais com ***param*** no nome) e verificar se o processamento em *hardware* terminou (***end_process***).

Quanto aos *multiports*, foram adicionadas as portas correspondentes tanto para envio e recebimento de dados por parte do PC (***wr_host*** para envio dos operandos na primeira memória, ***rd_host*** para leitura da matriz processada na segunda memória), as portas de comunicação com o *FPGA*, as portas de leitura dos operandos na primeira memória e a porta de escrita da resposta na segunda.

Concluída a configuração do ambiente, pode-se fazer a geração dos códigos HDL e C++. A HDL escolhida foi Verilog, em conformidade com a linguagem utilizada para desenvolver o algoritmo de adição. Verifica-se que após a geração do *hardware*, é criado um

template do ***control_adders***, com apenas as portas explicitadas no ambiente e algumas genéricas para comunicação com o *multiport*, além de um *top-level* com um mapeamento genérico do algoritmo, e um arquivo de projeto no ambinte de síntese, o Quartus II, com as configurações de síntese otimizadas para a *FPGA* da placa. Antes de sintetizar o hardware, é preciso então substituir o *template* do algoritmo pelo projeto de *hardware* desenvolvido, ou seja, o ***control_adders*** e seu módulo interno ***sum_int***. Além disso, também se faz necessário o ajuste do mapeamento no *top-level* para que o código gerado possa ter a comunicação adequada com o algoritmo. Feitas todas as alterações, o próximo passo é realizar é a geração do *bitstream*, um arquivo **.rbf**, de configuração do *FPGA*. Quando à parte da geração de C++, observa-se que o código é um arquivo **.h** (*header*), o qual está a classe representante da placa no software, a qual herda as características da classe *Proc*, como já mencionado. A única alteração relevante a fazer nesse arquivo é colocar a localização correta do rbf gerado do projeto em *hardware*.

Para concluir a integração do *hardware* ao *software* local, falta apenas a implementação das chamadas da *API ProcAPI*. Para esta etapa, foi criada uma nova classe, a **Add_driver**, cuja funcionalidade principal é realizar o papel efetivo de *driver* do dispositivo *PCI*, concentrando todas as chamadas da *API* referida, como inicialização da placa, envio e recebimento de dados via canal *DMA*, e fechamento correto da *PROCeIII*. Então ficou definido que essa classe conterá uma instância da classe gerada pelo *ProcWizard*, além de *buffers* para os parâmetros, as matrizes operandos e de resposta.

Além das funções de controle do *hardware* reconfigurável, essa classe também deve ter funções de interface com a função **soma_matrizes**, a fim de poder substituir as rotinas de *software* dessa função por essas funções de interface, de maneira já mostrada anteriormente. Terminada toda a integração, o novo *software* local está pronto para os testes de corretude. Caso haja algum erro, é preciso voltar ao *ProcWizard* para realizar as alterações e gerar novamente todos os arquivos necessários, podendo até ser gerado um novo .rbf, para a realização de novos testes. Concluídos os testes, o trecho local adaptado está pronto para ser integrado à aplicação distribuída.

Para a adaptação do novo software local à aplicação distribuída, é necessário apenas adicionar as funções que implementam o protocolo *MPI* do servidor e do cliente. Assim, a nova aplicação distribuída pode ser compilada e testada no *cluster* híbrido, cuja implantação com as configurações já definidas pode também ser concretizada.

6. Resultados

O projeto desenvolvido neste TG foi importante para validar o fluxo de desenvolvimento de um cluster híbrido em conjunto com a aplicação distribuída adaptada para o mesmo. Então, possíveis otimizações para aumentar o desempenho massivo da aplicação escolhida para estudo de caso foram descartadas, sendo tendo então o enfoque apenas na corretude da execução do programa em rede.

O cluster implantado foi composto por três máquinas, contendo cada uma um *PC* convencional e uma placa PROCeIII como coprocessador.

O gráfico da figura 20 demonstra o desempenho em quatro cenários, onde as aplicações realizam a operação de adição matricial. Um deles é a execução do *software* local, em apenas um nó. Outro é a execução de uma aplicação distribuída puramente *software*. Um terceiro é a aplicação local com sua rotina massiva sendo um algoritmo executado na FPGA. E como último cenário, temos a aplicação distribuída, com seus processos locais realizando suas operações principais através do algoritmo FPGA implantado na placa. Os tempos foram considerados apenas durante a execução do trecho massivo de cada programa.

