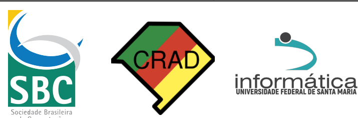
MINICURSOS

Minicursos da XX Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)

15 a 17 de abril de 2020
Santa Maria – RS, Brasil

Organização e Edição
André Du Bois
Márcio Castro

Realização



Organização

Patrocínio Diamante

Patrocínio Ouro

Agências de Fomento

ANDRÉ DU BOIS
MÁRCIO CASTRO

MINICURSOS DA XX ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD/RS)

Porto Alegre
Sociedade Brasileira de Computação – SBC
2020

Dados Internacionais de Catalogação na Publicação (CIP)

E73

Escola Regional de Alto Desempenho da Região Sul (20. : 2020: Santa Maria, RS).

Minicursos da XX Escola Regional de Alto Desempenho da Região Sul [recurso eletrônico] : 15 a 17 de abril de 2020, UFSM – Santa Maria / coordenadores André Du Bois, Márcio Castro. – Porto Alegre : SBC, 2020.

1 recurso eletrônico. (131 p.)

ISBN 978-65-87003-00-9 (e-book)

1. Processamento de Alto Desempenho. 2. Arquitetura de Computadores. 3. Programação Paralela. I. Du Bois, André. II. Castro, Márcio. III. Sociedade Brasileira de Computação. IV. Universidade Federal de Pelotas (UFPEL). V. Universidade Federal de Santa Catarina (UFSC). VI. Universidade Federal de Santa Maria (UFSM). VII. Título.

CDD 004

Ficha catalográfica elaborada pelo bibliotecário Samoel Luiz Paulini – CRB-14/1661

ERAD/RS 2020

XX Escola Regional de Alto Desempenho da Região Sul
15 a 17 de abril de 2020
Universidade Federal de Santa Maria (UFSM), Santa Maria, Rio Grande do Sul, Brasil
https://erad2020.inf.ufsm.br/

A XX Escola Regional de Alto Desempenho da Região Sul (ERAD/RS 2020) foi realizada entre os dias 15 a 17 de abril de 2020, na cidade de Santa Maria, no campus sede da Universidade Federal de Santa Maria (UFSM). A ERAD/RS é promovida anualmente pela Sociedade Brasileira de Computação (SBC) e pela Comissão Regional de Alto Desempenho da Região Sul (CRAD/RS).

O público alvo da ERAD/RS 2020 são alunos, profissionais e professores/pesquisadores que atuam direta ou indiretamente na computação de alto desempenho e em áreas correlatas. O evento engloba a região sul do Brasil (RS, SC e PR).

Os principais objetivos são:

- Qualificar os profissionais do sul do Brasil nas áreas que compõem o processamento de alto desempenho;
- Prover um fórum regular onde possam ser apresentados os avanços recentes nessas áreas;
- Discutir formas de ensino de processamento de alto desempenho nas universidades.

A programação da ERAD/RS 2020 contou com 4 palestras, 8 minicursos, os fóruns de iniciação científica e pós-graduação com apresentação de 58 trabalhos, além de uma sessão especial sobre os 20 anos da ERAD/RS. O Fórum de Iniciação Científica deste ano foi coordenado pelos professores Vinicius Garcia Pinto (UFRGS) e Carlos Rolim (URI). O Fórum de Pós-Graduação foi coordenado pelos professores Arthur Lorenzon (UNIPAMPA) e Guilherme P. Koslovski (UDESC). Os minicursos foram coordenados pelos professores André Du Bois (UFPEL) e Márcio Castro (UFSC). A equipe organizadora local foi coordenada pelas professoras Marcia Pasin, Juliana Vizzoto e Patricia Pitthan da Universidade Federal de Santa Maria (UFSM), junto com os discentes Claiton Dernardi Paulus, Matheus Dalmolin da Silva e João Vitor Meller, todos da UFSM.

Índice

Mensagem da Coordenação Geral iii
Mensagem da Coordenação dos Minicursos iv
Comitês Organizadores v
Minicursos vi

Mensagem da Coordenação Geral

Saudamos e damos as boas vindas à vigésima edição da Escola Regional de Alto Desempenho da Região Sul, a ERAD/RS 2020, que neste ano aconteceu na cidade de Santa Maria, coração do estado do Rio Grande do Sul, entre os dias 15 e 17 de abril.

A ERAD/RS é um evento anual, promovido pela Sociedade Brasileira de Computação (SBC), por meio da Comissão Regional de Alto Desempenho da Região Sul (CRAD-RS), desde 2001. O objetivo deste encontro, de caráter essencialmente regional, abrangendo a região sul do Brasil, é de qualificar profissionais da região nas áreas que compõem o Processamento de Alto Desempenho (PAD) e de proporcionar um fórum regular no qual se possa tanto apresentar os avanços recentes nessas áreas quanto discutir as formas de ensino de processamento de alto desempenho nas Instituições de Ensino Superior (IES) do sul do Brasil.

Além disso, as atividades da ERAD/RS aproximam academia, sociedade e indústria. A troca de experiências e conhecimento de novidades é estimulada através da intercalação de palestras com pessoas de renome na área, apresentações acadêmicas e relatos do mercado/indústria. Essa sinergia é ampliada pela interação de professores, estudantes e profissionais de diversas regiões do Brasil, a qual possibilita analisar questões por um viés diferente da sua realidade regional.

Para potencializar seus resultados, A ERAD/RS é itinerante, sendo abrigada, a cada ano, por uma instituição diferente. Neste ano, coube à Universidade Federal de Santa Maria (UFSM) gerenciar a execução deste encontro, que será realizado em Santa Maria – RS, com o apoio da Universidade do Estado de Santa Catarina (UDESC), da Universidade Federal do Rio Grande do Sul (UFRGS), da Universidade Federal de Santa Catarina (UFSC), da Universidade Federal do Pampa (UNIPAMPA), da Universidade Federal de Pelotas (UFPEL) e da Universidade Regional Integrada do Alto Uruguai e das Missões (URI).

A região sul do Brasil é, há muitos anos, referência no país na área de PAD. A ERAD/RS reflete este fato e, mais ainda, reforça esta posição de destaque quando se preocupa em formar novos pesquisadores e em manter atualizados os pesquisadores que criaram suas bases nos estados do sul. Portanto, é com satisfação que nos envolvemos neste já tradicional evento, nos tornando parte dessa história e auxiliando o desenvolvimento científico do país. Assim, é nesse momento que reconhecemos a presença de instituições que acompanham a evolução desta história colaborando de forma efetiva para sua realização. Neste ano, a ERAD/RS contou com fomento governamental da FAPERGS e do CNPq, além do patrocínio diamante da empresa Atos e do patrocínio ouro da empresa SDC.

Agradecemos, em especial, a todos os autores que submeteram trabalhos às sessões técnicas, às empresas e aos convidados diversos que aceitaram nosso convite e engrandecem a ERAD/RS com as suas participações. Por fim, agradecemos a prontidão com que diversos colegas do corpo multi-institucional deste evento, que tomaram para si diversos encargos e os conduziram a termo com pleno sucesso.

Obrigado pela presença de todos e aproveitem a ERAD/RS 2020.

João Vicente Ferreira Lima (UFSM), Charles Christian Miers (UDESC) e Lucas Mello Schnorr (UFRGS)
Coordenadores da ERAD/RS 2020

Mensagem da Coordenação dos Minicursos

A Escola Regional de Alto Desempenho da Região Sul (ERAD/RS) é um evento anual promovido pela Sociedade Brasileira de Computação (SBC) e pela Comissão Regional de Alto Desempenho da Região Sul (CRAD/RS). A escola, que neste ano completa os seus vinte anos, foi realizada entre os dias 15 e 17 de abril de 2020, na cidade de Santa Maria/RS, no campus sede da Universidade Federal de Santa Maria (UFSM).

Um dos objetivos da ERAD/RS é qualificar profissionais da região sul nas diversas áreas do Processamento de Alto Desempenho (PAD). Com este intuito, todo o ano, são selecionados minicursos introdutórios e avançados em tópicos de interessa à comunidade. Não diferente, neste ano festivo, foram selecionados oito minicursos, dos quais seis viraram capítulos para este livro. Os minicursos aqui representados, apresentam tópicos de ponta da área de PAD, os quais irão certamente agradar os participantes do evento.

Os coordenadores dos minicursos agradecem aos autores, por compartilharem seus conhecimentos através da submissão de minicursos de alto nível para esta edição da escola, aos coordenadores e organizadores da ERAD/RS, pelo apoio dado na seleção dos minicursos e na realização do evento, além de desejar uma boa ERAD/RS a todos!

Andre Du Bois (UFPEL) e Márcio Castro (UFSC)
Coordenadores dos Minicursos da ERAD/RS 2020

Comitês Organizadores

Coordenação Geral

- João Vicente Ferreira Lima (UFSM)
- Charles Christian Miers (UDESC)
- Lucas Mello Schnorr (UFRGS)

Coordenação Local

- Marcia Pasin (UFSM)
- Juliana Vizzoto (UFSM)
- Patricia Pitthan (UFSM)

Fórum de Pós-Graduação

- Arthur Lorenzon (UNIPAMPA)
- Guilherme Piêgas Koslovsk (UDESC)

Fórum de Iniciação Científica

- Vinicius Garcia Pinto (UFRGS)
- Carlos Rolim (URI)

Minicursos

- Andre Du Bois (UFPEL)
- Márcio Castro (UFSC)

Equipe Local

- Claiton Denardi Paulus (UFSM)
- Matheus Dalmolin da Silva (UFSM)
- João Vitor Meller (UFSM)

Minicursos

Minicurso I

Boas Práticas para Experimentos Computacionais de Alto Desempenho 1
Vinícius Garcia Pinto (UFRGS), Lucas Leandro Nesi (UFRGS), Lucas Mello Schnorr (UFRGS)

Minicurso II

Introdução à Programação com OpenACC 20
Evaldo B. Costa (UFRJ), Gabriel P. Silva (UFRJ)

Minicurso III

Programação Paralela em Memória Compartilhada e Avaliação de Desempenho com Contadores de Hardware 50
Matheus da Silva Serpa (UFRGS), Claudio Schepke (UNIPAMPA)

Minicurso IV

Introdução ao Desenvolvimento de Aplicações Paralelas com o Paradigma Orientado a Tarefas e o Runtime StarPU 70
Lucas Leandro Nesi (UFRGS), Vinícius Garcia Pinto (UFRGS), Marcelo Cogo Miletto (UFRGS), Lucas Mello Schnorr (UFRGS), Samuel Thibault (INRIA)

Minicurso V

Programando Aplicações com Diretivas Paralelas 89
Natiele Lucca (UNIPAMPA), Claudio Schepke (UNIPAMPA)

Minicurso VI

Estudo de Networks on Chip (NoCs) em FPGAs 105
Maurício Acconcia Dias (FHO), Marcílio F. de Oliveira Neto (FHO)

 DOI: 10.5753/sbc.4400.9

Capítulo 1

Boas Práticas para Experimentos Computacionais de Alto Desempenho

Vinícius Garcia Pinto, Lucas Leandro Nesi, Lucas Mello Schnorr
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Resumo

Este minicurso tem foco na análise de desempenho de aplicações paralelas para computação de alto desempenho. O objetivo é sensibilizar os participantes sobre os fatores que impactam a coleta de medidas representativas para que os experimentos sejam mais confiáveis. O minicurso tem três partes: (a) motivar cuidados essenciais na realização de experimentos computacionais para controlar a variabilidade experimental; (b) apresentação das principais formas de controlar parâmetros em sistemas Linux; e (c) como analisar os dados coletados de maneira reprodutível com linguagens de programação e ferramentas modernas de manipulação de dados.

1.1. Introdução

Um dos pilares do método científico é o uso de experimentos para validar ou refutar hipóteses e teorias que tentam explicar fenômenos naturais. Para ser confiável, um experimento deve ser reprodutível, de forma que outros pesquisadores possam refazer as observações independentemente. A reprodutibilidade se torna possível no momento que se exerce um controle sobre a maior quantidade possível de variáveis que possam afetar o fenômeno sob investigação. A prática consiste também em registrar o valor das variáveis não controladas (ou não controláveis) de forma que elas possam servir de contexto observado do fenômeno.

Experimentos computacionais na grande área de sistemas de computação não são diferentes. O controle e registro de variáveis dos sistemas computacionais é passo obrigatório para tornar qualquer observação computacional mais confiável. Em um cenário com um único computador, esse controle se exerce através do estabelecimento de configurações de todas as camadas do computador, das configurações de hardware (por exemplo, da BIOS) até as de software (sistema operacional e arcabouços). Na área de processamento

paralelo de alto desempenho onde múltiplos computadores são utilizados conjuntamente, esse controle deve ser exercido também sob os elementos da rede de interconexão.

Tal controle experimental tem desvantagens e vantagens. Por um lado, os experimentos se tornam um processo mais burocrático, envolvendo preparação adicional, cuidado maior no antes, durante e depois dos experimentos, e disciplina reforçada. Tais procedimentos podem fazer com que o avanço da investigação seja mais lento. Por outro lado, um controle reforçado permite que a investigação seja conduzida a partir de um efeito real, claro, fazendo com que as conclusões delineadas a respeito do fenômeno sejam mais perenes e significativas. Nesta mesma linha, tal controle torna o relato das observações realizadas, como artigos científicos ou relatórios de pesquisa, enriquecido e mais facilmente reprodutível. Por exemplo, os dados coletados podem ser retrabalhados sem a necessidade de realizar uma nova longa bateria experimental.

As boas práticas para experimentos computacionais em *clusters* de alto desempenho se estendem portanto ao relato das conclusões. Os dados coletados devem ser trabalhados através de programas de computadores (*scripts*) de maneira a retirar o humano da preparação de estatísticas, gráficos e tabelas. Por consequência, toda transformação de dados deve ser realizada de maneira automática através de programas de computador preparados pelo analista. Assumindo que um cuidado elevado seja empregado pelo analista na criação destes programas, isso garante que a transformação dos dados reflita exatamente a mensagem que se deseja transmitir, ou evidencie o efeito observado.

Levando-se em conta este contexto, este minicurso tem por objetivo sensibilizar os participantes às questões de reprodutibilidade em sistemas computacionais de alto desempenho. Apresenta portanto um conjunto de boas práticas a serem aplicadas desde a realização de experimentos computacionais em *clusters* de alto desempenho até a transformação e análise dos dados. Caracterizando-se como uma atividade multidisciplinar, o minicurso envolve conceitos de sistemas operacionais, redes, programação, análise de dados, e processamento paralelo. Ele está estruturado em duas partes:

- **Parte 1: Controle e Coleta** – apresentação de uma lista não exaustiva com as principais formas de controlar sistemas computacionais, focados em ambientes para processamento de alto nível (SO Linux, múltiplos nós, redes de baixa latência), e de projeto experimental (JAIN, 1991), para realização de baterias de experimentos com relevância estatística.
- **Parte 2: Análise de Dados** – como analisar os dados coletados com linguagens de programação e ferramentas modernas de transformação de dados que habilitam a reprodutibilidade desta análise, envolvendo conceitos como programação literária (KNUTH, 1984) e análise de dados com a linguagem R (R Core Team, 2018).

Esta separação em duas partes reflete um processo metodológico de duas fases. Primeiro, os experimentos são realizados através de mecanismos automáticos enriquecidos com coleta de dados, guiados por um projeto experimental onde constam as variáveis controladas e observadas. Segundo, os dados registrados são analisados pelo analista através de mecanismos, também automáticos, de tratamento de dados. Há portanto uma clara divisão onde a interpretação dos dados observados é realizada *à posteriori*. Considera-se

tal divisão importante pois permite um isolamento da fase de coleta. Os dados coletados podem ser analisados sob múltiplas facetas, permitindo interpretações diversas. Essa separação também traz a vantagem de forçar o experimentador e se preocupar com o registro da maior quantidade de informações do sistema computacional. A preocupação é induzida propositalmente pois uma vez finalizada a primeira parte, ao perceber que variáveis não controláveis não foram registradas, o experimentador deve realizar os experimentos com todos os custos de tempo e recurso associados. Portanto, somente uma reexecução completa garante que as configurações observadas refletem a máquina utilizada no experimento. Este texto está organizado da seguinte forma. A Seção 1.2 apresenta as principais formas de controlar sistemas computacionais, focados em ambientes para processamento de alto nível (SO Linux, múltiplos nós, redes de baixa latência), e de projeto experimental. A Seção 1.3 apresenta métodos de análise com linguagens de programação e ferramentas modernas de manipulação de dados que habilitam a reprodutibilidade desta análise. Enfim, a Seção 1.4 conclui este texto com um sumário do que foi descrito e ponteiros para aprofundar os conceitos apresentados.

1.2. Controle e coleta

A parte de controle e coleta envolve a fase da realização do experimento computacional. No âmbito do processamento de alto desempenho, consideramos que os experimentos são realizados em um *cluster* onde os computadores estão interconectados através de uma rede de interconexão específica para a comunicação de mensagens da aplicação paralela. Um exemplo com quatro nós computacionais está ilustrado na esquerda da Figura 1.1. Um conjunto de processos será executado sobre os vários nós deste tipo de plataforma. Habitualmente, executa-se um processo por *core* disponível nos nós computacionais.

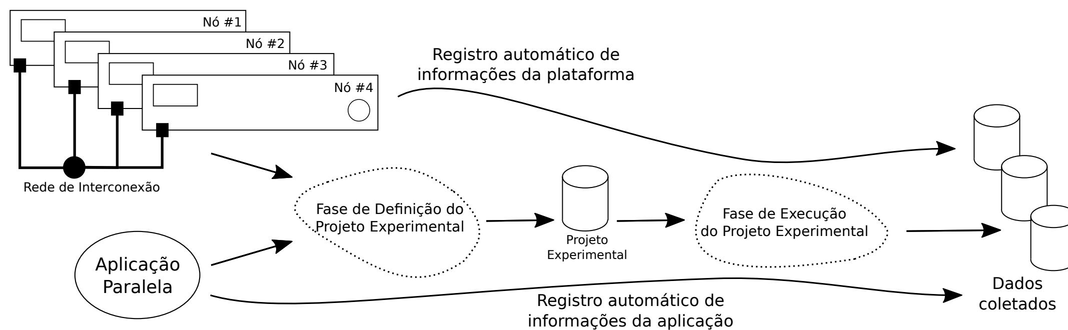


Figura 1.1. Panorama geral do controle e coleta em experimentos computacionais: um *cluster* de computadores com quatro nós e sua rede de interconexão combinado com uma aplicação paralela são sujeitos da definição de um projeto experimental que depois é executado para se coletar os dados para análise.

Tal execução paralela envolve características que impactam o controle e coleta de dados: o indeterminismo da execução paralela, a aparição de anomalias, e a complexidade do sistema computacional. É natural a existência do **indeterminismo** na ordem que as operações são de fato executadas, devido ao caráter concorrente da execução paralela. O aparecimento de **anomalias** inesperadas, em qualquer camada do sistema computacional, pode causar tempos maiores na execução de uma determinada sequência de operações. Enfim, temos a **complexidade** do sistema computacional, com inúmeras camadas em ní-

vel de hardware e software. Essa complexidade torna difícil considerar todas as possíveis facetas configuráveis de um *cluster* de computadores.

A combinação dessas características aumenta a *variabilidade* dos experimentos computacionais. Ou seja, o efeito combinado do indeterminismo, da aparição de anomalias, da complexidade, torna o comportamento de qualquer experimento mais sujeito a variações nas medições. Um exemplo disso é a avaliação do tempo de execução de uma aplicação paralela: além de calcular a média de um determinado conjunto de execuções, o experimentador também calcula a variabilidade da média considerado o intervalo de confiança desejado. Quando maior a dispersão, menos confiável é a média e por consequência qualquer conclusão que possa se tirar do experimento no momento de comparações. Qualquer ação do experimentador para reduzir tal dispersão é benéfico para melhor estudar determinado sistema computacional.

Das três características listadas, pouco pode ser feito em relação ao indeterminismo e ao aparecimento de anomalias. O indeterminismo é de certa forma desejado pois ele permite a execução concorrente, paralela, grande objetivo do processamento de alto desempenho. O aparecimento de anomalias inesperadas é natural em qualquer sistema computacional devido a grande quantidade de camadas de controle existentes, desde o baixo nível do hardware até a aplicação sendo executada. Enfim, para diminuir a variabilidade dos experimentos computacionais, resta controlar manualmente a maior quantidade de configurações possíveis do sistema computacional, diminuindo a sua complexidade.

Esta seção apresenta um resumo de conceitos e boas práticas para controlar um sistema computacional e obter medidas mais significativas. A Seção 1.2.1 apresenta conceitos a respeito da metodologia experimental separada em duas fases. A Seção 1.2.2 apresenta uma *checklist* com boas práticas para controle da complexidade de sistemas computacionais. A Seção 1.2.3 apresenta uma discussão e formas de registrar informações sobre a plataforma e ambiente de execução automaticamente. A Seção 1.2.4 apresenta ferramentas para instalação de dependências para a pilha de software sobre o sistema operacional. A Seção 1.2.5 lista ferramentas de virtualização através de *containers* do Linux para controlar também o sistema operacional. Enfim, a Seção 1.2.6 apresenta um estudo de caso que mostra como tais conceitos e práticas podem ser operacionalizados.

1.2.1. Metodologia experimental

Segundo Jain (JAIN, 1991), um experimento computacional se inicia através da definição de um projeto experimental. Ele deve ser constituído levando-se em conta os objetivos da investigação, definindo quais são as variáveis de controle – os **fatores** – e quais são as **variáveis de resposta**, ou seja, o que será observado. O objetivo é entender como os diferentes valores dos fatores – os **níveis** –influenciam a resposta. Como exemplo, podemos utilizar uma aplicação paralela. Uma variável de resposta pode ser simplesmente o tempo de execução ou a aceleração obtida com a paralelização. Como fatores, podemos considerar que o número de processos (seguindo a quantidade de núcleos de processamento), a quantidade de nós computacionais (de acordo com a disponibilidade do *cluster*), a frequência do processador (no intervalo de valores aceito pelo hardware) e a rede de interconexão (configurações alternativas de largura de banda) podem ter uma influência direta nas variáveis de resposta.

Existem vários tipos de projetos experimentais. Na sua versão mais simples, um projeto é capaz de estudar o impacto dos valores de um único fator, sendo que os valores dos demais fatores se mantém fixos. Tal tipo de projeto não permite o estudo da interação que possa existir entre os fatores. No exemplo anterior, seria inviável estudar a relação entre a quantidade de processos e a rede de interconexão. Tais fatores têm possivelmente uma relação devido a contenção da rede, mais fácil de ser atingida com um número maior de processos comunicantes. Por outro lado, o exemplo mais representativo de um projeto experimental mais complexo é o fatorial completo. Ele permite estudar a influência de todas as combinações de valores de todos os fatores nas variáveis de resposta. Tal projeto é bastante caro de ser executado, pois sua natureza combinatória o torna proibitivo de ser executado com um número já moderado de valores e fatores. Um exemplo intermediário é o projeto fatorial fracionário, onde alguns fatores se mantém fixos enquanto os demais são estudados com todas as combinações. A escolha do projeto experimental depende do recurso de tempo e de plataforma que se deseja investir para entender o fenômeno que se estuda.

A esquerda da Figura 1.1 ilustra a fase de *definição do projeto experimental* culminando na definição do **Projeto Experimental** no centro da imagem. Na prática, este projeto experimental pode consistir em uma tabela onde as colunas representam os fatores, e cada linha representa uma determinada configuração a ser executada na plataforma. Os valores das células nas colunas representam os níveis dos fatores que devem ser adotados por aquela execução específica. A ordem aleatória dos itens do projeto experimental é fundamental, pois permite absorver anomalias inesperadas durante a execução da bateria experimental.

Definido o projeto experimental, passa-se a fase de *execução do projeto experimental*, como ilustrado na direita da Figura 1.1. Essa fase pode ser representada através de programa de computador (idealmente escrito em linguagem de *script)* que lê o projeto experimental e executa a aplicação paralela na plataforma alvo de acordo com os valores de fatores preestabelecidos. É portanto fundamental que tal *script* tenha controle das configurações da plataforma e da aplicação. Embora existem arcabouços que possam tornar genérica tal fase de execução, frequentemente são construídos procedimentos específicos para cada combinação de plataforma e aplicação, dada a especialização obrigatória desta fase. Um conjunto de dados observados, incluindo as variáveis de resposta, é registrado em arquivos de dados. Tais arquivos contém também todas as informações de variáveis não controladas e configurações de sistema.

1.2.2. Boas práticas para controle da complexidade

Como anteriormente discutido, o controle da complexidade da plataforma computacional permite diminuir a variabilidade dos fenômenos sendo estudados. Esse controle visa a reduzir a quantidade de variáveis controláveis, fixando e registrando suas configurações para valores conhecidos de forma que possam ser utilizados mais tarde para a análise dos dados. Quais configurações devem ser realizadas depende bastante de qual tipo de experimento está se conduzido. A listagem a seguir não é exaustiva e se propõem a simplesmente dar uma noção de quais configurações são úteis em determinados contextos.

- ☐ Vinculação (*binding*) fixa de fluxos de execução (*threads*), permite evitar a migração

automática pelos algoritmos de balanceamento de carga embutidos no sistema operacional. Embora esses algoritmos tenham sido concebidos para eventualmente melhorar o desempenho, a migração de *threads* acontece de maneira não explícita, ou seja, a aplicação não fica sabendo e é relativamente difícil rastrear em qual núcleo de processamento (*core*) ela efetivamente está sendo executada em cada intervalo de tempo.

- ☐ Controle de frequência dos núcleos de processamento (*cores*) do processador, permite evitar que o HW ou o SW (neste caso, o sistema operacional), realize mudanças da frequência de processamento. Esse tipo de controle pode ser executado através da fixação de uma política de frequência por usuário, estabelecendo o uso da frequência máxima. Deve-se ter atenção ao fato que o HW, por possuir diversos elementos fechados, pode adotar uma política de frequência inadvertidamente.

- ☐ Desativar *turboboost* (em processadores Intel) pois este faz com que, sob altas demandas de processamento, a frequência seja a máxima possível para aquele processador. Como a ativação deste recurso é de maneira não transparente ao sistema operacional ou à aplicação, cabe desativá-lo para evitar que tal variabilidade afete o entendimento dos fenômenos sendo investigados.

- ☐ Desativar *hyperthreading* (em processadores Intel), ou seja, desativar os núcleos de processamento lógicos, tendo em vista que seus recursos são mais limitados que os núcleos físicos (*cores*). Esse tipo de recurso computacional, por mais que dobre a quantidade de *cores* visíveis em nível de sistema operacional, aumenta a variabilidade experimental. Isso acontece principalmente em aplicações limitadas pela CPU, embora aplicações limitadas pelo acesso à memória possam ter algum ganho de desempenho.

- ☐ Detectar a configuração NUMA do nó computacional e estabelecer uma política fixa de distribuição de fluxos de execução, em *chips* com múltiplos processadores.

- ☐ Configurar uma política TCP/Ethernet adequada para a rede de interconexão, sabendo em várias ocasiões o *kernel* do Linux vem configurado com tamanhos de pacote e outras configurações relacionadas específicas para redes 100 Mbit Ethernet. Esse tipo de configuração pode impactar negativa nas redes de interconexão de alto desempenho tais como 10 GBit Ethernet ou Infiniband.

Outras informações, incluindo outras configurações passíveis de verificação específicas para o sistema operacional Linux, podem ser obtidas em um trabalho relacionado (STANISIC et al., 2017).

1.2.3. Registro automático de informações sobre a plataforma

Usualmente os usuários registram manualmente informações sobre a plataforma na qual os experimentos estão sendo executados. Tais informações, de forma geral, compreendem apenas características básicas do *hardware*, tais como modelo da CPU, tamanho da memória e tipo da interface de rede, e do software como sistema operacional, versão do compilador e distribuição MPI. Além de ser pouco prática, tal estratégia pode incorrer em informações incompletas e até mesmo incorretas. Podemos imaginar uma situação

na qual dados referentes a CPU são coletados antes do início do experimento, neste momento a CPU está operando com uma frequência de 1200 MHz, entretanto o controle de frequência (`governor`) está configurado na opção `ondemand`, o que provavelmente fará com que, durante a execução do experimento, o processador passe a operar em uma frequência bem mais alta (e.g., 2300 MHz).

Para evitar tais situações, é conveniente adotar uma estratégia de registro automático de informações sobre a plataforma. É recomendável que estas informações sejam coletadas toda vez que uma execução for realizada e que sejam armazenadas juntamente com os resultados. Para coletar informações sobre o *hardware*, podemos partir do seguinte conjunto de ferramentas, assumindo um ambiente baseado em Linux. Cabe ressaltar que, além dos comandos abaixo, outros específicos podem ser necessários caso os experimentos envolvam outros recursos de *hardware* como GPUs, FPGAs ou interfaces de rede proprietárias.

Sistema operacional, topologia de hardware e frequência do processador (HW)

- **lstopo** – fornecido pela ferramenta `hwloc`, permite obter a topologia do sistema, incluindo hierarquia de memória cache, nós NUMA, núcleos físicos e lógicos bem como dispositivos PCI conectados.
- **cpufreq-info** – fornecido pela ferramenta `cpufrequtils`, permite obter a frequência atual, mínima e máxima para cada núcleo do processador, de maneira genérica independente do fabricante do processador. Informações sobre a política de controle de frequência atual (`governor`) e as demais disponíveis também podem ser obtidas.
- **pstate driver** – trata-se de um módulo de *kernel* específico para processadores Intel com funcionalidade semelhante àquela fornecida pelo `cpufreq`.
- **lspci** – lista todos os dispositivos PCI conectados ao sistema.
- **ifconfig** (ou **ip** em um Linux moderno) – este comando permite obter as configurações da interface de rede.

Informações associadas à aplicação paralela (SW)

Quanto ao software, além das informações básicas como versão do sistema operacional e do compilador, pode-se obter algumas informações adicionais com os seguintes comandos.

- **ompi-info** – assumindo que a aplicação paralela faz uso da implementação OpenMPI da especificação MPI, este comando permite listar todas as configurações que controlam o comportamento interno da implementação, como *buffers* e protocolos de envio/-recepção.
- **ldd** – mostra as bibliotecas compartilhadas requeridas por um executável e o onde elas se encontram (`PATH`) na árvore de diretórios.

- **env** (assumindo um *shell* baseado em `sh`) – lista as variáveis de ambiente no *shell* corrente.
- **nm** – lista todos os símbolos de arquivos objeto, um programa útil para se utilizar como informação de assinatura de binários executáveis junto com **md5sum**.

1.2.4. Ferramentas para instalação de dependências

Aplicações paralelas que executam em *clusters* de alto desempenho frequentemente apresentam uma pilha de software extensa, incluindo diversos níveis de dependências e parâmetros opcionais. Dessa forma, a configuração do ambiente experimental implica em obter, compilar e ligar dezenas de bibliotecas. Tal cenário motiva a utilização de gerenciadores de pacotes. Entretanto, em ambientes compartilhados como *clusters*, não é viável que os usuários tenham permissão para utilizar o gerenciador de pacotes do sistema (e.g. `dpkg`, `apt`, `rpm`).

Spack (GAMBLIN et al., 2015) é um gerenciador de pacotes que permite aos usuários obter, compilar e instalar programas e bibliotecas em seus próprios diretórios sem fazer uso de privilégios de administrador nem de comandos específicos do sistema operacional. Em oposição a gerenciadores similares de uso geral como o `homebrew` (HOWELL; MCQUAID, 2020), o Spack oferece funcionalidades específicas para ambientes de computação de alto desempenho, entre elas, configurações e dependências personalizadas, instalações não-destrutivas e coexistência de múltiplas instalações. Tais funcionalidades permitem testar e avaliar uma ampla gama de configurações. Os comandos abaixo, ilustram como o Spack pode ser usado para gerenciar diferentes configurações da biblioteca `Boost` que podem coexistir simultaneamente na mesma árvore de diretórios.

- Configuração da biblioteca Boost na versão 1.69.0 com compilador padrão (`gcc`) ligado com a distribuição OpenMPI para prover suporte à interface MPI:

SH

```sh
spack install -v boost@1.69.0+mpi^openmpi
```

- Configuração da biblioteca Boost na versão 1.68.0 com compilador padrão (`gcc`) ligado com a distribuição MPICH para prover suporte à interface MPI:

SH

```sh
spack install -v boost@1.68.0+mpi^mpich
```

- Configuração da biblioteca Boost na versão 1.69.0 com compilador `clang` ligado com a distribuição OpenMPI com compilador `gcc` para prover suporte à interface MPI:

SH

```sh
spack install -v boost@1.69.0+mpi%clang^openmpi%gcc
```

1.2.5. Controle em nível de sistema operacional

Embora o Spack seja uma ferramenta que permita um controle preciso da instalação de bibliotecas e arcabouços, ele não é capaz de gerenciar totalmente a pilha de software. Por exemplo, a forma como as chamadas de sistema são realizadas no sistema operacional, tanto em nível de usuário (através da `libc`), quando em nível de superusuário, se mantém sem controle. Existem alternativas para controlar também o sistema computacional, através de métodos nativos ou virtualizados.

Métodos nativos exigem algum suporte de hardware, tal como a necessidade de gerenciar e utilizar perfis PXE em/de servidores TFTP, disparar comandos de reinicialização através de IPMI ou uma PDU gerenciável, etc. Ferramentas como Kadeploy3 (JEANVOINE; SARZYNIEC; NUSSBAUM, 2013) utilizam tal infraestrutura de hardware para manter em cada nó computacional um sistema operacional principal em uma partição, ao mesmo tempo que possibilita a instalação completa de outros sistemas operacionais em outras partições. O usuário do cluster pode então instalar seu próprio sistema operacional em todos os nós gerenciados por Kadeploy3, se tornando superusuário, com controle completo da pilha de software.

Métodos virtualizados, principalmente aqueles baseados em *containers* Linux, permitem obter o mesmo tipo de controle sem a necessidade de reinicializar a máquina ou de se tornar superusuário. Não exigem também nenhum tipo de hardware específico pois são baseados em recursos do sistema operacional Linux. Exemplos de ferramentas que permitem essa alternativa incluem CharlieCloud (PRIEDHORSKY; RANDLES, 2017) ou Singularity (KURTZER; SOCHAT; BAUER, 2017). Estudos (ALLES; SCHNORR; CARISSIMI, 2018) baseados em *containers* identificaram que esse tipo de controle tem pouco impacto no desempenho de aplicações paralelas quando comparado com execuções nativas.

1.2.6. Estudo de caso com a fatoração de Cholesky

Para ilustrar como um experimento de coleta de dados é realizada na prática, utilizaremos um estudo de caso baseado no uso do *Cholesky Denso* da aplicação *Chameleon*(Agullo et al., 2017), um solucionador de álgebra linear baseado em tarefas que pode utilizar o *runtime* StarPU(AUGONNET et al., 2011).

O projeto experimental envolve utilizar dois tamanhos de matrizes diferentes para realizar a fatoração (4800×4800) e (9600×9600) e dois escalonadores diferentes do StarPU (`random` e `dmdas`). Podemos utilizar o Spack para a instalação do *Chameleon* com o seguinte comando:

SH

```
spack install chameleon@0.9.2+starpu~mpi~cuda ^starpu@1.3.1~fast
   +fxt~mpi~cuda~openmp~examples
spack view soft chameleon chameleon
```

Código 1.1. Configurações iniciais em exemplo para experimento `EXP20` com Chameleon.

Podemos gerar um *Design* de experimentos utilizando o pacote `DoE` do `R`. Exportando para um arquivo chamado `projeto-experimental.csv`.

R

```
suppressMessages(library(DoE.base))

set.seed(0)

btmz_erad <-
    fac.design(factor.names =
                   list(
                       size = c(4800, 9600),
                       scheduler = c("random", "dmdas")),
               replications = 3,
               randomize = TRUE)

export.design(btmz_erad,
              filename = "projeto-experimental",
              type = "csv",
              replace = TRUE)
```

Código 1.2. Criação do projeto experimental

Podemos realizar o experimento utilizando o seguinte código abaixo. Lembrando que este é apenas o script de execução. Cada experimento tem um arquivo `.stdout` gerado na pasta `$EXPEDIR`. No final da execução, todos os tempos estão salvos no arquivo `times`.

SH

```
# ... continuação

# Diretório geral para contar todos os resultados
export EXPEDIR=CHA

# Verificar se projeto experimental é fornecido
PROJETO=projeto-experimental.csv
if [[ -f $PROJETO ]]; then
  echo "O projeto experimental é o seguinte"
  cat $PROJETO | sed -e "s/^/PROJETO|/"
  # Salva o projeto no diretório corrente (da saída)
  cp $PROJETO .
else
  echo "Arquivo $PROJETO está faltando."
  exit
fi
mkdir -p $EXPEDIR
#Criar arquivo de tempos
touch $EXPEDIR/"times"
# Ler o projeto experimental, e para cada experimento
tail -n +2 $PROJETO |
```

```
while IFS=, read -r name runnoinstdorder runno runnostdrp \
        size scheduler Blocks
do
    # Limpar valores
    export name=$(echo $name | sed 's/\"//g')
    export scheduler=$(echo $scheduler | sed 's/\"//g')
    export size=$(echo $size | sed 's/\"//g')
    export KEY="$name-$schduler-$size"
    export STARPU_SCHED=$scheduler
    ./chameleon/bin/timing/time_dpotrf_tile  --nb=960 --n_range=
       $size:$size:$size --nowarmup > $EXPEDIR/${KEY}."stdout"
    out=($(cat $EXPEDIR/${KEY}."stdout" | tail -n 1))
    echo ${out[3]} >> $EXPEDIR/"times"
done
```

Código 1.3. Parte central do experimento CHA com Chameleon.

1.3. Análise de dados

A etapa de análise de dados envolve a fase pós-execução do experimento. Usualmente, esta etapa é executada no computador pessoal do usuário que, em geral, não é a mesma plataforma na qual os experimentos foram executados. A análise de dados consiste em um processo iterativo e reflexivo, no qual o usuário parte de uma análise em alto nível dos dados inicialmente coletados e a partir desta aprofunda-se em pontos específicos. Frequentemente, a análise de dados desenrola-se de maneira iterativa, onde uma análise anterior permite identificar e delimitar cenários e configurações que alimentam uma nova execução da etapa de controle e coleta como detalhado na Seção 1.2. Esta nova execução gerará mais dados, que implicarão em uma nova iteração do processo de análise.

As características do processo de análise de dados motivam a adoção de uma estratégia sistematizada, que permita, facilmente, reexecutar algumas etapas do processo de análise bem como revisar o fluxo de experimentos, reflexões e conclusões que levou a uma determinada suposição ou resultado. A adoção de tal estratégia é benéfica, não apenas ao usuário durante o desenvolvimento do trabalho, mas também às outras partes envolvidas no processo científico e que não, necessariamente, estarão próximas temporal ou fisicamente do usuário, tais como orientadores, revisores de publicações, autores de trabalhos relacionados ou até mesmo o próprio autor em momento futuro. Assim, esta seção tem por objetivo ilustrar ferramentas e conceitos que permitam sistematizar e disponibilizar a análise de desempenho.

O restante desta seção apresentará conceitos e ferramentas que podem ser empregados no desenvolvimento de uma análise de dados reprodutível. A Seção 1.3.1 ilustra como a programação literária pode ser usada no processo de análise de dados experimentais. Já a Seção 1.3.2 discute conceitos relacionados a reprodutibilidade da análise de desempenho em si tais como formato e plataforma de distribuição.

