MINICURSOS

XX ERAD

ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL

De 14 a 16 de Abril | Joinville/SC

Fomento:

fapesc

Patrocínio Diamante:

LANIAQ scientific computing

NEC

Patrocínio Ouro:

Patrocínio Prata:

unisociesc

Realização:

Organização:

Colaboração:

ANDREA CHARÃO
MATHEUS S. SERPA

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

Porto Alegre
Sociedade Brasileira de Computação – SBC
2021

Dados Internacionais de Catalogação na Publicação (CIP)

E74 Escola Regional de Alto Desempenho da Região Sul (21. : 2021 : Joinville, SC)

Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul [recurso eletrônico] / Organizadores: Andrea Charão, Matheus Serpa. – Porto Alegre : SBC, 2021.

ISBN 978-65-87003-50-4

1. Computação – Evento. 2. Processamento de alto desempenho. I. Charão, Andrea. II. Serpa, Matheus. III. Universidade Federal de Santa Maria. IV. Universidade Federal do Rio Grande do Sul. V. Universidade do Estado de Santa Catarina. VI. Título.

CDU 004 (059)

Catalogação elaborada por Francine Conde Cabral
CRB-10/2606

ERAD/RS 2021

XXI Escola Regional de Alto Desempenho da Região Sul
14 a 16 de abril de 2021
Evento Online
http://labp2d.joinville.udesc.br/erad2021/

A XXI Escola Regional de Alto Desempenho da Região Sul (ERAD/RS 2021) foi planejada para ser uma edição histórica para a comunidade de Computação de Alto Desempenho (CAD) da Região Sul do Brasil, pois a escola romperia as fronteiras do Rio Grande do Sul pela primeira vez desde a sua criação. A cidade escolhida foi Joinville, no estado de Santa Catarina, que acolheria os pesquisadores no campus da Universidade do Estado de Santa Catarina (UDESC) entre os dias 14 e 16 de abril de 2021. Todavia, devido a pandemia de COVID-19, não foi possível realizar um evento presencial. Portanto, o evento ocorreu de forma online e com transmissão gratuita para toda a comunidade através do Youtube da ERAD/RS.

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 escola são alunos, profissionais e professores/pesquisadores que atuam direta ou indiretamente na área de Computação de Alto Desempenho (CAD) e em áreas correlatas. O evento engloba a região sul do Brasil (RS, SC e PR).

Os principais objetivos da escola 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 2021 foi composta por sessões técnicas com apresentações de 50 trabalhos nos fóruns de Iniciação Científica (IC) e de Pós-Graduação (PG). Além disso, o evento proporcionou aos participantes 3 palestras científicas, 5 palestras empresariais, 7 minicursos e uma maratona de programação paralela. Ainda, com intuito de incentivar a participação de mulheres na área de CAD, o evento abriu espaço para um workshop e roda de conversa sobre mulheres na área de CAD, promovido pelo Women in HPC (WHPC).

A edição de 2021 da escola foi coordenada pelos professores Maurício Aronne Pillon (UDESC), Márcio Castro (UFSC) e Claudio Schepke (UNIPAMPA). O fórum de IC foi coordenado pelos professores Odorico Mendizabal (UFSC) e Marco A. Zanatta Alves (UFPR). O fórum de PG foi coordenado pelos professores Guilherme P. Koslovski (UDESC) e Tiago Ferreto (PUCRS). Os minicursos foram coordenados pela professora Andrea Charão (UFSM) e pelo doutorando Matheus Serpa (UFRGS). A maratona de programação paralela foi coordenada pelos professores Dalvan Griebler (PUCRS/SETREM) e João Vicente F. Lima (UFSM). A equipe organizadora local foi coordenada pelos professores Charles Christian Miers (UDESC) e Ricardo Pfitscher (UFRGS/UNISOCIESC).

Índice

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

Mensagem da Coordenação Geral

É com imenso prazer e entusiasmo que saudamos e damos as boas vindas à vigésima primeira edição da Escola Regional de Alto Desempenho da Região Sul (ERAD/RS 2021), que neste ano aconteceu de forma virtual entre os dias 14 e 16 de abril de 2021. 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.

Coordenar e organizar um evento da magnitude da ERAD/RS é, por si só, um grande desafio. Trazê-lo, pela primeira vez na história do evento, para fora do Estado do Rio Grande do Sul, fez com que nossa responsabilidade aumentasse ainda mais. Por fim, organizá-lo e executá-lo em meio a uma pandemia ampliou ainda mais os riscos e as preocupações envolvidas em todo o processo.

É inegável que a impossibilidade imposta pela pandemia de realizarmos a primeira ERAD/RS fora do estado do Rio Grande do Sul nos gerou certa frustração. Afinal, gostaríamos de ter acolhido os estudantes e pesquisadores da área de Computação de Alto Desempenho (CAD) em Joinville, a cidade mais populosa do estado de Santa Catarina. Porém, durante a organização do evento, nós coordenadores não medimos esforços para fazer um evento virtual de qualidade dentro das limitações impostas pela crise na saúde que afetou a vida de todos nós.

Neste ano, a ERAD/RS contou com fomento governamental da FAPESC, além do patrocínio na categoria diamante das empresas NEC, LANIAQ e de uma parceria conjunta entre DELL e NVIDIA. Ainda, o evento foi patrocinado pela SDC (patrocínio ouro) e pela UNISOCIESC (patrocínio prata).

Agradecemos, em especial, a todos os(as) autores(as) que submeteram seus trabalhos às sessões técnicas, às empresas patrocinadoras e aos diversos convidados que aceitaram prontamente nosso convite e, sem dúvida alguma, engrandeceram a ERAD/RS com as suas participações.

Por fim, agradecemos todos os coordenadores dos eventos da ERAD/RS, que tomaram para si diversos encargos e os conduziram com muito sucesso.