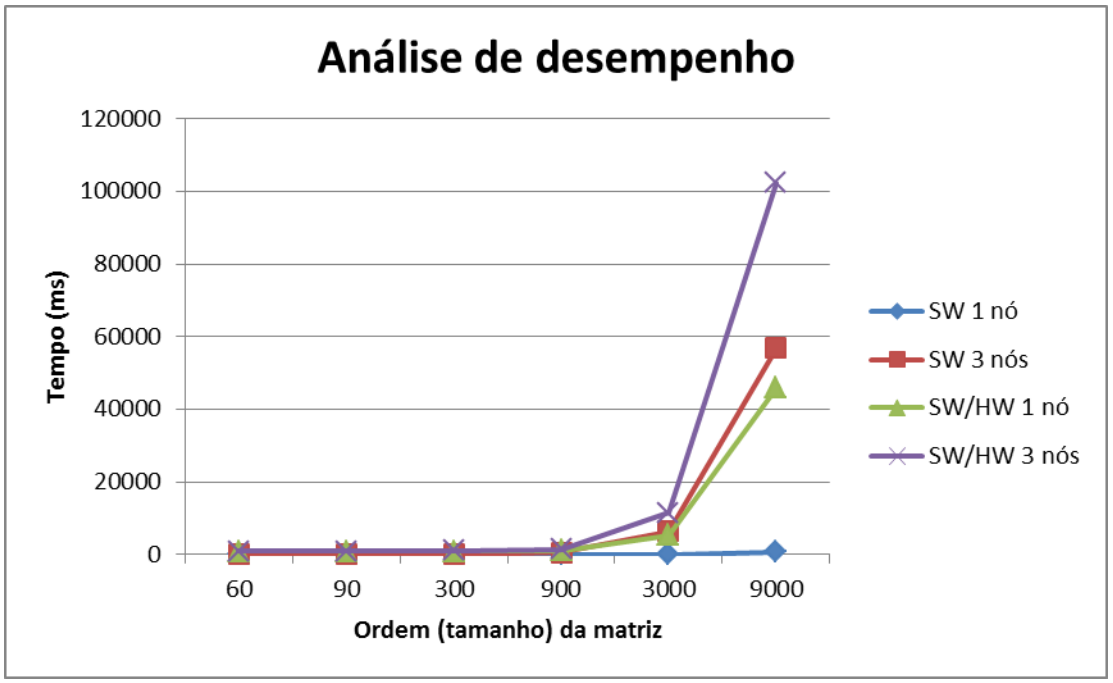

Figura 20: gráfico de análise de desempenho da aplicação desenvolvida

Os resultados indicam que as aplicações locais ainda foram mais eficientes em relação às distribuídas. Uma das possíveis causas foi a falta de otimização do algoritmo em FPGA, haja vista que a operação de adição entre matrizes requer nenhum reuso de dados para sua realização. Outro fator relevante para a perda de performance pode estar no tempo necessário para a comunicação via *MPI*, pois o subconjunto escolhido da referida API (ponto-a-ponto e bloqueante) abstrai as rotinas de sincronismo dos dados, sob a penalidade de aumentar o número de rotinas para entregar os dados para cada nó.

7. Conclusões

O *cluster* projetado neste Trabalho de Graduação proporcionou testar os conhecimentos adquiridos pelo aluno durante a graduação, além de fornecer novos conceitos e técnicas durante a pesquisa acadêmica necessária para o desenvolvimento da proposta inicial. A metodologia adotada mostrou-se satisfatória para a adaptação da aplicação, visto que sua funcionalidade se manteve bastante semelhante à aplicação original, e o objetivo da transparência das adaptações para o usuário foi alcançado.

O uso das ferramentas e conhecimentos já existentes no mercado e no meio acadêmico foram muito importantes para a conclusão desse projeto, pois além de auxiliar no desenvolvimento dos produtos necessários em cada passo da metodologia, o aluno pôde aperfeiçoar sua capacidade de manusear programas usados em grandes organizações, além de permitir ao estudante conhecer projetos semelhantes que foram ou estão sendo desenvolvidos no cenário científico atual.

A aplicação escolhida para o estudo de caso mostrou-se aproveitar pouco o paralelismo potencial do *FPGA*, pois os núcleos de processamento dependeram fortemente do fluxo sequencial dos dados fornecidos pela memória da placa, e também sua característica de ausência de reuso favoreceu ainda mais desempenho em apenas *SW*, como visto na sessão de resultados.

8. Trabalhos Futuros

Como trabalhos futuros, podemos inicialmente sugerir a melhoria no algoritmo a ser executado na *FPGA*, com otimizações que permitam a utilização máxima possível dos recursos oferecidos pelo dispositivo. Também pode-se melhorar com relação ao particionamento dos dados, fazendo com que ele possa ser utilizado por qualquer quantidade de nós.

Quanto à metodologia, pode-se também verificar a utilização da mesma em outras aplicações, como a de multiplicação de matrizes densas por exemplo. Outra linha de pesquisa sugerida seria aplicar a mesma metodologia em outras placas da *Gidel* como a *ProcStarIII*, inclusive esta outra placa contém mais *FPGA*'s e mais módulos de memória, as quais podem aumentar significativamente a quantidade de unidades de processamento implementados na plataforma, aumentando assim o paralelismo do algoritmo em *hardware*.