1.3.1. Programação literária

A Programação Literária proposta por Donald Knuth (KNUTH, 1984) tem por base duas operações (*weave* e *tangle*) que permitem converter um documento fonte em duas representações distintas, um formato legível para humanos e outro apto para execução em computadores. Esta funcionalidade é bastante útil na análise de resultados experimentais pois permite manter em um mesmo documento tanto as anotações preliminares (expectativas, suposições e reflexões acerca do experimento) quanto os comandos usados para (1) a execução do projeto experimental e (2) posterior processamento e visualização dos seus resultados.

A extensão Org-mode (DOMINIK, 2010) do editor de texto Emacs (STALLMAN et al., 2017) oferece, entre outros recursos, funcionalidades de programação literária. Arquivos criados com esta extensão (arquivos `org`) são arquivos em texto puro que podem ser abertos e lidos em qualquer editor de texto, embora seja conveniente o uso do editor Emacs para aproveitamento de todas as funcionalidades. O pacote `Babel` possibilita a definição de blocos ativos de código e de dados dentro de documentos `org`, tais blocos podem ser avaliados e a saída correspondente é capturada e incluída no documento. Os blocos podem ser escritos em diferentes linguagens de programação, e podem ser encadeados de forma que os dados de saída produzidos por um bloco sejam usados como entrada de outro. Os trechos de código abaixo ilustram o comportamento desta funcionalidade. O bloco a seguir é escrito em `shell script`, com a possível saída produzida representada na tabela abaixo.

SH

```
for n in $(seq 5);
do
    printf "%d $RANDOM \n" $n ;
done
```

Tabela 1.1. Possível saída produzida pelo trecho de código em `shell script`

1	21020
2	20873
3	7597
4	19882
5	30785

A avaliação do trecho de código acima produzirá uma saída, que será encadeada como entrada do código abaixo escrito na linguagem R. Quando avaliado, o código abaixo produzirá como saída, uma imagem contendo um gráfico construído a partir dos dados gerados pelo primeiro trecho de código.

R

```
library(ggplot2)
library(tidyverse)
dados %>%
    ggplot(aes(x = V1, y = V2)) +
    geom_point() +
    theme_bw()
```

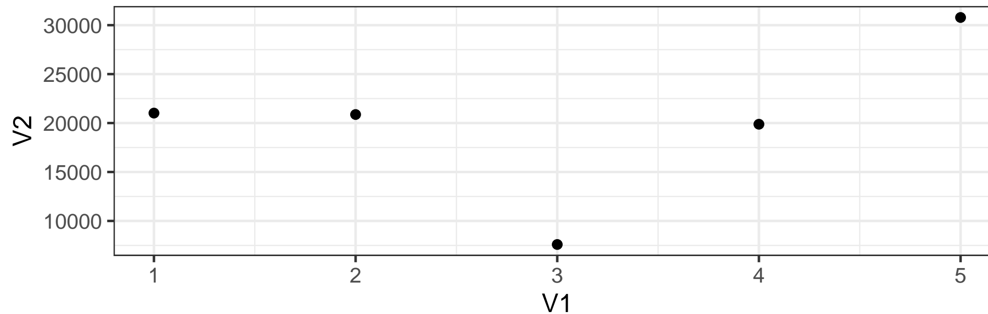


Figura 1.2. Gráfico gerado pelo código R utilizando a saída do código shell como entrada

Um mesmo conjunto de dados pode ser usado inúmeras vezes como entrada para blocos de código diversos. Além do gráfico da Figura 1.2, podemos usar os dados da Tabela 1.1 para calcular valores estatísticos como mínimo, máximo, média, mediana e quartis. O trecho de código abaixo ilustra o comando em R que permite a obtenção destas informações.

R

```
dados$V2 %>% summary()
```

```
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
7597   19882   20873   20031   21020   30785
```

A programação literária, por si só, já é uma prática aconselhada para facilitar o registro e análise de resultados. Entretanto, ela não garante que os gráficos gerados na análise em questão sejam claros e diretos. O resultado do processo de criação de gráficos pode ser aprimorado com a aplicação de alguns passos de verificação e controle (JAIN, 1991; SCHNORR; VINCENT, 2018; POLDRACK, 2018) apresentados na Tabela 1.2.

Tabela 1.2: *Checklist* para gráficos

Dados	✓	O tipo do gráfico é adequado para a natureza do dado (curva, barras, setores, histograma, nuvem de pontos, etc)
	✓	As aproximações/interpolações fazem sentido
	✓	As curvas são definidas com um número suficiente de pontos
	✓	O método de construção da curva é claro: interpolação (linear, polinomial, regressão, etc)
	✓	Os intervalos de confiança são visualizados (ou informados separadamente)
	✓	Os passos do histograma são adequados
	✓	Histogramas visualizam probabilidades (de 0 a 1)
Objetos Gráficos	✓	Os objetos gráficos são legíveis na tela, na versão impressa (P&B), em vídeo, etc
	✓	O intervalo do gráfico é padrão, sem cores muito similares, sem verde (vídeo)
	✓	Os eixos do gráfico estão claramente identificados e rotulados
	✓	Escalas e unidades estão explícitas
	✓	As curvas se cruzam sem ambiguidade
	✓	As grades ajudam o leitor
Anotações	✓	Eixos são rotulados por quantidades
	✓	Rótulos dos eixos são claros e autocontidos
	✓	Unidades estão indicadas nos eixos
	✓	Eixos são orientados da esquerda para a direita e de baixo para cima
	✓	Origem é (0,0), caso contrário deve estar claramente justificada
	✓	Sem buracos nos eixos
Anotações (2)	✓	Para gráficos de barras/histogramas a ordem das barras segue a ordenação clássica (alfabética, temporal, do melhor pro pior)
	✓	Cada curva tem uma legenda
	✓	Cada barra tem uma legenda
Informação	✓	Curvas estão na mesma escala
	✓	O número de curvas em um mesmo gráfico é pequeno (menor que 6)
	✓	Compare as curvas no mesmo gráfico
	✓	Uma curva não pode ser removida sem redução de informação
	✓	O gráfico fornece informações relevantes ao leitor
	✓	Se o eixo vertical mostra médias, as barras de erro devem estar presentes
	✓	Não é possível remover qualquer objeto sem modificar a legibilidade do gráfico
Contexto	✓	Todos os símbolos são definidos e referenciados no texto
	✓	O gráfico produz mais informação que qualquer outra representação (escolha da variável)
	✓	O gráfico tem um título

Continua na próxima página

Continuação da página anterior

✓	Histogramas visualizam probabilidades (de 0 a 1)
✓	O título é suficientemente autocontido para a compreensão parcial do gráfico
✓	O gráfico é referenciado no texto
✓	O texto comenta a figura
✓	**A representação gráfica deve ser elegante**

Ao aplicarmos as orientações da Tabela 1.2 à Figura 1.2, notamos que o gráfico em questão pode ser aprimorado. Inicialmente, devemos adicionar rótulos aos eixos vertical e horizontal. Dada a natureza aleatória dos dados, não há unidades a serem indicadas nos eixos. Em seguida, adicionamos um título ao gráfico, e por fim verificamos que a origem deve ser o ponto (0,1) e não (0,0) pois as observações foram numeradas a partir de 1. O código R abaixo produz a Figura 1.3 que contém a versão aprimorada do gráfico.

R

```
library(ggplot2)
library(tidyverse)
dados %>%
    ggplot(aes(x = V1, y = V2)) +
    theme_bw() +
    geom_point() +
    ylab("Valor Aleatório") +
    xlab("Observação") +
    ggtitle("Geração de Números Aleatórios em shell script") +
    lims(y = c(0, NA), x = c(1, NA))
```

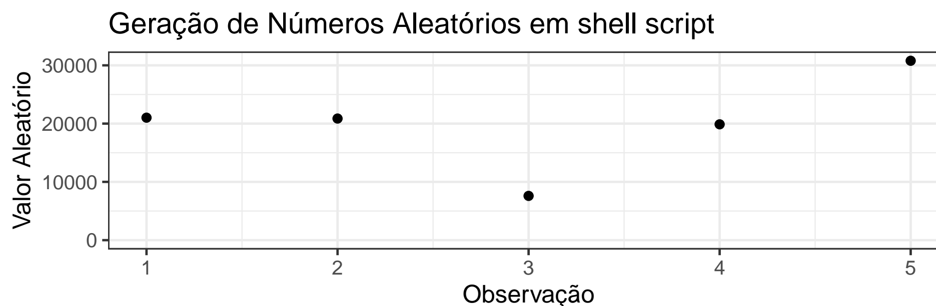


Figura 1.3. Gráfico gerado pelo código `R` utilizando a saída do código `shell` como entrada (versão aprimorada)

1.3.2. Reprodutibilidade da análise de desempenho

Demonstrabilidade e reprodutibilidade são conceitos-chave no método científico. Entretanto, frequentemente, tais processos ficam limitados ou comprometidos devido à falta de informações além do texto científico. No contexto da computação, e em especial da área de análise de desempenho, a disponibilização do código fonte e dos dados de entrada e saída são essenciais para permitir a reprodutibilidade dos experimentos.

Existem dois aspectos principais que devem ser considerados na disponibilização de anexos de publicações científicas. O primeiro deles se refere ao formato utilizado que deve ser aberto e de estrutura simples. O formato CSV, por exemplo, é um formato adequado para disponibilização de resultados numéricos brutos, pois é simples e de fácil leitura tanto por seres humanos quanto por ferramentas automatizadas. Para disponibilização de resultados qualitativos, que incluam não apenas os dados brutos mas também análises e reflexões, pode-se usar formatos que ofereçam alguma estrutura hierárquica e permitam criar uma espécie de caderno de laboratório, tais como Org-mode (apresentado na Subseção 1.3.1), R Markdown (BAUMER et al., 2014) ou IPython (PÉREZ; GRANGER, 2007).

O segundo aspecto refere-se à plataforma utilizada para disponibilização dos dados. Lamentavelmente, os repositórios de textos científicos, tais como IEEE Xplore[1], ACM DL[2] e Portal de Conteúdo da SBC[3], ainda não oferecem espaço para armazenamento de anexos de artigos. Idealmente, esses dados deveriam estar disponíveis juntamente com o texto científico.

A alternativa passa a ser a disponibilização do material complementar em outras plataformas não necessariamente científicas, incluindo, no texto científico, uma referência (*link*) para o material. Neste caso, os principais pontos a serem considerados são a livre acessibilidade, a perenidade, versionamento e o espaço disponível. Embora de fácil acesso, soluções baseadas em computação em nuvem como Dropbox, Onedrive e Google Drive tendem a ser limitadas em termos de perenidade, versionamento e espaço disponível. O uso de páginas pessoais em servidores institucionais tende a ser mais flexível, porém está sujeito a política de cada instituição. Plataformas de hospedagem e versionamento de código fonte como GitHub[4], Bitbucket[5] e GitLab[6] são boas opções em termos de acessibilidade e versionamento porém implicam restrições quando é necessário armazenar dados não-textuais ou ainda em grande volume. Por fim, pode-se citar plataformas voltadas para armazenamento de dados científicos como o Figshare[7] e o Zenodo[8]. Estas plataformas permitem o armazenamento de qualquer formato de arquivo com poucas restrições de tamanho. Cada registro recebe um identificador DOI, o que facilita a busca e a citação dos conjuntos de dados. A principal limitação dessas plataformas está relacionada a impossibilidade de corrigir ou apagar registros já publicados, o que pode ser um limitante para trabalhos em andamento ou sob revisão. Tal limitação, entretanto, pode ser contornada por meio da integração nativa com plataformas como GitHub, o que facilita a publicação de *releases* ou de resultados consolidados.

[1]<https://ieeexplore.ieee.org/>
[2]<https://dl.acm.org>
[3]<https://portaldeconteudo.sbc.org.br/>
[4]<http://github.com/>
[5]<https://bitbucket.org/>
[6]<https://gitlab.com/>
[7]<http://figshare.com/>
[8]<https://zenodo.org/>

1.4. Conclusão

Este minicurso sensibiliza os participantes da importância do emprego de boas práticas na realização de experimentos computacionais na área de processamento paralelo de alto desempenho. Após uma breve motivação, o minicurso se divide em duas partes, uma primeira que trata dos procedimentos de controle e coleta de dados experimentais, seguindo de uma segunda parte que trata da análise dos dados de maneira reprodutível.

As boas práticas apresentadas neste minicurso não são exaustivas. O enfoque dado foi em experimentos computacionais limitados pela CPU, levando a verificações relacionadas a vinculação de fluxos de execução aos núcleos de processamento, por exemplo. Caso os experimentos tenham um enfoque na rede, em entrada/saída (disco), em memória `RAM`, em uso de GPUs, ou algum outro aspecto computacional, novas diretivas de controle e verificação devem ser concebidas. Essas novas diretivas podem envolver também elementos de software (bibliotecas, arcabouços, *middlewares*). De maneira colaborativa foi instituído um repositório intitulado “Good Practices for Computational Experiments in High Performance Clusters”[9] para organizar tais diretivas.

Enfim, lembramos que qualquer decisão experimental requer claramente um ponto de vista crítico no seu emprego. Cada experimento tem suas particularidades que devem ser levadas em conta na hora de escolher quais tipos de controle devem ser executados antes, durante o experimento. Espera-se mesmo assim que o texto deste minicurso ressalte a importância de procedimentos sistemáticos na condução de experimentos computacionais, de forma a culminar em resultados credíveis.

Agradecimentos

Este trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001, do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), e dos projetos: FAPERGS ReDaS (19/711-6), MultiGPU (16/354-8) e GreenCloud (16/488-9), do projeto CNPq 447311/2014-0, do projeto CAPES/Brafitec 182/15 e CAPES/Cofecub 899/18, e com apoio do projeto Petrobras (2018/00263-5).

Referências

Agullo, E. et al. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, p. 1–1, 2017. ISSN 2161-9883. páginas 9

ALLES, G.; SCHNORR, L. M.; CARISSIMI, A. Assessing the computation and communication overhead of linux containers for hpc applications. In: *Anais do Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*. [S.l.]: Sociedade Brasileira de Computação, 2018. páginas 9

AUGONNET, C. et al. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, v. 23,

[9]<https://gitlab.com/schnorr/experiments>

n. 2, p. 187–198, 2011. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1631>. páginas 9

BAUMER, B. et al. R markdown: Integrating a reproducible analysis tool into introductory statistics. *arXiv preprint arXiv:1402.1894*, 2014. páginas 15

DOMINIK, C. *The Org Mode 7 Reference Manual - Organize Your Life with GNU Emacs*. [S.l.]: Network Theory Ltd., 2010. ISBN 9781906966089. páginas 12

GAMBLIN, T. et al. The spack package manager: Bringing order to hpc software chaos. In: IEEE. *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. [S.l.], 2015. p. 1–12. páginas 8

HOWELL, M.; MCQUAID, M. *Homebrew: The Missing Package Manager for macOS (or Linux)*. 2020. <https://brew.sh/>. páginas 8

JAIN, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. [S.l.]: Wiley, 1991. ISBN 9780471503361. páginas 2, 4, 13

JEANVOINE, E.; SARZYNIEC, L.; NUSSBAUM, L. Kadeploy3: Efficient and Scalable Operating System Provisioning. *USENIX ;login:*, v. 38, n. 1, p. 38–44, fev. 2013. páginas 9

KNUTH, D. E. Literate Programming. *The Computer Journal*, Oxford University Press, v. 27, n. 2, p. 97–111, 2 1984. ISSN 0010-4620. páginas 2, 11

KURTZER, G. M.; SOCHAT, V.; BAUER, M. W. Singularity: Scientific containers for mobility of compute. *PloS one*, Public Library of Science, v. 12, n. 5, p. e0177459, 2017. páginas 9

PÉREZ, F.; GRANGER, B. E. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, IEEE, v. 9, n. 3, p. 21–29, 2007. páginas 15

POLDRACK, R. A. *Statistical Thinking for the 21st Century*. Self-published, 2018. Disponível em: <https://statsthinking21.org>. páginas 13

PRIEDHORSKY, R.; RANDLES, T. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In: ACM. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. [S.l.], 2017. p. 36. páginas 9

R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2018. Disponível em: <https://www.R-project.org/>. páginas 2

SCHNORR, L. M.; VINCENT, J.-M. *Literate Programming and Statistics (CMP595)*. 2018. <https://github.com/schnorr/lps>. páginas 13

STALLMAN, R. et al. *GNU Emacs Manual*. 17. ed. Boston, USA: Free Software Foundation, 2017. 635 p. Disponível em: <https://www.gnu.org/software/emacs/manual/pdf/emacs.pdf>. páginas 12

STANISIC, L. et al. Characterizing the Performance of Modern Architectures Through Opaque Benchmarks: Pitfalls Learned the Hard Way. In: *IPDPS 2017 - 31st IEEE International Parallel & Distributed Processing Symposium (RepPar workshop)*. Orlando, United States: [s.n.], 2017. Disponível em: <https://hal.inria.fr/hal-01470399>. páginas 6

 DOI: 10.5753/sbc.4400.9

Capítulo 2

Introdução à Programação com OpenACC

Evaldo B. Costa, Gabriel P. Silva
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brasil

Resumo

O OpenACC (programação para aceleradores) é um modelo de programação para computação paralela desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre vários tipos de arquiteturas: multicore, manycore e GPUs. Este minicurso tem por objetivo apresentar este novo modelo de programação e suas facilidades de uso. A proposta deste minicurso é oferecer uma introdução à programação com OpenACC através de uma abordagem expositiva como: uma visão geral dos conceitos, diretivas e cláusulas; acrescidas da utilização de exemplos com os diversos tipos de arquiteturas alvo.

2.1. Introdução

As arquiteturas paralelas tem alcançado um alto grau de paralelismo com a utilização de um número cada vez maior de processadores, como podem ser encontrados nas arquiteturas *multicore, manycore* e GPUs (Figura 2.1). Esses tipos de arquitetura, com uso de processamento paralelo maciço, são usados extensivamente em aplicações nas áreas de geofísica, sequenciamento genético, simulações de modelos matemáticos e previsão do tempo, entre outras (SILVA, 2018) (COSTA; SILVA; TEIXEIRA, 2018).

Entre os tipos de arquiteturas existentes destacam-se as GPUs e *manycore*. Desenvolvida pela NVIDIA nos anos 90 a GPU (*Graphical Processing Unit*) se baseia em um grande número de núcleos para processamento paralelo maciço com foco na eficiência energética e para aplicações com demandas que melhorem o *throughput*. Inicialmente desenvolvidos com o objetivo de atender a área de jogos, rapidamente mostrou-se muito eficiente em outras áreas (A.; M., 2012).

Em 2010 a Intel iniciou os primeiros estudos para o desenvolvimento da arquitetura MIC (*Many Integrated Core*). A arquitetura do Intel MIC (*Many Integrated Core*) oferece paralelismo maciço e vetorização com foco em computação de alto desempenho

(HPC), que utiliza grandes demandas de dados para processamento. Atualmente essas arquiteturas estão presentes em todos os setores, como a engenharia, medicina, economia e finanças (RAHMAN, 2013).

Na programação dessas arquiteturas são utilizados modelos de programação como CUDA, OpenCL e OpenACC. Neste minicurso usaremos o modelo de programação OpenACC, que surgiu em 2011 utilizando diretivas de compilação de alto nível e foi desenvolvido para o uso em aceleradores como os produzidos pela NVIDIA e Intel (LARKIN, 2018).

O OpenACC foi desenvolvido pelos principais fabricantes de hardware e software como a Cray, CAPS, NVIDIA e PGI, com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e tornando possível a portabilidade do código independente de qual arquitetura foi desenvolvida inicialmente (OPENACC-STANDARD.ORG, 2015) (CHEN, 2017).

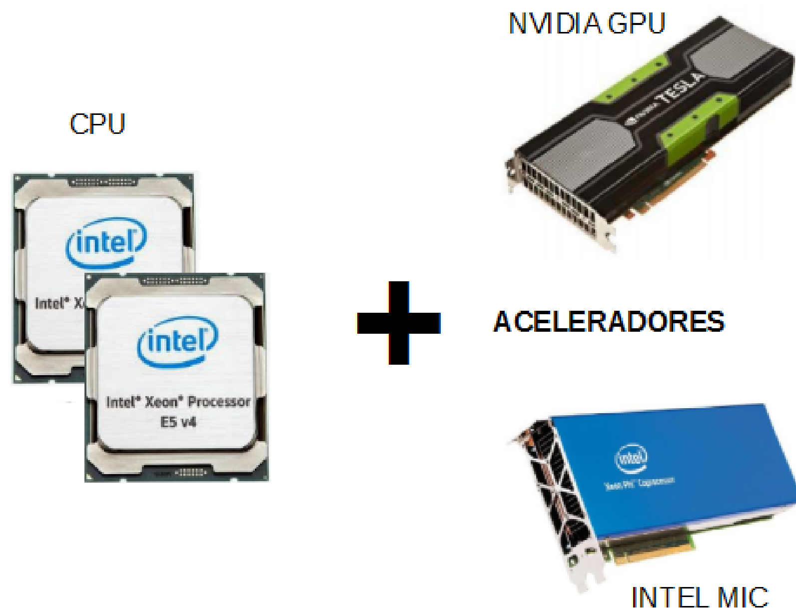


Figura 2.1: Tipos de arquiteturas: multicore, GPU e manycore

As *threads* são executados por processadores escalares. Os *thread blocks* são executados em multiprocessadores e não podem migrar. Diversos *thread blocks* podem residir em um multiprocessador, limitados pelo total de recursos disponíveis no multiprocessador, tais como memória compartilhada e banco de registradores. Um *kernel* é lançado como uma grade de *thread blocks*. Veja na Figura 2.2.

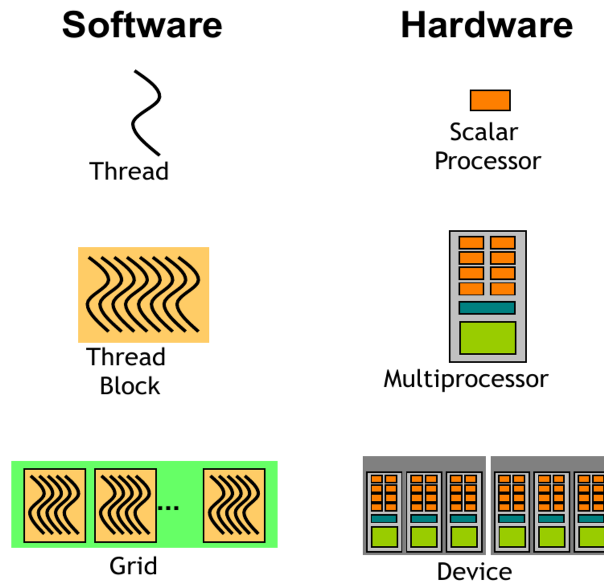


Figura 2.2: Modelo de Execução (ABBOTT, 2017)

Este minicurso introdutório à programação OpenACC consiste de três partes: na primeira parte serão abordados os conceitos iniciais, como definição do modelo, vantagens e desvantagens e onde melhor se aplica o seu uso; na parte seguinte estudaremos quais as principais diretivas usadas; finalmente na última parte veremos quais compiladores suportam o OpenACC e como compilar um programa, sendo que também examinaremos um exemplo com análise dos resultados.

2.2. Referencial teórico

2.2.1. Avaliação de desempenho

Antes de iniciarmos o estudo do OpenACC é importante destacar alguns conceitos sobre como avaliar o desempenho de um programa paralelo.

Uma das questões que surgem ao elaborar um programa paralelo é saber se o mesmo apresenta um desempenho adequado quando executado em um ambiente paralelo. Deste modo, para avaliar o desempenho de um sistema de processamento paralelo, as métricas mais importantes para serem consideradas são: *speedup* (também conhecido como ganho de desempenho ou aceleração), eficiência e escalabilidade.

São diversos fatores que influenciam essas métricas, tais como o custo de sincronização e comunicação, a distribuição das tarefas entre os processadores e o percentual do tempo de execução do programa que é passível de paralelização.

2.2.1.1. Speedup

O ***speedup***, ou aceleração, mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador ($T(1)$) e o tempo gasto na execução com P

processadores ($T(P)$), como visto na Equação 1:

$$S(P) = \frac{T(1)}{T(P)} \quad (1)$$

Em condições ideais, quando o *speedup* é sempre igual a P, onde P é o número de processadores em uso, temos o chamado *speedup* linear. Mas, em geral, o *speedup* é menor do que P, devido principalmente à sobrecarga de comunicação entre os diferentes fluxos de execução do programa, perdas por sincronização e decomposição de tarefas mal feita. Quando isso acontece, chamamos o *speedup* de sublinear. Essa situação pode se deteriorar até o ponto em que a adição de mais processadores diminui, em vez de aumentar, o ganho obtido, caracterizando o que se chama de "retorno negativo".

Em algumas situações especiais, *speedups* superiores a P podem ser obtidos (denominados "*speedups* superlineares"). Exemplos destas situações são aplicações onde o volume de dados manipulados é grande o suficiente para exceder o tamanho da memória cache de um único processador. Nesse caso, ao dividir a aplicação entre P processadores, o volume de dados manipulado por cada processador é aproximadamente dividido por P, sendo este volume agora pequeno o suficiente para poder ser armazenado integralmente, ou com baixo nível de interferência destrutiva, na memória cache de cada processador. Dentro deste quadro, e respeitadas as condições mencionadas anteriormente, é razoável se esperar um *speedup* superior a P no processo de paralelização, já que o desempenho com um único processador fica muito prejudicado pela baixa taxa de acerto nos acessos à memória cache.

Uma situação análoga ocorre quando são feitas buscas em grandes bases de dados, tais como a busca de dados genômicos realizadas por diversas implementações paralela do programa BLAST. Neste caso, a quantidade de memória RAM acumulada de cada um dos nós em um *cluster* permite que a base de dados se mova do disco inteiramente para a memória, reduzindo portanto dramaticamente o tempo requerido, por exemplo, para o mpiBLAST percorrer toda a base de dados (CORREA; SILVA, 2012).

O *speedup* superlinear pode ocorrer também, sob determinadas condições, quando da execução de algoritmos de *backtracking* e *branch and bound* paralelos (SILVA, 2018). A Figura 2.3 ilustra diferentes curvas de *speedups*.

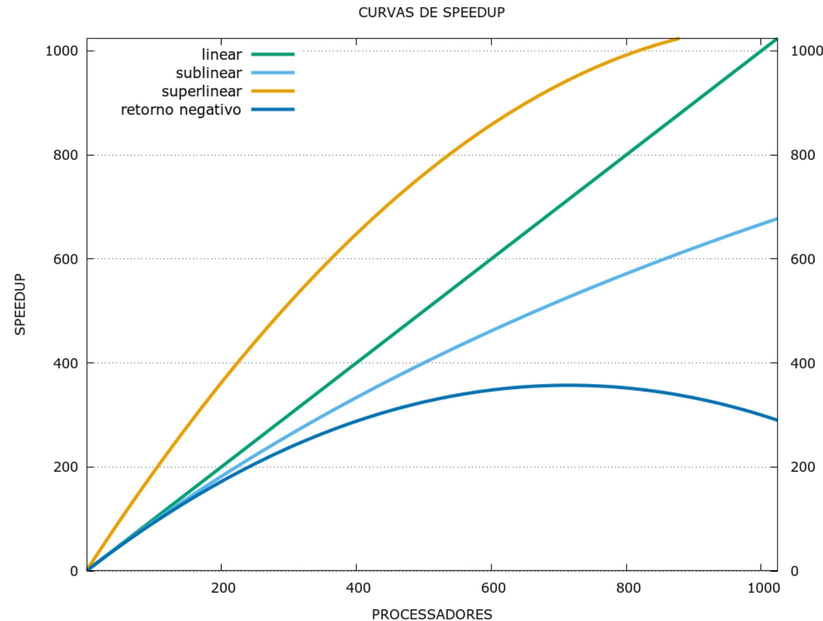


Figura 2.3: Curvas de *speedup*

2.2.1.2. Eficiência

A **eficiência** é uma medida de quão efetiva é a adição de novos processadores para ajudar na resolução de um problema. Seu valor é obtido pela razão entre o *speedup* ($S(P)$) e o número de processadores (P) utilizados para obter este *speedup*, conforme podemos observar na equação 2:

$$E(P) = \frac{S(P)}{P} \tag{2}$$

Novamente, como o *speedup* em geral é menor do que P por conta da sobrecarga do processamento paralelo, a eficiência tipicamente assume um valor menor do que 1, mas, preferencialmente, próximo de 1. Para se obter *speedup* muito próximo de P e, consequentemente, eficiência muito próxima de 1, as seguintes condições devem normalmente ser satisfeitas:

- no código a ser paralelizado, o percentual de código não paralelizável (que continuará sendo executado de forma sequencial) é muito pequeno;
- no processo de paralelização, a distribuição de carga de trabalho entre os P processadores deve ser homogênea;
- os processadores trabalham nos trechos de código executados em paralelo de forma bastante independente, requerendo muito pouca comunicação ou sincronização entre eles.

2.2.1.3. Escalabilidade

Um sistema é dito escalável quando sua eficiência se mantém constante à medida que o número de processadores P aplicado à solução do problema cresce. Se o tamanho do problema é mantido constante e o número de processadores aumenta, o *overhead* de comunicação tende a crescer e a eficiência a diminuir. Na prática, uma análise da escalabilidade deve considerar a possibilidade de se aumentar proporcionalmente o tamanho do problema a ser resolvido à medida que P cresce de forma a contrabalançar o natural aumento do *overhead* de comunicação quando P cresce.

Considere como exemplo um problema de tamanho S. Usando P processadores esse problema leva um tempo T para ser executado. O sistema é dito escalável se um problema de tamanho $2S$ executado em $2P$ processadores leva o mesmo tempo T. Escalabilidade é frequentemente uma propriedade mais desejável que o *speedup*.

É importante ter essas métricas em mente, para sabermos se os programas paralelos que iremos desenvolver em OpenACC estão realmente tendo o desempenho esperado.

2.3. Programação em OpenACC

O OpenACC é um modelo de programação aberta para computação paralela desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre diversos tipos de arquiteturas: *multicore, manycore* e GPUs.

O OpenACC é compatível com os modelos de programação OpenMP e MPI, ambas as abordagens podem ser combinadas com o OpenACC. Em geral, as diretivas do OpenACC são muito semelhantes às do OpenMP. Em relação ao CUDA, OpenACC é totalmente compatível tornando a necessidade de alteração do código a menor possível.

O modelo de programação OpenACC possui algumas características que o tornam fácil e simples de utilizar, como:

- Independente de fabricante;
- Oculta a complexidade do *hardware* dos programadores;
- Requer poucas modificações ao código fonte;
- Mais fácil de programar e depurar que o CUDA;
- Possui algumas facilidades que o CUDA não oferece;
- Mesmo código por ser usado em *multicore, manycore* e GPUs;
- Similar ao OpenMP (familiaridade);
- Fácil transição para o OpenMP 4.5 (futuro).

Antes de iniciar o processo de programação utilizando o OpenACC é recomendado que se faça uma análise do código a ser paralelizado seguindo o ciclo de desenvolvimento e análise de código conforme descrito:

1. Analisar o código para determinar quais as regiões mais prováveis que podem ser paralelizadas ou otimizadas.
2. Paralelizar o código iniciando com as regiões com o maior tempo de execução.
3. Otimizar o código para melhorar o tempo de execução observado a partir das alterações executadas.

Na Figura 2.4 é apresentado o ciclo de desenvolvimento e análise do código.

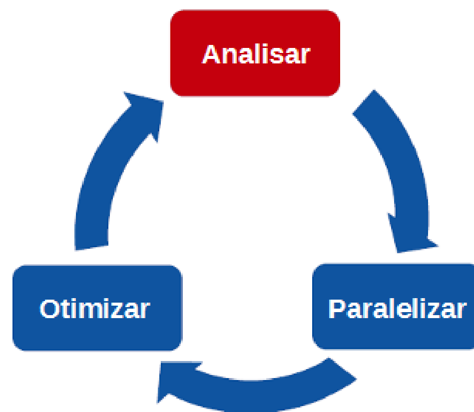


Figura 2.4: Ciclo de desenvolvimento e análise do código

O OpenACC pode ser utilizado em códigos programados em linguagens de programação C/C++ ou Fortran, deve-se atentar a sintaxe correta para tipo de linguagem de programação correta conforme a Figura 2.5.

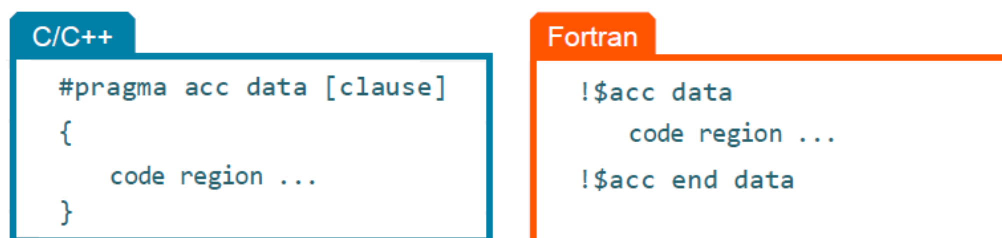


Figura 2.5: Sintaxe do OpenACC em C/C++ e Fortran

2.4. Diretivas e cláusulas

O modelo de programação usado no OpenACC é baseado em diretivas e cláusulas. As diretivas são comandos de instrução passadas pelo programador ao compilador. Cláusulas são os parâmetros adicionais atribuídos às instruções usadas nas diretivas.

As diretivas do OpenACC são muito parecidas com as diretivas do OpenMP. As diretivas são escritas na forma de **pragma**. Existem vantagens em usar diretivas, uma delas é o fato de o código precisar de pequenas modificações, as mudanças podem ser

feitas de forma incremental, um **pragma** de cada vez. Com o uso desse processo torna-se especialmente útil para fins de depuração, já que fazer uma única alteração permite identificar rapidamente um erro.

O uso do OpenACC pode ser desativado em tempo de compilação. Quando o suporte OpenACC está desabilitado na compilação, o **pragma** referente ao OpenACC é considerado um comentário, sendo ignorado pelo compilador. Com isso, um único código pode ser usado para compilar tanto um código com suporte a acelerador quanto uma versão sequencial.

2.4.1. Movimentação de dados

Um grande fator de impacto de desempenho no processamento paralelo é a movimentação de dados, principalmente quando se faz o processamento dos dados em lugares diferentes. Quando se usa processamento em aceleradores nem sempre é possível carregar todos os dados para o acelerador, isso ocorre em geral porque a memória da CPU é maior a dos aceleradores, embora os aceleradores tenham maior largura de banda (Figura 2.6).

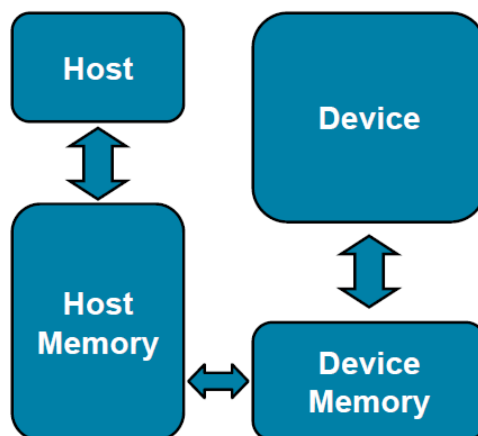


Figura 2.6: Modelo básico de movimentação de dados entre host e o acelerador (CHEN, 2017)

A movimentação de dados entre o *host* e o acelerador é feita através do barramento, que é lento em comparação com largura de banda de memória. Por sua vez o acelerador não pode executar o processamento dos dados até que eles estejam na sua memória local.

Para realizar a movimentação de dados entre o *host* e o acelerador durante a execução do programa é necessário o uso das cláusulas de dados. As cláusulas de movimentação de dados podem ser usadas nas diretivas **kernels** ou **parallel** e tem como principais caraterísticas:

- Define a região do código na qual os dados permanecem no acelerador;
- Define quais dados são compartilhados entre todos os *kernels* de uma região paralela;

- Realiza transferências de dados explícitas.

```
#pragma acc data [clause]
```

Cláusula	Descrição
copy	Cria espaço para as variáveis listadas no dispositivo, inicia as variáveis copiando dados para o dispositivo no início da região, copia os resultados de volta para o *host* no final da região e finalmente libera o espaço no dispositivo quando terminar.
copyin	Cria espaço para as variáveis listadas no dispositivo, inicia a variável copiando os dados para o dispositivo no início da região e libera o espaço no dispositivo quando terminar, sem copiar os dados de volta para o *host*.
copyout	Cria espaço para as variáveis listadas no dispositivo, mas não as inicia. No final da região, copia os resultados de volta para o *host* e libera o espaço no dispositivo.
create	cria espaço para as variáveis listadas e as libera no final da região, mas não copia nenhum dos dados de/para o dispositivo.
present	As variáveis listadas já estão presentes no dispositivo, portanto, nenhuma outra ação precisa ser executada. Isso é usado com mais frequência quando existe uma região de dados em uma rotina de maior nível.
deviceptr	As variáveis listadas usam a memória do dispositivo que foi gerenciada fora do OpenACC, portanto as variáveis devem ser usadas no dispositivo sem qualquer conversão de endereço. Esta cláusula é geralmente usada quando o OpenACC é misturado com outro modelo de programação.

Tabela 2.1: Cláusulas da Diretiva Data

2.5. Diretiva parallel

A paralelização usando o construtor **parallel** identifica uma região de código que será paralelizada, quando executada em conjunto com a diretiva **loop**, o compilador gerará uma versão paralela do laço para o acelerador. Desta forma o compilador executa o paralelismo do código de forma direta.

```
#pragma acc paralell [clause-list]
```

Neste modo de programação, os laços que serão paralelizados precisam ser definidos no código, o compilador não consegue identificar os laços que precisam ser paralelizados, se a diretiva **loop** não for especificada não será feita a sua paralelização.

2.5.1. Cláusulas da diretiva Parallel

Algumas cláusulas são usadas para melhorar o desempenho da região a ser paralelizada, essas cláusulas permitem ter um maior controle e processamento dos laços.

Cláusula	Descrição
private	A cláusula privada especifica que cada iteração do laço tem a sua própria cópia das variáveis listadas
reduction	A redução é executada com as operações suportadas: + * max min
async	Retira as barreiras implícitas no final da região paralela

Tabela 2.2: Cláusulas da Diretiva Parallel

A cláusula **private** da construção **parallel** vai privatizar as variáveis listadas para cada *gang* (veremos mais detalhes na seção seguinte) na região paralela.

A cláusula **reduction** funciona de forma similar à cláusula **private** de forma que é gerada uma cópia privada das variáveis, mas existe uma redução ao final da região paralela de todas as cópias privadas em um único resultado final, que é retornado ao sair da região paralela. As operações de redução possíveis são soma, multiplicação, máximo e mínimo. O formato para a cláusula de redução é como a seguir:

```
#pragma acc parallel reduction(operator:variable)
```

2.6. Diretiva parallel loop

A construção **loop** fornece ao compilador informações adicionais sobre o próximo laço no código-fonte. A diretiva **loop** será vista nesta seção em conexão com a diretiva **parallel**, embora também seja válida com **kernels**.

Para fazer o paralelismo de vários laços, é necessário que cada laço seja acompanhado de uma diretiva **parallel loop**.

A diretiva **parallel loop** é uma afirmação do programador de que é seguro e desejável paralelizar o laço afetado. Isso depende se o programador identificou corretamente o paralelismo no código e removeu qualquer coisa no código que poderia tornar perigosa a sua paralelização. Se o programador afirmar incorretamente que o laço pode ser paralelo, e ele não pode, a aplicação resultante pode produzir resultados incorretos.