Obrigado pela presença de todos.

Esperamos que a ERAD/RS 2021 seja muito proveitosa.

Um abraço,
Maurício Pillon (UDESC), Márcio Castro (UFSC) e Claudio Schepke (UNIPAMPA)
Coordenadores gerais da ERAD/RS 2021

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 e um anos, foi realizada entre os dias 14 e 16 de abril de 2021, na cidade de Joinville/SC, no campus sede da Universidade do Estado de Santa Catarina (UDESC). Essa foi a primeira vez que a escola foi realizada fora do estado do Rio Grande do Sul.

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, intermediários e avançados em tópicos de interessa à comunidade. Não diferente, neste ano, foram selecionados seis minicursos, dos quais todos viraram capítulos para este livro. Os minicursos aqui representados, apresentam tópicos de ponta da área de PAD, os quais irão certamente contribuir e 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!

Andrea Charão (UFSM) e Matheus S. Serpa (UFRGS)
Coordenadores dos Minicursos da ERAD/RS 2021

Comitês Organizadores

Coordenação Geral

- Maurício Aronne Pillon (UDESC – Joinville)
- Márcio Castro (UFSC – Florianópolis)
- Claudio Schepke (UNIPAMPA – Alegrete)

Coordenação Local

- Charles Christian Miers (UDESC – Joinville)
- Ricardo Pfitscher (UFRGS/UNISOCIESC)

Fórum de Pós-Graduação

- Guilherme Piêgas Koslovski (UDESC – Joinville)
- Tiago Ferreto (PUCRS)

Fórum de Iniciação Científica

- Odorico Mendizabal (UFSC)
- Marco A. Zanatta Alves (UFPR)

Minicursos

- Andrea Charão (UFSM)
- Matheus Serpa (UFRGS)

Minicursos

Minicurso 1

Desvendando o Uso de Contadores de Hardware para Otimizar Aplicações de Inteligência Artificial 1
Valéria Girelli (UFRGS), Félix Dal Pont Michels (UFRGS), Francis Birck Moreira (UFPR), Philippe Olivier Alexandre Navaux (UFRGS)

Minicurso 2

Otimização de Programas Paralelos com uso do OpenACC 30
Evaldo B. Costa (UFRJ), Gabriel P. Silva (UFRJ)

Minicurso 3

Are you root? Experimentos Reprodutíveis em Espaço de Usuário 70
Jessica Dagostini (UFRGS), Vinicius Garcia Pinto (UFRGS), Lucas Nesi (UFRGS), Lucas Mello Schnorr (UFRGS)

Minicurso 4

Além de Simplesmente: #pragma omp parallel for 86
João Vicente Ferreira Lima (UFSM), Natiele Lucca (UNIPAMPA), Claudio Schepke (UNIPAMPA)

Minicurso 5

Ambiente de Nuvem Computacional Privada para Teste e Desenvolvimento de Programas Paralelos 104
Anderson Maliszewski (UFRGS), Adriano Vogel (PUCRS), Dalvan Griebler (PUCRS/SETREM), Claudio Schepke (UNIPAMPA), Philippe Olivier Alexandre Navaux (UFRGS)

Minicurso 6

Desenvolvimento de Aplicações Baseadas em Tarefas com OpenMP Tasks 129
Lucas Leandro Nesi (UFRGS), Vinícius Garcia Pinto (UFRGS), Marcelo Cogo Miletto (UFRGS), Lucas Mello Schnorr (UFRGS)

Capítulo 1

Desvendando o Uso de Contadores de *Hardware* para Otimizar Aplicações de Inteligência Artificial

Valéria S. Girelli
vsgirelli@inf.ufrgs.br
Grupo de Processamento Paralelo e Distribuído (GPPD)
Universidade Federal do Rio Grande do Sul (UFRGS)
Sala 209, Prédio 67, Instituto de Informática - Campus do Vale
91501-970 - Porto Alegre - RS - Brasil

Félix D. P. Michels
felix.junior@inf.ufrgs.br
Grupo de Processamento Paralelo e Distribuído (GPPD)
Universidade Federal do Rio Grande do Sul (UFRGS)
Sala 201, Prédio 67, Instituto de Informática - Campus do Vale
91501-970 - Porto Alegre - RS - Brasil

Francis B. Moreira
fbm@inf.ufpr.br
High Performance Systems (HiPES)
Universidade Federal do Paraná (UFPR)
Sala 84, Departamento de Ciência da Computação - Centro Politécnico
81531-980 - Curitiba - PR - Brasil

Philippe O. A. Navaux
navaux@inf.ufrgs.br
Grupo de Processamento Paralelo e Distribuído (GPPD)
Universidade Federal do Rio Grande do Sul (UFRGS)
Sala 210, Prédio 67, Instituto de Informática - Campus do Vale
91501-970 - Porto Alegre - RS - Brasil

Resumo

O desempenho dos sistemas computacionais aumentou consideravelmente nas últimas décadas. Tal avanço se deu por meio de mecanismos que nem sempre são visíveis para o usuário final, como o sistema de memória e o sistema de prefetching, que possuem grande impacto no desempenho de processadores modernos. Ao mesmo tempo, algoritmos de Inteligência Artificial (IA) se tornam cada vez mais relevantes em diversas áreas da computação e da sociedade, e requerem um crescente poder computacional. Com isso, para se obter o máximo desempenho dessas aplicações, é necessário garantir que as mesmas estejam utilizando da melhor forma esses recursos. Para auxiliar na avaliação da utilização desses mecanismos transparentes ao usuário, muitos processadores e aceleradores modernos fornecem contadores de hardware, estruturas que permitem o monitoramento de eventos internos, como o número de acessos à memória e a porção de dados encontrados em cada nível de memória. Portanto, neste capítulo abordaremos a utilização de contadores das arquiteturas Intel Xeon Cascade Lake e NEC SX-Aurora TSUBASA para analisar o desempenho das cada vez mais frequentes aplicações de IA. Por meio das ferramentas Linux perf e NEC FTRACE é possível acessar esses contadores e utilizar os resultados para identificar gargalos nessas aplicações.