9. Referências

[1] Araujo, Rodrigo (2010), “Um Cluster De Pc's Usando Nós Baseados Em Módulos Aceleradores De Hardware (FPGA) Como Co-Processadores.”, Dissertação de Mestrado, UFPE, Recife, Pernambuco.

[2] C. Pascoe, A. Lawande, et al, “Reconfigurable Supercomputing with Scalable Systolic Arrays and In-Stream Control for Wavefront Genomics Processing”, Paper, CHREC, ICBR, University of Florida, USA.

[3] Site Altera (2011), www.altera.com

[4] Site GIDEL (2011), www.gidel.com

[5] GIDEL ProcWizard, http://www.gidel.com/procwizard.htm

[6] Site OpenMPI (2011), www.open-mpi.org

[7] Site Nvidia (2011), www.nvidia.com

[8] Backus, J., “Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs”, Artigo, Communicationsof the ACM, 1978.

[9] Site Intel (2011), Arquitetura multicore, <http://www.intel.com/portugues/products/processor/core2quad/index.htm>

[10] Ferreira, Thiago, “Introdução ao MPI”, <http://www.fisiocomp.ufjf.br/seminarios/IntroducaoMPI.pdf>

[11] Gropp, William, “An Introduction to MPI Parallel Programming with the Message Passing Interface”, <http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiintro/index.htm>, 1998.

[12] Introdução ao MPI, <http://www.cenapad.unicamp.br/servicos/treinamentos/apostilas/apostila_MPI.pdf>, 2011

[13] Site Tutorial MPI (2011), <https://computing.llnl.gov/tutorials/mpi/>.

[14] Moraes, F., Mesquita, D., “Tendências em Reconfiguração dinâmica de FPGAs”, Artigo, PUCRS, Porto Alegre, Rio Grande do Sul.

[15] Rocha, Rodrigo, (2010) “Desenvolvimento De Uma Plataforma Reconfigurável Para Modelagem 2d, Em Sísmica, Utilizando FPGA’s.”, Dissertação de Mestrado, Cin, UFPE

[16] Xilinx. Virtex-4 User Guide. <http://www.xilinx.com/support/documentation/user_guides/ug070.pdf>. Consultado em: 17 de Julho de 2010.

[17] GiDEL. Plataforma PROCe III. <http://www.gidel.com/PROCe%20III.htm>. Consultado em: 16 de Julho de 2010.
[18] Altera. Overview da Stratix III. <http://www.altera.com/products/devices/stratix-fpgas/stratix-iii/overview/st3-overview.html>. Consultado em: 16 de Julho de 2010.
[19] Altera. Accelerating High-Performance Computing With FPGAs, Agosto 2007. URL: <http://www.altera.com/literature/wp/wp-01029.pdf>. Consultado em: 14 de julho de 2010.
[20] Aschermann, N., Roberto, P., Arquitetura Multicore. <http://www.ic.unicamp.br/~rodolfo/Cursos/mc722/2s2007/trabalhos/g20_texto.pdf>. Consultadoem: 20 de outubro de 2010.
[21] Site Mitrionics, "Mitrion User Guide". <www.mitrionics.com> Consultado em: 20 de abril de 2010.
[22] Visual Studio, <http://www.microsoft.com/visualstudio/pt-br>, consultado em dezembro de 2011
[23] GiDEL. ProcMultiport,<http://www.gidel.com/procMultiPort.htm>, consultado em dezembro de 2011
[24] El-Ayat K., Agarwal R.. "The Intel 80386- Architecture and Implementation", Artigo, IEEE, 1985.
[25] Introduction to Computer Organization and Architecture <http://cnx.org/content/m29431/latest/>, 2011
[26] HPC Wire, GPU Architecture, <http://www.hpcwire.com/hpcwire/2008-09-10/compilers_and_more_gpu_architecture_and_applications.html>, consultado em dezembro de 2011
[27] Setup cluster MPI, <http://techtinkering.com/2009/12/02/setting-up-a-beowulf-cluster-using-open-mpi-on-linux/>, consultado em dezembro de 2011.
[28] Araújo, Cristiano, "InterfPISH – Uma ferramenta para geração automática de interfaces em hardware/software Codesign", <http://www.cin.ufpe.br/~cca2/phd/documentos/msc_thesis_cca2.pdf>
[29] Debian GNU Linux, <http://www.debian.org/>, consultado em dezembro de 2011.
[30] Red Hat Brasil, < http://www.br.redhat.com/>, consultado em dezembro de 2011.
[31] Site MPICH, http://www.mcs.anl.gov/research/projects/mpich2/, consultado em dezembro de 2011.
[32] Modelsim- Advanced Simulation and Debugging, <www.model.com>, consultado em dezembro de 2011.