O programador identifica o paralelismo sem dizer ao compilador como explorar esse paralelismo. Isso significa que o código OpenACC pode ser portado para outros dispositivos além do dispositivo para o qual o código foi primeiramente desenvolvido,

porque detalhes sobre como paralelizar o código são deixados para o compilador decidir, ao invés de ser especificado explicitamente no código fonte.

Cada região **parallel loop** pode ter diferentes laços e cada laço pode ser paralelizado e otimizado de forma independente entre eles. Porém alguns cuidados devem ser tomados.

```
#pragma acc parallel loop
for (int i = 0 ; i < N; i++)
   a[i] = 0;
#pragma acc parallel loop
for (int j = 0 ; j < M; j++)
   b[j] = 0;
```

Exemplo 2.1: Diretiva Parallel Loop

Por exemplo, essa é a maneira recomendada de paralelizar vários laços. Tentando paralelizar múltiplos laços dentro da mesma região paralela pode ocasionar problemas de desempenho ou resultados inesperados.

2.6.1. Cláusulas da diretiva parallel loop

Como o OpenACC serve como uma linguagem para aceleradores genéricos existem três níveis de paralelismo que podem ser usados no OpenACC. Eles especificam o nível de paralelismo contidos na rotina, são chamados de **gang, worker e vector**. Uma *gang* é composta por um ou vários *workers*. Todos os *workers* de uma *gang* podem compartilhar os mesmos recursos, como memória cache ou processador.

Os níveis de paralelismo usado no OpenACC podem ser comparados ao níveis de execução usados na programação CUDA. Podendo assim admitir a relação entre eles: **gang = block**, **worker = warp** e **vector = threads**.

Cláusula	Descrição
gang	Particiona o laço entre as *gangs*
worker	Particiona o laço entre os *workers*
vector	Vetoriza o laço
seq	Não particiona o laço, que é executado sequencialmente

Tabela 2.3: Cláusulas da Diretiva Parallel Loop

Essas diretivas também podem ser combinadas em um laço específico. Por exemplo, um laço **gang vector** pode ser particionado entre *gangs*, cada uma delas com 1 *worker* implicitamente, e depois vetorizado.

A especificação OpenACC reforça que o laço mais externo deve ser um laço de uma *gang*, o laço paralelo mais interno deve ser um laço *vector* e um laço *worker* pode aparecer no meio. Um laço seqüencial (**seq**) pode aparecer em qualquer nível.

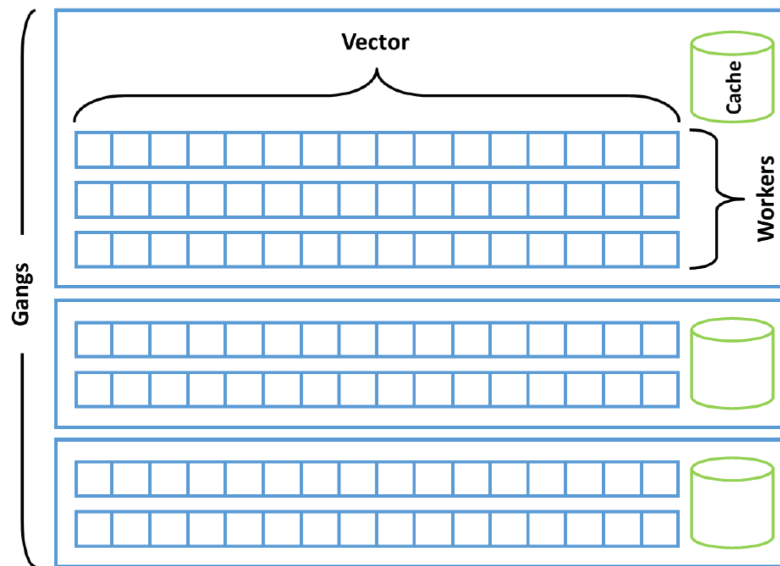


Figura 2.7: Gangs x Workers x Vector (OPENACC-STANDARD.ORG, 2015)

O uso dos níveis de paralelismo são aplicados na diretiva **parallel loop** para gerar maior ganho na execução do laço. Também podem ser usadas da diretiva **kernels**.

```
#pragma acc parallel loop gang
for(i = 0 ; i < size; i++)
  #pragma acc loop worker
  for(j = 0 ; j < size; j++)
    #pragma acc loop vector
    for(k = 0 ; k < size; k++)
       c[i][j]+=a[i][k]*b[k][j];
```

Exemplo 2.2: Cláusulas da Diretiva Parallel Loop

2.6.2. Cláusula reduction

A construção **parallel loop** possui uma cláusula com o mesmo nome **reduction** e com funcionamento similar a construção **parallel**, ou seja, uma cópia privada da variável é gerada para cada iteração do laco, mas é feita uma redução em um único resultado final, que é retornado da região.

Por exemplo, o valor máximo de todas as cópias privadas pode ser necessário ou talvez a soma. A redução pode apenas ser especificada em uma variável **escalar** e apenas as operações **+,*, max e min** e algumas operações bit a bit, tais como **&, |, ^, &&, ||**.

O formato da cláusula **reduction** é mostrado a seguir, onde *operador* deve ser substituído com a operação desejada e *variavel* deve ser substituída com a variável sendo reduzida.

```
reduction(operador:variavel)
```

Um exemplo pode ser visto a seguir:

```
{}
sum = 0.0;
#pragma acc parallel loop reduction(+:soma)
for( int i = 1; i < n-1; i++ )
    soma = A[i] + B[i];

   printf("Soma= %6f\n", soma);
}
```

2.6.3. Cláusula private

A cláusula **private** especifica que cada iteração do laço requer sua própria cópia das variáveis listadas. Por exemplo, se cada laço contiver uma matriz pequena e temporária denominada *tmp* usada durante seu cálculo, essa variável deverá ser tornada privada para cada iteração do laço, a fim de garantir resultados corretos. Se *tmp* não for declarada privada, as *threads* que executam diferentes iterações do laço podem acessar essa variável *tmp* compartilhada de maneiras imprevisíveis, resultando em uma condição de corrida e em resultados potencialmente incorretos. Abaixo está a sintaxe da cláusula privada.

```
private (variavel)
```

Há alguns casos especiais que devem entendidos sobre variáveis escalares em lacos. Primeiro, os iteradores do laco são privatizados por padrão, então eles não precisam ser listados na diretiva **private**. Segundo, a menos que seja especificado em contrário, qualquer variável escalar acessada dentro de um laco paralelo será definida *first private* por padrão, o que significa que uma cópia será feita para cada iteração do laço e que terá um valor inicial igual ao que havia na variável escalar ao entrar na região. Finalmente, quaisquer variáveis (escalares ou não) que são declaradas dentro de um laço em C ou C++ serão feitas privadas para as iterações do laço por padrão.

Nota: a construção **parallel** também tem uma cláusula **private** que vai privatizar a lista de variáveis para cada *gang* na região paralela.

2.7. Diretiva routine

As chamadas de função ou sub-rotina em laços paralelos podem ser problemáticas para os compiladores, pois nem sempre é possível para o compilador ver todos os laços de uma só vez. Os compiladores OpenACC 1.0 eram forçados a fazer *inline* de todas as rotinas chamadas em regiões paralelas ou a não paralelizar laços contendo chamadas de rotina.

O OpenACC 2.0 introduziu a diretiva **routine**, que instrui o compilador a criar uma versão de dispositivo da função ou sub-rotina para que possa ser chamada de uma região de dispositivo. Para leitores já familiarizados com a programação CUDA, essa funcionalidade é semelhante ao especificador da função *__device__*.

Para orientar a otimização, você pode usar cláusulas para informar ao compilador se a rotina deve ser criada para paralelismo de nível de **gang, work, vector ou seq** (sequencial). Você pode especificar várias cláusulas para rotinas que podem ser chamadas com vários níveis de paralelismo.

Fazer isso corretamente exige que você coloque uma cláusula **routine** apropriada antes da definição da rotina para chamar a rotina com o nível certo de paralelismo.

```
#pragma acc routine vector
void foo(float* v, int i, int n) {
  #pragma acc loop vector
  for ( int j=0; j<n; ++j) {
    v[i*n+j] = 1.0f/(i*j);
  }
}

#pragma acc parallel loop
for ( int i=0; i<n; ++i) {
  foo(v,i);
  //chamada no dispositivo
}
```

Exemplo 2.3: Diretiva Routine

Quando a rotina *foo* é chamada a partir do código do *host*, ele será executado no *host*, incrementando os valores do *host*. Quando chamado de dentro de uma construção paralela do OpenACC, ela incrementará os valores do dispositivo.

Teoricamente esta diretiva permitira o uso de funções recursivas, contudo há alguns fatores que que limitam a profundidade da recursão. Por exemplo, os dispositivos NVIDIA estão limitados a 16 níveis de recursão, assim como dispositivos AMD possuem outros limites.

Nota: a partir da versão 14.9 do compilador PGI, uma diretiva **routine** sem nenhuma cláusula de nível de paralelismo (**gang, worker ou vector**) será tratada como se uma cláusula **seq** estivesse presente.

2.8. Diretiva kernels

Quando é utilizada a diretiva **kernels**, isto significa que é informado ao compilador que existem regiões do código que podem ser paralelizadas e que o compilador será o responsável por identificar quais são as regiões e qual a estratégia que será utilizada. A estratégia definida pelo compilador pode ser extrair o máximo de paralelismo do código ou executar somente o mínimo de paralelismo.

Com uso da diretiva **kernels**, o compilador analisará o código e apenas paralelizará quando tiver certeza de que é seguro fazê-lo. Em alguns casos, o compilador pode não ter informações suficientes para determinar se é seguro paralelizar um laço, neste caso essa paralelização não será feita. Os passos do processo de compilação do código usando a diretiva **kernels** são:

- Analisar o código para identificar regiões de paralelismo
- Se encontrado, identificar quais dados devem ser transferidos
- Criar um *kernel*
- Movimentar os dados para dispositivo

```
#pragma acc kernels [clause-list]
```

2.9. Diretiva kernels vs. diretiva parallel

Um dos maiores pontos de confusão para os novos programadores de OpenACC é o motivo pelo qual a especificação possui as diretivas **paralell** e **kernels**, que parecem fazer a mesma coisa. Embora ela estejam fortemente relacionadas, existem diferenças sutis entre elas, que possuem características distintas para cada tipo de aplicação e são usadas de acordo com a necessidade de execução do código.

Existem códigos que são fáceis de alterar e obtém melhor desempenho usando a diretiva **parallel**, porém existem códigos que possuem grande dificuldade de alteração não sendo possível usar diretiva **parallel**, neste caso é usado a diretiva **kernels**, pois as alterações são as mínimas possíveis.

A construção **kernels** fornece ao compilador uma margem de manobra maior para paralelizar e otimizar o código da forma que ele considerar adequada para o tipo de acelerador utilizado, mas também depende muito da capacidade do compilador de paralelizar automaticamente o código. Como resultado, o programador pode ver diferenças de desempenho e de nível de otimização em compiladores diferentes.

A diretiva **parallel loop** é uma afirmação do programador de que é seguro e desejável paralelizar o laço afetado. Isso depende do programador ter identificado corretamente o paralelismo no código e removido qualquer coisa no código que não seja segura para

paralelizar. Se o programador afirmar incorretamente que o laço pode ser paralelizado, o programa resultante poderá produzir resultados incorretos. Em outras palavras: a construção **kernels** pode ser pensada como uma dica para onde o compilador deve procurar paralelismo, enquanto a diretiva **paralell** é uma afirmação para o compilador onde há paralelismo.

Uma coisa importante a ser observada sobre a construção **kernels** é que o compilador analisará o código e apenas paralelizará quando estiver certo de que é seguro fazê-lo. Em alguns casos, o compilador pode não ter informações suficientes em tempo de compilação para determinar se um laço é seguro para ser paralelizado; nesse caso,isso não será feito, mesmo que o programador possa ver claramente que o laço pode ser paralelizado com segurança.

Por exemplo, no caso do código C/C ++, em que as matrizes são passadas para as funções como ponteiros, o compilador nem sempre pode ser capaz de determinar que duas matrizes não compartilham a mesma área de memória, também conhecido como *aliasing* de ponteiros. Se o compilador não puder saber que os dois ponteiros não possuem *alias*, não será capaz de paralelizar um laço que acessa essas matrizes.

Uma prática recomendada para os programadores em C é usar a palavra-chave *restrict* (ou o decorador __*restrict* em C ++) sempre que possível, para informar ao compilador que os ponteiros não têm *alias*, o que frequentemente fornecerá ao compilador informações suficientes para paralelizar laços que não o seriam de outra forma. Além da palavra-chave *restrict*, declarar variáveis constantes usando a palavra-chave *const* pode permitir que o compilador use memória apenas de leitura para essa variável, se essa memória existir no acelerador.

O uso de *const* e *restrit* é uma boa prática de programação em geral, pois fornece ao compilador informações adicionais que podem ser usadas na otimização do código. Um benefício adicional que a construção **kernels** fornece é que, se os dados forem movidos para o dispositivo para uso em laços contidos na região, esses dados permanecerão no dispositivo por toda a extensão da região ou até que sejam necessários novamente no *host* dessa região.

Isso significa que, se vários laços acessarem os mesmos dados, eles apenas serão copiados uma vez para o acelerador. Quando o laço paralelo é usado em dois laços subsequentes que acessam os mesmos dados, o compilador pode ou não copiar os dados entre o *host* e o dispositivo entre os dois laços, o que pode resultar em menor movimentação de dados por padrão.

Nos Exemplos 2.4 e 2.5 usamos um trecho de um código que possui dois laços comparando o processo de paralelização utilizando as diretivas de **kernels** e **parallel**.

Note-se que no processo de execução usando a diretiva **kernels** são gerados dois *kernels*, um para cada laço. Existe um barreira entre os laços, deste modo o segundo laço só será iniciado quando o primeiro laço terminar.

No processo de execução usando a diretiva **parallel** é gerado um único *kernel*. Não existe barreira entre os laços, assim podem ser executados de forma independente.

```
#pragma acc kernels
{
 for (i=0; i<n; i++)
  a[i] = 3.0f*(float)(i+1);
 for (i=0; i<n; i++)
  b[i] = 2.0f*a[i];
}
```

Exemplo 2.4: Diretiva kernels

```
#pragma acc parallel
{
 #pragma acc loop
 for (i=0; i<n; i++)
 a[i] = 3.0f*(float)(i+1);
 #pragma acc loop
 for (i=0; i<n; i++)
 b[i] = 2.0f*a[i];
}
```

Exemplo 2.5: Diretiva parallel

2.10. Operações atômicas

Quando uma ou mais iterações de um laço precisam acessar um elemento na memória ao mesmo tempo, condições de corrida podem ocorrer. Por exemplo, se uma iteração do laço está modificando o valor contido em uma variável e outra está tentando ler a mesma variável em paralelo, diferentes resultados podem ocorrer dependendo de qual iteração ocorra primeiro.

Em programas seriais, os laços sequenciais garantem que a variável será modificada e lida em uma ordem previsível, mas os programas paralelos não garantem que uma iteração específica de um laço irá ocorrer antes da outra. Em casos simples, como encontrar uma soma, valor máximo ou mínimo, uma operação de redução irá garantir que o programa será executado corretamente.

Para operações mais complexas, a diretiva **atomic** garantirá que não haverá duas *threads* (gang, work ou vector) executando a operação nela contida simultaneamente. O uso da diretiva **atomic** é às vezes uma parte necessária do processo de paralelização para garantir a correção do código.

A diretiva **atomic** aceita uma das quatro cláusulas seguintes para declarar o tipo de operação contida na região:

- A operação **read** assegura que duas iterações de um laço não farão leituras da região ao mesmo tempo;
- A operação **write** garantirá que não haja duas iterações realizando escrita na região ao mesmo tempo;
- Uma operação **update** é uma operação de leitura e de escrita combinadas;
- Finalmente, uma operação **capture** executa uma atualização, mas salva o valor calculado nessa região para ser utilizada no código seguinte à região.

Se nenhuma cláusula for definida, uma operação **update** é assumida.

Um histograma é uma técnica comum para contar quantas vezes os valores ocorrem em um conjunto de entrada de acordo com o seu valor. O código do exemplo abaixo percorre uma série de números inteiros de um intervalo conhecido e conta o total de ocorrências de cada número nesse intervalo. Como cada número no intervalo pode ocorrer várias vezes, precisamos garantir que cada elemento no vetor de histograma seja atualizado atomicamente. O código abaixo demonstra usando a diretiva **atomic** para gerar um histograma.

```
#pragma acc parallel loop
    for(int i=0; i < HN; i++)
        h[i]=0;
#pragma acc parallel loop
    for(int i=0; i < N; i++) {
#pragma acc atomic update
        h[a[i]]+=1; }
```

Exemplo 2.6: Diretiva atomic

Observe que as atualizações no vetor do histograma *h* são executadas atomicamente. Como estamos incrementando o valor do elemento de um vetor, uma operação **update** é usada para ler o valor, modificá-lo e gravá-lo novamente.

2.11. Cláusula device_type

O OpenACC permite que os programadores consigam otimizar suas diretrizes para aceleradores específicos com o uso da cláusula **device_type**, com isso é possível obter melhores desempenhos. Com o OpenACC 1.0, diretivas de pré-processador eram necessárias para ajustar as diretivas para uso em aceleradores específicos. Além de dificultar a manutenção do código, devido à duplicação de diretivas, isso significa que é impossível oferecer suporte a vários tipos de dispositivos no mesmo executável. Já com a versão do OpenACC 2.0 ele permite que determinadas cláusulas sejam fornecidas especificamente para arquiteturas específicas.

2.12. Cláusula vector_length

Outra cláusula importante é a **vector_length**, esta cláusula é utilizada para especificar o tamanho do vetor que será usado no laço.

No exemplo abaixo foi especificado um comprimento de vetor, dependendo do tipo de acelerador que será usado.

```
#pragma acc parallel loop \
            device_type(nvidia) vector_length(256) \
            device_type(radeon) vector_length(512) \
                                vector_length(64)
for ( int i=0; i<n; ++i) {
      c[i] = a[i] + b[i];
```

```
    }
```

Exemplo 2.7: Suporte a múltiplos aceleradores

Neste exemplo, se o laço for utilizar um acelerador **NVIDIA**, o compilador utilizará um comprimento vetorial de 256; se for utilizar um acelerador **Radeon**, o compilador usa um comprimento de vetor de 512; e para qualquer outro acelerador que não seja especificado será usado comprimento vetorial de 64. Ambas as cláusulas podem ser utilizadas em conjunto com as demais cláusulas do OpenACC.

2.13. Cláusula tile

Existe um novo recurso disponível no OpenACC 2.0, é a adição da cláusula **tile** à diretiva **acc loop**. Com a cláusula **tile** é possível otimizar o laço através da operação de blocos menores para explorar o acesso aos dados. Considere o seguinte exemplo de transposição de matriz.

```
#pragma acc parallel loop private(i,j) tile(8,8)
for(i=0; i<rows; i++)
    {
     for(j=0; j<cols; j++)
      {
        out[i*rows + j] = in[j*cols + i];
      }
    }
```

Exemplo 2.8: Cláusula tile

Ao adicionar a cláusula tile (8,8) ao laço paralelo, serão criados automaticamente pelo compilador dois laços adicionais que funcionam em um *chunk* 8x8 (tile) da matriz antes de passar para o próximo *chunk*. Com isso o compilador faz a otimização dentro do bloco, com o objetivo de obter melhor desempenho. Embora uma transposição de matriz não tenha muita reutilização de dados, outros algoritmos podem ter uma melhora significativa no desempenho, explorando a localidade e a reutilização de dados nos laços disponíveis.

2.14. Cláusula collapse

A execução de um laço em OpenACC está associada ao laço imediatamente a seguir. Uma diretiva é necessária para cada laço. Isso tende a ser complicado, especialmente se vários laços devem ser tratados da mesma maneira. A cláusula **collapse** é útil nesse caso. O argumento para a cláusula **collapse** é um número inteiro positivo constante, que especifica quantos laços fortemente aninhados serão associados para a criar um novo laço.

```
#pragma acc parallel loop collapse(2)
for(int i=0; i < N; i++)
```

```
    for(int j=0; j < M; j++)

#pragma acc parallel loop
for(int ij=0; ij < N*M; ij++)...
```

Exemplo 2.9: Cláusula collapse

Quais as vantagens em usar a cláusula **collapse**?

- colapsar os laços externos para permitir a criação de mais *gangs*.
- colapsar os laços internos para permitir comprimentos de vetor mais longos.
- colapsar todos os laços, quando for possível, para fazer as duas coisas: ter mais *gangs* criadas e vetores maiores.

2.15. Compilação

Antes de compilar qualquer código é importante saber quais dispositivos aceleradores estão configurados para uso no sistema. Existem alguns comandos que fornecem informações de modelos e características desses dispositivos.

Para aceleradores da NVIDIA existes os comandos *nvidia-smi* e *nvidia-settings*, estes comandos fornecem de informações de configuração como: modelo, cpus, cuda core, memória. Com o compilador da PGI também existe um comando chamado *pgaccelinfo* que fornece as principais características dos aceleradores instalados no sistema.

```
# nvidia-smi -q | grep "Product Name"
    Product Name                        : Quadro K420
    Product Name                        : Tesla K80
    Product Name                        : Tesla K80

# pgaccelinfo | grep "Device Name"
    Device Name:                       Tesla K80
    Device Name:                       Tesla K80
    Device Name:                       Quadro K420
```

Para compilar os códigos feitos em OpenACC, gerando código para execução em GPUs, é necessário o uso de compiladores que suportem OpenACC. A PGI (Portland Group), tem um versão disponibilizada para uso público sem a necessidade de licença, também existe uma versão com uso de licença que permite ter acesso à equipe de suporte da PGI.

A Cray também tem seu compilador o CCE 8.6.5, para usá-lo é necessário a aquisição de uma licença, não existe um licença para uso público.

Nos exemplos apresentados neste minicurso usaremos o compilador da PGI disponibilizado para a comunidade, o compilador pode ser baixado do sítio da PGI através no endereço https://www.pgroup.com/products/community.htm.

Para compilar códigos em C usaremos o comando *pgcc*, e para compilação de códigos em C++ usar o comando *pgc++*. Algumas parâmetros básicos devem ser usados durante a execução dos comandos *pgcc* e *pgc++*. Esses parâmetros definem em qual dispositivo o código é executado de acordo com a arquitetura. Relação dos principais parâmetros usados no comando *pgcc*:

Parâmetro	**Descrição**
-fast	faz a otimização do código
-acc	habilita o uso de diretivas OpenACC
-Minfo=accel	informações sobre quais partes do código foram aceleradas
-Minfo=opt	informações sobre todas as otimizações de código
-Minfo=all	informações sobre todas as saídas de código
-ta=host	compila o código em modo serial
-ta=multicore	compila o código usando *threads* em CPU
-ta=nvidia	compila o código usando NVIDIA

Tabela 2.4: Parâmetros de Compilação pgcc

Alguns exemplos de uso do compilador PGI e seus parâmetros básicos.

```
$ pgcc -acc -ta=nvidia -Minfo=accel main.c
$ pgc++ -acc -ta=nvidia -Minfo=accel main.cpp
```

Uma outra alternativa é o uso do compilador GNU GCC a partir da versão 6 o compilador tem suporte ao OpenACC. Como o GNU GCC é um compilador *opensource* não requer licença para uso. Para usar o GNU GCC especificar o parâmetro *-fopenacc*.

```
$ gcc -fopenacc main.c
```

2.16. Exemplos

Após a introdução do conceito e das principais diretivas usadas no OpenACC, veremos alguns exemplos da aplicações dessas diretivas, como elas se comportam e quais as melhores opções de uso das diretivas para gerar maior ganho.

Os códigos foram executados em um servidor com dois processadores Intel Xeon E5-2609 (2,40 GHz, 4 núcleos cada, cache de 10 MB), com 128 GB de memória compartilhada, discos locais de alta velocidade SSD (Solid-State Drive) e um acelerador NVIDIA Tesla K80 (24 GB de memória, 4992 CUDA cores). O sistema operacional usado foi a

versão 7.3 da distribuição Centos Linux de 64 bits. Todos os códigos foram compilados usando o PGI Community Edition Version 19.4.

2.16.1. Cálculo de Pi

Iniciaremos com o exemplo básico do cálculo do número Pi. O valor de Pi é definido pela relação entre o perímetro de uma circunferência e seu diâmetro, para a maioria dos cálculos simples é comum usar a aproximação do valor de Pi para 3,1415. Em computação existem algoritmos que podem ser utilizados para o cálculo aproximado do Pi, como: Gauss-Legendre e Monte Carlo. A seguir é apresentada uma implementação simples para o cálculo sequencial (serial) de Pi.

```
#include <stdio.h>
#define N 1000000000

int main(void) {
   double pi = 0.0f; long i;
   for (i=0; i<N; i++)   {
     double t=(double) ((i+0.5)/N);
     pi += 4.0/(1.0+t*t);
   }
   printf("pi=%f\n",pi/N);
   return 0;
}
```

Exemplo 2.10: Cálculo de Pi sequencial

A execução do cálculo de Pi dentro do laço na versão sequencial será feito por uma *thread*, independente de quantidade de processadores que existam no sistema.

Podemos paralelizar o código em OpenMP adicionando a linha **#pragma omp parallel for reduction(+: pi)** antes do laço, desta forma pode-se utilizar mais de uma *thread* para o cálculo.

```
   #pragma omp parallel for reduction(+: pi)
   for (i=0; i<N; i++)
   {
     double t=(double) ((i+0.5)/N);
     pi += 4.0/(1.0+t*t);
   }
```

Exemplo 2.11: Cálcudo de Pi com OpenMP

Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Da mesma forma que foi usado no OpenMP, adicionaremos a linha **#pragma acc parallel loop reduction(+: pi)** antes do laço.

```
#pragma acc parallel loop reduction(+: pi)
for (i=0; i<N; i++)
{
  double t=(double) ((i+0.5)/N);
  pi += 4.0/(1.0+t*t);
}
```

Exemplo 2.12: Cálculo de Pi com OpenACC

Para avaliar o tempos de execução do cálculo de Pi, foram executados os códigos sequencial, com OpenMP e com OpenACC. Para o cálculo em OpenMP foram utilizadas 16 *threads*, a quantidade máxima de processadores no servidor.

O tempo de execução sequencial do cálculo de Pi foi de 5,6 segundos, o tempo com OpenMP foi de 0,60 segundos e com OpenACC foi de 0,15 segundos.

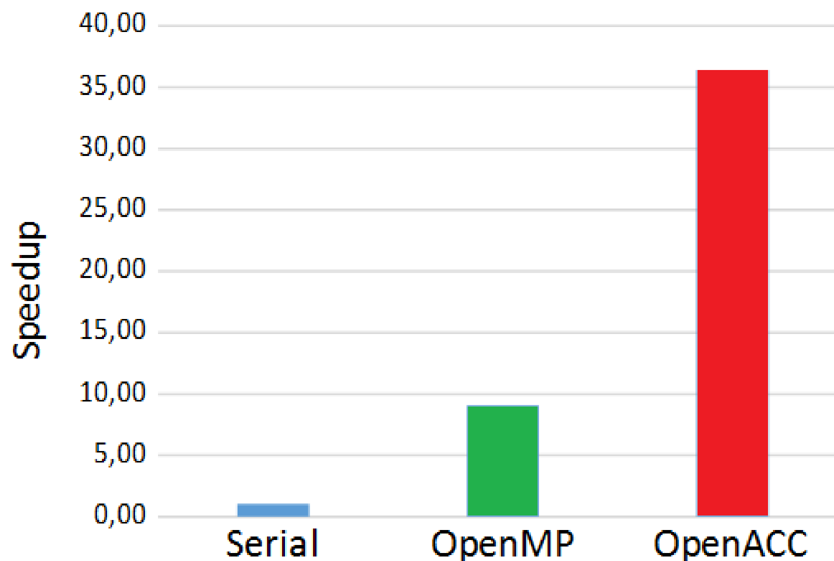
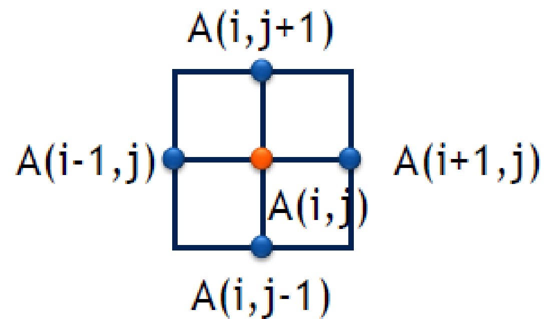


Figura 2.8: Seepdup - Cálculo do valor de Pi

O *speedup* encontrado usando programação OpenMP foi de 9,10 em comparação a execução em sequencial, enquanto que usando OpenACC o *speedup* foi de 36,40 em relação ao executado em sequencial. Os resultados são apresentados na Figura 2.8

2.16.2. Método Jacobi

O Método de Jacobi é um procedimento iterativo para a resolução de sistemas lineares. Converte iterativamente para o valor correto, calculando novos valores em cada ponto a partir da média dos pontos vizinhos. Neste exemplo faremos o cálculo da temperatura na placa usando a equação de Laplace: $\nabla^2 f(x,y) = 0$.



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

A seguir é apresentada uma implementação sequencial (serial) para o cálculo da temperatura da placa usando o método Jacobi.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define COLUMNS    1000
#define ROWS       1000
#define MAX_TEMP_ERROR 0.01

double Anew[ROWS+2][COLUMNS+2];
double A[ROWS+2][COLUMNS+2];

void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

    int i, j;
    int max_iterations=1000;
    int iteration=1;
    double dt=100;

    initialize();

    while ( dt > MAX_TEMP_ERROR && iteration
          <= max_iterations ) {

        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
```

```
                Anew[i][j] = 0.25 * (A[i+1][j] +
                A[i-1][j] + A[i][j+1] + A[i][j-1]);
            }
        }

        dt = 0.0;

        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
              A[i][j] = Anew[i][j];
            }
        }

        iteration++;
    }

    printf("\n Erro maximo na iteracao %d era %f\n",
           iteration-1, dt);
}

void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            A[i][j] = 0.0;
        }
    }

    for(i = 0; i <= ROWS+1; i++) {
        A[i][0] = 0.0;
        A[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    for(j = 0; j <= COLUMNS+1; j++) {
        A[0][j] = 0.0;
        A[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

Exemplo 2.13: Método de Jacobi Sequencial

O primeiro laço dentro do *while* de convergência calcula o novo valor para cada elemento com base nos valores atuais de seus vizinhos. Armazenando em uma matriz

temporária, garantindo que todos os valores sejam calculados usando o estado atual de **A** antes que **A** seja atualizado. Como resultado, cada iteração do laço é completamente independente uma da outra.

Esse laço também calcula um máximo valor de erro. O valor do erro é a diferença entre o novo valor e o antigo. Se a quantidade máxima de alteração entre duas iterações estiver dentro de alguma tolerância, o problema será considerado convergido e o laço externo será encerrado. O segundo laço simplesmente atualiza o valor de **A** com os valores calculados em **Anew**.

A execução do cálculo da temperatura dentro dos laços serão feitos por uma *thread*, independente de quantidade de *threads* que existam no sistema.

Podemos paralelizar o código em OpenMP adicionando as linhas **#pragma omp parallel for** antes do primeiro laço e a linha **#pragma omp parallel for reduction(max:dt)** antes do segundo laço, desta forma pode-se utilizar mais de uma *thread* para o cálculo.

```
#pragma omp parallel for
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
        Anew[i][j] = 0.25 * (A[i+1][j] +
        A[i-1][j] + A[i][j+1] + A[i][j-1]);
    }
}

dt = 0.0;

#pragma omp parallel for reduction(max:dt)
for(i = 1; i <= ROWS; i++){
    for(j = 1; j <= COLUMNS; j++){
      dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
      A[i][j] = Anew[i][j];
    }
}
```

Exemplo 2.14: Método de Jacobi com OpenMP

Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Da mesma forma que foi usado no OpenMP, adicionar a linha **#pragma acc parallel loop**, e a linha **#pragma acc parallel loop reduction(max:dt)** no segundo laço.

```
#pragma acc parallel loop
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
        Anew[i][j] = 0.25 * (A[i+1][j] +
        A[i-1][j] + A[i][j+1] + A[i][j-1]);
    }
```

```
        }

        dt = 0.0;

        #pragma acc parallel loop reduction(max:dt)
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
              A[i][j] = Anew[i][j];
            }
        }
```

Exemplo 2.15: Método de Jacobi com OpenACC - Versão 1

Para avaliar os tempos de execução dos cálculos da temperatura, foram executadas as versões sequencial, com OpenMP e com OpenACC. Para o cálculo em OpenMP foram utilizadas 16 *threads*, quantidade máxima de processadores no servidor.

Foram observados os seguintes tempos de execução: o código sequencial teve um tempo total de 11 segundos, sendo que com OpenMP o tempo de execução foi de 4,30 segundos e com OpenACC o valor medido foi de 10 segundos.

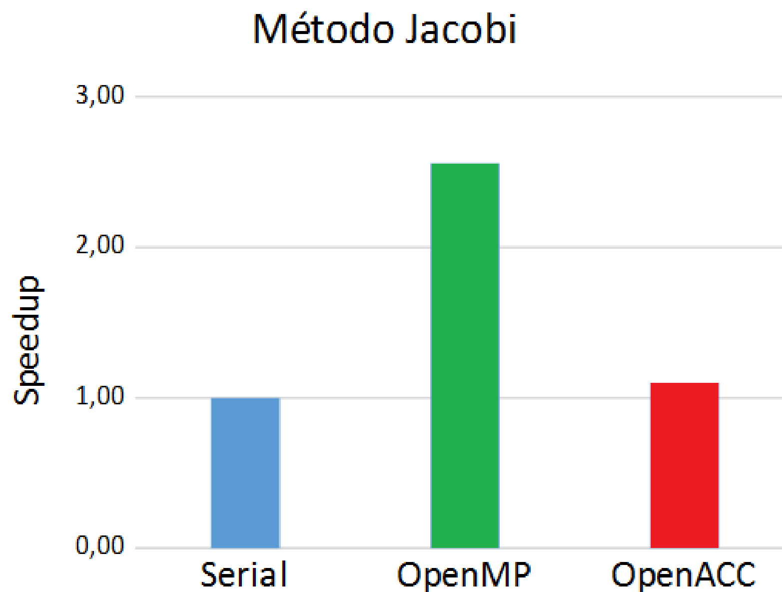


Figura 2.9: Seepdup - Método Jacobi

O *speedup* encontrado na execução do código com OpenMP foi de 2,56 em comparação à execução em sequencial, enquanto que usando OpenACC o *speedup* foi de 1,10. Os resultados são apresentados na Figura 2.9.

Como visto, o resultado do *speedup* encontrado usando OpenACC foi quase idêntico aos resultados encontrados na execução do código sequencial. Isso ocorre porque a

matriz de cálculo não está armazenada no acelerador. Toda vez que o acelerador executa uma operação as informações são gravadas na matriz que está na memória do *host*.

Para resolver este problema é necessário fazer a cópia da matriz para o acelerador de modo que não seja mais necessário gravar as informações no *host* toda vez que for realizada uma operação pelo acelerador.

Usaremos a diretiva **data** do OpenACC. Adicionar a linha **#pragma acc data copy(A) create (Anew)** antes dos dois laços para fazer a cópia da matriz para a memória do acelerador.

```
#pragma acc data copy(A) create(Anew)
while ( dt > MAX_TEMP_ERROR && iteration <=
max_iterations ) {

   #pragma acc parallel loop
   for(i = 1; i <= ROWS; i++) {
       for(j = 1; j <= COLUMNS; j++) {
           Anew[i][j] = 0.25 * (A[i+1][j] +
           A[i-1][j] + A[i][j+1] + A[i][j-1]);
       }
   }

   dt = 0.0;

   #pragma acc parallel loop reduction(max:dt)
   for(i = 1; i <= ROWS; i++){
       for(j = 1; j <= COLUMNS; j++){
         dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
         A[i][j] = Anew[i][j];
       }
   }
```

Exemplo 2.16: Método de Jacobi com OpenACC - Versão 2

O total de tempo para a execução sequencial foi de 11 segundos, sendo que o tempo total com OpenMP se manteve em 4,30 segundos e o tempo de execução com OpenACC foi reduzido para 0,80 segundos.

O *speedup* para a execução com OpenMP continou em 2,56 em comparação a execução sequencial, enquanto que usando OpenACC o *speedup* aumentou para 13,75. Os resultados são apresentados na Figura 2.10.

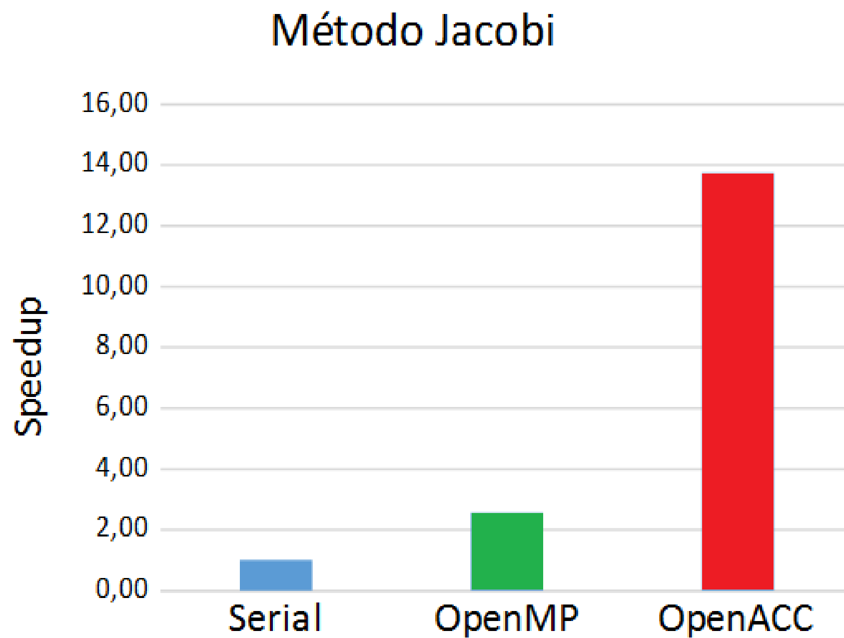


Figura 2.10: Seepdup - Método Jacobi

2.17. Conclusão

O modelo de programação OpenACC foi desenvolvido pelos principais fabricantes de hardware e software do mercado, com o objetivo de simplificar a programação paralela, tornando possível a portabilidade do código.

Com o uso do OpenACC é possível atingir alto níveis de paralelismo usando arquitetura baseada em aceleradores. Entre suas principais características se destacam:

- O OpenACC é fácil de usar;
- Usa uma abordagem baseada em diretivas de compilação;
- Em alguns casos são feitas pequenas alterações no código;
- O código pode ser implementado em qualquer acelerador.

Referências

A., Z. L. F.; M., M. *Arquitetura e programação de GPU nvidia*. [S.l.]: UNICAMP, 2012.

ABBOTT, S. *Advanced OpenACC*. [S.l.]: NVIDIA Corporation, 2017.

CHEN, S. *Introduction to OpenACC*. [S.l.]: Research Computing Services Information Services and Technology Boston University, 2017.

CORREA, J. C.; SILVA, G. P. Analysis and performance evaluation of parallel BLAST. *I. J. Comput. Appl.*, v. 20, n. 2, p. 112–122, 2012.

COSTA, E. B.; SILVA, G. P.; TEIXEIRA, M. Avaliação de desempenho do montador daligner em arquiteturas manycore. In: *WSCAD 2018 - WCH ()*. São Paulo - SP, Brazil: [s.n.], 2018.

LARKIN, J. *Introduction to OpenACC*. [S.l.]: NVIDIA, 2018.