1.1. Introdução

Aplicações de Inteligência Artificial (IA) vêm ganhando cada vez mais relevância nos mais diversos espaços da sociedade (A et al., 2019). De jogos como Xadrez e Poker, ao tratamento de doenças como câncer, análise de mudanças climáticas, reconhecimento visual, de fala e detecção de fraudes em transações bancárias, são quase incontáveis os campos do nosso dia a dia nos quais a IA tem se introduzido. Diversos trabalhos também propõem a aplicação de IA para aprimorar o desempenho de sistemas computacionais, como mecanismos de *prefetching* (Liao et al., 2009; Peled et al., 2015; BHATIA et al., 2019) e de predição de desvio (Zangeneh et al., 2020; Zhang et al., 2020), que fazem uso de heurísticas na tomada de decisão.

A crescente relevância da área é acompanhada pelo aumento na quantidade de dados sobre os quais as aplicações de IA trabalham. A quantidade de dados e informações digitais no mundo hoje ultrapassa os 44 trilhões de gigabytes. Dessa forma, surge a necessidade de um poder computacional cada vez maior, e muitas empresas migram seus serviços para grandes servidores de processamento de dados com milhares de núcleos e centenas de GPUs em busca de tempos de execução menores. Com a utilização desses sistemas computacionais, surge também a preocupação com o consumo energético e com a refrigeração. É necessário, portanto, garantir que aplicações de Inteligência Artificial estejam utilizando eficientemente os recursos computacionais a sua disposição, e desenvolvedores e desenvolvedoras das diversas bibliotecas voltadas à IA empregam grandes esforços na otimização de suas ferramentas.

Além de aplicações de IA, diversas aplicações de *High-Performance Computing* (HPC) fazem uso desses sistemas equipados com centenas de núcleos e GPUs. A importância de se alcançar o mais alto desempenho possível nessas aplicações levou a uma grande variedade de ferramentas de análise de desempenho. Tais ferramentas permitem a identificação de comportamentos que levam à redução de desempenho e auxiliam no

desenvolvimento de otimizações nas aplicações. Exemplos dessas ferramentas são o *framework* HPCToolkit (ADHIANTO et al., 2009), Periscope (GERNDT; FÜRLINGER; KEREKU, 2005), o projeto TAU (SHENDE; MALONY, 2006), Vampir (KNÜPFER et al., 2008) e Score-P (MEY et al., 2012). No entanto, as informações providas por esses mecanismos são geralmente de mais alto nível e menos detalhadas. Com isso, a identificação de gargalos provenientes da utilização ineficiente dos componentes disponíveis na arquitetura se torna mais desafiadora. Além disso, por serem ferramentas complexas e que coletam informações em diferentes níveis do sistema ao mesmo tempo, o *overhead* e ruído sobre a execução das aplicações tendem a ser altos.

Uma alternativa a essas ferramentas é a utilização de contadores de *hardware*, estruturas encontradas em muitos processadores e aceleradores modernos que permitem o monitoramento de eventos internos a essas arquiteturas. Alguns desses eventos são o número de instruções executadas, o número de ciclos, o número de acessos à memória, dentre outros. Por meio da utilização desses contadores é possível realizar a coleta de informações de forma mais específica e detalhada se comparado às informações obtidas com outras ferramentas de mais alto nível. O usuário tem a possibilidade de identificar as informações que estão disponíveis para sua arquitetura específica e combinar diferentes contadores afim de investigar aspectos distintos. Além disso, contadores de diferentes núcleos também podem ser combinados, analisando-se o sistema como um todo. Dessa forma, utilizar contadores de *hardware* para analisar o desempenho de aplicações paralelas permite que o usuário tenha mais controle sobre o processo, com menos ruído sobre a aplicação.

Portanto, ao longo deste capítulo iremos estudar aspectos de duas arquiteturas distintas, alguns de seus contadores de *hardware* e como utilizar ferramentas de *profiling* para acessar essas informações. Com base nisso, será possível analisar o desempenho de aplicações de Inteligência Artificial e propor otimizações em seus códigos.

O capítulo está organizado da seguinte maneira: na Seção 1.2 é feita uma introdução à arquitetura de computadores, demonstrando conceitos como *pipeline*, arquiteturas superescalares, entre outros. Em seguida, na Seção 1.3 temos uma discussão sobre hierarquia de memória e sistema de *prefetching*. A Seção 1.4 é apresentada uma breve explicação sobre vetorização, exemplificando com a arquitetura vetorial NEC SX-Aurora TSUBASA. A Seção 1.5 expõe os ambientes de execução utilizados na elaboração desse minicurso. A Seção 1.6 apresenta os contadores de *hardware* e a utilização das ferramentas Linux perf e o NEC FTRACE. Na Seção 1.7, conceitos de Inteligência Artificial e Aprendizado de Máquina são apresentados, bem como a aplicação utilizada neste minicurso. Por fim, na Seção 1.8 utilizamos exemplos práticos aplicados em uma implementação de retro-propagação, otimizando-a por meio da utilização de contadores de *hardware*, seguindo para a conclusão deste curso, na Seção 1.9.