OPENACC-STANDARD.ORG. *OpenACC Programming and Best Practices Guide*. [S.l.]: OpenACC-Standard.org, 2015.

RAHMAN, R. *Intel Xeon Phi Coprocessor Architecture and Tools The Guide for Application Developers*. [S.l.]: Apress Open, 2013.

SILVA, G. P. *Programação Paralela com MPI Um Curso Introdutorio*. [S.l.]: Amazon, 2018.

 DOI: 10.5753/sbc.4400.9

Capítulo 3

Programação Paralela em Memória Compartilhada e Avaliação de Desempenho com Contadores de Hardware

Matheus da Silva Serpa
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Claudio Schepke
Universidade Federal do Pampa
Alegrete, Brasil

Resumo

No passado, o aumento de desempenho das aplicações dava-se de forma transparente aos programadores devido ao aumento do paralelismo a nível de instruções e de frequência dos processadores. Entretanto, isto já não se sustenta mais há alguns anos. Atualmente para se ganhar desempenho nas arquiteturas modernas, é necessário ter conhecimentos sobre programação paralela e vetorial. Todos estes paradigmas são tratados de alguma forma em muitos cursos de computação, mas geralmente não aprofundados. Neste contexto, este capítulo objetiva propiciar um maior entendimento sobre os paradigmas de programação paralela e vetorial, de forma que seja possível aprender a otimização adequada de aplicações para arquiteturas atuais. Além disso, conceitos de avaliação de desempenho com contadores de hardware via Linux perf e PAPI são apresentados. Desta forma, o capítulo fornece aos estudantes a oportunidade de aprender e praticar conceitos de programação paralela em aplicações de alto desempenho.

3.1. Introdução

A introdução de circuitos integrados, *pipelines*, aumento da frequência das operações, execução fora de ordem e previsão de desvios constituem parte importante das tecnologias introduzidas até o final do século XX. Recentemente, tem crescido a preocupação com o consumo energético, com o objetivo de se atingir a computação em nível *exascale* de forma sustentável. Entretanto, as tecnologias até então desenvolvidas não possibilitam

atingir tal fim, devido ao alto custo energético de se aumentar a frequência e estágios de *pipeline*, assim como a chegada nos limites de exploração do paralelismo a nível de instrução (BORKAR; CHIEN, 2011; COTEUS et al., 2011).

Este capítulo envolve o estudo de aspectos relacionados à arquitetura de computadores e a elaboração, execução e teste de programas concorrentes. Neste sentido, pretende-se inicialmente identificar as arquiteturas de *hardware* que existem atualmente e que podem ser utilizadas para a construção de máquinas de alto desempenho. Em um segundo momento, a interface de programação OpenMP será apresentada para ambientes de memória compartilhada. Com base nesta interface, almeja-se elaborar aplicações paralelas otimizadas.

Para os programas a serem desenvolvidos são disponibilizados códigos fonte sequenciais e paralelos previamente criados e testados. Os exemplos de código serão introduzido de forma incremental, isto é, variações do mesmo código serão fornecidas para testar aspectos distintos oferecidos pelas interfaces de programação. Por fim, alguns tópicos de testes, depuração e medição de desempenho serão citadas, com o intuito de mostrar como são feitas avaliações de performance de aplicações paralelas.

A estrutura do capítulo é dividida em 7 partes. Inicialmente será apresentada uma introdução sobre as arquiteturas paralelas na Seção 3.2. Na sequência será explicitado, na Seção 3.3, como pode ser feita a modelagem de aplicações paralelas. A Seção 3.4 discute como se pode programar paralelamente uma arquitetura de memória compartilhada usando a biblioteca de diretivas OpenMP. Essa seção também aborda a programação vetorial usando instruções SIMD. A avaliação de desempenho com contadores de *hardware* do tipo Linux perf e Performance Application Programming Interface (PAPI) é apresentada na Seção 3.5. A Seção 3.6 mostra um estudo de caso sobre o desempenho de uma aplicação de geofísica utilizando os conceitos apresentados no capítulo e, finalmente, a Seção 3.7, traz-se a conclusão, abordando as potencialidades de paralelismo com outras interfaces de programação.

3.2. Arquiteturas paralelas

Algumas das primeiras perguntas a serem feitas em relação ao desenvolvimento de aplicações é: por que estudar programação paralela?, os programas já não são rápidos o suficiente? as máquinas já não são rápidas o suficiente? Um dos principais pontos de motivação é que os requisitos necessários estão sempre mudando. Os usuários desejam executar aplicações e jogos cada vez mais detalhistas e com tempo de resposta menor. Além de que, desde 2005, é difícil encontrar um processador de um só *core* no mercado (FRUEHE, 2005; GEPNER; KOWALIK, 2006).

Outros motivos para utilizar programação paralela são: (i) reduzir o tempo necessário para solucionar um problema e (ii) resolver problemas mais complexos e de maior dimensão em um tempo aceitável. Existem além desses, outros motivos como: utilizar recursos computacionais subaproveitados; ultrapassar limitações de memória quando a memória disponível num único computador é insuficiente para a resolução do problema; e também ultrapassar os limites físicos que atualmente começam a restringir a possibilidade de construção de computadores sequenciais cada vez mais rápidos.

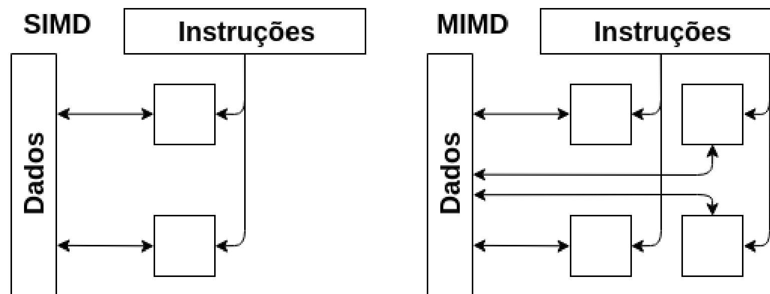
Nesse sentido, a computação de alto desempenho tem sido responsável por uma revolução científica. A evolução das arquiteturas de computadores melhorou o poder computacional, aumentando a gama de problemas e a qualidade das soluções que poderiam ser resolvidas no tempo requerido como, por exemplo, a previsão do tempo. Entretanto, devido a limitações de consumo de energia, dissipação de calor, dimensão do processador e uma melhor distribuição das *threads* para processamento, a indústria mudou seu foco para arquiteturas paralelas e sistemas distribuídas (HSU, 2015; BORKAR; CHIEN, 2011; COTEUS et al., 2011).

A principal característica dessas arquiteturas é a presença de vários núcleos de processamento operando simultaneamente. No entanto, o desenvolvimento de *software* foi afetado por essa mudança de paradigma e diversas aplicações sofreram reengenharias para tornar possível o aproveitamento dos recursos através da execução paralela (GROPP; SNIR, 2013; MITTAL; VETTER, 2015; CRUZ et al., 2016).

3.2.1. Classificação de arquiteturas paralelas

A classificação de Flynn (FLYNN; RUDD, 1996) baseia-se no fato de um computador executar uma sequência de instruções sobre uma sequência de dados. As instruções e os dados são separados em um ou vários fluxos de instruções (*instruction stream*) e um ou vários fluxos de dados (*data stream*). Essa classificação possui quatro classes, mas vamos nos concentrar apenas nas duas classes que representam as arquiteturas paralelas: *Single Instruction Multiple Data* (SIMD) e *Multiple Instruction Multiple Data* (MIMD). A Figura 3.1 apresenta os diagramas das classes exemplificadas a seguir.

Figura 3.1. Diagramas das classes SIMD (esquerda) e MIMD (direita).



Em uma arquitetura SIMD, uma única instrução é executada ao mesmo tempo sobre múltiplos dados. Esse processamento é controlado por uma única unidade de controle que é alimentada por um único fluxo de instruções. Cada instrução é enviada para todos processadores que executam as instruções em paralelo de forma síncrona sobre diferentes fluxos de dados. Essa arquitetura é encontrada nas unidades MMX/SSE de processadores *multicore* e nas *Graphics Processing Units* (GPUs).

Em arquiteturas MIMD, cada unidade de controle recebe um fluxo de instruções próprio e repassa-o para seu respectivo processador. Dessa forma, cada processador executa suas instruções em seus dados de forma assíncrona. O princípio dessa classe é bastante genérico, pois se um computador de um grupo de máquinas for analisado separadamente, este pode ser considerado uma máquina MIMD. Nessa classe encontram-se as arquiteturas paralelas *multicore*.

Arquiteturas paralelas atuais combinam ambas arquiteturas SIMD e MIMD. Por exemplo, um processador *multicore* possui vários *cores* cada um trabalhando sobre um conjunto de instruções e um conjunto de dados, ou seja, MIMD. Em cada *core* do mesmo processador existe uma unidade especial de ponto flutuante que explora SIMD. Sistemas distribuídos são compostos por várias arquiteturas paralelas, logo, combinando SIMD e MIMD.

3.2.2. Arquiteturas multicore e manycore

Desde 2003, a indústria vem seguindo duas abordagens para o projeto de microprocessadores (KIRK; WEN-MEI, 2016). A abordagem multicore é orientada à latência, onde instruções são executadas em poucos ciclos de *clock*. Por outro lado, as arquiteturas manycore tem uma abordagem focada ao *throughput*, ou seja, um grande número de instruções é executado por unidade de tempo.

O projeto das arquiteturas multicore e manycore é diferente ao ponto que dependendo da aplicação, o desempenho pode ser muito grande em uma arquitetura e muito pequeno na outra (COOK, 2012). A arquitetura multicore utiliza uma lógica de controle sofisticada para permitir que instruções de uma única *thread* sejam executadas em paralelo. Grandes memórias *cache* são fornecidas para reduzir latências de acesso às instruções e dados de aplicações que tem acesso à memória predominante. Por fim, as operações das Unidades Lógicas e Aritméticas (ULA) também são projetadas visando otimizar a latência.

A arquitetura manycore tira proveito de um grande número de *threads* de execução. Pequenas memórias *cache* são fornecidas para evitar que múltiplas *threads*, acessando os mesmos dados, precisem ir até a memória principal. Além disso, a maior parte do *chip* é dedicada a unidades de ponto flutuante. Arquiteturas desse tipo são projetadas como mecanismos de cálculo de ponto flutuante e não para operações convencionais, que são realizadas por arquiteturas multicore. Algumas aplicações poderão utilizar tanto multicore quanto manycore em conjunto, sendo cada arquitetura melhor para um tipo de operação.

3.3. Modelagem de aplicações paralelas

A programação paralela possibilita utilizar ao máximo os recursos de *hardware*. Com isso, muitos problemas antes impossíveis de serem solucionados podem ser executados sem muito esforço. A demanda de desempenho necessária para a realização de uma tarefa está relacionada com a quantidade de dados ou variáveis envolvidas durante o processamento e quais as operações pelas quais estes terão de passar até o resultado. Quanto mais eficiente for a implementação de um algoritmo, menor a demanda por desempenho.

Segundo Foster (FOSTER, 1995), a modelagem de um problema de forma paralela passa por quatro fases: o particionamento, a comunicação, o agrupamento e o escalonamento.

1) Particionamento - Primeiramente, os dados são divididos de maneira que cada tarefa possa ser executada independentemente das demais. Com isso obtém-se a menor granularidade possível para cada tarefa.

2) Comunicação - Em um segundo momento, devido ao fato de os dados normalmente estarem inter-relacionados, é necessário que haja a troca de informações entre os processos. Nessa fase é definida a forma de comunicação paralela adotada, caso seja utilizado uma arquitetura multiprocessada.

3) Agrupamento - Em seguida, em uma terceira fase, as operações ou dados são agrupados a fim de realizar um melhor uso dos processadores. O objetivo dessa fase é aumentar a granularidade das operações realizadas por um único processador. Assim, operações que envolvam um conjunto de dados vizinho são executadas em um mesmo processador, diminuindo a interdependência entre os dados.

4) Escalonamento - Por fim, na quarta etapa, ocorre o mapeamento, que é a fase que define como serão distribuídas as tarefas entre os processadores. Essa distribuição busca casar a granularidade das tarefas com a capacidade de processamento dos processadores e a dependência entre os processos que se encontram em processadores distintos.

A Figura 3.2 ilustra cada uma das etapas descritas anteriormente. Inicialmente um conjunto de dados é particionado. Posteriormente são destacadas as interações entre dois pontos vizinhos de granularidade fina. Após o agrupamento entre alguns pontos é feito um mapeamento, que distribui as tarefas entre 5 processadores ($P1, ..., P5$).

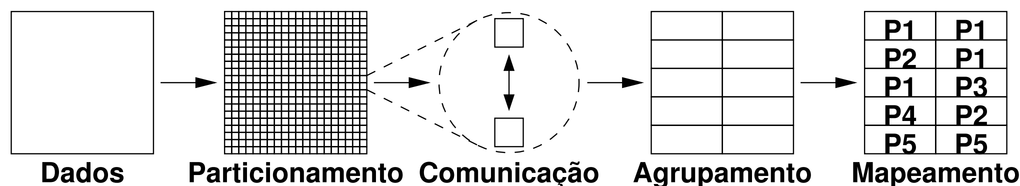


Figura 3.2. Principais etapas na paralelização de um algoritmo.

Mais conceitos de projeto e desenvolvimento de programas paralelos podem ser estudados no material base do minicurso Projetando e Construindo Programas Paralelos, apresentando na ERAD/RS 2019[1].

3.4. Programação paralela e vetorial em OpenMP

Os modelos de programação paralela são divididos em modelos de memória compartilha e de memória distribuída. Exemplos de modelos de programação em memória compartilhada são OpenMP (*Open Multi-Processing*) (OPENMP, 2008; CHAPMAN; JOST; PAS, 2008), Cilk (REINDERS, 2012; ROBISON, 2013) e CUDA (*Compute Unified Device Architecture*) (COOK, 2012; KIRK; WEN-MEI, 2016). Nesse ambiente, a programação é feita utilizando *threads*. A decomposição utilizada é na sua maioria a decomposição do domínio ou a funcional, com diferentes granularidades. No caso dos modelos de memória distribuída, o MPI (*Message Passing Interface*) (GROPP; THAKUR; LUSK, 1999;

[1] https://www.setrem.com.br/erad2019/data/pdf/minicursos/mc02.pdf

PITT-FRANCIS; WHITELEY, 2017) é um dos mais utilizados. Nesse modelo, a programação é feita utilizando processos distribuídos. A decomposição utilizada é do domínio, buscando a maior granularidade possível.

Neste capítulo, optou-se por utilizar OpenMP, focando em memória compartilhada. OpenMP é uma API (*Application Programming Interface*) de programação paralela portável para arquiteturas de memória compartilhada. OpenMP surgiu da dificuldade no desenvolvimento de programas paralelos em arquiteturas de memória compartilhada, além da ausência de APIs padronizadas para tais arquiteturas. A interface proporciona diretivas que possibilitam expressar paralelismo de dados, em trechos de código e laço, e paralelismo de tarefas, introduzido em sua versão 3.0 (AYGUADÉ et al., 2008). Sua API é constituída de diretivas de compilação, métodos de biblioteca e variáveis de ambiente. Em sua versão 4.0 (MARTINEAU; MCINTOSH-SMITH; GAUDIN, 2016), OpenMP inclui suporte para dependências de dados em tarefas e suporte a aceleradores (OPENMP, 2008). No momento da escrita desse minicurso, OpenMP estava em sua versão 5.0 (SUPINSKI et al., 2018).

Alguns fatores limitam o desempenho dos códigos paralelos. O primeiro fator é o próprio código sequencial. Existem partes do código que são inerentemente sequenciais como, por exemplo, iniciar e terminar a computação. Essas partes do código não são aceleráveis. Outro fator é a concorrência, ou seja, o número de tarefas pode ser escasso ou de difícil definição. Por exemplo, pode-se ter um processador com 40 *cores*, mas a aplicação possuir apenas 10 iterações de laço que podem ser divididas. Outros dois pontos que limitam muito o desempenho das aplicações são a comunicação e a sincronização. Existe sempre um custo associado à troca de informação e enquanto as tarefas sincronizam essa informação, elas não contribuem para a computação. A partilha de dados entre as várias tarefas pode levar a problemas de contenção no acesso à memória e enquanto as tarefas ficam à espera da sincronização elas não podem computar nada. Por fim, o balanceamento de carga é muito importante, pois ter os processadores maioritariamente ocupados durante toda a execução é decisivo para o desempenho global do sistema.

3.4.1. Programando com OpenMP

A API OpenMP é composta basicamente por diretivas de compilação e métodos da biblioteca. As diretivas são anotações no código e os métodos OpenMP dependem da compilação com a biblioteca. As diretivas de compilação, *pragmas* em linguagem C/C++, do OpenMP começam com `#pragma omp` e são seguidos por construções e cláusulas que se aplicam a um bloco estruturado. As construções descrevem seções paralelas, dividem dados ou tarefas entre *threads* e controlam sincronização. Por sua vez, as cláusulas modificam ou especificam aspectos das construções.

O primeiro exemplo é um *Olá Mundo*. A Figura 3.3 ilustra o primeiro exemplo em OpenMP. A construção `parallel` indica um bloco de execução paralela, ou seja, faz com que o bloco estruturado especificado entre as linhas 8 e 14 seja executado uma vez para cada *thread* criada. O ambiente OpenMP irá alocar um determinado número de *threads*, e todas elas executarão as linhas de comando contidas dentro do `parallel`. O número de *threads* varia, sendo responsabilidade do programador garantir que o resultado esperado seja atingido independentemente do número de *threads*. A compilação de

tal programa com o compilador `gcc` necessita da opção `-fopenmp`, como no exemplo abaixo:

```
$ gcc -fopenmp -o hello hello.c
```

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(){
5     int myid, nthreads;
6
7     #pragma omp parallel private(myid)
8     {
9        myid = omp_get_thread_num();
10       nthreads = omp_get_num_threads();
11
12       printf("Hello world. I am thread %d of %d\n",
13                 myid, nthreads);
14    }
15    return 0;
16 }
```

Figura 3.3. Exemplo de um *Hello world* em OpenMP.

A execução ocorre da mesma forma que qualquer outro programa em um terminal. Se nenhum argumento é especificado, o programa utilizará todos os *cores* disponíveis no processador. Em nosso exemplo, assumindo que a máquina possui um processador de dois *cores*, a execução será:

```
$ ./hello
0 of 2 - hello world!
1 of 2 - hello world!
```

Na linha de comando, pode-se alterar o número de *threads* com a variável de ambiente `OMP_NUM_THREADS`. Por exemplo, com 4 *threads*:

```
$ OMP_NUM_THREADS=4 ./hello
0 of 4 - hello world!
1 of 4 - hello world!
2 of 4 - hello world!
3 of 4 - hello world!
```

3.4.2. Modelo de execução

O paralelismo em OpenMP é chamado *fork/join*, ou seja, o programa inicia com uma *thread*, a *thread* inicial. Ao encontrar uma construção `parallel`, o programa cria ou bifurca (*fork*) um grupo de *threads* que executam um bloco estruturado de código. Essas *threads* são então unidas (*join*) ao final do bloco.

A Figura 3.4 mostra um exemplo de execução OpenMP com três regiões paralelas. A *thread* inicial, que encontra a construção `parallel`, é chamada de *thread* **master**. Ela é responsável por criar um grupo de *threads* que executará o bloco paralelo. As regiões sequenciais são aquelas fora da construção `parallel` e são executadas pela *thread* master. Por outro lado, as regiões paralelas executam nos *cores* disponíveis e podem variar o número de *threads* no decorrer da execução. Nesse exemplo (Figura 3.4) existem três regiões paralelas com quatro, seis e três *threads*, respectivamente.

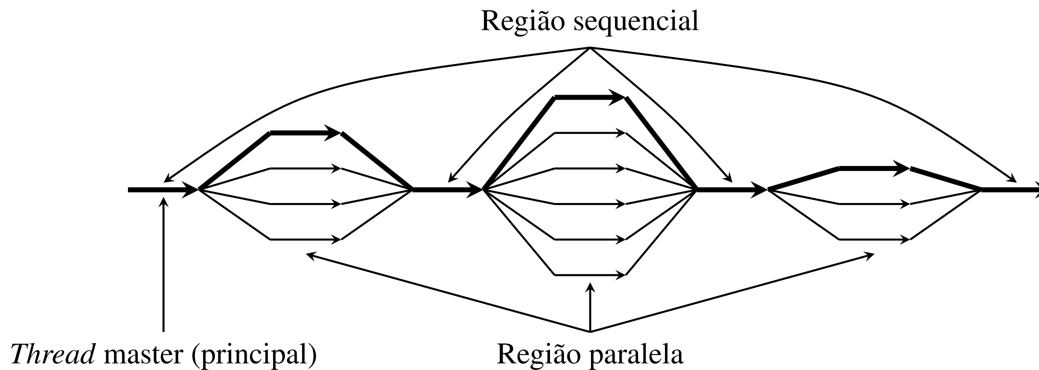


Figura 3.4. Modelo de execução *fork/join* do OpenMP.

A execução dentro de um bloco `parallel` é SPMD (*single program multiple data*), ou seja, as *threads* do grupo executam o mesmo código. A execução em SPMD é amplamente utilizada em alto desempenho e principalmente conhecida por seu uso em programas MPI. Cada *thread* possui um identificador como veremos a seguir.

3.4.3. Métodos de biblioteca

Os métodos da biblioteca OpenMP atuam para modificar e monitorar *threads*, processos e a região paralela do programa. Elas são ligadas como funções externas em C. É necessário incluir a biblioteca no arquivo fonte do código (com `#include <omp.h>`). A seguir são listadas as principais funções de OpenMP:

`void omp_set_num_threads(int N)`

Modifica o número de *threads* da próxima região paralela.

`int omp_get_num_threads()`

Retorna o número de *threads* ativas naquele momento da execução.

`int omp_get_thread_num()`

Retorna o identificador da *thread* atual, também conhecido como *id*.

Um exemplo de uso dessas funções pode ser visto na Figura 3.3.

3.4.4. Cláusulas de dados

O OpenMP é uma API de programação paralela para memória compartilhada, então grande parte das variáveis em memória são compartilhadas. Porém, nem todas as varáveis podem ser compartilhadas. Por exemplo, variáveis da pilha de funções e automáticas (de blocos de código) dentro de uma região paralela são privadas.

O OpenMP permite especificar e modificar o modo de acesso dentro de construções `parallel` por meio de cláusulas. As cláusulas para dados em OpenMP são:

private - cria uma cópia local da memória para cada *thread*. Não inicializa as cópias criadas e não mantém o valor após o fim da execução da região paralela.

shared - indica que a variável é compartilhada entre todas as *threads*. Esse é o padrão quando nada é especificado.

firstprivate - cria uma cópia local da memória para cada *thread*, e inicializa cada uma com o último valor fora da região paralela.

lastprivate - copia o valor da última iteração dentro da região paralela para a variável única após a região paralela.

3.4.5. Laços paralelos

Os laços paralelos são uma das principais construções do OpenMP devido a sua popularidade e ocorrência em aplicações paralelas. O laço paralelo distribui as iterações entre as *threads* disponíveis, o que justifica a construção ser chamada **worksharing**.

A Figura 3.5 mostra um exemplo de laço paralelo em OpenMP, onde a soma das posições do vetor `v` será dividido entre as *threads* da região paralela. As construções `parallel` e `for` podem ser combinadas em uma única linha como em `#pragma omp parallel for`, isso, caso exista apenas uma região `for` dentro da região paralela.

3.4.6. Exclusão mútua

Uma pergunta que surge é como as *threads* de programas paralelos interagem? Como foi visto, OpenMP é um modelo de memória compartilhada. Nesse sentido, as *threads* comunicam-se através de variáveis compartilhadas. Ao longo da execução do programa, podem acontecer compartilhamentos não intencionais de dados causando condições de corrida. Uma condição de corrida ocorre quando a saída do programa muda caso as *threads* sejam escalonadas de uma forma diferente. O problema existe quando duas ou mais *threads* tentam alterar as mesmas posições de memória como na Tabela 3.1. Nesta representação, deseja-se fazer a soma de todos elementos de um vetor, considerando que há 2 *threads* e que todos os elementos do vetor são iguais a 1. Entre os tempos 1 e 4 tudo ocorre como esperado. Porém, nos tempos 5 e 6, as operações de ambas as *threads* se sobrepõem, fazendo literalmente que uma das operações de soma seja perdida, causando um resultado errado.

Para resolver tal problema, utilizou-se a sincronização. A sincronização é necessária em programação paralela a fim de coordenar a execução e evitar condições de

```
long long int sum(int *v, long long int N){
  long long int i, sum_local, sum = 0;

  #pragma omp parallel private(i, sum_local)
  {
    sum_local = 0;

    #pragma omp for
    for(i = 0; i < N; i++)
      sum_local += v[i];

    #pragma omp atomic
    sum += sum_local;
  }
  return sum;
}
```

Figura 3.5. Laço paralelo com OpenMP.

Tabela 3.1. Exemplo de execução do código de soma dos elementos de um vetor com o problema das condições de corrida.

Tempo	Thread 0	Thread 1
1	Ler sum=0	
2	Escrever sum=1	
3		Ler sum=1
4		Escrever sum=2
5	Ler sum=2	Ler sum=2
6	Escrever sum=3	Escrever sum=3

corrida. Em OpenMP pode-se encontrar diversas formas de sincronização desde controle de ordem de execução até regiões críticas. Vale a pena lembrar que a sincronização é cara e, por isso, tenta-se mudar a forma de acesso aos dados para minimizar a necessidade de sincronizações.

A sincronização assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução. As duas formas mais comuns de sincronização são a barreira e a exclusão mútua. Na barreira, cada *thread* espera na barreira até a chegada de todas as demais. Já a exclusão mútua, define um bloco de código onde apenas uma *thread* pode executar por vez.

Para a barreira, utiliza-se a diretiva `barrier`. Para exclusão mútua, pode-se usar duas diretivas: `critical` e `atomic`. A diretiva `critical` especifica que o bloco de código é uma região crítica e apenas uma *thread* por vez executa a região. A diretiva `atomic` tem o mesmo objetivo, entretanto, diferente da `critical` que é implementada

em *software*, a `atomic` é implementada em *hardware* utilizando instruções especiais da arquitetura. A diretiva `atomic` é muito veloz em relação a `critical`, entretanto, ela só implementa um conjunto de operações específicas que incluem incrementos, atribuições e operações simples. A Figura 3.5 mostra um exemplo de uso do `atomic`, onde um valor é acumulado. A acumulação é atômica e concorrente.

Além dessas diretivas, existem outras, como a `master`, que define uma região em que apenas a *thread* 0 executa. Caso não seja necessário que a *thread* 0 execute, mas apenas uma das *threads*, pode ser utilizada a construção `single`. Outra diferença entre a diretiva `single` e a `master` é que a `single` adiciona uma barreira implícita após seu término. Isto é, apesar de apenas uma *thread* executar o bloco `single`, todas as outras *threads* ficam aguardando a execução finalizar para prosseguir. Caso não seja necessária a barreira, deve-se adicionar a diretiva `nowait` ao comando resultando em `#pragma omp single nowait`.

3.4.7. Redução

Em algumas situações, as aplicações paralelas precisam reduzir ou acumular um certo valor de forma concorrente dentro de um laço. Essa situação é bem comum, e chama-se redução. O suporte a tal operação é fornecido pela maioria dos ambientes de programação paralela. Tal funcionalidade é suportada em OpenMP com a cláusula `reduction`. Basicamente, uma redução é a combinação de variáveis locais de uma *thread* em uma variável única.

Uma redução em OpenMP possui a sintaxe `reduction (op : list)`, onde `op` é a operação e `list` é a lista de variáveis a serem acumuladas. Dentro de um bloco cada variável de `list` gera uma cópia local (por *thread*) e é inicializada de acordo com a operação (ex.: `0` para a operação `+`). Atualizações por iteração acontecem localmente em cada *thread* e, ao fim do bloco (*join*), as cópias locais são reduzidas em um valor único e combinadas com o valor original. Note que as variáveis em `list` devem ser compartilhadas (`shared`) dentro da região paralela.

```
1 long long int sum(int *v, long long int N){
2    long long int i, sum = 0;
3
4    #pragma omp parallel for private(i) reduction(+ :  sum)
5    for(i = 0; i < N; i++)
6      sum += v[i];
7
8    return sum;
9 }
```

Figura 3.6. Exemplo do cálculo de soma de vetor com redução em OpenMP.

Na Figura 3.6 o cálculo da soma de todos os elementos de um vetor `v` é utilizado como exemplo. Anteriormente calculou-se este resultado utilizando somas locais. O exemplo difere do anterior por conter a adição da construção `parallel for` com a operação de redução `+` para acumular os resultados na variável `sum`. As operações

suportadas pela redução são `+`, `−`, `*`, `min`, `max`, `&`, `|`, `^`, `&&` e `||`.

3.4.8. Vetorização

O paralelismo com execução vetorial ocorre de forma diferente do paralelismo em *multicore*. Enquanto na execução normal cada instrução opera em apenas um dado, na instrução vetorial a mesma operação é executada em vários dados de forma independente (SATISH et al., 2012). Considerando o laço apresentado na Figura 3.7, que soma dois vetores e armazena o resultado em um terceiro, pode-se perceber que as iterações do laço são independentes. Supondo que há instruções para ler e escrever 8 operandos na memória, e somar 8 operandos, pode-se visualizar o mesmo laço sendo operado vetorialmente, operando 8 por unidade de tempo utilizando o `#pragma omp simd`.

A lógica deste comando é semelhante ao `pragma omp for`, com a diferença que agora o paralelismo dá-se vetorialmente. Ele também aceita a cláusula `reduction`, sendo que, é responsabilidade do programador assegurar que as iterações são independentes.

Em relação ao exemplo, cada iteração do laço carrega 8 operandos a partir da posição i dos vetores b e c, soma-se cada par $(b[i], c[i])$ de forma independente, e depois o bloco de 8 operandos é escrito no vetor a a partir da posição i. O número de operandos por unidade de tempo depende tanto do tamanho do dado quanto do tamanho da unidade vetorial do processador alvo.

```
1 void sum(int *a, int *b, int *c, long long int N){
2    long long int i;
3
4    #pragma omp simd
5    for(i = 0; i < N; i++)
6      a[i] = b[i] + c[i]
7 }
```

Figura 3.7. Exemplo do cálculo de soma de dois vetores com SIMD.

As instruções vetoriais já estão presentes há muitos anos em processadores x86. A cada nova geração, aumenta-se a quantidade de dados processados por instrução, bem como o número de instruções vetoriais disponíveis. É importante ressaltar que, para maior eficiência, os endereços acessados no laço em iterações sucessivas devem ser consecutivos.

3.5. Avaliação de desempenho com contadores de *hardware*

Em arquitetura de computadores, contadores de desempenho de *hardware* são um conjunto de registradores de finalidade especial, os quais são incorporados nos processadores modernos para armazenar a contagem de atividades relacionadas ao seu funcionamento e desempenho. Usuários, pesquisadores e desenvolvedores utilizam esses contadores para analisar e otimizar o desempenho de processadores.

O número de contadores de *hardware* disponíveis em um processador é limitado,

sendo que isso varia de modelo de processador. Também existem limitações a quantos contadores podem ser medidos ao mesmo tempo. Para ler esses registradores podem ser necessárias permissões especiais no sistema operacional, além da utilização de alguma ferramenta disponível no sistema. Duas ferramentas conhecidas que realizam as chamadas de sistema necessárias para isso são o perf (*Performance Counters for Linux*) e o PAPI (*Performance Application Programming Interface*). Ambos serão vistos a seguir.

3.5.1. Linux perf

O Linux *perf* é uma ferramenta de análise de desempenho disponível no Linux, desde o *kernel* 2.6 (MELO, 2010; WEAVER, 2013). A ferramenta é chamada a partir da linha de comando e permite criar perfis estatísticos de todo sistema e de aplicações específicas. Na Tabela 3.2, podemos ver alguns exemplos de eventos que o *perf* mede.

Tabela 3.2. Alguns eventos medidos pelo Perf

Evento Perf	Descrição
`L1-dcache-loads`	*Loads* na *cache* L1
`L1-dcache-load-misses`	*Misses* na *cache* L1
`LLC-loads`	*Loads* na *cache* de último nível
`LLC-load-misses`	*Misses* na *cache* de último nível
`simd_fp_256.packed_single`	Operações de precisão simples AVX-256
`simd_fp_256.packed_double`	Operações de precisão dupla AVX-256
`instructions`	Total de instruções
`cycles`	Total de ciclos
`SMT_2T_Utilization`	Fração de ciclos que utilizou SMT
`GFLOPs`	Giga operações de ponto flutuante por segundo
`IPC`	Instruções por ciclo
`ILP`	Paralelismo em nível de instrução
`MLP`	Paralelismo em nível de memória

Para listar os eventos do perf disponíveis na arquitetura alvo basta digita o comando:

```
$ perf list
List of pre-defined events (to be used in -e):

branch-instructions
branch-misses
cache-misses
cache-references
cpu-cycles
instructions
```

Após, é possível medir os contadores de uma aplicação especifica com o comando:

```
$ perf stat -e cache-misses,cache-references ./mult 2048
```

```
    Performance counter stats for mult 2048:

        8.175.407.685     cache-misses          #95,2%
        8.591.351.092     cache-references
```

A partir disso, pode-se fazer alguma otimização de *cache* e executar novamente:

```
$ perf stat -e cache-misses,cache-references ./mult 2048
    Performance counter stats for mult 2048:

        112.800.963     cache-misses          #34,1%
        330.311.356     cache-references
```

3.5.2. Performance application programming interface (PAPI)

Outra forma de medir contadores de desempenho de *hardware* é utilizando a ferramenta PAPI(TERPSTRA et al., 2010; WEAVER et al., 2012; JOHNSON et al., 2012). Essa ferramenta, fornece acesso a vários contadores de *hardware* do processador, como por exemplo o número de instruções de um determinado tipo e a taxa de acerto da memória *cache*. Na Tabela 3.3, pode-se ver alguns exemplos de eventos que o PAPI mede.

Tabela 3.3. Alguns eventos medidos pelo PAPI

Evento PAPI	Descrição
PAPI_L1_TCM	*Misses* na *cache* L1
PAPI_L2_TCM	*Misses* na *cache* L2
PAPI_L3_TCM	*Misses* na *cache* L3
PAPI_BR_INS	Instruções de *branch*
PAPI_VEC_SP	Instruções de ponto flutuante de precisão simples
PAPI_VEC_DP	Instruções de ponto flutuante de precisão dupla
PAPI_INT_INS	Instruções de inteiro
PAPI_LD_INS	Instruções de *load*
PAPI_SR_INS	Instruções de *store*
PAPI_TOT_INS	Total de instruções

Diferente do Linux Perf, a ferramenta PAPI é mais baixo nível sendo necessárias alterações no código fonte da aplicação ou a criação de uma biblioteca a ser carregada no código binário via `LD_PRELOAD` (PULO, 2009; CIESLAK, 2015). A compilação de um programa OpenMP que utilizada PAPI, com o compilador `gcc`, necessita da opção `-lpapi`, como no exemplo abaixo:

```
$ gcc -fopenmp -o hello hello.c -lpapi
```

A Figura 3.8 ilustra as funções e a ordem necessária para inicializar o PAPI e medir o desempenho de um código em OpenMP, por exemplo. Na linha 1, incluiu-se a biblioteca PAPI. Após, na linha 4 inicializou-se a biblioteca. Na linha 7 definiu-se um conjunto de eventos. Nesse conjunto, na linha 10, adicionou-se o evento `PAPI_TOT_INS`, que mede

o número total de instruções que o programa executa. Na linha 13, definiu-se o início da medição de desempenho. A linha 15 representa uma ou mais linhas de um programa em C que se deseja medir o desempenho. Esse programa pode ser todo o programa ou apenas um trecho de código como, por exemplo, uma função específica. Após a execução desse programa, na linha 18, parou-se a medição do PAPI. Na linha 21, removeu-se o evento `PAPI_TOT_INS`. Por fim, nas linhas 24 e 27, desligou-se o conjunto de eventos e a biblioteca PAPI. Na linha 29 é mostrado na tela o resultado da métrica medida.

```
1  #include<papi.h>
2
3  /* Inicia a biblioteca PAPI */
4  PAPI_library_init(PAPI_VER_CURRENT);
5
6  /* Inicia o conjunto de eventos */
7  PAPI_create_eventset(&EventSet);
8
9  /* Adiciona o evento PAPI_TOT_INS */
10 PAPI_add_named_event(EventSet, "PAPI_TOT_INS");
11
12 /* Inicia o PAPI */
13 PAPI_start(EventSet);
14
15 /* PROGRAMA QUE DESEJA-SE MEDIR O DESEMPENHO */
16
17 /* Para o PAPI */
18 PAPI_stop(EventSet, value);
19
20 /* Remove o evento PAPI_TOT_INS */
21 PAPI_remove_named_event(EventSet, "PAPI_TOT_INS");
22
23 /* Desliga o conjunto de eventos */
24 PAPI_destroy_eventset(&EventSet);
25
26 /* Desliga o PAPI */
27 PAPI_shutdown();
28
29 /* Mostra o resultado na tela */
30 printf("PAPI_TOT_INS = %d\n", value[0]);
```

Figura 3.8. Exemplo de como medir desempenho utilizando PAPI.

3.6. Estudo de caso: aplicação geofísica

Nesta seção, é visto como avaliar e otimizar o desempenho de uma aplicação de geofísica em um processador *multicore*. Primeiro apresenta-se a aplicação e após, o ambiente de execução e a discussão das otimizações e dos resultados.

3.6.1. Modelagem Fletcher

A Modelagem Fletcher (FLETCHER; DU; FOWLER, 2009) simula a propagação de ondas em meio anisotrópico em um domínio ao longo do tempo. As ondas são emitidas por uma fonte, tipicamente no interior ou na borda do domínio, o qual é um paralelepípedo tridimensional. O código[2] foi escrito em linguagem *C* e a discretização foi feita utilizando diferenças finitas.

A modelagem simula a coleta de dados em um levantamento sísmico, como na Figura 3.9. De tempos em tempos, equipamentos acoplados ao navio emitem ondas que refletem e refratam as mudanças de meio no subsolo. Eventualmente essas ondas voltam à superfície do mar, sendo coletadas por microfones específicos acoplados a cabos rebocados pelo navio. O conjunto de sinais recebidos por cada fone ao longo do tempo constitui um traço sísmico. Para cada emissão de ondas, gravam-se os traços sísmicos de todos os fones do cabo. O navio continua trafegando e emitindo sinais ao longo do tempo.

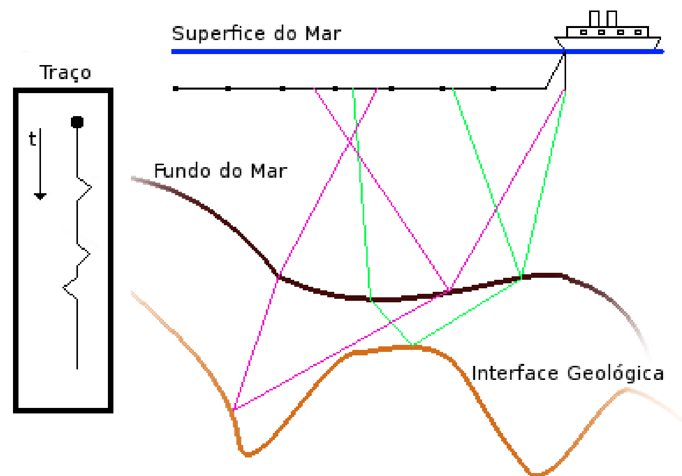


Figura 3.9. Coleta de dados em levantamento sísmico marítimo.

3.6.2. Avaliação e otimização de desempenho

Os resultados apresentados nessa seção foram medidos no sistema *draco* do Grupo de Processamento Paralelo e Distribuído[3] (GPPD) da Universidade Federal do Rio Grande do Sul (UFRGS). Cada nó da *draco* consiste de 2 processadores Intel Xeon E5-2630 de 8 núcleos, totalizando 16 *threads (Hyper-Threading)*, além de 64 GB DDR4.

Nessa atividade, o objetivo é implementar uma versão paralela e vetorial otimizada da aplicação de Geofísica. O primeiro passo é verificar o desempenho da *cache*, buscando após, encontrar um dos laços para vetorizar. Utilizando o comando:

```
$ perf stat -e cache-misses,cache-references src/6-kernel.exec
```

verifica-se a taxa de acerto de *cache* da versão original. A taxa retornada nesse

[2]Esse código é parcialmente financiado por recursos do projeto Petrobras 2016/00133-9.
[3]<http://gppd-hpc.inf.ufrgs.br/>

caso foi de 27%, sendo que essa é considerada baixa. Buscando melhorar o desempenho da aplicação, otimizando os acessos a *cache*, utilizou-se a técnica de *loop interchange* para alterar a ordem dos laços. A taxa de acerto para diferentes combinações pode ser vista na Tabela 3.4. Uma vez que a ordem `k, Y, X` obteve a maior taxa de acerto, continuou-se com essa, agora buscando efetuar a vetorização.

Tabela 3.4. Taxa de Acerto da Aplicação Geofísica

Ordem dos laços	Taxa de Acerto (%)
`X, Y, k`	26.9%
`Y, k, X`	93.6%
`k, Y, X`	95.8%

No caso da vetorização, identificou-se que o laço mais interno pode ser vetorizado. Isso é possível, pois a maior parte dos acessos à memória é feita de forma contígua em direção aos pontos em X. O perf também pode ser utilizado para analisar o uso de instruções SIMD via comando:

```
$ perf stat -e
   simd_fp_256.packed_single,simd_fp_256.packed_double
   src/6-kernel.exec
```

verificou-se que após adicionar a diretiva `omp simd`, o número de instruções `simd` de precisão simples mudaram de 0 para 3429629397.

Por fim, a Tabela 3.5 apresenta os resultados de tempo de execução das diferentes versões utilizando como entrada um cubo de tamanho 512. A versão na qual melhorou-se o desempenho da *cache* foi 5,2× mais rápida que a versão sequencial. A versão vetorial utilizando unidades vetoriais melhorou o desempenho da aplicação em 3,6× em relação a versão com *cache* otimizada. A versão paralela em OpenMP foi executada com 32 *threads*, melhorando o desempenho da aplicação em 4,1×. A versão final, que combina todas otimizações, teve o melhor desempenho, de 75,5× em relação a versão sequencial. Essas e outras propostas de otimização para essa aplicação de geofísica podem ser vistas em diversos trabalhos (SERPA et al., 2017, 2018b, 2018a, 2019).

Tabela 3.5. Desempenho da Aplicação Geofísica

Versão	Tempo de Execução (s)
Sequencial	90,6
Cache	17,5
Vetorizada	4,9
OpenMP	1,2

3.7. Conclusão

O uso de arquiteturas paralelas e vetoriais é uma solução para incrementar a capacidade de execução, tornando possível a computação eficiente de aplicações com grande demanda

de processamento. Para tanto, é preciso fazer uso de interfaces de programação paralela. Neste capítulo foram apresentados detalhes da interface *OpenMP*, a qual é amplamente utilizada em aplicações de alto desempenho para ambientes de memória compartilhada. Com isso, é possível criar aplicações com múltiplas *threads* de forma prática. Desta forma, é possível a solução eficiente de problemas que possuem algum tipo de concorrência.

Além disso, mostrou-se como utilizar as ferramentas perf e PAPI para analisar o desempenho de aplicações paralelas. Essas ferramentas leem contadores de *hardware* do processador e auxiliam no entendimento de otimizações propostas. Desta forma, como mostrado no estudo de caso de uma aplicação geofísica, é possível maximizar o uso dos recursos computacionais disponíveis nas arquiteturas atuais, com isso, melhorando o desempenho de aplicações sintéticas e reais.

No futuro, planeja-se abordar outras interfaces de programação paralela como Intel TBB, MPI, CUDA, entre outras.

Exercícios e soluções deste capítulo estão disponíveis em:

<https://gitlab.com/msserpa/prog-paralela-erad>.

Referências

AYGUADÉ, E. et al. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 20, n. 3, p. 404–418, 2008. páginas 6

BORKAR, S.; CHIEN, A. A. The future of microprocessors. *Communications of the ACM*, ACM New York, NY, USA, v. 54, n. 5, p. 67–77, 2011. páginas 2, 3

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. [S.l.]: MIT press, 2008. v. 10. páginas 5

CIESLAK, R. Dynamic linker tricks: Using ld_preload to cheat, inject features and investigate programs. *Retrieved March*, v. 12, p. 2015, 2015. páginas 14

COOK, S. *CUDA programming: a developer's guide to parallel computing with GPUs*. [S.l.]: Newnes, 2012. páginas 4, 5

COTEUS, P. W. et al. Technologies for exascale systems. *IBM Journal of Research and Development*, IBM, v. 55, n. 5, p. 14–1, 2011. páginas 2, 3

CRUZ, E. H. et al. Lapt: A locality-aware page table for thread and data mapping. *Parallel Computing (PARCO)*, Elsevier, v. 54, p. 59–71, 2016. páginas 3

FLETCHER, R. P.; DU, X.; FOWLER, P. J. Reverse time migration in tilted transversely isotropic (tti) media. *Geophysics*, Society of Exploration Geophysicists, v. 74, n. 6, p. WCA179–WCA187, 2009. páginas 16

FLYNN, M. J.; RUDD, K. W. Parallel architectures. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 28, n. 1, p. 67–70, 1996. páginas 3

FOSTER, I. *Designing and building parallel programs: concepts and tools for parallel software engineering*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1995. páginas 4

FRUEHE, J. Multicore processor technology. *Reprinted from Dell Power Solutions www. dell. com/powersolutions (Obtained from the Internet on Mar. 23, 2012)*, p. 67–72, 2005. páginas 2

GEPNER, P.; KOWALIK, M. F. Multi-core processors: New way to achieve high system performance. In: IEEE. *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*. [S.l.], 2006. p. 9–13. páginas 2

GROPP, W.; SNIR, M. Programming for exascale computers. *Computing in Science and Engineering*, IEEE, v. 15, n. 6, p. 27–35, 2013. páginas 3

GROPP, W.; THAKUR, R.; LUSK, E. *Using MPI-2: Advanced features of the message passing interface*. [S.l.]: MIT press, 1999. páginas 5, 6

HSU, J. Three paths to exascale supercomputing. *IEEE Spectrum*, IEEE, v. 53, n. 1, p. 14–15, 2015. páginas 3

JOHNSON, M. et al. Papi-v: Performance monitoring for virtual machines. In: IEEE. *2012 41st International Conference on Parallel Processing Workshops*. [S.l.], 2012. p. 194–199. páginas 14

KIRK, D. B.; WEN-MEI, W. H. *Programming massively parallel processors: a hands-on approach*. [S.l.]: Morgan kaufmann, 2016. páginas 4, 5

MARTINEAU, M.; MCINTOSH-SMITH, S.; GAUDIN, W. Evaluating openmp 4.0's effectiveness as a heterogeneous parallel programming model. In: IEEE. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. [S.l.], 2016. p. 338–347. páginas 6

MELO, A. C. D. The new linux'perf'tools. In: *Slides from Linux Kongress*. [S.l.: s.n.], 2010. v. 18, p. 1–42. páginas 13

MITTAL, S.; VETTER, J. S. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 47, n. 4, p. 1–35, 2015. páginas 3

OPENMP, A. Openmp application program interface v3. 0. *OpenMP Architecture Review Board*, 2008. [Online; accessed 15-January-2020]. páginas 5, 6

PITT-FRANCIS, J.; WHITELEY, J. An introduction to parallel programming using mpi. In: *Guide to Scientific Computing in C++*. [S.l.]: Springer, 2017. p. 197–224. páginas 5, 6

PULO, K. Fun with ld_preload. In: *linux. conf. au*. [S.l.: s.n.], 2009. v. 153. páginas 14

REINDERS, J. An overview of programming for intel xeon processors and intel xeon phi coprocessors. *Intel Corporation, Santa Clara*, v. 1, p. 1550002, 2012. páginas 5

ROBISON, A. D. Composable parallel patterns with intel cilk plus. *Computing in Science and Engineering*, IEEE Computer Society, v. 15, n. 2, p. 66–71, 2013. páginas 5

SATISH, N. et al. Can traditional programming bridge the ninja performance gap for parallel computing applications? In: IEEE. *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. [S.l.], 2012. p. 440–451. páginas 12

SERPA, M. S. et al. Strategies to improve the performance of a geophysics model for different manycore systems. In: IEEE. *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. [S.l.], 2017. p. 49–54. páginas 17

SERPA, M. S. et al. Optimization strategies for geophysics models on manycore systems. *The International Journal of High Performance Computing Applications*, SAGE Publications Sage UK: London, England, v. 33, n. 3, p. 473–486, 2019. páginas 17

SERPA, M. S. et al. Improving oil and gas simulation performance using thread and data mapping. In: SPRINGER. *Symposium on High Performance Computing Systems*. [S.l.], 2018. p. 55–68. páginas 17

SERPA, M. S. et al. Optimizing geophysics models using thread and data mapping. In: IEEE. *2018 Symposium on High Performance Computing Systems (WSCAD)*. [S.l.], 2018. p. 135–141. páginas 17

SUPINSKI, B. R. de et al. The ongoing evolution of openmp. *Proceedings of the IEEE*, IEEE, v. 106, n. 11, p. 2004–2019, 2018. páginas 6

TERPSTRA, D. et al. Collecting performance data with papi-c. In: *Tools for High Performance Computing 2009*. [S.l.]: Springer, 2010. p. 157–173. páginas 14

WEAVER, V. M. Linux perf_event features and overhead. In: *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*. [S.l.: s.n.], 2013. v. 13. páginas 13

WEAVER, V. M. et al. Measuring energy and power with papi. In: IEEE. *2012 41st International Conference on Parallel Processing Workshops*. Pittsburgh, PA, USA, 2012. p. 262–268. páginas 14

Capítulo 4

Introdução ao Desenvolvimento de Aplicações Paralelas com o Paradigma Orientado a Tarefas e o Runtime StarPU

Lucas Leandro Nesi, Vinícius Garcia Pinto, Marcelo Cogo Miletto, Lucas Mello Schnorr
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Samuel Thibault
LaBRI, STORM, INRIA
Bordeaux, França

Resumo

As novidades e a heterogeneidade de recursos computacionais na área da computação de alto desempenho, como aceleradores GPGPUs, continuam contribuindo para a complexidade da programação paralela. O paradigma orientado a tarefas permite algumas facilidades nesta programação por que transfere para um runtime muitas responsabilidades que seriam anteriormente realizadas pelo programador nos paradigmas tradicionais. Desta forma, é responsabilidade do runtime escalonar as tarefas, gerenciar a memória, e escolher o recurso/implementação à ser utilizada. Para isso, neste paradigma basta o programador definir as tarefas, suas implementações em recursos heterogêneos diferentes, e suas dependências de dados. Neste minicurso, será apresentado uma introdução ao paradigma de programação paralela orientado a tarefas, e como construir programas que executam em recursos heterogêneos utilizando o runtime StarPU. Demonstraremos como programar tarefas com múltiplas implementações, utilizando CPU e GPU. Apresentaremos exemplos de programas como a soma e multiplicação de matrizes e como encontrar o maior elemento de uma lista. Por fim, faremos a introdução a ferramentas e métodos de como analisar o desempenho destes programas, com o emprego do framework StarVZ.

4.1. Introdução

Ao longo dos anos o cenário de HPC passou por diversas transformações drásticas, com o objetivo de fornecer cada vez mais poder computacional. A busca por um maior desempenho levou à larga adoção de sistemas heterogêneos. Estes são frequentemente formados por um conjunto de processadores multicore, aliados a aceleradores específicos, como as GPUs. Ao mesmo tempo em que a heterogeneidade nos traz uma maior capacidade computacional, também tem-se um maior desafio ao construir programas que utilizem dessa capacidade em sua totalidade. Programas paralelos devem considerar aspectos como o balanceamento de carga, custos de comunicação, e a escalabilidade e portabilidade de desempenho, o que se torna ainda mais difícil nestes cenários onde existem vários nós computacionais com diferentes tipos de recursos. A heterogeneidade é vista como um caminho para alcançarmos o patamar da computação Exascale (Dongarra et al., 2017).

Tais mudanças arquiteturais demandam o uso de diferentes estratégias para que se possa explorar, de forma eficiente, os recursos computacionais presentes nos supercomputadores de hoje. Dada a maior complexidade dos sistemas de HPC, o trabalho do programador de desenvolver uma aplicação paralela envolve diversos níveis, tornando-se algo igualmente complexo. Para facilitar este trabalho, determinados paradigmas de programação podem ser usados.

O paradigma orientado a tarefas permite a construção de programas paralelos através dos conceitos de tarefas e dependência de dados (AUGONNET et al., 2011). Uma tarefa é a unidade computacional básica, e contém um trecho de código específico que ela irá executar sobre um conjunto de dados. Desta forma, é possível escrever códigos voltados para diferentes dispositivos que realizam uma mesma operação. Cada tarefa acessa determinadas regiões de dados, sendo que os conflitos de leitura e escrita em uma mesma região de memória causam dependências entre diferentes tarefas. A combinação destes dois conceitos implica em um grafo acíclico dirigido (DAG), onde os vértices são as tarefas e as arestas as dependências. Assim, a aplicação pode ser vista nesta estrutura de grafo.

A grande vantagem de se modelar uma aplicação desta maneira é que ela pode ser tratada por outro componente que faz parte deste paradigma: o sistema de *runtime* (THIBAULT, 2018). Esse sistema encontra-se em uma camada entre a aplicação e as interfaces dos recursos computacionais, tendo uma visão geral da aplicação como um DAG, e a disponibilidade e capacidade dos recursos computacionais. Essa camada facilita uma série de tarefas que antes eram de responsabilidade do programador, como por exemplo o controle dos recursos, o escalonamento das tarefas de forma paralela, e o gerenciamento dos dados entre os diferentes recursos. Isso torna este paradigma uma opção interessante, flexível e versátil para o desenvolvimento de aplicações paralelas.

O restante deste documento está estruturado da seguinte forma. A Seção 4.2 apresenta as noções básicas sobre o Paradigma de Programação Paralela Orientada a Tarefas e os conceitos a serem utilizados. A Seção 4.3 apresenta o *Runtime* StarPU e suas particularidades. A Seção 4.4 mostra passo a passo como construir a primeira aplicação utilizando o StarPU. A Seção 4.5 traz outros exemplos de programas utilizando o StarPU. A Seção 4.6 apresenta uma introdução de como analisar estes programas. A Seção 4.7 conclui este minicurso.

4.2. Paradigma de programação paralela orientado a tarefas

O Paradigma de Programação Paralela Orientado a Tarefas (*task-based programming* ou *data flow scheduling*) é um conceito (BRIAT et al., 1997) que utiliza uma estratégia mais declarativa na programação. Isso permite que um *runtime* (*midleware*) interprete estas declarações e seja responsável por ações como escalonamento e gerenciamento dos dados. Em outros paradigmas tradicionais, estas ações devem ser programadas diretamente pelo desenvolvedor da aplicação paralela. No geral, esta estratégia facilita na programação e reduz a complexidade no desenvolvimento de aplicações paralelas, entretanto, reduz o controle do programador sobre a execução (Dongarra et al., 2017). Este paradigma está se tornando cada vez mais popular desde a década de 2000, quando diferentes projetos começaram a utilizá-lo (THIBAULT, 2018; Dongarra et al., 2017) tendo em vista a heterogeneidade dos sistemas computacionais de alto desempenho.

A estrutura de uma aplicação paralela orientada a tarefas consiste em um conjunto de tarefas e um conjunto de bloco de dados. Cada tarefa, que normalmente é uma região de código sequencial (em CPU), trabalha em alguns destes blocos de dados, aplicando suas operações neles e modificando-os, ou salvando seus resultados em outros blocos de dados. Um bloco de dados é simplesmente uma subdivisão do domínio a ser trabalhado. Se a aplicação trabalha em uma matriz, cada bloco de dados pode ser uma submatriz, e pode ser identificado com uma coordenada, por exemplo. Se o domínio da aplicação é uma lista, cada bloco de dados por ser uma sub lista contínua da posição i até a posição j. Uma tarefa possui uma implementação e pode ser executada várias vezes sobre conjunto de dados diferentes. Como por exemplo, uma tarefa de multiplicar submatrizes de uma matriz, pode ser aplicada em submatrizes de coordenadas diferentes várias vezes. Neste caso, cada tarefa em dados diferentes possui um identificador único, mas o nome da tarefa e a implementação é igual para todos elas. Alguns *runtimes* permitem a submissão de tarefas apenas informado os blocos de dados utilizados. Desta maneira, a interação entre as tarefas ocorre devido a utilização dos mesmos blocos de memória por elas. Assim, se uma tarefa T^A, submetida primeiro, modifica um bloco de dados D^A, e em seguida, uma tarefa T^B, submetida depois de T^A, lê D^A, existe uma dependência implícita entre T^A e T^B. Na maneira que T^A é uma dependência de T^B, e deve ser executada antes de T^B para garantir a coerência da aplicação.

Usualmente, uma aplicação paralela orientada a tarefas pode ser representada por um grafo acíclico dirigido (*Directed Acyclic Graph* – DAG). Neste grafo, os nós são as tarefas, e as arestas as dependências entre elas. Uma aresta dirigida da tarefa T^A para a T^B significa que T^A deve ser executada antes de T^B. Estas dependências podem ser herdadas da reutilização de dados (como explicado anteriormente) ou, em alguns *runtimes,* explicitamente inseridas pelo desenvolvedor.

Discretizar um algoritmo para o paradigma orientado a tarefas nem sempre é um processo trivial, e nem todos os algoritmos são bons candidatos para tal conversão (THIBAULT, 2018). Entretanto, isso não significa que os problemas não admitam algoritmos que utilizem este paradigma de forma eficiente.

Vamos selecionar um problema simples, uma operação de redução em um vetor, para demonstrar a criação de um algoritmo orientado a tarefas e a construção do DAG correspondente. Neste exemplo, vamos procurar o maior valor em um vetor. Sabemos

que podemos dividir a lista em sub listas para procurar o maior elemento em cada uma e depois procurar o maior entre os resultados. Desta maneira, podemos definir uma tarefa como responsável por procurar o maior valor em uma sub lista e retornar o maior valor. Para fins didáticos, vamos considerar uma lista de 27 elementos. Podemos criar uma tarefa que descobre o maior número entre três elementos, vamos chamá-la de `MD3`, *maior dos três*. Em um primeiro momento, podemos aplicar ela nove vezes nesta lista, em nove triplas diferentes, gerando nove resultados. Sabemos que o maior número da lista é um dentre este nove resultados, basta analisar eles novamente, aplicando a mesma tarefa em cada uma das três possíveis triplas resultantes da primeira etapa. A Figura 4.1 demonstra o grafo da execução destas tarefas com o comportamento dos dados da lista de 27 elementos. Todas as tarefas utilizadas são do tipo `MD3` e são aplicadas em triplas contínuas da lista. As tarefas possuem um identificador (`id`) mostrando uma possível ordem de criação delas. Neste exemplo, o maior elemento é o número 99, primeiramente encontrado pela tarefa 6, e então pela tarefa 11 e finalmente pela última tarefa 13. O DAG desta aplicação mostrando as dependências entre as tarefas pode ser visualizado na Figura 4.2.

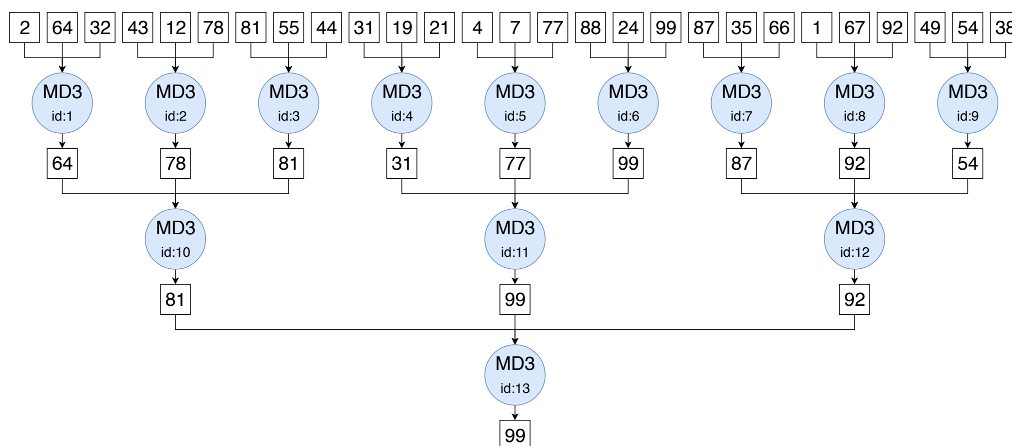


Figura 4.1. Grafo de execução com os dados da aplicação de encontrar o maior elemento em uma lista de 27 elementos utilizando uma tarefa de três entradas chamada de MD3. Fonte: Autores.

O DAG presente na Figura 4.2 exemplifica as oportunidades de paralelismo desta execução. Por exemplo, todas as tarefas do primeiro nível (ids de 1 a 9) são independentes, isso quer dizer, trabalham em dados diferentes e podem ser executadas em paralelo. Ainda, tarefas de níveis diferentes que não possuem dependência, por exemplo, tarefa 1 e tarefa 11 podem também ser executadas em paralelo. Este exemplo mostra os graus de liberdade para possíveis paralelismos. Como a programação é mais declarativa, afinal o programador apenas definiu as tarefas e suas entradas, não foi necessário especificar que tarefa x pode executar ao mesmo tempo com a tarefa y, nem em que momento elas devem executar ou como devem ser mapeadas para os recursos computacionais. Em um exemplo simples como esse, o programador poderia ter descrito manualmente as possibilidades de paralelismo, ou seja, especificar que instruções em quais dados podem ser executadas paralelamente. No entanto, na medida que os problemas se tornam mais complexos, com interações de dados mais robustas, tal processo manual se torna demasiadamente laborioso. Com a especificação e declaração das tarefas pode-se inferir o paralelismo com base

no DAG, e utilizar qualquer combinação dos conjuntos possíveis paralelos de tarefas. Esta inferência pode ser feita durante a execução da aplicação utilizando um *runtime*.

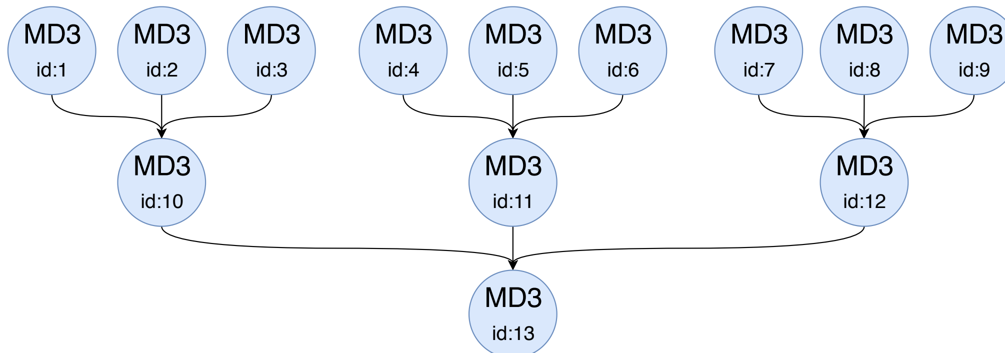


Figura 4.2. DAG da aplicação de encontrar o maior elemento em uma lista de 27 elementos utilizando uma tarefa de três entradas chamada de MD3. Fonte: Autores.

Na computação de alto desempenho, podemos encontrar diversos exemplos de *runtimes*. Duas ferramentas muito populares de programação paralela `OpenMP` (DAGUM; MENON, 1998) e `MPI` (GROPP; LUSK; SKJELLUM, 1999) utilizam runtimes para suas execuções. Isto é, existe uma entidade que é executada juntamente com a aplicação, realizando ações que não foram tomadas diretamente pelo programador. Na programação de aplicações baseadas em tarefas, alguns exemplos de *runtimes* existentes são PaRSEC (BOSILCA et al., 2012), OmpSs (DURAN et al., 2011), OpenMP >4.0(OpenMP, 2013), XKaapi (GAUTIER et al., 2013), e StarPU (AUGONNET et al., 2011). Entre os exemplo citados, o StarPU é um *runtime* que permite a utilização de recursos heterogêneos (CPUs e GPGPUs) e multi-nó com seu módulo StarPU-MPI (AUGONNET et al., 2012).

O *runtime* pode realizar diversas otimizações baseado no DAG. A primeira delas é fazer o mapeamento das tarefas para os recursos, e identificar as oportunidades de paralelismo. Os *runtimes* podem disponibilizar diversos escalonadores para realizar estas operações. Outro exemplo é realizar comunicações em paralelo com as computações, efetuando operações de gerenciamento de memória.

4.3. O runtime StarPU

O StarPU é um *runtime* orientado a tarefas de propósito geral para programação de aplicações paralelas para computadores heterogêneos *multicore*. Ele foi inicialmente desenvolvido por Cédric Augonnet em sua Tese de Doutorado (AUGONNET, 2011) na *Université de Bordeaux*. O StarPU provê uma interface para que aplicações submetam suas tarefas em recursos. A Figura 4.3 apresenta a pilha de software de uma aplicação paralela orientada a tarefas e a localização da aplicação e do *runtime* sobre a arquitetura da máquina.

O StarPU usa o modelo STF (*Sequential Task Flow*) (AGULLO et al., 2016), onde a aplicação pode submeter sequencialmente as tarefas durante a execução e o *runtime* dinamicamente as escalona para os recursos. Neste modelo, o DAG não precisa ser inteiramente conhecido, e as tarefas podem ir sendo submetidas gradativamente, sendo

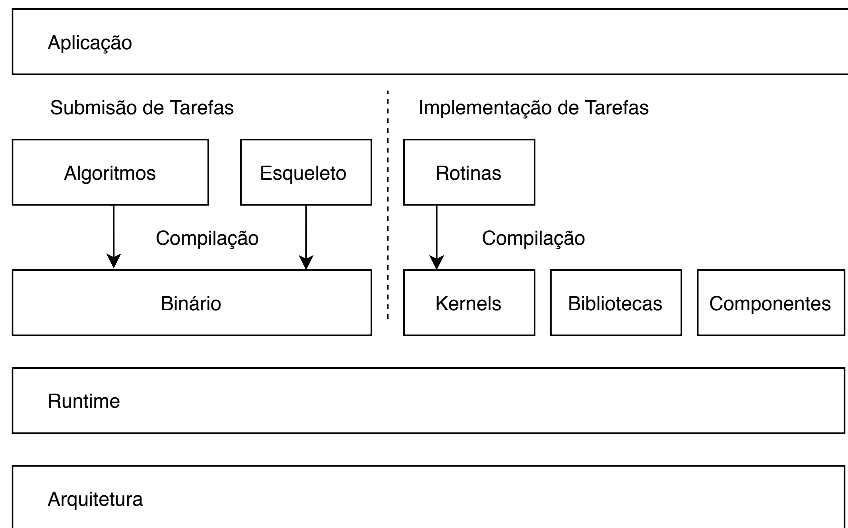


Figura 4.3. *Software stack* Utilizando o runtime StarPU. Fonte: Thibault (2018).

criadas dinamicamente durante a execução. O StarPU associa a cada recurso computacional um trabalhador, sendo que este trabalhador pode executar uma tarefa por vez sem preempção. Por exemplo, cada *core* de um processador e cada GPU é um trabalhador. Ainda, as tarefas podem ter múltiplas implementações (CPU/GPU), e a decisão de qual implementação utilizar é feita pelo *runtime* durante a execução. Isso permite a execução da aplicação em ambientes heterogêneos com uma certa facilidade ao programador.

As dependências entre as tarefas que estruturam o DAG são herdadas da reutilização de dados entre as tarefas. Os blocos de dados no `StarPU` são organizados na estrutura `data_handle`, e devem ser definidos antes da criação das tarefas que os utilizam. Desta maneira, no StarPU, as dependências entre as tarefas são implícitas. Esta característica é mais simples e diferente de outras abordagens como OpenMP-Tasks (v. 3.0) e MPI, onde o programador deve informar as dependências ou comunicações manualmente. No OpenMP 4.0, o programador pode definir as dependências de memória usando a cláusula `depend`.

Para escalonar as tarefas nos trabalhadores, o StarPU pode utilizar diferentes heurísticas de escalonamento. Dependendo da disponibilidade dos trabalhadores e da heurística utilizada, o escalonador pode escolher dinamicamente uma das implementações da tarefa e executá-la. Exemplos de heurísticas de escalonamento são `lws` (local work-stealing), `eager` (deque centralizado), `dm` (deque model) baseado no algoritmo heft (TOPCUOGLU; HARIRI; WU, 2002) onde modelos de desempenho das tarefas são utilizados, `dmda` (deque model data aware) uma versão modificada do `dm` para considerar os dados. O escalonador a ser utilizado pode ser definido pela variável de ambiente `STARPU_SCHED` no momento da execução. Mais informações sobre os escalonadores disponíveis podem ser encontrados na página do StarPU[1].

A análise de desempenho das aplicações em StarPU é possível pela utilização de

[1]<http://starpu.gforge.inria.fr/doc/html/Scheduling.html>

rastros de execução. O StarPU pode ser configurado para gerar estes rastros em formato FxT (DANJEAN; NAMYST; WACRENIER, 2005), que podem ser posteriormente convertidos para outros formatos de arquivo como Pajé (SCHNORR; STEIN; KERGOMMEAUX, 2013) que, por sua vez, pode ser utilizado por diferentes *frameworks* para a análise de desempenho.

4.4. Como construir seu primeiro programa

Esta Seção descreve como construir o primeiro programa utilizando a abordagem orientada a tarefas e o *runtime* StarPU. Primeiramente é descrito como realizar a instalação das bibliotecas necessárias. Após é realizada a elaboração do primeiro programa passo a passo.

4.4.1. Instalação do StarPU e ambiente

A página do StarPU[2] oferece maneiras alternativas de instalação. Para o propósito deste minicurso, utilizaremos o gerenciador de pacotes para supercomputadores `spack`[3]. Para baixar o `spack` é necessário apenas o comando `git` para a clonagem do repositório. Após essa operação, devemos configurar o ambiente carregando um arquivo de configuração. Todos os comandos estão também disponíveis publicamente no *companion*[4]

```bash
git clone https://github.com/spack/spack.git
source spack/share/spack/setup-env.sh
```

Código 4.1. (bash) Baixar Spack

O `StarPU` pode ser obtido utilizando o gerenciador `spack` pelo repositório do `solverstack`[5], que pode ser adicionado pelos seguintes comandos.

```bash
git clone https://gitlab.inria.fr/solverstack/spack-repo.git
spack repo add spack-repo/
```

Código 4.2. (bash) Adicionar repositório no Spack

Com essa configuração podemos instalar o `StarPU` utilizando o `spack`. Este gerenciará a instalação de quase todas as dependências necessárias. Pode-se configurar a instalação utilizando algumas variações, como a instalação do `StarPU` com suporte a `CUDA`. Para fins deste minicurso o seguinte comando instala o `StarPU` com os devidos pacotes necessários e gera um diretório com todos os cabeçalhos e bibliotecas.

```bash
spack install starpu@1.3.1~fast+fxt~mpi+cuda~openmp ^cuda@10.0.130
export STARPU_HOME=$(pwd)/starpu
export PKG_CONFIG_PATH=$STARPU_HOME/pkgconfig:\
       $STARPU_HOME/lib/pkgconfig:$PKG_CONFIG_PATH
export LD_LIBRARY_PATH=$STARPU_HOME/lib:\
       $STARPU_HOME/lib64:$LD_LIBRARY_PATH
spack view soft $STARPU_HOME starpu@1.3.1
```

Código 4.3. (bash) Instalar StarPU e configurar ambiente

[2]<http://starpu.gforge.inria.fr/doc/html/BuildingAndInstallingStarPU.html>
[3]<https://spack.io/>
[4]<https://gitlab.com/lnesi/minicurso-starpu-erad-2020>
[5]<https://gitlab.inria.fr/solverstack/spack-repo>

Podemos utilizar a aplicação `pkg-config` para buscar os cabeçalhos e bibliotecas necessárias para compilar com o GCC um programa escrito com StarPU, assumindo que a variável de ambiente `PKG_CONFIG_PATH` foi devidamente configurada. No caso de uma instalação de StarPU com CUDA, as bibliotecas de CUDA também devem ser especificadas. Podemos compilar o programa `hello world` abaixo com o comando do Código 4.5. Este programa cria uma tarefa que imprime *"Hello World"* e mostra a topologia dos recursos identificados. Nas próximas seções será explicado como construir uma aplicação de somar matrizes.

```
#include <stdlib.h>
#include <limits.h>
#include <starpu.h>

void func_cpu(void *buffers[], void *args)
{
    printf("Hello World!\n");
}

struct starpu_codelet codelet_world =
{
    .cpu_funcs = { func_cpu },
    .nbuffers = 0,
    .name = "hello_world",
};

int main(){
    starpu_init(NULL);
    starpu_topology_print(stdout);
    starpu_task_insert(&codelet_world, 0);
    starpu_task_wait_for_all();
    starpu_shutdown();
}
```

Código 4.4. (C) Esqueleto de uma aplicação StarPU

```
gcc $(pkg-config --cflags starpu-1.3 cuda-10.0) ./hello_world.c\
    $(pkg-config --libs starpu-1.3 cuda-10.0)
```

Código 4.5. (bash) Compilar uma aplicação StarPU

4.4.2. Bases de um programa StarPU

A estrutura de um programa StarPU deve começar com o *include* do cabeçalho da biblioteca do StarPU, `starpu.h`. As funções do StarPU só poderão ser utilizadas após a inicialização da biblioteca, realizada pela função `starpu_init()`. Esta função faz uma verificação no sistema para determinar quantos trabalhadores existem e inicializa as estruturas de dados necessárias. Ao fim do programa, a função `starpu_shutdown()` precisa ser invocada para finalizar o StarPU. Esta chamada é importante porque algumas ações como a gravação dos rastros ocorre somente quando esta função é chamada. Desta maneira, o esqueleto de um programa StarPU pode ser visto no Código 4.6.

```
#include <starpu.h>
```

```
int main(){
    starpu_init(NULL);
    //Criar e Submeter Tarefas
    starpu_shutdown();
    return 0;
}
```

Código 4.6. (C) Esqueleto de uma aplicação StarPU

4.4.3. Descritor de funções (*codelet*)

Codelet é uma estrutura do StarPU (`struct starpu_codelet`) que contém as diversas implementações de um kernel. Basicamente uma tarefa tem um *codelet* associado e irá executar uma das implementações previstas. Para definir um *codelet* é necessário definir pelo menos uma implementação, por exemplo, uma implementação em CPU. Para tal, define-se a variável interna `cpu_funcs` com um ponteiro para uma função, por exemplo `func_cpu`. Utilizamos como exemplo a construção de uma tarefa que faz a soma de duas submatrizes A e B, e salva em C. Precisamos definir quantos blocos de dados este *codelet* usa, usando a variável `nbuffers`, por exemplo 3, e os modos de acesso a cada um dos buffer utilizando a variável `modes`. Utilizamos `STARPU_R` para somente leitura, `STARPU_W` para somente escrita, e `STARPU_RW` para leitura e escrita. Definimos esses valores em uma lista. Neste caso, se queremos os dois primeiros buffers como leitura e o terceiro como escrita, utilizamos `{ STARPU_R, STARPU_R, STARPU_W }`. Ainda, opcionalmente, podemos definir um nome para este *codelet*, neste caso `soma_bloco`. O Código 4.7 apresenta a construção completa deste *codelet* (assumindo a existência de uma função `func_cpu` com a implementação da funcionalidade da tarefa.

```
struct starpu_codelet codelet_soma =
{
    .cpu_funcs = { func_cpu },
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_W },
    .name = "soma_bloco"
};
```

Código 4.7. (C) Exemplo da declaração de codelet

As implementações das funções dos *codelets* sempre tem a assinatura `void func(void *buffers[], void *args)`. Para acessar os blocos de memória dentro da função é utilizada a macro `STARPU_VECTOR_GET_PTR` (para o nosso caso onde utilizaremos vetores) com o buffer e identificador corretos. Neste caso, como queremos realizar a operação $C = A + B$, criamos três variáveis e utilizamos a macro para acessar os endereços de memória corretos. Após isso realizamos a nossa operação. Após a finalização da tarefa, o StarPU automaticamente irá gerenciar a memória caso necessário (salvar ou mover blocos que foram escritos). Um exemplo desta implementação está presente no Código 4.8.

```
void func_cpu(void *buffers[], void *args)
{
    float *A = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
    float *B = (float *)STARPU_VECTOR_GET_PTR(buffers[1]);
    float *C = (float *)STARPU_VECTOR_GET_PTR(buffers[2]);
    for(int i=0; i < BLOCK_TOTAL_SIZE; i++){
```

```
        C[i] = A[i] + B[i];
    }
}
```

Código 4.8. (C) Exemplo de implementação de tarefa

4.4.4. Descritor de dados (*data handle*)

Para as tarefas utilizarem os blocos de memória, devemos registrá-los no StarPU na forma de `starpu_data_handle_t`. Basicamente, realizamos a alocação convencional de uma variável (utilizando `malloc`, por exemplo), e depois utilizamos a função `starpu_matrix_data_register` para criar um handle. Podemos então utilizar esta área alocada para inicializar estes dados na RAM.

```
float* matrix_a[NUMBER_BLOCKS];
starpu_data_handle_t matrix_a_handle[NUMBER_BLOCKS];
for(int i=0; i<NUMBER_BLOCKS; i++){
    matrix_a[i]=(float*)malloc(WIDTH * WIDTH * sizeof(float));
    starpu_matrix_data_register(&matrix_a_handle[i], STARPU_MAIN_RAM, (
   uintptr_t)matrix_a[i], WIDTH, WIDTH, WIDTH, sizeof(float));
}
```

Código 4.9. (C) Esqueleto de uma aplicação StarPU

Antes da finalização do starpu, com `starpu_shutdown` é necessário desalocar os `data_handles` utilizando a função `starpu_data_unregister()`. No nosso caso, podemos realizar isso com o seguinte bloco de código.

```
for(int i=0; i < NUMBER_BLOCKS; i++){
    starpu_data_unregister(matrix_a_handle[i]);
    starpu_data_unregister(matrix_b_handle[i]);
    starpu_data_unregister(matrix_c_handle[i]);
}
```

Código 4.10. (C) Esqueleto de uma aplicação StarPU

4.4.5. Tarefa (*Task*)

Finalmente, podemos submeter a tarefa com o *codelet* e os blocos de memória corretos. Neste caso, como queremos realizar a operação de soma de duas matrizes A e B e salvar na matriz C, e como as matrizes estão subdivididas em blocos, devemos definir cada tarefa como realizando a operação $bloco_c[i] = bloco_a[i] + bloco_b[i]$. O Código 4.11 apresenta a submissão da tarefas em todas as submatrizes.