1.2. Arquitetura de Computadores

A rápida evolução na área de arquitetura de computadores é baseada em três fatores relacionados: tamanho dos componentes, paralelismo entre componentes, e especulação (HENNESSY; PATTERSON, 2017). O tamanho dos componentes, ou tamanho do processo de fabricação, define a largura em nanômetros dos transistores do sistema. Quanto menor

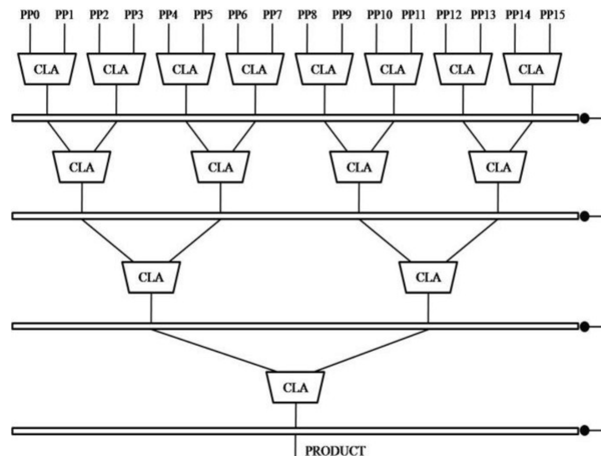


Figura 1.1. Multiplicador de 32 bits com carry pré-calculado. Fonte: Bokade et al. (BOKADE; DAKHOLE, 2016), pág. 5.

o transistor, mais *hardware* conseguimos colocar dentro do *chip* do processador ou memória, menor a latência de fio dos componentes e entre os mesmos, e portanto maior a frequência de operação (FLOYD, 2010). Já o paralelismo implica em usar mais *hardware* para produzir mais trabalho no mesmo período de tempo, sendo usado em todos os níveis de arquitetura. Um exemplo claro é a diferença existente entre um somador por propagação e um somador com *carry* pré-calculado. O somador normal precisa de um tempo igual à soma dos tempos de todos somadores já que cada somador depende do resultado do somador predecessor. Já o somador com *carry* pré-calculado utiliza *hardware* adicional para calcular *carries* em paralelo, permitindo que todos os somadores terminem seu serviço em paralelo. Este paralelismo pode ser considerado o paralelismo a nível de "operação", onde torna-se uma operação mais eficiente ao adicionar *hardware* para tal fim (FLOYD, 2010).

1.2.1. *Pipeline*

Uma técnica muito utilizada para paralelismo é a técnica de *pipeline* (HENNESSY; PATTERSON, 2017). Como operações de multiplicação e divisão são muito demoradas, é comum dividir elas em vários estágios com o uso de registradores intermediários, como na Figura 1.1. Assim é possível reduzir o caminho crítico do sistema e manter uma frequência de operação mais alta. Com isso, tem-se capacidade de realizar mais operações em menos tempo, pois conforme uma instrução termina um estágio, este está pronto para começar uma nova instrução. A ideia de *pipeline* foi adotada em larga escala para todo o funcionamento do processador. Processadores antigos utilizam de frequência mais baixa e um conjunto grande de instruções complexas, recebendo a caracterização de CISC (*Complex Instruction Set Computer*) (ISEN; JOHN; JOHN, 2009). Em 1981, Hennessy desenvolveu o processador MIPS (*Microprocessor without Interlocked Pipeline Stages*) (HENNESSY et al., 1982), que implementa um conjunto pequeno de instruções simples, permitindo uma grande inovação: o uso de *pipeline* para a operação de todas as instruções. Hoje este tipo de computador é conhecido como RISC (*Reduced Instruction Set Computer*) (ISEN; JOHN; JOHN, 2009).

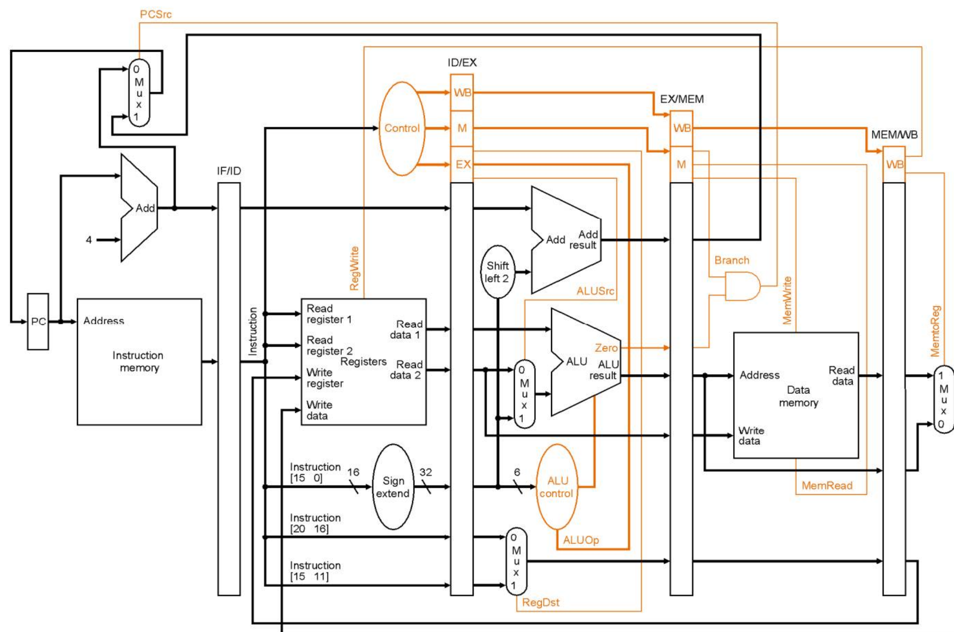


Figura 1.2. *pipeline* da arquitetura MIPS. Fonte: Hennessy & Patterson (PATTERSON; HENNESSY, 2004), pág. 387

A ideia, demonstrada na Figura 1.2, é separar as instruções em 5 estágios: busca (IF), decodificação (DEC), execução (EXE), acesso à memória (MEM), e escrita (WB). Assim, quando uma instrução termina de executar um estágio e passa para o próximo, o processador já pode receber uma nova instrução. Isto permite um paralelismo a nível de operação do processador, pois temos múltiplas instruções no processador, utilizando seus diferentes estágios, embora realisticamente ainda entregaremos no máximo uma instrução por ciclo. O *pipeline* também permite um grande aumento na frequência ao reduzir o caminho crítico para o tempo de um estágio, e faz melhor uso dos recursos do processador, pois boa parte dos recursos ficava inativa enquanto as instruções não estivessem usando elas. Um dos problemas do *pipeline* é que existem dependências entre certas operações como saltos e leitura após escrita, onde é necessário inserir bolhas no *pipeline* para operações terem os valores corretos. Assim o número de instruções entregues por ciclo geralmente é menor que um, o que é resolvido através de técnicas de especulação, como predição de salto (YEH; PATT, 1991).

1.2.2. Arquiteturas Superescalares

Neste mesmo princípio, nota-se que várias unidades funcionais ficam inativas quando outra operação está sendo executada. Assim, desenvolveu-se a arquitetura superescalar (THORNTON, 1980), com o propósito de aumentar o paralelismo a nível de instrução (*instruction level parallelism* – ILP). A arquitetura de *pipeline* superescalar é basicamente uma arquitetura *pipeline* de maior capacidade: o processador é capaz de buscar múltiplas instruções em um ciclo, decodificar múltiplas instruções em um ciclo, mandar múltiplas instruções para execução nas diferentes unidades funcionais, mandar múltiplas requisições para a memória, e fazer múltiplas escritas nos registradores. Uma arquitetura

superescalar não é necessariamente uma arquitetura *pipeline*, pois as técnicas são distintas (HENNESSY; PATTERSON, 2017). Para o correto funcionamento, são necessárias várias adições ao processador, como *buffers* entre os estágios, a adição de estágios como a renomeação de registradores (o qual elimina dependências falsas entre as instruções) e o estágio de despacho (o qual possui lógica inteligente adicional para despachar instruções na melhor ordem conforme o estado do processador), um *buffer* de reordenação de instruções (*reorder buffer* - ROB), *buffers* de ordem para requisições à memória (*memory order buffer* - MOB), entre outros (FOG, 2012).

1.2.3. Threading

A adição de tantas estruturas para melhorar o desempenho de um *pipeline* superescalar inseriu problemas nas arquiteturas modernas. Problemas como a espera por resolução de saltos e por acessos à memória limitam o número de instruções alcançadas em testes reais. O número de instruções por ciclo (*instructions per cycle* – IPC) na maior parte dos programas não passa de 2, apesar de processadores dimensionados para mais de 4 instruções em paralelo. Para dados contíguos, adotou-se vetorização, que é o paralelismo no nível de operação da instrução, a qual agora recebe mais dados para uma operação. Para programas mais complexos, criou-se um novo nível de paralelismo: o paralelismo de *threads* (*Thread Level Parallelism* – TLP). Agora, o processador é capaz de escalonar mais de um fluxo de programa ao manter múltiplos contadores de programa (*Program Counter* - PC) e conjuntos de registradores lógicos para representar o estado de diferentes *threads* (fios). Este suporte ocorre em dois níveis: arquiteturas *multithreaded* e arquiteturas multi-core.

Em arquiteturas *multithreaded*, o núcleo de processamento possui a capacidade de dividir os seus recursos entre múltiplas *threads* (TULLSEN; EGGERS; LEVY, 1995). O *buffer* de reordenamento, as estações de espera por execução em cada unidade funcional, o banco de registradores, e todas as outras estruturas de controle são divididas entre as múltiplas *threads* para que todas ocupem de forma eficiente as unidades funcionais do core. Entre as formas de *multithreading* podemos citar:

- *Multithreading* entrelaçado, no qual o núcleo busca uma instrução de cada *thread* a cada ciclo;
- *Multithreading* em bloco, onde a cada período de ciclos o processador busca várias instruções da mesma *thread*, e troca a *thread* quando acaba o período;
- *Multithreading* simultâneo, hoje adotado pela Intel como *Hyperthreading* (MARR et al., 2002), onde instruções de todas as *threads* são buscadas no mesmo ciclo.

Em arquiteturas multi-core, possuímos múltiplos *cores*, os quais podem ter apenas uma *thread*, ou podem ser *multithreaded* (BLAKE; DRESLINSKI; MUDGE, 2009). O sistema operacional encarrega-se de gerenciar os recursos dos múltiplos *cores*, permitindo o processamento paralelo e o mapeamento da execução conforme desejado nos *cores* disponíveis.

1.2.4. Especulação

Através da evolução da arquitetura, vários problemas foram identificados na busca de execução eficiente. Mecanismos de especulação tentam contornar estes problemas ao prever o comportamento da aplicação em relação a eles. Por exemplo, a latência da memória se tornou cada vez maior devido à evolução muito mais rápida do processador, tornando o acesso a instruções e dados um gargalo. Para diminuir a latência da memória, a memória *cache* foi adotada, a qual serve como armazenamento temporário para linhas de memória, com tamanho e latência muito menores (JACOB; WANG; NG, 2010). A primeira premissa da memória *cache* é de que um dado recentemente usado será reusado em breve, o que é uma especulação ou predição em relação ao comportamento da aplicação, onde assume-se localidade temporal. A segunda premissa comum em memória *cache* ao usar linhas longas com múltiplos endereços é de que dados contíguos serão usados em um curto espaço de tempo, portanto assume-se localidade espacial, o que é praticamente sempre válido para acessos à instruções na memória, por exemplo. Entre outros mecanismos de especulação, podemos citar a desambiguação de leituras, a predição de saltos, *prefetching*, *buffers* de linha na memória principal (*Dynamic Random-Access Memory* – DRAM) (JACOB; WANG; NG, 2010), entre outros.