```
for(int i=0; i<NUMBER_BLOCKS; i++){
        starpu_task_insert(&codelet_soma, STARPU_R, matrix_a_handle[i],
   STARPU_R, matrix_b_handle[i], STARPU_W, matrix_c_handle[i], 0);
}
```

Código 4.11. (C) Esqueleto de uma aplicação StarPU

4.4.6. Heterogeneidade e implementações em GPU

Para utilizar a heterogeneidade dos recursos o StarPU permite múltiplas implementações do *codelet*. Para isso, devemos definir a variável `cuda_funcs`, para uma função em

CPU que executará uma chamada de função para um recurso CUDA. Neste exemplo, utilizaremos diretamente a biblioteca `cublas`. Ressalta-se que a função a ser definida em `cuda_funcs` deve ser uma função normal em CPU, que chama um *kernel* ou uma função de uma biblioteca em CUDA. Podemos ainda utilizar a variável do *codelet* `where` para obrigar a execução do *codelet* na GPU. O seguinte *codelet* apresenta as modificações.

```
struct starpu_codelet codelet_soma =
{
    .cpu_funcs = { func_cpu },
    .cuda_funcs = { func_gpu },
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_W },
    .name = "soma_bloco",
    .where = STARPU_CUDA
};
```

Código 4.12. (C) Esqueleto de uma aplicação StarPU

Um exemplo de função que invoca a biblioteca `cublas` para realizar a soma de matrizes é dada no código abaixo.

```
#include <cuda_runtime.h>
#include <cublas_v2.h>
#define CHECK_CUBLAS(x) if(x!=CUBLAS_STATUS_SUCCESS){printf("Cublass
   error: %d\n", x);};
cublasHandle_t cublas_mainhandle;

void func_gpu(void *buffers[], void *args)
{
    float alpha_beta = 1;
    float *A = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
    float *B = (float *)STARPU_VECTOR_GET_PTR(buffers[1]);
    float *C = (float *)STARPU_VECTOR_GET_PTR(buffers[2]);
    cublasSetStream(cublas_mainhandle, starpu_cuda_get_local_stream());
    CHECK_CUBLAS(cublasSgeam(cublas_mainhandle,
                CUBLAS_OP_N, CUBLAS_OP_N, WIDTH, WIDTH,
                &alpha_beta, A, WIDTH, &alpha_beta,
                B, WIDTH, C, WIDTH));
    cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

Código 4.13. (C) Esqueleto de uma aplicação StarPU

Um detalhe de implementação é que para utilizar a biblioteca `cublas` devemos inicializar ela utilizando a função `cublasCreate();` no main do nosso programa. A compilação agora do programa deve informar a biblioteca `cublas`.

```
gcc $(pkg-config --cflags starpu-1.3 cuda-10.0) \
    ./soma_matrix_cuda.c \
    $(pkg-config --libs starpu-1.3 cudart-10.0 cublas-10.0)
```

Código 4.14. (bash) Compilar uma aplicação StarPU

4.5. Exemplos de aplicações

Apresentamos nesta seção duas aplicações paralelas baseadas em tarefas com o *runtime* StarPU: a clássica multiplicação de matrizes em sua versão com blocos, seguida de um exemplo de como encontrar o maior valor em um vetor. A versão completa de todos estes códigos está disponível publicamente no *companion*[6].

Multiplicação de matrizes

Para realizar a multiplicação de matrizes podemos utilizar a versão do algoritmo baseada em blocos. Sendo k a largura do bloco, o bloco $C[i][j]$ é por:

$$C[i][j] = SUM(A[k][j] \times B[i][k]) \tag{1}$$

Desta forma, cada tarefa pode computar $C = C + A \times B$. Neste exemplo, duas tarefas que atualizem o mesmo C não podem executar ao mesmo tempo. Criando uma série de dependências. A estrutura da aplicação fica igual a anterior com as seguintes diferenças. O código do *codelet* fica conforme listado em 4.15, sendo que a listagem do Código 4.16 apresenta a inserção das tarefas na *thread* sequencial do programa principal (*main*).

```
void block_mm_cpu (void *buffers[], void *args)
{
    float *A = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
    float *B = (float *)STARPU_VECTOR_GET_PTR(buffers[1]);
    float *C = (float *)STARPU_VECTOR_GET_PTR(buffers[2]);

    for(int i=0; i < WIDTH; i++){
       for(int j=0; j < WIDTH; j++){
          for(int k=0; k < WIDTH; k++){
             C[j * WIDTH + i] += A[j * WIDTH + k] * B[k * WIDTH + i];
          }
       }
    }
}

struct starpu_codelet codelet_multi =
{
    .cpu_funcs = { block_mm_cpu },
    .cpu_funcs_name = { "block_mm_cpu" },
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_W },
    .where = STARPU_CPU,
    .name = "multiplica_por_bloco"
};
```

Código 4.15. (C) Codelet de uma multiplicação de matrizes em blocos.

```
for(int i=0; i<NUMBER_BLOCKS_WIDTH; i++){
    for(int j=0; j<NUMBER_BLOCKS_WIDTH; j++){
        for(int k=0; k<NUMBER_BLOCKS_WIDTH; k++){
```

[6]<https://gitlab.com/lnesi/minicurso-starpu-erad-2020>

```
            starpu_task_insert(&codelet_multi,
                STARPU_R, matrix_a_handle[j * NUMBER_BLOCKS_WIDTH + k],
                STARPU_R, matrix_b_handle[k * NUMBER_BLOCKS_WIDTH + i],
                STARPU_W, matrix_c_handle[j * NUMBER_BLOCKS_WIDTH + i],
                0);
        }
    }
}
```

Código 4.16. (C) Esqueleto de uma aplicação StarPU

Encontrar o maior valor em um vetor

Este problema esta descrito na Seção 4.2. O grande diferencial deles é definir os dados de tal forma a criar corretamente as dependências. Pode-se utilizar algumas abordagens diferentes para criar e particionar os `data_handles`. Iremos assumir que cada tarefa realiza a operação em uma lista de tamanho três. Podemos então realizar a alocação de `data_handles` com tamanho três. Entretanto, as respostas das tarefas são apenas um valor. Caso utilizássemos estes `data_handles` de tamanho três como resultado para múltiplas tarefas, um problema de dependência iria existir. Veja que a tarefa teria permissão de leitura/escrita sobre os dados, então se duas tarefas escrevem sobre os mesmos dados uma dependência existirá para garantir a coerência entre elas. Entretanto, sabemos que cada tarefa só salvaria seu resultado em uma posição específica. Para descrever este comportamento em StarPU, devemos dividir os `data_handles` em sub-`data_handles`. Para isso, criamos um filtro informando como será esta divisão. Como queremos dividir um vetor de tamanho três em blocos de tamanho um, utilizamos o filtro a seguir.

```
struct starpu_data_filter filter_resposta =
{
    .filter_func = starpu_vector_filter_block,
    .nchildren = 3
};
```

Código 4.17. (C) Filtro para divisão de um data_handle.

Devemos então informar a utilização deste filtro para o StarPU com a função `starpu_data_partition()`. A submissão de tarefas segue normalmente com um `data_handle` resultante da função `starpu_data_get_sub_data()` como demonstrado no código a seguir.

```
starpu_data_partition(handles_resposta[reposta_id], &filter_resposta);

starpu_data_handle_t sub_resposta = starpu_data_get_sub_data(
    handles_resposta[reposta_id], 1, i%DIVISAO);

starpu_task_insert(&mycodelet,
                   STARPU_R, handles_entrada[i],
                   STARPU_W, sub_resposta,
                   0);
```

Código 4.18. (C) Dividindo um data_handle em múltiplos e utilizando os sub-data_handles na submissão de tarefas.

4.6. Analisando aplicações StarPU

Esta Seção apresenta métodos para analisar as aplicações StarPU. Primeiramente apresentamos algumas métricas especializadas para avaliar as aplicações. Segundo, iremos apresentar porque estas métricas são insuficientes, demonstrando uma solução com uso do *workflow* StarVZ.

4.6.1. Métricas

As métricas fundamentais para análise de desempenho de aplicações paralelas são o *makespan*, o *speed-up* e a eficiência. Entretanto, em cenários como o apresentado neste minicurso, que incluem recursos de processamento heterogêneos e escalonamento dinâmico, tais métricas se mostram insuficientes (PINTO et al., 2016; PINTO et al., 2018b; PINTO et al., 2018a; NESI et al., 2019).

Na seção a seguir, apresentaremos um breve exemplo de análise de desempenho de uma aplicação implementada com tarefas executando sobre o StarPU. Esta análise será conduzida com auxilio do *framework* StarVZ[7]. Embora a funcionalidade base do StarVZ seja a visualização de rastros, diversos recursos adicionais são disponibilizados para auxiliar o analista a identificar e compreender a origem de gargalos ou anomalias. Entre estes recursos podemos citar a exibição de arestas de dependências entre as tarefas, cálculo de estimativas para o tempo de execução, computação da ociosidade dos trabalhadores e identificação de tarefas com duração anômala. Além disso, por meio de painéis gráficos complementares, podemos adicionar outras informações relevantes como a quantidade de tarefas submetidas, quantidade de tarefas prontas para execução em cada instante e a progressão da computação ao longo do tempo.

4.6.2. Visualizações e StarVZ

A criação de visualizações com o *framework* StarVZ é composta de duas etapas. A etapa inicial consiste em um pré-processamento dos rastros gerados no final da execução da aplicação com StarPU. A segunda etapa, que consiste na análise propriamente dita, é realizada com auxílio do pacote R do StarVZ. Para baixar o *framework* StarVZ e instalar o pacote R, é necessário executar os comandos abaixo.

```bash
git clone https://github.com/schnorr/starvz.git
apt install -y r-base libxml2-dev libssl-dev \
    libcurl4-openssl-dev libgit2-dev libboost-dev
./starvz/R/install.R
# pajeng
apt install -y git cmake build-essential \
    libboost-dev asciidoc flex bison
git clone git://github.com/schnorr/pajeng.git
mkdir -p pajeng/b ; cd pajeng/b
cmake ..
make
```

Código 4.19. (bash) *Download*, configuração e instalação do StarVZ e suas dependências

Para obter rastros de uma execução com StarPU é necessário definir a variável de

[7] <https://github.com/schnorr/starvz>

ambiente STARPU_GENERATE_TRACE. Além disso, é necessário que o StarPU tenha sido compilado com suporte ao FxT (veja Código 4.3). A variável STARPU_FXT_PREFIX permite escolher o local onde serão disponibilizados os arquivos contendo os rastros.

```
gcc $(pkg-config --cflags starpu-1.3) ./exemplos/mult_matrix.c \
    $(pkg-config --libs starpu-1.3) -o mult_matrix
STARPU_FXT_PREFIX=$PWD/ STARPU_GENERATE_TRACE=1 ./mult_matrix
```

Código 4.20. (bash) Compilação e Execução de Aplicação `mult_matrix` da seção anterior com rastreamento habilitado

Após a execução serão gerados arquivos com os rastros da aplicação (um ou mais arquivos com nome prof_file_*). O pré-processamento destes rastros com StarVZ se dá pela forma abaixo.

```
export PATH=starvz/:$PATH
export PATH=pajeng/b:$PATH
export PATH=$STARPU_HOME/bin:$PATH
./starvz/src/phase1-workflow.sh ./ ""
```

Código 4.21. (bash) Pré-processamento dos rastros com StarVZ

Em seguida, os rastros pré-processados devem ser carregados na linguagem R por meio do pacote do StarVZ, assumindo que os dados encontram-se no diretório corrente. As visualizações produzidas com StarVZ são altamente customizáveis. Dessa forma, devemos utilizar um conjunto básico de parâmetros que permite ativar ou desativar painéis conforme desejado ou conforme as informações disponíveis nos rastros coletados. O comando abaixo carrega uma configuração de base que já está inclusa no pacote StarVZ baixado anteriormente. Em seguida, devemos escolher quais dados queremos adicionar ou remover da visualização. Em seguida, podemos chamar a função the_master_function() que cria o gráfico. O exemplo completo em R está no Código 4.22 e um exemplo de resultado está presente na Figura 4.4.

```
library(starvz)
dtrace <- the_fast_reader_function("./")
pajer <- config::get(file = "starvz/full_config.yaml")

# desativa visualizacao dos estados internos do StarPU
pajer$starpu$active = TRUE
# habilita curvas de tarefas submetidas e ativas
pajer$submitted$active = TRUE
pajer$st$abe$active = TRUE

# seleciona tarefas para desenhar as dependencias
pajer$st$tasks$active = TRUE
pajer$st$tasks$levels = 5
pajer$st$tasks$list = dtrace$State %>% group_by(X, Y) %>%
     filter(End==max(End), !is.na(JobId)) %>% .$JobId
the_master_function(dtrace)
```

Código 4.22. (R) Geração de visualizações com StarVZ

A visualização presente na Figura 4.4 é composta de quatro painéis. O painel superior apresenta a visualização espaço-tempo da aplicação, onde o eixo vertical *y* representa os recursos de processamentos que, neste caso representam os *cores* do processador

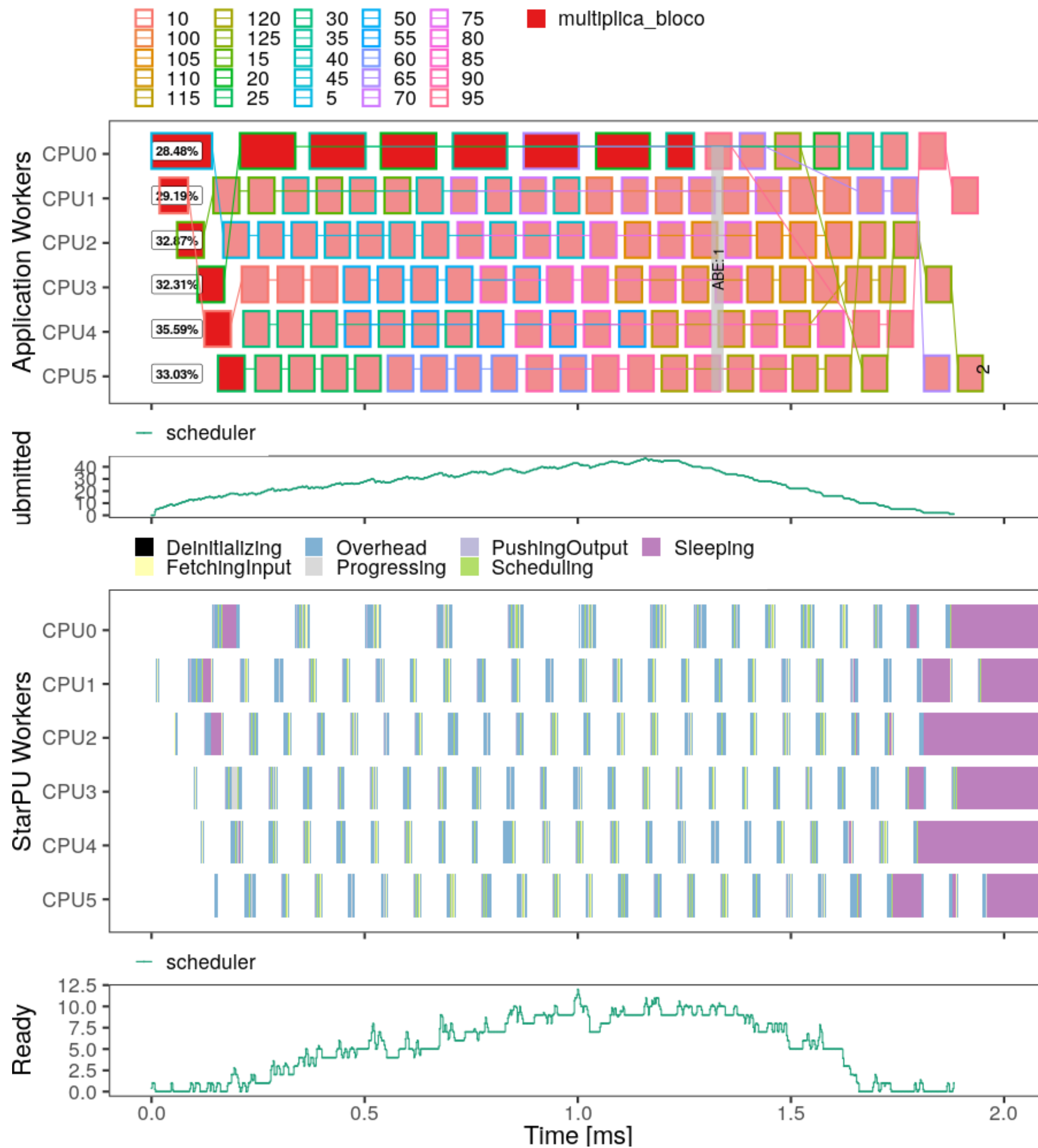


Figura 4.4. Exemplo de visualização produzida com o *framework* StarVZ para a aplicação `mult_matrix`. Fonte: Autores.

e o eixo horizontal x representa o tempo de execução (duração das tarefas). O eixo x deste e dos demais painéis é mantido sincronizado de forma a facilitar a correlação entre os diferentes painéis. O segundo painel apresenta o número de tarefas submetidas ao longo do tempo (eixo y). O terceiro painel apresenta a visualização espaço-tempo do *runtime* StarPU de forma análoga ao primeiro painel. O último painel apresenta o número de tarefas prontas para execução ao longo do tempo (eixo y).

A análise do primeiro painel permite observar que a ociosidade dos trabalhadores é consideravelmente elevada, oscilando em torno de 30%. Inicialmente, poderíamos supor que a ociosidade decorre da falta de paralelismo. Entretanto, a correlação com o

último painel mostra que, ao menos em parte do tempo de execução, o número de tarefas prontas para execução se mantém elevado o suficiente para ocupar todos os trabalhadores. Descartada esta primeira suposição, podemos partir para a análise do diagrama espaço-tempo do *runtime*. Vemos que os períodos de ociosidade no processamento das tarefas da aplicação coincidem com a execução de atividades de gerenciamento do próprio StarPU. Em geral, o *overhead* gerado pelo StarPU costuma ser pouco significante no tempo total de execução da aplicação, porém, neste exemplo, a curta duração das tarefas (em média 0.059 ms) faz com que ele se torne perceptível. A taxa de ociosidade elevada também pode explicar a lacuna entre o limite teórico calculado pelo StarVZ (barra vertical cinza em 1,33 ms) e o *makespan* observado.

O diagrama espaço-tempo da aplicação mostra também que a duração de algumas tarefas, em especial aquelas executadas pelo trabalhador CPU0, foi considerada como anômala (*outlier*) pelo StarVZ. Estas tarefas são representadas pela cor vermelha em destaque em oposição ao vermelho esmaecido usado para colorir as demais tarefas de mesmo tipo. Ao comparar este diagrama com o gráfico de tarefas submetidas (segundo painel), podemos perceber que a ocorrência destes *outliers* na CPU0 coincide com o crescimento da curva de tarefas submetidas. Após a estagnação do crescimento do número de tarefas submetidas (em torno do instante 1,25 ms), a duração média das tarefas executadas pela CPU0 se aproxima daquelas executadas nos demais trabalhadores. Isto nos permite supor que a *thread* principal da aplicação (`main`) está alocada no mesmo *core* físico do trabalhador CPU0. A solução para tal conflito reside no uso da variável de ambiente `STARPU_MAIN_THREAD_BIND`, dedicando um *core* específico para a submissão das tarefas.

4.7. Conclusão

Este minicurso tem por objetivo apresentar como se programar aplicações paralelas utilizando o paradigma de programação orientado a tarefas. Este paradigma de programação tem se tornado cada vez mais útil tendo em vista que com ele a complexidade da programação de recursos heterogêneos se torna menor. O paradigma orientado a tarefas permite algumas facilidades nesta programação por que transfere para um ambiente de execução (*runtime*) muitas responsabilidades que seriam anteriormente realizadas pelo programador nos paradigmas tradicionais. Espera-se que o texto deste minicurso traga os conceitos básicos iniciais para se permitir a programação utilizando o paradigma orientado a tarefas. Para ir além do que é apresentado neste minicurso, sugerirmos consultar a vasta documentação, tutoriais e minicursos disponíveis no site do StarPU: <http://starpu.gforge.inria.fr/tutorials/>.

Agradecimentos

Este trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001, do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), e dos projetos: FAPERGS ReDaS (19/711-6), MultiGPU (16/354-8) e GreenCloud (16/488-9), do projeto CNPq 447311/2014-0, do projeto CAPES/Brafitec 182/15 e CAPES/Cofecub 899/18, e com apoio do projeto Petrobras (2018/00263-5).

Referências

AGULLO, E. et al. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Tr. Math. Softw.*, ACM, New York, NY, USA, v. 43, n. 2, 2016. ISSN 0098-3500.

AUGONNET, C. *Scheduling Tasks over Multicore machines enhanced with acelerators: a Runtime System's Perspective*. Tese (Theses) — Université Bordeaux 1, dez. 2011. Disponível em: <https://tel.archives-ouvertes.fr/tel-00777154>.

AUGONNET, C. et al. Starpu-mpi: Task programming over clusters of machines enhanced with accelerators. In: TRÄFF, J. L.; BENKNER, S.; DONGARRA, J. J. (Ed.). *Recent Advances in the Message Passing Interface*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 298–299. ISBN 978-3-642-33518-1.

AUGONNET, C. et al. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Conc. Comp.: Pract. Exp., SI:EuroPar 2009*, John Wiley and Sons, Ltd., v. 23, 2011.

BOSILCA, G. et al. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, Elsevier, v. 38, n. 1-2, p. 37–51, 2012.

BRIAT, J. et al. Athapascan runtime: Efficiency for irregular problems. In: SPRINGER. *European Conference on Parallel Processing*. [S.l.], 1997. p. 591–600.

DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, IEEE, v. 5, n. 1, p. 46–55, 1998.

DANJEAN, V.; NAMYST, R.; WACRENIER, P.-A. An efficient multi-level trace toolkit for multi-threaded applications. In: SPRINGER. *European Conference on Parallel Processing*. [S.l.], 2005. p. 166–175.

Dongarra, J. et al. With extreme computing, the rules have changed. *Computing in Science Engineering*, v. 19, n. 3, p. 52–62, May 2017. ISSN 1521-9615.

DURAN, A. et al. Ompss: a proposal for programming heterogeneous multi-core architectures. *Paral. Proces. Letters*, World Scientific, v. 21, n. 02, 2011.

GAUTIER, T. et al. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: *IEEE Intl. Symposium on Parallel and Distributed Processing*. [S.l.: s.n.], 2013. ISSN 1530-2075.

GROPP, W.; LUSK, E.; SKJELLUM, A. *Using MPI: portable parallel programming with the message-passing interface*. [S.l.]: MIT press, 1999. v. 1.

NESI, L. L. et al. Visual performance analysis of memory behavior in a task-based runtime on hybrid platforms. In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019. p. 142–151. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/CCGRID.2019.00025>.

OpenMP. *OpenMP Application Program Interface Version 4*. OpenMP Architecture Review Board, 2013. Disponível em: <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>.

PINTO, V. G. et al. Detecção de Anomalias de Desempenho em Aplicações de Alto Desempenho baseadas em Tarefas em Clusters Híbridos. In: *WPerformance 2018 - 17º Workshop em Desempenho de Sistemas Computacionais e de Comunicação*. Natal, Brazil: [s.n.], 2018. p. 1–14. Disponível em: <https://hal.inria.fr/hal-01842038>.

PINTO, V. G. et al. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. *Concurrency and Computation: Practice and Experience*, v. 30, n. 18, p. e4472, 2018. E4472 cpe.4472. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4472>.

PINTO, V. G. et al. Analyzing dynamic task-based applications on hybrid platforms: An agile scripting approach. In: *Third Workshop on Visual Performance Analysis, VPA@SC 2016, Salt Lake, UT, USA, November 18, 2016*. [s.n.], 2016. p. 17–24. Held in conjunction with SC16. Disponível em: <https://doi.org/10.1109/VPA.2016.008>.

SCHNORR, L.; STEIN, B. de O.; KERGOMMEAUX, J. C. de. Paje trace file format, version 1.2.5. *Laboratoire d'Informatique de Grenoble, France, Technical Report*, 2013.

THIBAULT, S. *On Runtime Systems for Task-based Programming on Heterogeneous Platforms*. Tese (Habilitation à diriger des recherches) — Université de Bordeaux, dez. 2018. Disponível em: <https://hal.inria.fr/tel-01959127>.

TOPCUOGLU, H.; HARIRI, S.; WU, M.-y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, IEEE, v. 13, n. 3, p. 260–274, 2002.

Minicursos da XX Escola Regional de Alto Desempenho da Região Sul. DOI: 10.5753/sbc.4400.9

Capítulo 5

Programando Aplicações com Diretivas Paralelas

Natiele Lucca, Claudio Schepke
Universidade Federal do Pampa
Alegrete, Brasil

Resumo

Uma das motivações para o desenvolvimento de programas paralelos é acelerar aplicações científicas. Aplicações deste tipo geralmente demandam de um grande tempo de computação para uma versão com um único fluxo de execução, o que pode levar minutos, horas ou até mesmo dias, dependendo do tamanho do domínio ou resolução do problema adotado. Uma das maneiras de gerar paralelismo a partir de um código é inserir diretivas (pragmas), os quais geram fluxos concorrentes de código, que podem ser executados tanto em arquiteturas multi-core como many-core. Neste sentido, este minicurso tem como objetivo apresentar técnicas de exploração de paralelismo em diferentes trechos de código para um conjunto de aplicações científicas usando as interfaces de programação OpenMP e OpenACC. Serão demonstrados exemplos reais do impacto do uso de pragmas no desempenho de códigos, incluindo situações em que a granularidade impede que se obtenha a aceleração do programa.

5.1. Introdução

Este Capítulo aborda a prática de programação com diretivas paralelos. Diretivas de pré-compilação possibilitam a geração de código específico e automatizado. Inicialmente serão vistos alguns conceitos sobre as arquiteturas de programação Multi-core e GPUs. Na sequência também serão introduzidas as interfaces de programação OpenMP e OpenACC, apresentando a estrutura e organização de suas diretivas. Por fim, serão mostrados os códigos-fonte de 2 aplicações científicas, destacando o objetivo de cada uma, bem como as funcionalidades existentes. A partir disso, uma avaliação do custo sequencial de cada função ou rotina de código será então realizada. Trechos de código previamente selecionados serão indicados para que abordagens de paralelização sejam utilizadas, avaliando o impacto na aceleração da aplicação. Com isso, o objetivo é testar e validar diretivas que possam reduzir o tempo total de execução da aplicação.

5.2. Introdução às arquiteturas multi-Core e GPU

O crescente desenvolvimento computacional previsto pela Lei de Moore (MOORE, 1965) possibilitou executar *softwares* cada vez mais rápidos, incluindo melhores funcionalidades e se tornando mais úteis para os usuários. Estes cada vez mais requeriam desta tecnologia, gerando uma fase positiva para a indústria de computadores, que se responsabilizava por manter o contínuo desenvolvimento, praticamente dobrando o número de transistores por processador a aproximadamente cada dois anos ao mesmo tempo em que mantinha os custos.

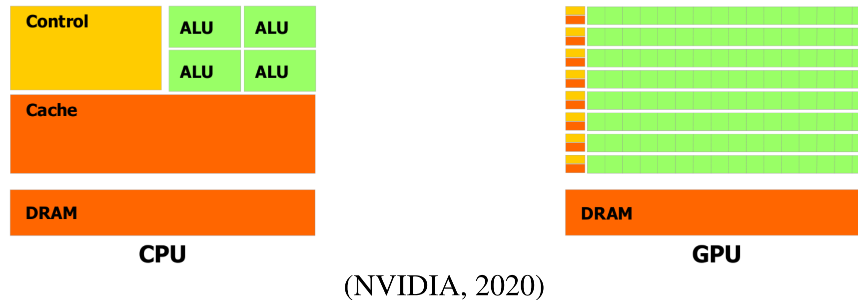
O fato é que essa tendência diminuiu a partir do ano de 2003 devido ao consumo elevado de energia e problemas com a dissipação de calor nos processadores (KIRK; WEN-MEI, 2016). Estes problemas se tornaram fatores limitantes para o aumento da velocidade de *clock* dos processadores, o que levou os fabricantes de processadores a buscarem por novas estratégias para se obter melhor desempenho, tendo um maior cuidado com a eficiência energética e a dissipação de calor. A solução para se obter melhor desempenho considerando os obstáculos mencionados foi a de começar a produzir processadores compostos por dois ou mais *cores*, que são processadores mais simples e com menor frequência de *clock*. Isto fez com que surgissem os processadores *multi-core*, que são processadores que contém múltiplas unidades de processamento, permitindo assim, a execução de instruções de forma paralela. Este fato, de grande importância, foi nomeado por Sutter e Larus (2005) como a revolução da concorrência, para a qual a indústria de *software* deve estar preparada, sendo capaz de produzir aplicações que utilizem os recursos oferecidos pelas arquiteturas paralelas de forma eficiente.

Em contraste com os processadores *multi-core*, existem os chamados processadores *manycore*, também desenvolvidos pensando na execução paralela de instruções. Apesar de que estas duas arquiteturas tenham sido desenvolvidas com o mesmo objetivo, existem diferenças na forma com que são organizadas e as estratégias que utilizam para se obter poder de computação. Enquanto as arquiteturas *multi-core* foram desenvolvidas para manter a velocidade de programas sequenciais enquanto faziam a transição para múltiplos processadores, os dispositivos de arquitetura *manycore* são voltados para a execução de dezenas, centenas ou até milhares de instruções simultâneas (KIRK; WEN-MEI, 2016).

Na Figura 5.2 pode-se perceber os elementos que representam as diferenças entre as duas arquiteturas. À esquerda tem-se a representação de uma CPU *multi-core* e à direita de uma GPU *manycore*.

A arquitetura *multi-core* é otimizada para a performance de código sequencial, contendo uma lógica de controle sofisticada que permite a execução em paralelo de instruções de uma *thread*, somado a existência de uma grande memória *cache* para reduzir a latência de acesso aos dados e instruções necessários por uma aplicação, além de cada ULA também ser otimizada para reduzir a latência. Já as arquiteturas *manycore* possuem um *hardware* especificamente projetado para suportar um grande número de *threads*, com memórias *cache* de pequeno porte, visando uma redução ao acesso à memória principal nos casos em que múltiplas *threads* acessam os mesmo dados, maximizando, desta forma, o espaço dedicado para as ULAs, que também são mais simplificadas.

Figura 5.1. Representação das arquiteturas *multi-core* e *manycore*



(NVIDIA, 2020)

Com isto pode-se perceber que cada arquitetura é especializada em um tipo de tarefa diferente, sendo os processadores *multi-core* voltados tanto para a performance paralela quanto sequencial, dando ênfase para o desempenho individual das *threads*, porém limitado a um número menor de *cores* quando comparado com um processador *manycore*. Arquiteturas *manycore* são totalmente focadas e otimizadas para terem níveis altíssimos de paralelismo, tendo muitos *cores* mais simples e com menos performance individual. Uma arquitetura heterogênea é aquela que combina as duas arquiteturas, aproveitando o melhor das duas estratégias para alcançar máxima eficiência computacional.

Para alcançar este melhor desempenho oferecido pelas arquiteturas paralelas, é necessário expressar o paralelismo nos programas através da programação paralela. Este paradigma de programação permite ao programador especificar as áreas de código que devem ser executadas concorrentemente entre núcleos de um dispositivo paralelo, garantindo assim uma utilização mais eficiente do poder computacional proporcionado por estes dispositivos. Dada a importância do desenvolvimento de aplicações paralelas, atualmente existem diversas *Application Programming Interface* (APIs) voltadas para este fim, a maioria delas já oferece suporte para a programação de GPUs ou foram projetadas para este fim, como é o caso das APIs CUDA, OpenACC e OpenCL.

5.3. A Interface de programação OpenMP

OpenMP é uma API para programação paralela de memória compartilhada e multiplataforma disponível em C/C++ e Fortran (OPENMP, 2020). A API é fundamentada no modelo de execução *fork-join*. Esse modelo possui uma *thread* mestre que inicia a execução e gera *threads* de trabalho para executar as tarefas em paralelo (CHAPMAN; MEHROTRA; ZIMA, 1998). O OpenMP aplica o modelo de execução *fork-join* em segmentos do código que são informados pelo programador (ou usuário). Dessa forma um código sequencial é executado pela *thread* mestre até um bloco ou área de execução paralela.

O inicio da área paralela é demarcado por uma diretiva OpenMP que é responsável por sinalizar que as *threads* de trabalho devem ser lançadas (*fork*). Todo o código seguinte é executado em paralelo pelas *threads* até o fim da área paralela que pode ser demarcado explicitamente como o símbolo de } ou implícito, como por exemplo, em um laço de repetição `for` onde o fim do laço de repetição também é o fim da área paralela. O fim da área paralela implica no encerramento das *threads* de trabalho, sincronização (*fork*) e retorno da *thread* mestre para a execução.

A API OpenMP possui um conjunto de diretivas de compilação, funções e variáveis de ambiente para a programação paralela (TORELLI; BRUNO, 2004). Uma diretiva é precedida obrigatoriamente por `#pragma omp` e seguida por `[atributos]`, sendo que os atributos são opcionais. Seguem algumas diretivas que compõem a API OpenMP (OPENMP, 2020).

- `parallel`: Essa diretiva descreve que a uma área do código será executada por *n threads*, sendo *n* o número de *threads* especificados por um atributo e/ou variável de ambiente.
- `for`: Essa diretiva especifica que as iterações do laço de repetição serão executadas em paralelo por *n threads*.
- `parallel for`: Especifica a construção de um laço paralelo, sendo que o laço será executado por *n threads*.
- `simd`: Essa diretiva descreve que algumas iterações de um laço de repetição podem ser executadas simultaneamente por unidades vetoriais.
- `for simd`: Essa diretiva especifica que um laço pode ser dividido em *n threads* que executam algumas iterações simultaneamente por unidades vetoriais.

Na sequência são apresentados alguns atributos da API OpenMP (OPENMP, 2020). Para todos os casos, `lista` representa uma ou mais variáveis.

- `private (lista)`: Esse atributo informa que o bloco paralelo possui variáveis privadas para cada uma das *n threads*. As variáveis do bloco que não são informadas na lista são públicas.
- `shared (lista)`: O atributo especifica que as variáveis são públicas e compartilhadas entre as *n threads*.
- `num_threads(int)`: Esse atributo determina o número *n* de *threads* utilizadas no bloco paralelo. O valor de *n* é válido apenas para o bloco em que foi definido.
- `reduction (operador: lista)`: A redução é utilizada para executar cálculos em paralelos. Cada *thread* tem seu valor parcial. Ao final da região paralela o valor final da variável é atualizado com os cálculos parciais. O operador pode ser, por exemplo $+$, $-$, $*$, *max* e *min*.
- `nowait`: Uma diretiva OpenMP possui uma barreira implícita ao seu fim, com o objetivo de garantir a sincronização. Entretanto a diretiva `nowait` omite a existência dessa barreira. Dessa forma, as *threads* não ficam em espera até que as demais também terminem o trabalho.

As variáveis de ambiente do OpenMP especificam características que afetam a execução dos programas. Seguem algumas variáveis (OPENMP, 2020):

- `OMP_NUM_THREADS`: Especifica o número *n* de *threads* utilizados nos blocos paralelos do algoritmo.
- `OMP_THREAD_LIMIT`: Descreve o número máximo de *threads*.
- `OMP_NESTED`: Permite ativar ou desativar o paralelismo aninhado.
- `OMP_STACKSIZE`: Especifica o tamanho da pilha para as *threads*.

5.4. A Interface de programação OpenACC

O OpenACC é uma API que contém um conjunto de diretivas de compilação para GPUs, similar às fornecidas por OpenMP para CPU (CHANDRASEKARAN; JUCKELAND, 2017). Enquanto uma área paralela no OpenMP é executada por *threads* de trabalho localizadas na CPU o OpenACC especifica instruções que criam *threads* localizadas na GPU (OPENACC, 2019).

Uma diretiva OpenACC em C/C++ possui o seguinte formato (OPENACC, 2019):

```
#pragma acc diretiva [clausulas [,] clausulas]
```

Em Fortran a diretiva possui o seguinte formato (OPENACC, 2019):

```
!$acc diretiva [clausulas [,] clausulas]
```

O OpenACC possui três níveis de paralelismo, são eles: *gang*, *worker* e *vector*. A *gang* possui um ou mais *workers*, sendo que nos *workers* podem ocorrer operações vetoriais (*vector*).

Seguem algumas diretivas que compõem a API OpenACC (OPENACC, 2019).

- `parallel`: Essa diretiva descreve um bloco em paralelo executado por *n threads* disponíveis.
- `kernels`: Especifica que o compilador dividirá o bloco paralelo em *kernels*.
- `loop`: Paraleliza o(s) loop(s) aninhados imediatamente após a diretiva.
- `serial`: Essa diretiva especifica um bloco de código que será executado sequencialmente.
- `data`: Define um bloco no qual os dados são acessíveis pela GPU.

Seguem algumas cláusulas que compõem a API OpenACC (OPENACC, 2019). Para todos os casos, *lista* representa uma ou mais variáveis.

- `copy (lista)`: Essa cláusula copia os dados do *host* para a GPU e da GPU para o *host*.
- `copyin (lista)`: Essa cláusula copia os dados do *host* para a GPU.
- `copyout (lista)`: Essa cláusula copia os dados da GPU para o *host*.

- `present (lista)`: Essa cláusula informa que os dados utilizados no bloco paralelo já foram copiados para a memória da GPU.

A API OpenACC possui variáveis de ambiente que especificam alterações na execução dos programas. Para as ocorrências, *type* representa o tipo do dispositivo e *size* o tamanho de memória que será alocado. Seguem algumas variáveis (OPENACC, 2019):

- `acc_get_num_devices (type)`: Retorna o número de dispositivos do tipo especificado.
- `acc_set_device_type (type)`: Define o tipo de dispositivo utilizado.
- `acc_get_device_type()`: Retorna o tipo de dispositivo que está sendo usado pela *thread*.
- `acc_wait_all()`: Aguarda até que todas as atividades assíncronas sejam concluídas.
- `acc_malloc(size)`: Retorna o endereço da memória alocada no dispositivo.
- `acc_is_present`: Testa se os dados do *host* já foram copiados para a GPU.

5.5. Exemplos de aplicações científicas

Duas aplicações científicas foram consideradas para a inserção de diretivas paralelas. A primeira aplicação engloba o uso de algoritmos bio-inspirados para a otimização de problemas. A segunda aplicação realiza a simulação numérica de uma câmara de combustão usando as Equações de Navier-Stokes.

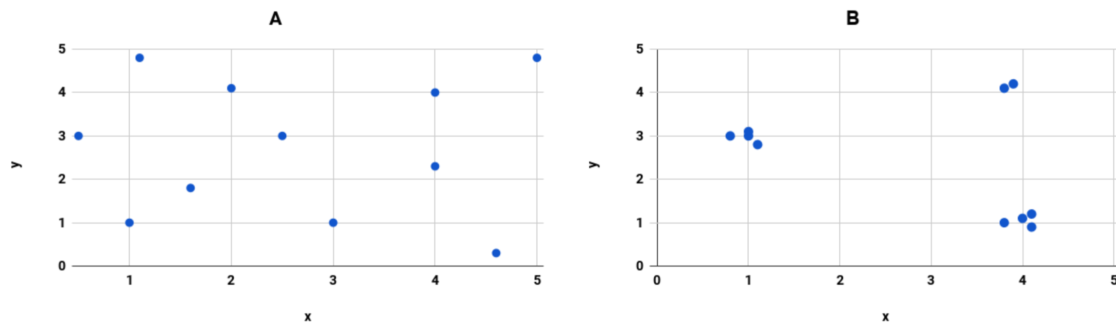
5.5.1. Algoritmos bio-inspirados

Problemas reais de engenharia, ciência e economia por exemplo, não podem ser resolvidos de forma exata e sequencial devido ao elevado tempo de computação para encontrar a solução ótima (IGNÁCIO; FERREIRA, 2002). A capacidade de processamento de um computador não é suficiente para realizar o esforço matemático exigido para encontrar a solução desses problemas. Para isso é necessário que os algoritmos utilizem abordagens que otimizem a execução das tarefas, assim podem solucionar esses problemas de forma eficiente (NIEVERGELT et al., 1995).