1.3. Hierarquia de Memória e Sistema de *Prefetching*

Em processadores modernos, uma hierarquia de *cache* em três níveis é comumente usada (MORGAN, 2017; CUTRESS, 2017). Nessa configuração, o primeiro nível de *cache* de dados (L1) e o segundo nível de *cache* de dados (L2) são normalmente privados a cada núcleo do processador. Esses níveis de *cache* são mais próximos fisicamente do processador, possuem menor capacidade de armazenamento, e permitem acesso aos dados de forma mais eficiente. Um terceiro nível de *cache* (L3, também conhecida por *Last Level cache* – LLC) é compartilhada entre todos os núcleos do processador. Seu tempo de resposta é frequentemente maior que o tempo de resposta dos níveis de *cache* privados, mas com a vantagem de permitir uma capacidade de armazenamento maior. Uma representação dessa hierarquia é observada na Figura 1.3.

Quando o processador emite uma requisição por um dado na memória, diversas situações podem ocorrer. Inicialmente, a requisição é entregue à *cache* L1, que é relativamente pequena (32 KiB) e possui uma baixa latência de acesso (4 ciclos de processador) (FOG, 2012; HENNESSY; PATTERSON, 2017). Caso o dado seja encontrado nesse nível de *cache* ele é rapidamente entregue ao processador. No entanto, devido à capacidade limitada da *cache* L1, por muitas vezes os dados não estão presentes, e a busca pelo dado é repetida no próximo nível de *cache*. Cada vez que um dado não é encontrado em um nível de *cache* tem-se um *cache miss* e a necessidade de repetir o procedimento de busca em um nível mais distante do processador, cujo tempo de acesso é maior (e somado aos tempos de acesso dos níveis de memória predecessores). Portanto, encontrar os dados solicitados em níveis de *cache* mais próximos do processador é preferível, caso contrário, a hierarquia da memória pode se tornar um grande gargalo para o desempenho das aplicações (BAKHSHALIPOUR et al., 2019).

Cabe ressaltar, no entanto, que diversas outras ações são necessárias juntamente com o processo de busca por dados descritos acima, como por exemplo o acesso à *Trans-*

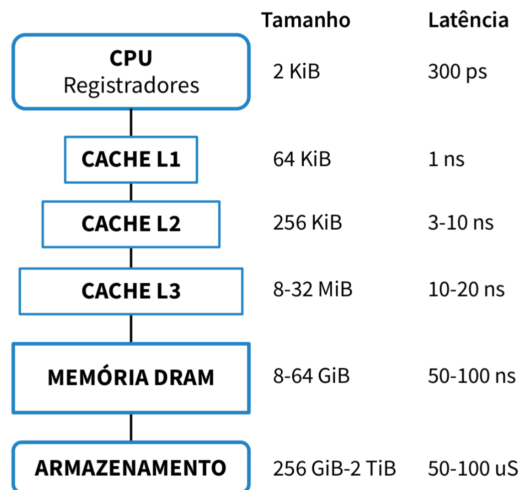


Figura 1.3. Exemplo da hierarquia de memória de um processador moderno. Conforme nos afastamos do processador, as memórias se tornam maiores e mais lentas.

lation Lookaside Buffer (TLB) (onde é possível verificar se a página de dados está fisicamente presente na memória principal e a tradução do endereço virtual para o físico (HENNESSY; PATTERSON, 2017)), possíveis acessos à memória principal para consulta da tabela de páginas (um procedimento custoso devido à alta latência da memória DRAM), e ainda transferências de dados entre duas *caches* de diferentes núcleos do sistema devido à ação do protocolo de coerência de *cache*.

Nos últimos anos, várias melhorias no desempenho do processador têm sido observadas, como o aumento do número de núcleos – o que requer memórias com maior largura de banda de transferência de dados para lidar com as requisições de dados emitidas por esses diversos núcleos, e a capacidade do processador de requisitar vários dados por ciclo (*multiple issue*) (HENNESSY; PATTERSON, 2017). No entanto, as tecnologias de memória não melhoraram tanto quanto os processadores, criando uma lacuna de desempenho referida na literatura como *Memory Wall* (WULF; MCKEE, 1996). Vários problemas podem surgir dessa disparidade de desempenho. Por exemplo, se uma instrução for um *load* (requisição de dado para leitura) e seus dados necessários não forem entregues rapidamente pelo sistema de memória, a execução dessa instrução e das instruções dependentes a ela podem ser interrompidas (HENNESSY; PATTERSON, 2017). Para evitar tais paralisações, deve-se reduzir o número de ciclos desde o momento em que o processador emite uma requisição até o momento em que pode realmente usar os dados deve ser o menor possível. Além disso, dada a natureza de *multiple issue* dos processadores modernos, um grande número de solicitações de memória pode ser emitido em apenas alguns ciclos, possivelmente criando contenção em algum nível da hierarquia de memória.

Diante desses vários problemas, o *prefetcher* foi criado para mitigar a latência da memória (BAER; CHEN, 1991). *Prefetching* é uma técnica implementada em *hardware* que visa prever quais serão os próximos endereços de memória a serem solicitados pelo processador. Ao monitorar as solicitações de memória anteriores, o *prefetcher* é capaz

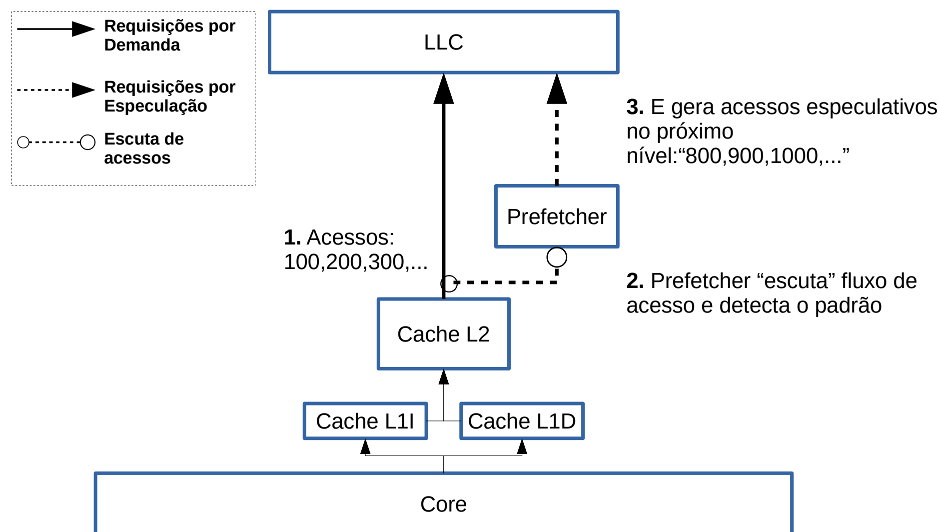


Figura 1.4. Abstração do comportamento de um *prefetcher*.

de identificar possíveis padrões de acesso. Com base nesses padrões, ele especula quais podem ser os próximos endereços a serem solicitados e, em seguida, realiza requisições com antecedência, antes que o processador realmente precise dos dados. Assim, quando o dado for finalmente solicitado pelo processador, ele já estará em níveis de *cache* mais próximos (HENNESSY; PATTERSON, 2017). A latência da memória principal acima mencionada é, portanto, ocultada por outras instruções anteriores à instrução que de fato realizou a requisição dos dados buscados pelo *prefetcher*.

Com os dados já em níveis mais próximos, (i) a crítica *load-to-use latency* pode ser reduzida (KANG; WONG, 2013; Guttman et al., 2015), e (ii) uma importante métrica de desempenho é melhorada, a taxa de acertos da *cache*, também conhecida como *cache hit*. A taxa de *hits* representa a porção de requisições que são encontradas em um determinado nível de *cache* sem a necessidade de se aprofundar na hierarquia de memória – que, consequentemente, resultaria em um tempo de execução maior. Esses ganhos de desempenho permitiram que os *prefetchers* se tornassem um mecanismo predominante nas arquiteturas atuais (MORGAN, 2017; CUTRESS, 2017; FOG, 2012; GIRELLI et al.,). Exemplos de padrões identificados por mecanismos de *prefetcher* comuns são *stride* (CHEN; BAER, 1995) e *stream* (LE et al., 2007).

A Figura1.4 mostra um exemplo de *prefetcher* da *cache* L2 detectando um padrão de acesso *stride*. A *cache* L2 encaminha requisições para a LLC (mostrado na Figura1.4 como o evento **1**). O *prefetcher* da L2, por sua vez, intercepta essas solicitações "escutando"a interconexão da *cache* (evento**2**) e identificando o padrão de acesso que está sendo gerado. Com base no padrão identificado, requisições especulativas são inseridas no *Miss Status Holding Register* (MSHR) da *cache* L2 (**3**), um *buffer* que mantém o controle de eventos de *miss* que ainda precisam ser tratados. Essas requisições especulativas inseridas no MSHR da L2 são feitas primeiramente à *cache* L2 para evitar a busca redundante de um dado que já reside na L2. Esses acessos são vistos como solicitações regulares feitas à L2 pelo *prefetcher*, de modo que a L2 não precisa realmente encaminhar a resposta para a

L1. Se o endereço especulado ainda não estiver presente na L2, a L2 encaminha a requisição por dado para os próximos níveis da hierarquia, como em um acesso normal. Assim, quando o processador precisar de um dado solicitado previamente pelo *prefetcher*, ele já estará em um nível de *cache* mais próximo (neste caso, a *cache* L2).

1.4. Vetorização

A característica chave das arquiteturas vetoriais é seu modelo *Single Instruction Multiple Data* (SIMD). Em processadores superescalares, o dado padrão é uma palavra de normalmente 32 *bits* sobre a qual um conjunto de instruções atua de maneira individual. Já em uma arquitetura vetorial, uma instrução vetorial é aplicada simultaneamente sobre uma coleção de palavras em formato de vetor (HENNESSY; PATTERSON, 2017). Desse modo, tem-se a execução de uma mesma instrução (*single instruction*) sobre múltiplos dados (*multiple data*) em um único ciclo. Por conta disso, processadores vetoriais têm a vantagem de que cada dado é independente entre si, o que permite que a mesma instrução seja realizada sobre todos eles ao mesmo tempo.

Por operar em um número maior de dados de uma única vez, instruções vetoriais resultam em menos buscas por dados e menos *branches* quando os dados estão contíguos. Com isso é possível reduzir-se o número de erros de predição e a latência de acesso à memória, favorecendo o tempo de execução de uma aplicação(KSHEMKALYANI, 2012). Porém, essa vantagem é relevante apenas quando há blocos de memória suficientemente grandes, onde haveria uma grande latência de acesso a memória em um processador escalar tradicional.

Um exemplo de arquitetura vetorial é a SX-Aurora TSUBASA da empresa NEC Corporation. Esse processador possui 8 núcleos de processamento executando com frequência de 1,408 GHz e 3 níveis de memória *cache* (KOMATSU et al., 2018). Uma das vantagens dessa arquitetura em relação as outras existentes é o tamanho das unidades vetoriais da mesma, podendo chegar a 256 elementos de 64 bits. Além disso, o compilador da NEC (NEC, 2020a) toma decisões automaticamente , identificando áreas nas quais é possível gerar código vetorizável sem que haja a necessidade de alteração do código fonte. Entretanto, o compilador ainda necessita de ajuda do programador para facilitar a interpretação do código. O programador pode utilizar diretrizes específicas e utilizar técnicas de otimização como *loop unrolling* e *inlining*.