A computação natural é uma estratégia de otimização que aplica conceitos da biologia na solução de problemas complexos (CASTRO et al., 2004). Esses problemas possuem um grande número de variáveis ou soluções potenciais, sendo que nem sempre é possível garantir que uma solução encontrada pelo algoritmo seja ótima (CASTRO, 2007).

São exemplos de problemas complexos, os clássicos caixeiro viajante e roteamento de veículos. Outras aplicações são: o trabalho de Lin e Lucas (2015) que apresenta um modelo de evacuação de aeronaves sob emergência com emoções para simular o efeito de passageiros com medo ou pânico; a pesquisa de Diciolla et al. (2015) que classifica e

Figura 5.2. Malha Aleatória x Malha Convergindo



gera uma previsão para pacientes com doença renal terminal (DRC); e o trabalho de Carvalho et al. (2016) que apresenta um algoritmo bio-inspirado para controlar o tráfego em sistemas de elevadores.

Os algoritmos bio-inspirados aplicam técnicas de inteligência de enxames ou inteligência de colônias, ou ainda inteligência coletiva, que simulam o comportamento coletivo de sistemas auto-organizados, distribuídos, autônomos, flexíveis e dinâmicos (SERAPIAO, 2009). Um enxame ou colônia é uma população de elementos que interagem e são capazes de otimizar um objetivo global através da busca colaborativa em um espaço seguindo regras específicas.(KENNEDY; EBERHART, 2001).

A Figura 5.2-*A* mostra uma malha aleatória e a Figura 5.2-*B* uma malha convergindo. A primeira malha representa a inicialização do algoritmo onde os indivíduos ou agentes são dispostos de forma aleatória pelo espaço de busca. Já a malha convergindo expressa a solução quando os indivíduos estão aglomerados em um ou mais grupos que representam as melhores soluções encontradas.

Nesse contexto, a Computação natural ou computação bio-inspirada é o processo de extrair ideias da natureza para desenvolver sistemas computacionais. Os algoritmos inspirados na natureza tem a capacidade de descrever e resolver problemas complexos. Portanto, a computação natural pode ser definida como um campo de pesquisa que, baseado ou inspirado na natureza, permite o desenvolvimento de novas ferramentas para solução de problemas, através da síntese de padrões e de comportamentos (CASTRO, 2007).

A computação inspirada na natureza é subdivida em algoritmos inteligentes que incluem redes neurais artificiais, computação evolutiva, inteligência de enxame, sistemas imunológicos artificiais e outros (ENGELBRECHT, 2007). A inteligência de enxame simula o comportamento de agentes diante da colônia. Esses são inteligentes e autônomos não só realizam tarefas, mas tem a capacidade de tomar decisões (KENNEDY; EBERHART, 2001). São exemplos ACO, ABC, BCO e PSO. A computação evolutiva é baseada na evolução natural das espécies, onde a população se reproduz transferindo e combinando aos outros indivíduos suas características genéticas (BANZHAF et al., 1998). São exemplos de algoritmos dessa área da computação bio-inspirada o GA, DE, SA, ESA

e ES. As redes neurais artificiais simulam a capacidade cognitiva do ser humano, através dos neurônios que interconectados permitem a troca de informação (RAUBER, 2005).

No minicurso é abordado o algoritmo de inteligência de enxame *Particle Swarm Optimization* (Enxame de Partículas - PSO).

5.5.2. Simulação de uma câmara de combustão

Um processo de combustão ocorre quando um material combustível reage com um material reagente. O uso de modelos para a investigação paramétrica de diferentes condições de fluxo relacionadas a problemas de engenharia aeroespacial permite análises rápidas e confiáveis. A camada de mistura compressível serve como um modelo para a análise de problemas de propulsão considerando a passagem de ar em alta velocidade, como a mistura de reagentes em uma câmara de combustão ou a geração de ruído nos bicos de exaustão. Em ambos os casos, duas correntes paralelas a velocidades diferentes podem ser compostas por diferentes espécies químicas ou com gradientes de alta temperatura. Nesses casos, a variação das propriedades pode resultar em diferenças significativas nas características de estabilidade do fluxo, como taxas de crescimento e topologia do fluxo (SILVA et al., 2017).

Para simular uma camada de mistura, tem-se a implementação de uma aplicação. Essa implementação pode ser usada para otimizar o tamanho da dimensão da camada mista e decidir o melhor combustível e condição para um cenário específico. O aplicativo discretiza a equação completa de Navier-Stokes para uma representação bidimensional. A solução numérica considera o método Runge-Kutta usando um esquema de 4ª ordem para o tempo e um esquema de 6ª ordem para o espaço.

No entanto, esse tipo de aplicativo exige muito tempo de execução para qualquer simulação simples. Para reduzir a execução por um tempo aceitável, pode-se explorar a concorrência das instruções nas arquiteturas Multi-Core e GPU. Essas arquiteturas tradicionalmente tem sido adotadas para o desenvolvimento de aplicações numéricas.

A Figura 5.3 apresenta uma relação sistemática entre as funções que compõem a aplicação e suas localizações nos arquivos. O código inicia com o início da execução das chamadas disponíveis no arquivo euler.f90. O primeiro passo a ser feito para iniciar a paralelização é identificar as funções que demandam mais tempo de processamento. Para uma aplicação sequencial, é possível utilizar a ferramenta Gprof (GRAHAM; KESSLER; MCKUSICK, 1982).

5.6. Avaliando a performance de trechos de código

Um código possui trechos que não são paralelizáveis. Dessa forma, é necessário um estudo prévio que realize uma análise e identifique a(s) área(s) que podem ser paralelizadas.

A dependência entre os dados é um fator importante na paralelização de aplicações, uma vez que, pode gerar resultados incorretos devido a acesso de memória com valores ainda não atualizados. Uma característica que deve ser evitada na programação paralela são sincronizações sucessivas em blocos paralelos. Um bloco paralelo realiza o lançamento de *n threads*, entretanto sucessivas pausas para sincronizar os resultados parciais reduzem significativamente a eficiência gerada pelo paralelismo.

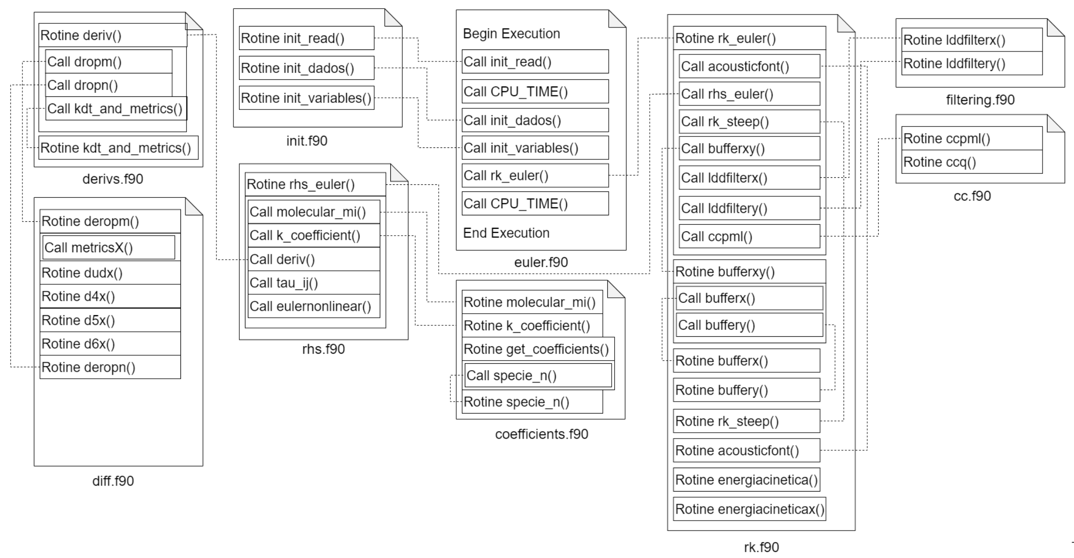


Figura 5.3. Relação esquemática entre as funções

O código que faz uso da GPU deve obter ganho de desempenho superior ao da execução do bloco em CPU. A GPU possibilita a execução de blocos com grande volume de dados em um tempo significativamente inferior ao da CPU. Mas, todos os dados manipulados no bloco paralelo devem ser copiados para a memória da GPU e os dados retornados devem ser copiados para a memória da CPU. Essas copias podem inviabilizar algumas paralelizações, pois a análise considera não apenas a execução.

A performance de um código OpenMP ou OpenACC pode ser avaliada pelo *speedup*(S). O *speedup* é definido como a razão entre o tempo de computação do algoritmo serial (T_{serial}) e o tempo de computação do algoritmo paralelo ($T_{paralelo}$), dado pela Equação 1. O *speedup* mostra o ganho efetivo do tempo de processamento do algoritmo paralelo sobre o algoritmo serial.

$$S = \frac{T_{serial}}{T_{paralelo}} \tag{1}$$

Quando o tempo paralelo é exatamente igual ao tempo sequencial, o *speedup* é igual a 1. Neste caso não há ganho de desempenho. Uma outra forma de mensurar o quanto uma versão paralela é melhor que a versão sequencial é considerar o percentual de ganho de desempenho apresentado na Equação 2.

$$S = \frac{T_{serial} - T_{paralelo}}{T_{paralelo}} * 100 \tag{2}$$

Nessa seção é apresentada uma aplicação que não obtém um ganho desempenho significativo devido as características presentes no domínio e ao volume de dados. A outra aplicação possibilita o ganho expressivo de desempenho em certas funcionalidades do código. Para ambos os casos, o mesmo ambiente de execução foi utilizado conforme

Tabela 5.1. Ambiente de execução: CPU

Características	Xeon E5-2650 (×2)
Frequência	2.00 GHz
Núcleos	8 (×2)
Threads	16 (×2)
Cache L1	32 KB
Cache L2	256 KB
Cache L3	20 MB
Memória RAM	128 GB

Tabela 5.2. Ambiente de execução: GPU

Características	Quadro M5000
Frequência	1.04 GHz
CUDA *cores*	2048
Cache L1	64 KB
Cache L2	2 MB
Memória Global	8 GB

descrito em na Tabela 5.1 e Tabela 5.2.

5.6.1. Aplicando diretivas OpenMP em algoritmos bio-inspirados

Serapiao (2009) define a inteligência de enxames ou inteligência de colônias, ou ainda inteligência coletiva, é um conjunto de técnicas baseadas no comportamento coletivo de sistemas auto-organizados, distribuídos, autônomos, flexíveis e dinâmicos. Os algoritmos que aplicam essa técnica simulam a forma com que o enxame (grupo de agentes) interagem e a forma com que tomam decisões.

A Figura 5.4 apresenta um algoritmo de inteligência de enxame genérico. Inicialmente o algoritmo gera uma população inicial que normalmente é aleatória (critério do desenvolvedor). Em seguida essa população é avaliada de acordo com uma função de qualidade, analisada e em caso de ótimo local armazenada. O critério de parada é uma maneira de controlar a quantidade de iterações que serão realizadas no algoritmo, caso não satisfeito as alterações são realizadas sob a população de acordo com a estratégia de cada algoritmo a fim de melhorar e alcançar a função objetivo; ao fim das iterações a melhor solução é retornada.

O algoritmo otimização por Enxame de Partículas (PSO) foi modelado a partir do comportamento social do bando de aves (ENGELBRECHT, 2007). No PSO, a população de indivíduos ou partículas é agrupada em um enxame (conjunto de soluções). Essas partículas simulam o comportamento de pássaros através do aprendizado próprio (componente cognitivo) e o aprendizado do bando (componente social), ou seja, imitam o seu próprio sucesso e o de indivíduos vizinhos (TAVARES; NEDJAH; MOURELLE, 2015). A partir desse comportamento as partículas podem definir novas posições que as direcionam para soluções ótimas.

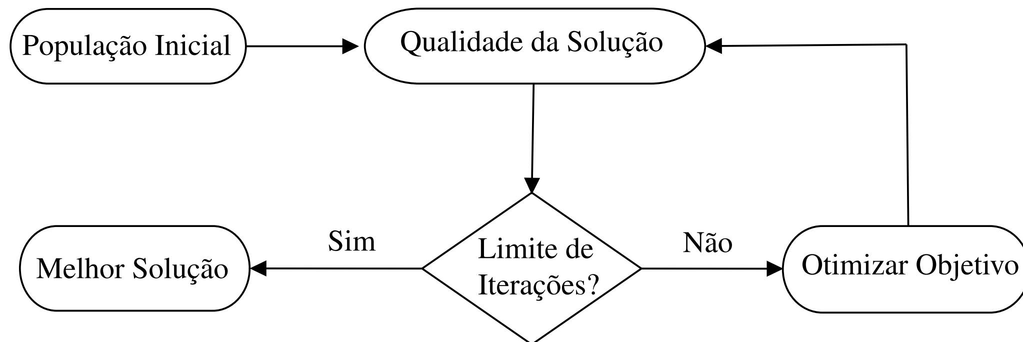


Figura 5.4. Representação da estrutura lógica de um algoritmos de inteligencia de enxame

Algoritmo 1: Pseudocódigo PSO

```
1  início
2      Inicializar as partículas ;                    // Soluções iniciais
3      Inicializar a velocidade das partículas ;
4      enquanto critério de parada não for satisfeito faça
5          Calcular a aptidão das soluções;
6          Memorizar a melhor solução local
7          Memorizar a melhor solução global
8          Atualizar a velocidade das partículas ;
9          Atualizar a posição das partículas
10     fim
11     retorna Melhor Solução Encontrada ;
12 fim
```

```
1 double sphere(vector<double> temp){
2   double sum=0;
3   #pragma omp simd reduction(+:sum)
4   for(int i=0; i<temp.size() ;i++){
5     sum += pow(temp[i],2);
6   }
7   return sum;
8 }
```

Figura 5.5. Segmento de código paralelo - PSO

Função	PSO			
	Melhor Solução	Tempo Sequencial (Desv. Padrão)	Tempo Paralelo (Desv. Padrão)	Taxa de Desempenho (%)
1. Alpine	1,95E+00	0,370068 (0,0253)	0,091106 (0,0113)	306,19
2. Booth	1,47E-05	0,012690 (0,0014)	0,001806 (0,0002)	602,84
3. Easom	-9,64E-01	0,104991 (0,0070)	0,026976 (0,0013)	289,20
4. Griewank	1,20E+02	3,792227 (0,3100)	1,167276 (0,1064)	224,88
5. Rastrigin	3,05E+02	3,947903 (0,3863)	1,047876 (0,1123)	276,75
6. Rosenbrock	1,30E+05	2,737766 (0,2457)	0,129914 (0,0711)	2007,38
7. Sphere	1,31E-02	0,085171 (0,0115)	0,006861 (0,0027)	1141,37

Tabela 5.3. Casos de Teste - Melhores Resultados.

As instruções OpenMP no algoritmo PSO foram inseridas na função objetivo e na função de atualização do enxame. As funções Booth e Easom são bidimensionais, logo o laço de repetição foi substituído por uma instrução que compreende o somatório das duas dimensões, as demais funções N-dimensionais possuem um laço de repetição como expresso na Figura 5.5. Esse laço é precedido pela diretiva OpenMP `#pragma omp simd reduction(+: sum)`. O outro segmento de código paralelizado é o método que atualiza a posição de cada partícula e em seguida atualiza a velocidade de cada partícula. A diretiva OpenMP aplicada é `#pragma omp simd`. Para a execução são inseridas *flags* de compilação para sinalizar ao compilador instruções vetoriais do tipo AVX.

O *benchmark* Sphere apresentado na Figura 5.5 foi testado para enxames de diversos tamanhos. A Tabela 5.3 mostra o tamanho da população testado, o tempo sequencial e paralelo com os respectivos desvios padrões, o ganho de desempenho expresso em percentual pela equação 2 e a média das soluções. A média é obtida a partir de 30 execuções.

5.6.2. Aplicando diretivas OpenACC em aplicação de combustão

A aplicação escolhida simula o processo de mistura do combustível de uma aeronave e o material reagente, como detalhado na Seção 5.5.2. O código possui duas versões, uma na linguagem de programação C++ e a outra na linguagem Fortran. A estrutura do código é robusta. Esse código é subdivido em diversos arquivos e métodos como expresso na 5.3.

A Tabela 5.4, apresenta o tempo de execução de cada uma das funções mais expressivas de tempo para uma execução de um domínio de 521 (x) por 481 (y) pontos

Nome da Função	Porcentagem do Tempo Total de Execução
deropn	24,11 %
rhs_euler	19,24 %
deropm	17,46 %
lddfilterx	7,92 %
lddfiltery	7,04 %
rk_euler	5,23 %
eulernonlinear	4,68 %
metricsx	3,72 %
metricsy	2,71 %
molecular_mi	1,93 %
bufferxy	1,48 %
k_coefficient	1,19 %
deriv	1,11 %

Tabela 5.4. Porcentagem do tempo de execução usando GProf

coletados com a ferramenta GProf. Pode-se observar que não há uma função que domina o tempo total de execução e que as funções cujas operações atuam sobre o eixo x demandam um tempo mais expressivo do que as sobre o eixo y, dado que o domínio em x é maior que o em y para o problema em questão. O fato de uma chamada demandar de mais tempo de processamento não significa que esta terá o maior potencial de paralelização. No caso de implementações em GPU por exemplo, o tempo de transferência e sincronização dos dados pode tornar o ganho paralelo pouco efetivo para compensar a transferência de dados.

Inicialmente o código é paralelizado com a API OpenMP, sendo registrado quais modificações foram realizadas e quais os respectivos tempos de execução. O tempo de execução é mensurado a partir da *flag* `time` precedendo o script de execução. Outra informação que deve ser registrada é se o resultado obtido entre a versão sequencial e a paralela é igual. Caso o resultado não for igual o paralelismo está incorreto e não pode ser utilizado, sem o devido tratamento das operações.

O código também pode ser paralelizado com diretivas da API OpenACC. Nem todo segmento de código que contenha um `pragma` de OpenMP pode ser substituído por um `pragma` de OpenACC. Lembrando que os dados utilizados nos métodos paralelizados devem ter sido copiados para a GPU. Através da paralelização do código em OpenACC é possível observar e comparar o tempo de execução da aplicação sobre diferentes APIs.

A Tabela 5.5 apresenta o ganho de desempenho aplicando algumas otimizações em apenas algumas funções da aplicação. Na tabela, pode-se observar a redução do tempo de processamento. Outras funções também podem ser otimizadas para diminuir mais ainda o tempo de processamento.

Tipo	Tempo (s)	Ganho de Desemp (%)
Sequencial	126,6606	-
OpenACC	113,0066	12,08

Tabela 5.5. Taxa de ganhos de desempenho.

5.7. Conclusão

Paralelizar uma aplicação possui desafios. Muitas vezes é necessário reescrever ou realizar adaptações no código sequencial. Posteriormente, deve-se partir para a paralelização do problema. Uma técnica escolhida para o paralelismo pode não ser a mais adequada ao domínio do problema ou as características que ele possui. Neste caso, outras abordagens devem ser testadas. Também é preciso garantir a equivalência numérica dos resultados, ou seja, uma versão paralela não pode resultar em valores inconsistentes da solução do problema.

Nem sempre é possível obter ganho de desempenho com a execução paralela de trechos de código, por mais fino que o paralelismo seja aplicado, uma vez que a granularidade é baixa, ou a cópia de dados envolve um sobrecusto não compensado pelo paralelismo. Este é o caso do algoritmo bio-inspirado avaliado nesse capítulo que serve de exemplo como estudo de caso para situações onde a granularidade paralela é bastante baixa.

Em relação a aplicação de simulação de combustão esta possui um grande conjunto de funções que podem se paralelizadas tanto em OpenMP como em OpenACC. A comparação realizada nas atividades permite perceber o impacto da granularidade das aplicações. Assim como a viabilidade do uso da GPU e custo de alterações. Estas mesmas soluções podem ser aplicadas a outras classes de aplicações.

Referências

BANZHAF, W. et al. *Genetic programming: an introduction*. California: Morgan Kaufmann San Francisco, 1998. v. 1. páginas 7

CARVALHO, G. C. de et al. Otimização com algoritmo bio-inspirado de controle de tráfego em sistemas de grupos de elevadores. *Revista Interdisciplinar de Pesquisa em Engenharia*, v. 2, n. 9, p. 56–76, 2016. páginas 7

CASTRO, L. N. de. Fundamentals of natural computing: an overview. *Physics of Life Reviews*, Elsevier, v. 4, n. 1, p. 1–36, 2007. páginas 6, 7

CASTRO, L. N. de et al. Computação natural: Uma breve visão geral. In: *Workshop em nanotecnologia e Computação Inspirada na Biologia*. Santos, São Paulo: Universidade Católica de Santos (UniSantos), 2004. páginas 6

CHANDRASEKARAN, S.; JUCKELAND, G. *OpenACC for Programmers: Concepts and Strategies*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2017. ISBN 0134694287. páginas 5

CHAPMAN, B.; MEHROTRA, P.; ZIMA, H. Enhancing openmp with features for locality control. In: CITESEER. *Proc. ECWMF Workshop" Towards Teracomputing-The Use of Parallel Processors in Meteorology*. Austrian: PSU, 1998. páginas 3

DICIOLLA, M. et al. Patient classification and outcome prediction in iga nephropathy. *Computers in biology and medicine*, Elsevier, v. 66, p. 278–286, 2015. páginas 6

ENGELBRECHT, A. P. *Computational intelligence: an introduction*. South Africa: John Wiley & Sons, 2007. páginas 7, 10

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 17, n. 6, p. 120–126, jun. 1982. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/872726.806987>. páginas 8

IGNÁCIO, A. A. V.; FERREIRA, V. J. M. F. Mpi: uma ferramenta para implementação paralela. *Pesquisa Operacional*, scielo, v. 22, p. 105 – 116, 06 2002. ISSN 0101-7438. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382002000100007&nrm=iso>. páginas 6

KENNEDY, J.; EBERHART, R. C. *Swarm Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN 1-55860-595-9. páginas 7

KIRK, D. B.; WEN-MEI, W. H. *Programming massively parallel processors: a hands-on approach*. [S.l.]: Morgan kaufmann, 2016. páginas 2

LIN, J.; LUCAS, T. A. A particle swarm optimization model of emergency airplane evacuations with emotion. *NHM*, v. 10, n. 3, p. 631–646, 2015. páginas 6

MOORE, G. E. *Cramming more components onto integrated circuits, Electronics Magazine*. 1965. páginas 2

NIEVERGELT, J. et al. *All the needles in a haystack: Can exhaustive search overcome combinatorial chaos?* Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. 254–274 p. ISBN 978-3-540-49435-5. Disponível em: <https://doi.org/10.1007/BFb0015248>. páginas 6

NVIDIA. *CUDA C Programming Guide*. 2020. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessado em: 2020-02-28. páginas 3

OPENACC. *What is OpenACC?* 2019. [Online; acesso 20 Fevereiro 2020]. Disponível em: <https://www.openacc.org/>. páginas 5, 6

OPENMP. *The OpenMP API specification for parallel programming*. 2020. [Online; accessed at January, 15 2020]. Disponível em: <https://www.openmp.org/>. páginas 3, 4

RAUBER, T. W. Redes neurais artificiais. *Universidade Federal do Espírito Santo*, 2005. páginas 8

SERAPIAO, A. Fundamentos de Otimização por Inteligência de enxames: Uma Visão Geral. *Controle y Automacao*, v. 20, p. 271–304, 07 2009. páginas 7, 10

SILVA, M. et al. Mixing layer stability analysis with strong temperature gradients. In: *17th Brazilian Congress of Thermal Sciences and Engineering (ENCIT 2018)*. [S.l.: s.n.], 2017. páginas 8

SUTTER, H.; LARUS, J. Software and the Concurrency Revolution. *Queue*, ACM, New York, NY, USA, v. 3, n. 7, p. 54–62, set. 2005. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/1095408.1095421>. páginas 2

TAVARES, Y.; NEDJAH, N.; MOURELLE, L. de M. Utilização de otimização por enxame de partículas e algoritmos genéticos em rastreamento de padrões. In: *12 Congresso Brasileiro de Inteligência Computacional*. Rio de Janeiro, Brasil: Diretoria de Sistemas de Armas da Marinha, 2015. páginas 10

TORELLI, J. C.; BRUNO, O. M. Programação paralela em smps com openmp e posix threads: um estudo comparativo. In: *Anais do IV Congresso Brasileiro de Computaçao (CBComp)*. São Carlos, SP: Instituto de Ciências Matemáticas e de Computação Universidade de São Paulo, 2004. v. 1, p. 486–491. páginas 4

 DOI: 10.5753/sbc.4400.9

Capítulo 6

Estudo de Networks on Chip (NoCs) em FPGAs

Maurício Acconcia Dias, Marcílio F. de Oliveira Neto
Centro Universitário da Fundação Hermínio Ometto
Araras, Brasil

Resumo

Processadores com mais de um núcleo de processamento, os multicores, são atualmente a base para o desenvolvimento de Systems on Chip (SoCs) tanto para aplicações de propósito geral quanto de propósito específico. A evolução da tecnologia de fabricação, aliada a necessidade de explorar o desempenho dos processadores em outros aspectos além da frequência de clock, gerou o cenário atual de desenvolvimento de unidades de processamento. Assim como existem as redes de computadores cujas características são analisadas em macro escala existem redes internas nos chips (chamadas de Networks on Chip, NoCs), desenvolvidas para realizar a comunicação entre todos os componentes de hardware de um determinado SoC de forma eficiente em diversos aspectos como latência, consumo de energia e área de chip ocupada. Considerando estas estruturas o objetivo principal deste trabalho é apresentar os conceitos básicos relacionados às NoCs juntamente com uma análise de qual o papel desempenhado e as modificações propostas pelos Field Programmable Gate Arrays (FPGAs) para auxiliar o desenvolvimento desta importante subárea da computação de alto desempenho.

6.1. Introdução

Com a evolução tecnológica dos materiais que compõe os processadores - transistores, bem como a evolução dos demais circuitos, arquiteturas que utilizam processadores *multicore* são a base da computação atual e são o foco também da computação de alto desempenho. Ao se colocar mais de um processador em um mesmo chip cria-se um problema de comunicação entre os núcleos que não deve ser solucionado com as ferramentas de barramentos atuais por não atingirem de forma satisfatória os requisitos de sistema como *throughput*, consumo de energia, confiabilidade. Então qual seria a solução para este problema?

O conceito de comunicação em rede surgiu no passado com objetivo de conectar de uma forma mais otimizada sistemas que compunham redes *peer-to-peer*. A tarefa prin-

cipal dos projetistas de redes é solucionar múltiplas instâncias de um problema complexo de otimização que possui diversas soluções possíveis. No caso das NoCs este problema retorna em um contexto diferente, porém não menos complexo. Redes on-chip estão prevalecendo no domínio dos servidores de alto desempenho integrados à sistemas embarcados on-chip (SoC - System on-chip) (PEH; JERGER, 2009). Ao se projetar uma NoC novos desafios para problemas antigos como roteamento, qualidade de serviço, fluxo, controle de congestionamento e confiabilidade são encontrados e precisam ser solucionados. Além disso novos desafios são incorporados ao problema como consumo de energia e área de chip ocupada que contribuem ainda mais para o aumento da complexidade das NoCs.

Os multiprocessadores SoC permitem soluções com alto poder de processamento em aplicações embarcadas e utilizam uma implementação de memória distribuída compartilhada (DSM - Distribuited Shared Memory) em hardware. Sistemas SoC são compostos por diversos núcleos de processamento, memória e coprocessadores de propósitos específicos, e baseiam-se em uma arquitetura de memória NUMA (Non-Uniform Memory Access), permitindo acesso variado à memória através da rede de alto desempenho (MONCHIERO et al., 2007). Na arquitetura de rede dos SoCs, barramentos compartilhados dominam as estruturas de interconexões em sistemas simples, sendo esses barramentos bidirecionais permitindo o tráfego de informações pelas vias. Tal estrutura compartilhada facilita a adição de novos dispositivos, além de permitir portabilidade de periféricos entre os sistemas (ABDALLAH, 2017).

Conforme o número de núcleos conectados ao barramento da estrutura da rede aumenta, há um aumento proporcional na demanda de largura de banda a fim de facilitar a utilização de todos os núcleos (PEH; JERGER, 2009) e no clock geral do sistema, ocasionando atrasos e gerando gargalos importante no desempenho da aplicação. Além disso, essa abordagem tradicional de comunicação falha quando requisitos de escalabilidade e consumo de energia são levados em consideração para buscar uma evolução de futuros SoCs. Assim, novos esquemas de interconexão estão sendo propostos, denominados *Networks on Chip - NoC* (ABDALLAH, 2017), a fim de mitigar o gargalo na comunicação de sistemas on-chip (TSAI et al., 2012). Este texto está organizado com a análise de camadas proposta por Benini e De Micheli (2006): a seção 1.2 trata da camada física seguida pela camada de enlace na seção 1.3, redes e transporte na seção 1.4, aplicação e implementação na seção 1.5 e, por fim, uma análise da implementação em FPGAs na seção 1.6. O texto encerra-se com a conclusão e as referências bibliográficas.

6.2. A camada física

Gordon Moore foi responsável por definir um dos principais guias responsáveis por orientar a o ramo da arquitetura de computadores, conhecido como Lei de Moore. Moore previu que o número de transistores em um chip dobraria a cada 12 meses, embora esse fator tenha sido alterado para cada 18 meses, a Lei de Moore foi por muito tempo um dos principais fatores para promover o avanço na área de circuitos integrados. Estudo atuais sobre a miniaturização dos transistores e o efeito do tunelamento quântico demonstram que não se pode diminuir o dispositivo a partir de um certo tamanho sem prejudicar seu funcionamento. Sendo assim a lei de Moore que previa que o número de transistores seria dobrado não é mais verdade, e sim que os mesmos transistores e outros componen-

tes serão utilizados de uma maneira mais inteligente formando hardware mais eficiente (TRACK; FORBES; STRAWN, 2017). Assim, diferentes arquiteturas que exploram diversos modelos de interconexões da rede utilizando inúmeros transistores, que visam o baixo consumo energético e a máxima frequência são oferecidas a fim de de uma melhor eficiência do sistema on-chip (ABDALLAH, 2017).

Em sistemas on-chip (SoC), cujo conceito arquitetural tem sido desenvolvido nas últimas décadas, cada processador - ou processadores, juntamente com memórias e seus conjuntos de periféricos estão interconectados através de um barramento implementado em um único chip. Entretanto, com o aumento do número de transistores, o design está a cada dia mais complexo, o que demanda novas técnicas para resolver o gargalo de comunicação e que possibilite que um chip resolva aplicações complexas de forma a aproveitar todos os benefícios fornecidos pelas tecnologias atuais (CHEN et al., 2012).

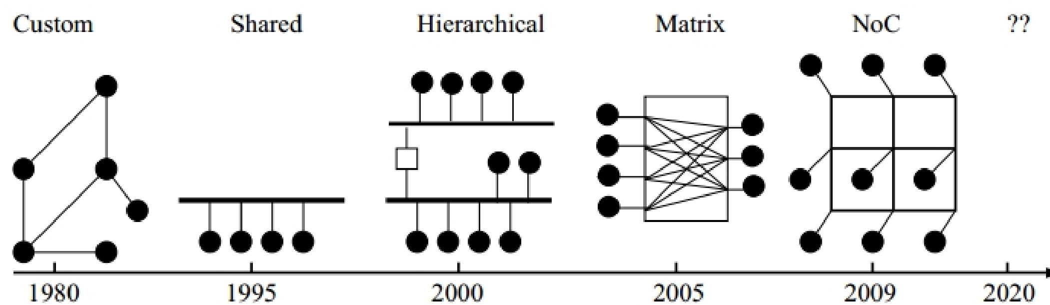


Figura 6.1. Linha do tempo da evolução da interconexão de sistemas on-chip. (ABDALLAH, 2017)

A interconexão é o principal elemento de sistemas multiprocessadores on-chip, uma vez que buscam fornecer baixa latência na camada de comunicação, a fim de minimizar o overhead gerado pelo sincronismo de processos, contrapondo a abordagem baseada em barramentos, o qual possui limitações físicas, garantindo alta largura de banda e paralelismo na comunicação devido à latência dos fios (Monchiero et al., 2006). Para tanto, sistemas on-chip precisam ser confeccionados cuidadosamente, pois podem consumir altos níveis de energia (PEH; JERGER, 2009). A Figura 6.1 exibe uma linha do tempo acerca das interconexões de sistemas on-chip, evidenciando que as arquiteturas de interconexão de redes que utilizam NoCs são as mais utilizadas nos últimos anos.

A estrutura básica de uma NoC é apresentada na Figura 6.2. Esta estrutura pode se alterar de acordo com a topologia escolhida, porém os componentes básicos são os mesmos. O *Switch* irá fazer a conexão entre os *Data Links* para transportar a informação dos Módulos de Processamento. As estruturas da camada física permitem que a camada de enlace possa executar suas operações.

6.3. Camada de enlace

No caso das NoCs a camada de enlace é responsável por garantir uma transferência confiável entre meios físicos de transmissão inerentemente não confiáveis (*data links*). Outra tarefa importante desta camada é regular o acesso a recursos compartilhados. Neste contexto existem dois problemas a serem tratados: a parte interna que precisa ser controlada

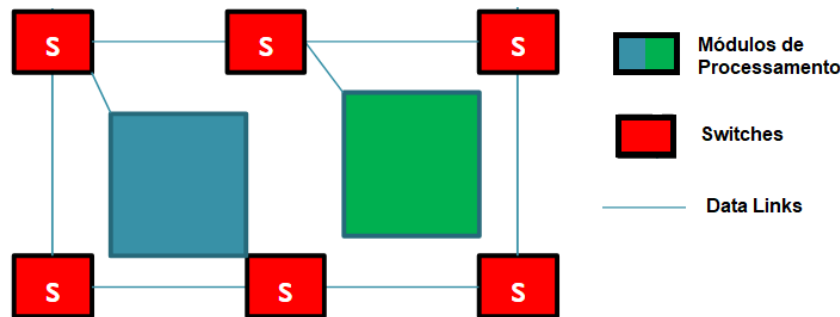


Figura 6.2. Estrutura básica de uma NoC genérica com foco em seus principais componentes.

da melhor forma possível com um escalonador, e a parte de interferências internas e externas (eletromagnéticas, ruídos térmicos, partículas alpha).

A disputa por recursos nas NoCs é praticamente inevitável. A existência de circuitos chamados árbitros permite que o pacote seja enviado e colida com outros para decidir de quem é o barramento. Como consequência a única maneira de se conseguir eliminar o árbitro é utilizar roteamento sem contenção que, caso seja utilizado, necessita que os pacotes sejam escalonados de forma a nunca utilizar o mesmo link ao mesmo tempo.

Os erros causados por interferências são normalmente resolvidos considerando a redundância de informações pela rede juntamente com técnicas de conferência de bits que são mais fáceis de serem implementadas em hardware.

6.4. Camadas de transporte e de rede

Apesar de todas as camadas desempenharem papeis importantes, as camadas de transporte e rede podem impactar significativamente o resultado de um projeto de NoC mesmo que as outras camadas estejam obtendo bom desempenho. O procedimento nesta etapa envolve o princípio de *switching* a ser utilizado (circuitos, pacotes), em seguida a topologia da rede deve ser escolhida e a conexão física realizada corretamente, por fim o esquema de endereçamento e roteamento devem ser definidos.

Neste momento o dispositivo onde a NoC será construída deve ser escolhido. Neste momento é importante ter em mente as vantagens e desvantagens de cada tipo de hardware sendo que o *chip* propriamente dito (ASIC) e os FPGAs são escolhas interessantes. A seção 1.5 irá tratar em mais detalhes as características de cada um dos dispositivos.

A questão da qualidade de serviço (QoS) de uma NoC é definida de diversas maneiras, porém três características são base para a definição: *data rate* que representa a taxa em que dados e instruções são processadas, a latência da rede, e a variação da latência (conhecida como *jitter*). Para garantir os requisitos de QoS é necessário definir com cuidado as partes da NoC, começando por sua topologia.

O principal foco quando se trata de arquiteturas multiprocessadas que utilizam plataformas SoC é a topologia das interconexões (PANDE et al., 2005) e quando o foco é a

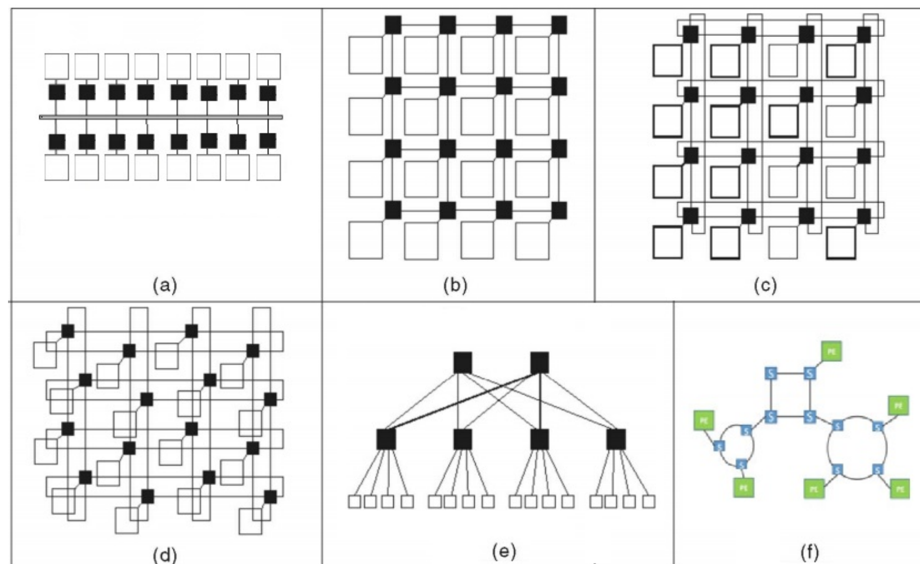


Figura 6.3. Topologias para NoCs. (a) Crossbar, (b) Mash, (c) Torus, (d) Folded Torus, (e) Tree, (f) Irregular(ABDALLAH, 2017)

perspectiva de comunicação, existem várias topologias que tangem os NoCs (AGARWAL; SHANKAR, 2009). O assunto sobre topologias de rede foi largamente explorado no contexto de computação paralela e distribuída. Neste trabalho serão discutidas as questões relevantes para NoCs. A diferença principal é que normalmente as questões de topologia em NoCs são concentradas em modelos 2D considerando o estado atual da fabricação de *chips*.

As topologias podem incluir redes de diversos tipos como apresentados na Figura 6.3. Tantas variedades buscam explorar diferentes topologias e implementações para plataformas SoCs que usufruem de NoCs (AGARWAL; SHANKAR, 2009). Os objetivos de escolha de uma determinada topologia envolvem requisitos de área de chip, desempenho e consumo de energia (BENINI; MICHELI, 2006).

O uso de malhas em 2D faz com que as NoCs sejam mais eficientes em termos de latência, consumo energético e facilidade na implementação (AGARWAL; SHANKAR, 2009). A topologia Crossbar (Figura 6.3(a)) apresenta todos os nós da rede interconectados, porém a escalabilidade é baixa. O Mesh (Figura 6.3(b)) é popular entre os projetos de NoCs (60% dos casos de topologias 2-D (PEH; JERGER, 2009)), a área cresce linearmente em relação ao tamanho do Mesh e o tipo de conexão acaba por concentrar um tráfego muito alto no centro. No caso do Torus (Figura 6.3(c)) a área ocupada e a dissipação de energia cresce linearmente com o número de núcleos e o desempenho é inversamente proporcional ao tamanho da estrutura pela demora na transmissão de linhas muito longas. Quando longos barramentos em linha existem, problemas de resistência e capacitância não podem ser ignorados.

Este problema pode ser resolvido com a topologia da Figura 6.3(d) que possui *bypasses* para otimizar a comunicação sendo o tipo mais utilizado em FPGAs. A topologia em árvore (Figura 6.3(e)) pode apresentar bons resultados em redes pequenas, porém

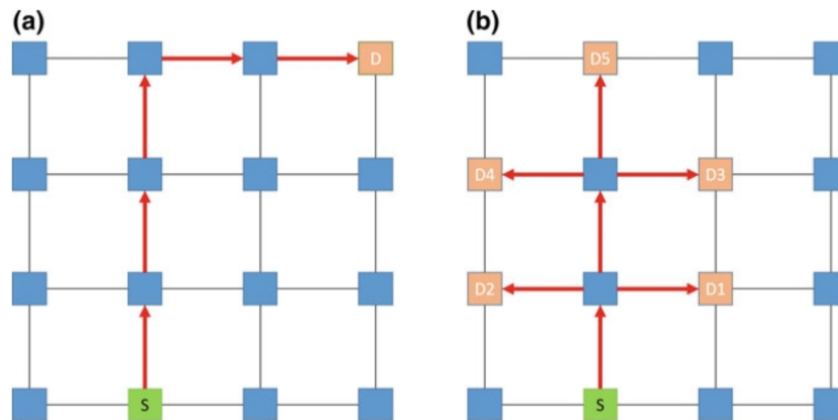


Figura 6.4. Categorização dos algoritmos de roteamento. a) *unicast*; b) *multicast* (ABDALLAH, 2017)

tende a ser pior que as duas anteriores em redes normais devido ao tempo de acesso por começar sempre na raiz e ter que percorrer a árvore. Topologias mistas podem ser empregadas no uso de aplicações específicas, uma vez que a o desenvolvimento pode exigir restrições rigorosas, e.g.: área e energia. Assim, as topologias regulares podem não ser a abordagem adequada, uma vez que topologias irregulares podem oferecer melhor flexibilidade (ABDALLAH, 2017). Um exemplo de topologia irregular pode ser visto na Figura 6.3(f).

As plataformas que utilizam NoC são baseadas em redes comutadas por pacotes, gerando novas e eficientes formas de roteamento. Segundo (AGARWAL; SHANKAR, 2009), os roteadores implementam várias funcionalidades, desde uma comutação simples até o roteamento inteligente, ou seja, suponha que um roteador baseado em topologia de malha possua quatro entradas e quatro saídas para outros roteadores e outras entradas e saídas para interfaces de rede (NI - Network Interface), dado esse cenário, o roteamento deve ser projetado baseado no uso do hardware. Para redes com comutação de circuitos, roteadores podem ser projetados utilizando filas (buffering), e para redes comutadas por pacotes, há a necessidade de se dispor de uma certa quantidade de buffers para suportar a transferência de dados intermitentes.

Os algoritmos de roteamento estão intrinsecamente ligados às topologias de rede. Tais algoritmos podem ser classificados, segundo (ABDALLAH, 2017), de acordo com vários critérios, sendo:

- **Número de destinações:** Conforme o número de nós de destino, os quais recebem os pacotes destinados a eles, os algoritmos de roteamento podem ser classificados em *unicast* e *multicast*. O roteamento *unicast* envia pacotes de um único nó de origem para um único nó destino. Já os *multicast* enviam pacotes de um único nó de origem para vários nós de destino. Além disso, o *multicast* pode ser dividido em roteamento baseado em árvore e baseado em caminho. A Figura 6.4 ilustra as duas categorias.
- **Decisão de localidade:** De acordo com as decisões de localidade, os algoritmos

de roteamento (sejam *multicast* ou *unicast*, podem ser classificados em roteamento de origem ou roteamento distribuído. No roteamento distribuído, haverá um cabeçalho - roteamento *unicast* - contendo o endereço do nó de destino. Com isso, as informações são calculadas *in place* toda vez que o cabeçalho entra em um nó do comutador. Já no roteamento de origem, os caminhos são calculados no nó de origem.

- **Adaptativo:** Em todas as implementações de roteamento, o algoritmo pode ser determinístico ou adaptativo. No roteamento determinístico, os caminhos gerados entre fonte e destino são calculados estaticamente e sempre serão parecidos. Já o algoritmo adaptativo pode desviar de regiões não favoráveis ou defeituosas na rede. Além disso, podem ser divididos em totalmente adaptativos ou parcialmente adaptativos.

- **Minimalidade:** Algoritmos que podem ser classificados em minimais ou não-minimais. Em linhas gerais, um algoritmo minimal não permite rotas afastadas do nó destino, garantindo sempre o caminho mínimo. O não-minimal permite desvios, acarretando em rotas distantes do nó de destino, que podem existir de formas aleatórias ou seguindo certas regras.

Além disso, em Agarwal e Shankar (2009) pode ser visto uma classificação dos algoritmos de roteamento baseados no número de nós de destino, sendo *unicast* e *multicast*. Tal classificação, além dos já citados acima, levam em consideração o número de caminhos gerados, o quão progressivo é o algoritmo e a forma de implementação, que podem utilizar tabelas de consulta ou máquinas de estados. As implementações que utilizam tabela de consulta são as mais populares, uma vez que são implementados em *software*, onde armazenam todos os nós em uma tabela. Já as implementações baseadas em máquinas de estados podem ser implementadas tanto em *software* quanto em *hardware*.

Na prática o modelo de NoC segue o padrão semelhante a um grid. Os módulos de interfaces de rede transformam os pacotes de dados gerados dos clientes - processadores - em sinais de controle de fluxo de comprimento fixo, denominados *flits*. Os *flits* associados ao pacote de dados consiste em fornecer um cabeçalho, uma saída e um identificador. Esse conjunto será roteado em direto ao nó destino, passando de um roteador ao outro. Neste modelo de NoC, cada roteador possui cinco portas de entradas e cinco portas de saídas, que são conectadas aos vizinhos, formando o grid. Assim, a função do roteador é rotear cada *flit* de uma porta de entrada à uma porta de saída apropriada. A fim de performar essa tarefa, os roteadores são preparados com buffers em cada porta, *switches* para redirecionar o fluxo e lógica de controle para garantir a corretude da execução (TSAI et al., 2012), conforme mostrado na Figura 6.5.

Protocolos podem requisitar diversas classes de mensagens, consequentemente, diversas classes de coerência devem existir a fim de retornar com a ação correta. Esse tráfego, quando não consistente, podem surpreender o algoritmo de roteamento e acarretar em *deadlocks*. Portanto, garantir que a rede esteja acessível em nível de protocolo é fundamental para que não ocorra impedimentos (*deadlocks*).Peh e Jerger (2009) ilustram essa situação na Figura 6.6, na qual a rede é exposta a inúmeras solicitações que não podem ser consumidas até a interface de rede iniciar a resposta, caracterizando uma

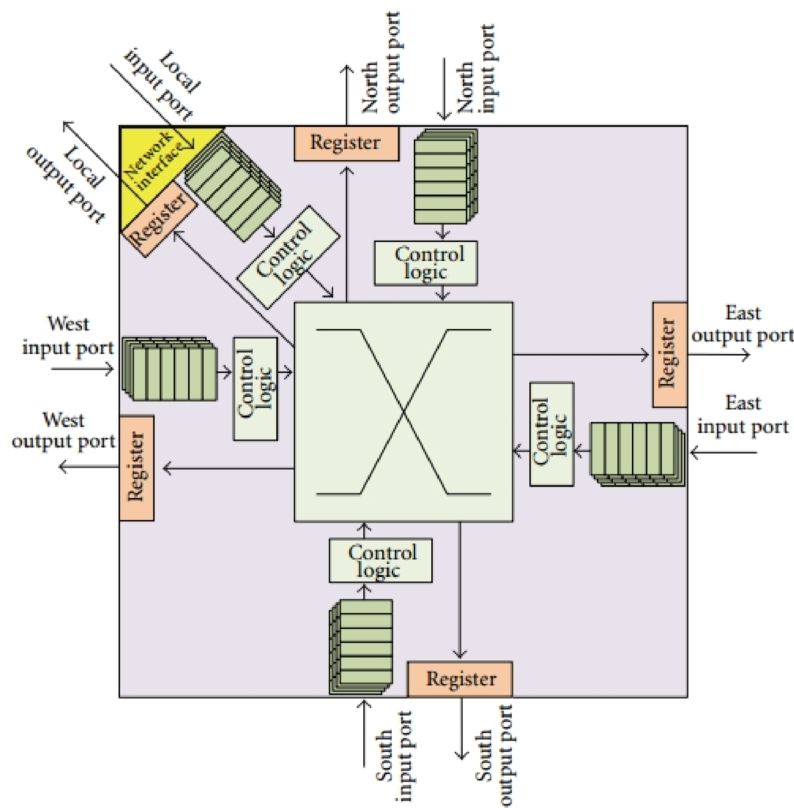


Figura 6.5. Arquitetura típica de roteador em NoC. (TSAI et al., 2012)

dependência cíclica. Assim, se duas unidades funcionais gerarem solicitações ao ponto de sobrecarregar a rede, essas unidades ficaram ociosas aguardando respostas. Se tais respostas utilizarem os mesmos recursos da rede que as solicitações, não haverá avanço no processo, caracterizando um *deadlock*. Portanto, a implementação de SoCs demandam grupos específicos de protocolos baseados diretamente na aplicação que o sistema comportará (TSAI et al., 2012).

6.5. Camada de aplicação e implementação

Sistemas que demandam poder computacional elevado se beneficiam de dois principais tipos de sistemas: multiprocessadores de chip de memória compartilhada e SoCs de multiprocessadores (MPSoCs). Com esses sistemas, a programação paralela se tornou amplamente utilizável e importante em inúmeras aplicações, tal crescimento se deve ao uso de um espaço de endereçamento global compartilhado, facilitando a implementação de aplicações que dependem de programação paralela. Assim, com o modelo de memória compartilhada, a comunicação ocorre de forma implícita através de instruções de *load* e *store* (PEH; JERGER, 2009). A arquitetura de um multiprocessador de chip de memória compartilhada pode ser visto na Figura 6.7

O modelo de memória compartilhada oferece uma forma intuitiva de realizar o compartilhamento. O espaço global de memória compartilhada é baseado na aplicação

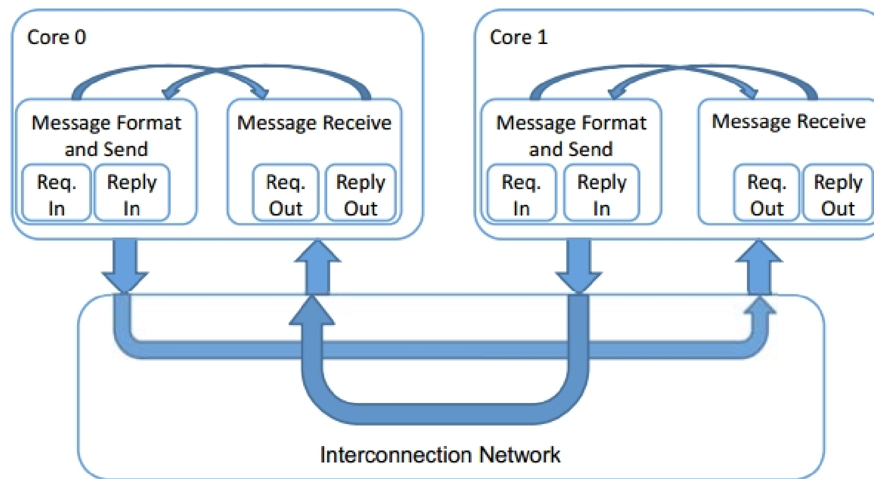


Figura 6.6. *Deadlock* a nível de protocolo. (PEH; JERGER, 2009)

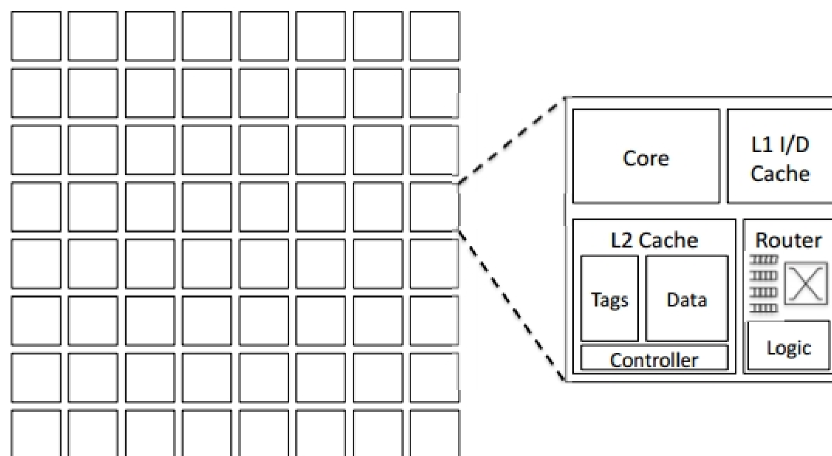


Figura 6.7. Arquitetura de chip de memória compartilhada. (PEH; JERGER, 2009)

que o sistema deve suportar, oferecendo acesso aos endereços de memória local e remota de maneira uniforme (Vasava; Rathod, 2015). Sendo que o acesso remoto ocorre através de uma rede interconectada que permite a troca de dados entre os processadores (RAUBER; RüNGER, 2010). Assim, sistemas que utilizam o espaço de memória compartilhado mantém várias cópias de dados através dos nós a fim de garantir a leitura desses dados, mesmo quando isso não é possível (Xu et al., 2017).

Para garantir a correta execução da aplicação, bem como garantir a consistência dos dados em memória, é necessário que protocolos de semântica e coerência de cache sejam aplicados. Sistemas que utilizam multiprocessadores utilizam um ou dois diferentes tipos de protocolos para garantir a coerência, cada um resultando em diferentes características de tráfego na rede interconectada (RAUBER; RüNGER, 2010). Protocolos de coerência de cache necessitam de diversos tipos de mensagens, como: *unicast*, *multicast*

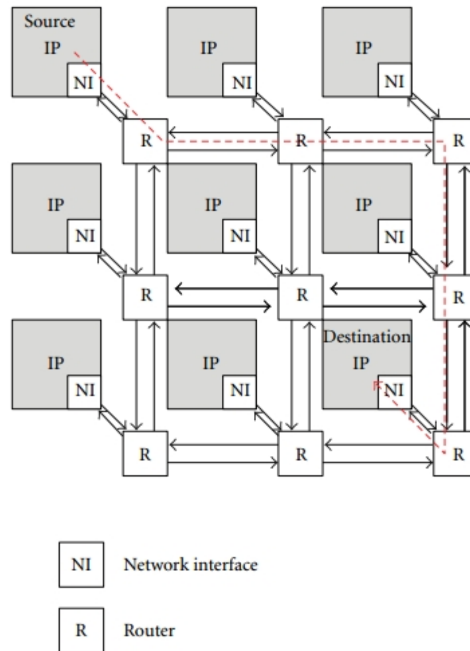


Figura 6.8. Arquitetura típica de NoC. (TSAI et al., 2012)

e *broadcast*. *Unicast* é um tráfico que possui uma única fonte e um único destino, e.g: nível L2 ao controlador de memória). *Multicast* possui um tráfego de uma única fonte para múltiplos destinos. Por fim, *Broadcast* envia uma mensagem de uma única fonte para toda rede interconectada (PEH; JERGER, 2009).

Não só os protocolos dependem intimamente da aplicação que um sistema SoC comportará, como também sua arquitetura e organização. Uma típica arquitetura de um SoC baseado em NoC consiste em múltiplos segmentos de fios e roteadores, como mostrado na Figura 6.8.

As características apresentadas sobre NoCs são aplicáveis independentemente dos dispositivos utilizados para sua implementação. Entretanto em cada caso é necessário considerar a melhor forma de aplicar a teoria para que o resultado na prática seja o melhor possível. Um dos dispositivos mais utilizados para o teste e implementação de NoCs são os FPGAs e, ao desenvolver redes para estes dispositivos, algumas práticas específicas são adotadas. A próxima seção apresenta os FPGAs e seu papel no desenvolvimento das NoCs.

6.6. NoCs em dispositivos reconfiguráveis

Os *Field Programmable Gate Arrays* (FPGAs) são ferramentas de desenvolvimento de hardware amplamente utilizadas para prototipação de sistemas permitindo sua descrição em diferentes níveis de abstração. O FPGA pode ser reconfigurado um número suficiente de vezes para justificar sua utilização como dispositivo de prototipação. Assim como a evolução do hardware que levou aos dispositivos *multicore*, os FPGAs também apresentam diferentes opções de plataformas e ferramentas de desenvolvimento com foco no

desenvolvimento de co-projetos de hardware/software.

Dentre as diversas configurações possíveis para o hardware de um FPGA a mais comum seria a organização hierárquica de blocos de células lógicas interconectadas por barramentos e, na camada mais externa, blocos responsáveis por operações de E/S de dados (BOBDA, 2008). A ideia básica do funcionamento de FPGA é a conversão do hardware descrito em funções lógicas básicas que podem ser implementadas por *Look-Up Tables*. Após esta conversão as funções podem ser armazenadas em elementos lógicos básicos que, em conjunto, formam os blocos de células lógicas programáveis do dispositivo. Os barramentos existentes promovem a conexão dos elementos lógicos que recebem informação dos bolocos de E/S (HAUCK; DEHON, 2008).

Porém, antes de analisar uma figura com a disposição desta arquitetura é conveniente a expansão deste conceito para arquiteturas modernas de FPGAs. A evolução do conceito de desenvolvimento de hardware para os co-projetos de hardware/software mostrou a necessidade de integração entre as soluções de hardware criadas e processadores que executavam os componentes de software (BERGER, 2001).

O movimento inicial nos FPGAs com relação a inclusão de processadores que poderiam ser utilizados juntamente com o hardware desenvolvido foram os *soft-processors* como os processadores Nios II da Intel®e MicroBlaze da Xilinx®. Estes processadores são totalmente descritos e configurados utilizando os componentes lógicos do FPGA e podem se comunicar com qualquer bloco desenvolvido utilizando a própria estrutura do FPGA. O baixo desempenho destas soluções para um grande número de aplicações ressaltou a questão de que comunicar os blocos com um processador é um requisito crítico do co-projeto e a solução foi a criação dos *Systems on a Chip* (SoCs).

Os SoCs apresentam um sistema completo (diversos componentes como processador, memória, lógica reconfigurável) implementado em um mesmo chip. Esta é uma solução que se tornou uma alternativa amplamente utilizada para diminuir os custos de comunicação de co-projetos de hardware/software (WOLF, 2016). No caso dos FPGAs foram desenvolvidos SoCs que contém toda a parte do desenvolvimento da lógica reconfigurável e a conexão com processadores (geralmente ARM®e RiscV®) feita totalmente dentro do chip. O tempo de comunicação neste caso diminui consideravelmente o que impacta diretamente na qualidade da solução final.

A Figura 6.9 apresenta um modelo de arquitetura moderna de FPGAs simplificado indicando a conexão entre um *hard-core processor*, células de memória e a parte responsável pela lógica reconfigurável. Os fabricantes possuem seus próprios modelos de arquitetura, porém de forma básica o que pode ser encontrado é muito similar ao apresentado na Figura 6.9. Um dos FPGAs mais modernos disponíveis no mercado, o Stratix X da fabricante Intel®, possui um bloco com um ARM Cortex®quad-core que pode chegar a 1.5Ghz de clock e um bloco de memória on-chip de 256MB de RAM[1].

Considerando a situação apresentada é possível analisar como os FPGAs são utilizados em relação ao desenvolvimento, criação, verificação, teste e utilzação de NoCs. A implementação de NoCs utilizando FPGAs possui alguns componentes básicos (de Lima et al., 2016):

[1] <https://www.intel.com.br/content/www/br/pt/products/programmable/soc/stratix-10.html>

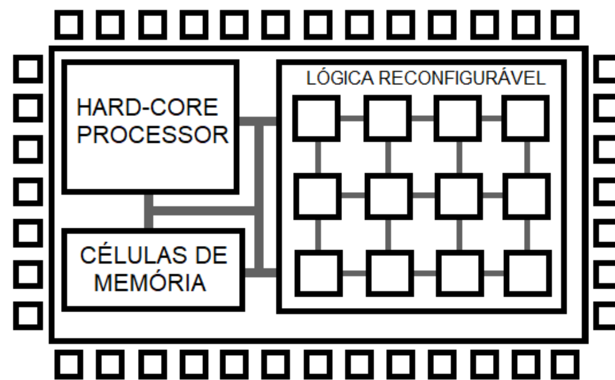


Figura 6.9. Arquitetura de SoCs com FPGAs.

- Host - parte do sistema responsável por receber os cenários de avaliação e enviar os resultados da avaliação.
- Cenários de Avaliação - são as configurações da NoC que está sendo implementada para avaliação. Deve considerar os parâmetros da NoC a serem avaliados e configurar o sistema.
- Resultados da Avaliação - a rede irá receber uma carga de trabalho, direcionar os dados e fazer o envio dos dados solicitados. A partir do comportamento da rede o host avalia todas as métricas desejadas. O relatório final com toda a avaliação deve ficar disponível no host para consulta.
- Geradores de Tráfego - esta parte é responsável por gerar o fluxo de dados que será tratado pela NoC em determinado cenário.
- Receptores de Tráfego - irão receber da NoC os dados após seu processamento e verificar todas as informações necessárias para o cálculo das métricas.
- NoC - a rede propriamente dita com os roteadores conectados e toda a estrutura necessária para seu funcionamento configurada pelos parâmetros descritos pelos Cenários de Avaliação.

A diferença entre os sistemas apresentados está normalmente em como estes componentes são desenvolvidos e onde estão implementados (de Lima et al., 2016). Uma das formas de implementar todos os componentes do sistema no FPGA, sendo que o problema é que devido ao número de elementos lógicos necessários para implementar todo o sistema a NoC que pode ser avaliada acaba por ter um tamanho reduzido. Nestes casos é possível implantar a NoC utilizando um conjunto de FPGAs permitindo que a estrutura avaliada seja maior e mais complexa.

Caso não seja possível implementar todo o sistema em hardware de uma forma funcional é possível utilizar o hardware do FPGA para acelerar o tempo da simulação da rede. Esta aceleração é possível pois implementa-se apenas o mínimo necessário para a

NoC executar o cenário, porém esta técnica não é indicada para testes de novas estruturas devido às simplificações efetuadas na implementação. Ainda assim o hardware do FPGA pode ser um limitante na execução. A saída neste caso é tentar implementar todos os roteadores da rede em um só, multiplexando informações. O tempo de simulação é maior, porém a limitação do hardware é solucionada.

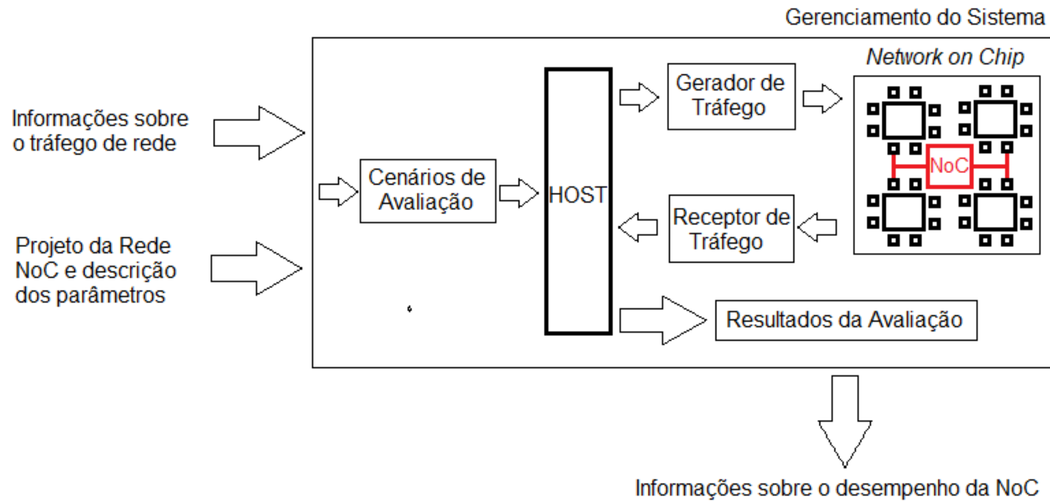


Figura 6.10. Modelo completo de implementação de NoCs utilizando hardware reconfigurável.

A camada de geração de tráfego na rede descreve uma distribuição espaço-temporal do tráfego de informação a ser trabalhado pela NoC. Duas abordagens são utilizadas neste caso: o tráfego sintético que tem por objetivo estressar a rede para descobrir seus problemas estruturais e de projeto de uma forma geral; e o tráfego orientado a aplicação queé importante pois irá utilizar o tráfego real das aplicações para as quais a NoC foi desenvolvida. A geração de tráfego de aplicativos reais para a NoC a ser testada pode seguir três técnicas basicamente (de Lima et al., 2016):

- *Trace-based* - inicialmente a aplicação é executada para se obter os rastros da comunicação realizada pela rede. Os resultados são utilizados para criar cenários baseados nos pontos mais críticos e de forma escalar. Este procedimento gera situações complexas de teste para a NoC. O tamanho reduzido da memória RAM nos FPGAs é o principal problema da técnica limitando os cenários de teste.
- *Stochastic* - análises estatísticas são feitas no comportamento da aplicação e então o tráfego é gerado para o teste da NoC de acordo com os resultados desta análise.
- *Application Cores* - este modelo gera o tráfego de acordo com a execução real dos núcleos das aplicações que irão utilizar a NoC projetada.

A Figura 6.10 representa todos os componentes do processo e como se comunicam. Toda esta estrutura é gerenciada por um sistema que está em uma camada acima do que foi descrito até o momento. No caso dos FPGAs os softwares dos fabricantes

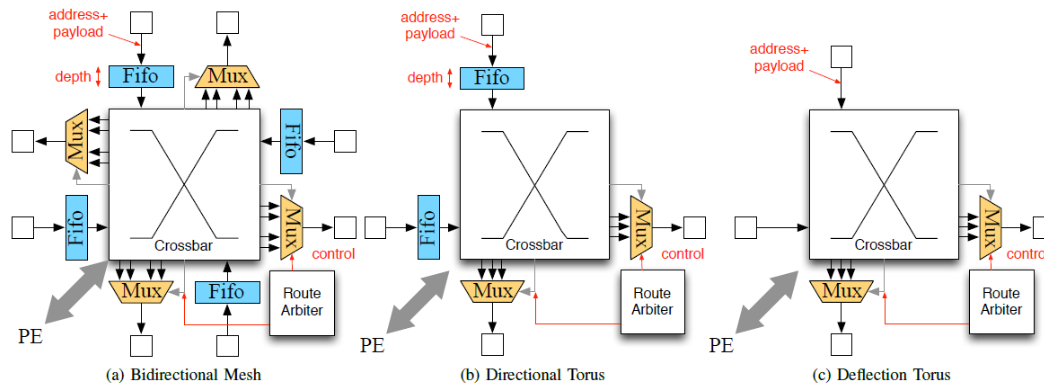


Figura 6.11. Imagens das possíveis arquiteturas de switches para NoCs em FPGAs (Kapre; Gray, 2015).

proveem uma plataforma que soluciona grande parte dos problemas encontrados já que implementam toda a interface de teste segundo padrão IEEE JTAG (IEEE, 2012).

O desenvolvimento da área nos últimos anos permitiu que a abordagem dos trabalhos recentes na área tivesse foco principalmente na otimização de cada um dos componentes apresentados no modelo da Figura 6.10 e no projeto específico para aplicações. A seguir são analisados alguns dos trabalhos recentes da área de NoCs em hardware reconfigurável e seus resultados.

O início da análise será um artigo de 2015 que apresentou uma NoC que seria a base de diversos trabalhos apresentados em conferências a Hoplite (Kapre; Gray, 2015). O trabalho apresenta inicialmente um gráfico comparativo que atesta na prática que a rede Hoplite que possui a topologia *deflection torus* e foi desenvolvida em nível RTL apresentava a melhor relação *Throughput x Utilização de LUTs (Area)* em comparação com a mesma topologia gerada automaticamente, com a Mesh e com a Torus.

A principal contribuição na arquitetura da rede foi a mudança da arquitetura dos *switches* da rede como mostra a Figura 6.11. Dentre as possibilidades, o switch proposto (letra (c) da Figura 6.11) elimina movimentos multidirecionais ao eliminar todas as estruturas FIFO e metade dos multiplexadores em comparação com a implementação mais completa (letra (a) da Figura 6.11). Quando os resultados do artigo foram apresentados a rede apresentava uma área de chip 3.5x menor do que a da melhor rede até o momento com o dobro de frequência máxima de clock e 2.5% de vantagem sobre o throughput da rede 2D bidirecional. Estes resultados expressivos fizeram com que esta rede fosse considerada o estado-da-arte em NoC para FPGAs.

Consumo de energia é uma das características mais importantes atualmente para os SoCs, devido a suas aplicações em sistemas críticos e dispositivos móveis. Uma comparação entre uma NoC implementada em um FPGA (chamado de *soft* pelos autores) e uma NoC implementada em um ASIC (chamado de *hard* pelos autores) é apresentada por Abdelfattah e Betz (2014). Os resultados demonstram que o consumo de energia de uma NoC em um ASIC é, em média, quase a metade do consumo total em comparação com a rede configurada em um FPGA. Porém o gráfico que mostra os resutlados apresenta

um consumo extremamente baixo FPGA até 3 nós principais da rede sugerindo que para redes pequenas os FPGAs são indicados. Além disso, existe a flexibilidade da estrutura que indiscutivelmente é maior quando se utiliza lógica reconfigurável (ABDELFATTAH; BETZ, 2014).

Algumas tecnologias presentes em dispositivos específicos podem auxiliar no desempenho das NoCs. O funcionamento das NoCs desenvolvidas para FPGAs é usualmente baseado em memória, sugerindo que caso a estrutura de memória possa ser otimizada o resultado da rede também será. A técnica de organização de memória RAM em cascata implementada pela fabricante Xilinx, que apresenta uma forma otimizada de conectar a memória melhorando o tempo de acesso, foi utilizada e os resultados apresentados melhoraram o *troughput* da rede em 40% e o consumo de energia em 10% no pior caso (KAPRE, 2017).

O foco de implantação de NoC em sistemas de tempo-real foi demonstrado inicialmente com a modificação da rede Hoplite proposta anteriormente por uma versão RT (WASLY; PELLIZZONI; KAPRE, 2017) onde foram modificados a função de roteamento que passou a priorizar deflecções e a taxa de vazão dos pacotes. Em seguida um algoritmo de roteamento baseado em prioridade foi implementado para otimizar o sistema o que ocasiou a mudança para uma topologia de anel direcional com *bypasses* (RIBOT; NELISSEN, 2019). Outra modificação para melhorar o desempenho da rede Hoplite foi o Hoplitebuf (GARG et al., 2019) que adicionou estruturas de *buffers* FIFO em cada roteador para diminuir a deflexão de pacotes e ordená-los no próprio roteador chegando a atingir uma taxa de até 50% de viabilidade com o aumento de até 20% na injeção de pacotes.

Ringnet (Siast; Łuczak; Domański, 2019), uma alternativa para o design da Hopnet, apresenta uma comunicação totalmente baseada em memória o que, segundo os autores, torna a implementação mais voltada para FPGAs. Outros recursos implementados foram o controle de injeção de pacotes exclusivamente pelos elementos lógicos e memória distribuída em pequenos *buffers*. Apesar dos resultados do artigo serem classificados como bons pelos autores nenhuma comparação com a Hopnet em nenhuma de suas versões foi apresentada diretamente.

O artigo mais recente encontrado sobre a questão é sobre a HopliteBuf (GARG et al., 2020) no que parece ser uma continuação do trabalho anterior (GARG et al., 2019). Neste artigo são apresentados com mais profundidade os conceitos incorporados à implementação inicial da NoC Hoplite e o resultado de ocupação de 30 a 40% menor de LUTs do FPGA utilizado no trabalho.

OS trabalhos relacionados analisados indicam que a direção principal do desenvolvimento de NoCs para FPGAs tem como base a implementação da NoC Hoplite e modificações que visam solucionar os problemas da estrutura para aplicações em sistemas críticos de tempo-real.

6.7. Conclusão

As NoCs são estruturas já consolidadas nos projetos de SoCs e são uma área de pesquisa ampla em constante desenvolvimento dentro do contexto da computação de alto desempe-

nho. O início da área que conhecemos hoje pode ser considerado o início dos anos 2000 com a publicação de materiais completos sobre o assunto (BENINI; MICHELI, 2006) e a primeira edição da principal conferência sobre o assunto o IEEE/ACM International Symposium on Networks on Chip (NOCS) em 2007.

Diversas tecnologias de projeto, implantação, verificação e fabricação estão sendo utilizadas e testadas com objetivo de encontrar as melhores soluções possíveis para um problema extremamente complexo. Este trabalho apresentou os conceitos básicos relacionados às NoCs de uma maneira geral, em seguida uma visão do mesmo contexto e foco em computação reconfigurável e, por fim, uma análise superficial do estado-da-arte do desenvolvimento de NoCs em FPGAs.

Para um panorama mais amplo do estado-da-arte da área existem conferências especializadas no assunto como, por exemplo, o IEEE/ACM International Symposium on Networks on Chip (NOCS), o International Forum on MPSoC for Software-Defined Hardware (MPSoC), e também as conferências de hardware reconfigurável como International Conference on Field-Programmable Logic and Applications (FPL) e o ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). Revistas (*journals*) com foco em desenvolvimento de hardware, circuitos e sistemas também apresentam artigos interessantes sobre o assunto.

O futuro da utilização de NoCs em computação reconfigurável caminha para a disponibilização de redes já implementadas nas arquiteturas dos chips dos FPGAs. As principais empresas do ramo já perceberam que o caminho são os chips *multicore* híbridos com *hard-processors* conectados a blocos de lógica programável e dispositivos de memória. Então nada mais natural que o próximo passo seja a criação em hardware das interconexões necessárias para a implantação de NoCs e que as ferramentas de design permitam apenas a configuração destas estruturas sem a necessidade do desenvolvimento das mesmas *from scratch*.

Referências

ABDALLAH, A. B. *Advanced Multicore Systems-On-Chip: Architecture, On-Chip Network, Design*. [S.l.: s.n.], 2017. ISBN 978-981-10-6091-5. páginas 2, 3, 5, 6

ABDELFATTAH, M. S.; BETZ, V. Energy-efficient embedded nocs on fpgas. In: . [S.l.: s.n.], 2014. páginas 15

AGARWAL, A.; SHANKAR, R. Survey of network on chip (noc) architectures and contributions. *Journal of Engineering, Computing and Architecture*, v. 3, 01 2009. páginas 5, 6

BENINI, L.; MICHELI, G. D. *Networks on Chips. Technology and Tools*. [S.l.]: Morgan Kaufmann, 2006. ISBN 978-0-12-370521-1. páginas 5, 16

BERGER, A. S. *Embedded Systems Design: An Introduction to Processes, Tools and Techniques*. 1st. ed. [S.l.]: CMP Books, 2001. páginas 11

BOBDA, C. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. 1st. ed. [S.l.]: Springer Netherlands, 2008. páginas 11

CHEN, S.-J. et al. Reconfigurable networks-on-chip. 01 2012. páginas 3

de Lima, O. A. et al. A survey of noc evaluation platforms on fpgas. In: *2016 International Conference on Field-Programmable Technology (FPT)*. [S.l.: s.n.], 2016. p. 221–224. páginas 11, 12, 13

GARG, T. et al. Hoplitebuf: Fpga nocs with provably stall-free fifos. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: Association for Computing Machinery, 2019. (FPGA '19), p. 222–231. páginas 15

GARG, T. et al. Hoplitebuf: Network calculus-based design of fpga nocs with provably stall-free fifos. *ACM Trans. Reconfigurable Technol. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 13, n. 2, 2020. páginas 15

HAUCK, S.; DEHON, A. *Reconfigurable computing: the theory and practice of FPGA-based computation*. 1st. ed. [S.l.]: Morgan Kaufmann, 2008. (Systems on Silicon). páginas 11

IEEE. *Std. 1149.1 - Standard Test Access Port and Boundary-Scan Architecture*. [S.l.]: Official IEEE Std. 1149.1 Standard Working Group, 2012. páginas 14

KAPRE, N. Implementing fpga overlay nocs using the xilinx ultrascale memory cascades. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. [S.l.: s.n.], 2017. p. 40–47. páginas 15

Kapre, N.; Gray, J. Hoplite: Building austere overlay nocs for fpgas. In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. [S.l.: s.n.], 2015. p. 1–8. páginas 14

Monchiero, M. et al. Exploration of distributed shared memory architectures for noc-based multiprocessors. In: *2006 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. [S.l.: s.n.], 2006. p. 144–151. ISSN null. páginas 3

MONCHIERO, M. et al. Exploration of distributed shared memory architectures for noc-based multiprocessors. *Journal of Systems Architecture*, v. 53, n. 10, p. 719 – 732, 2007. páginas 2

PANDE, P. P. et al. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Trans. Comput.*, IEEE Computer Society, USA, v. 54, n. 8, p. 1025–1040, ago. 2005. ISSN 0018-9340. Disponível em: <https://doi.org/10.1109/TC.2005.134>. páginas 4

PEH, L.-S.; JERGER, N. E. *On-Chip Networks*. 1st. ed. [S.l.]: Morgan and Claypool Publishers, 2009. ISBN 1598295845. páginas 2, 3, 5, 8, 9, 10

RAUBER, T.; RüNGER, G. *Parallel Programming - for Multicore and Cluster Systems.* [S.l.: s.n.], 2010. ISBN 978-3-642-04817-3. páginas 9

RIBOT, Y.; NELISSEN, G. Design and implementation of an fpga-based noc for real time systems. In: *CISTER Research Centre Conference Paper*. [S.l.: s.n.], 2019. p. 1–3. páginas 15

Siast, J.; Łuczak, A.; Domański, M. Ringnet: A memory-oriented network-on-chip designed for fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 27, p. 1284–1297, 2019. páginas 15

TRACK, E.; FORBES, N.; STRAWN, G. The end of moore's law. *Computing in Science Engineering*, v. 19, n. 2, p. 4–6, 2017. páginas 3

TSAI, W.-C. et al. Networks on chips: Structure and design methodologies. *J. Electrical and Computer Engineering*, v. 2012, 01 2012. páginas 2, 7, 8, 10

Vasava, H. D.; Rathod, J. M. Software based distributed shared memory (dsm) model using shared variables between multiprocessors. In: *2015 International Conference on Communications and Signal Processing (ICCSP)*. [S.l.: s.n.], 2015. p. 1431–1435. páginas 9

WASLY, S.; PELLIZZONI, R.; KAPRE, N. Hoplitert: An efficient fpga noc for real-time applications. In: *2017 International Conference on Field Programmable Technology (ICFPT)*. [S.l.: s.n.], 2017. p. 64–71. páginas 15

WOLF, M. *Computers as Components: Principles of Embedded Computing System Design*. 5th. ed. [S.l.]: Morgan Kaufmann Publishers, 2016. páginas 11

Xu, P. et al. The research of distributed shared memory technology in power system. In: *2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. [S.l.: s.n.], 2017. p. 1309–1313. ISSN null. páginas 9