1.5. Ambiente de Execução

O ambiente de execução utilizado nos experimentos estão presentes na infraestrutura PCAD[1], no INF/UFRGS. Na Tabela 1.1 tem-se as características do *Vector Engine* (VE) TSUBASA. A máquina vetorial consiste em um ambiente com 8 cores, 48GB de memória global a 900 MHz e *cache* L3 compartilhada de 2 MB. Cada core possui memórias *caches* privadas, L1 de instrução e dados de 32KB cada e L2 de 256KB, uma unidade de processamento escalar (*Scalar Processing Unit* – SPU) e uma unidade de processamento vetorial (*Vector Processing Unit* – VPU), sendo que cada VPU contém *load buffer*, *store buffer*, e 32 *pipelines* paralelos vetoriais (*Vector Parallel Pipeline* – VPP) (NEC, 2020b). A maior parcela dos cálculos é feita pela VPU, sendo a SPU responsável pelos traba-

[1]<http://gppd-hpc.inf.ufrgs.br>

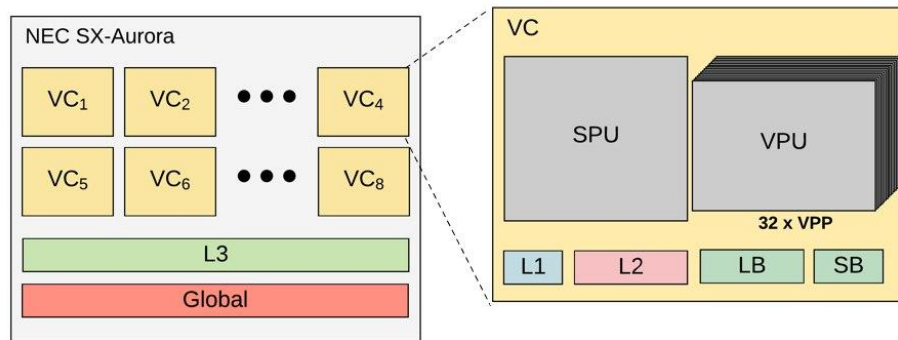


Figura 1.5. Ambiente de execução SX-Aurora.

Tabela 1.1. Arquitetura SX-Aurora (Vector Engine Type 10BE).

Processador	8 *cores* @ 1408 MHz
Microarquitetura	SX-Aurora
Cache	8 X 32 KB L1I; 8 X 32 KB L1D; 8 X 256 KB L2; 8 X 2 MB L3
Memória	HBM2 48 GB, 900 MHz

lhos sequenciais, bem como as tarefas do sistema operacional. Na figura 1.5 temos uma representação gráfica desses componentes.

Já o *Vector Host* (VH) é representado pela microarquitetura Intel Cascade Lake. Na Tabela 1.2 tem-se as especificações do processador Intel Xeon Gold 6226, que possui 12 núcleos operando a uma frequência entre 2.7 GHz e 3.7 GHz. Cada núcleo possui 32 KB de *cache* L1 de dados e instruções, bem como uma cache L2 também privada de 1 MB. A L3, compartilhada entre todos os núcleos, possui capacidade de 16,5 MB, e a máquina ainda apresenta 192 GB de memória DRAM. A microarquitetura Cascade Lake é bastante semelhante à sua predecessora Skylake, porém a Cascade Lake introduz suporte à instruções de Rede Neural Vetorial AVX-512 (*AVX-512 Vector Neural Network Instructions* – VNNI) (PEREZ et al., 2018).

1.6. Contadores de Hardware

Após compreendermos melhor o funcionamento de arquiteturas superescalares e vetoriais e alguns de seus pontos de perda de desempenho, agora é necessário compreender-

Tabela 1.2. Microarquitetura Cascade Lake (Xeon Gold 6226).

Processador	12 *cores* @ 2700 - 3700 MHz;
Microarquitetura	Cascade Lake
Cache	12 X 32 KB L1I; 12 X 32 KB L1D; 12 X 1 MB L2; 16,5 MB L3
Memória	DDR4 192 GB, 2933 MHz

mos como obter informações a respeito do *hardware* no momento da execução de uma aplicação afim de melhorarmos o desempenho de uma aplicação. Arquiteturas modernas permitem a utilização de contadores de *hardware*, estruturas internas que permitem o monitoramento de eventos da execução e funcionamento do processador ou acelerador. Como já citado anteriormente, diversas ferramentas de *profiling* foram desenvolvidas que permitem acesso a contadores de *hardware*. Neste capítulo, nós apresentamos os contadores existentes na arquitetura Intel Cascade Lake e no processador vetorial SX-Aurora TSUBASA e como acessá-los por meio das ferramentas Linux perf e NEC FTRACE, respectivamente.

1.6.1. Linux perf

Linux perf é uma ferramenta de *profiling* para sistemas Linux que permite acesso à interface `perf_events` de eventos de desempenho presentes em arquiteturas superescalares. A utilização da ferramenta é feita por linha de comando por meio do comando `perf`, e é possível listar os eventos disponíveis na plataforma por meio do comando `perf list`. Um comando comumente utilizado para análise de desempenho é o comando `stat`, que permite a coleta de informações referentes à execução de uma aplicação. Com a utilização da opção `-e` é possível listar os eventos que se deseja coletar. O exemplo abaixo representa a linha de comando necessária para a coletar informações a respeito do número de instruções e do número de ciclos de CPU necessários para a execução do comando `ls`:

```
1 perf start -e instructions,cpu-cycles ls
```

Além de realizar a execução do comando `ls` e listar o conteúdo do diretório atual, a execução do comando `perf stat` com os eventos `instructions` e `cpu-cycles` dá um resultado semelhante ao observado abaixo:

```
1  Performance counter stats for 'ls':
2          1.309.712   instructions   # 0,71 insnt per cycle
3          1.848.886   cycles
4        0,000800891 seconds time elapsed
```

É importante notar que as arquiteturas superescalares normalmente possuem um número limitado de contadores de *hardware*. Uma vez que cada contador de *hardware* pode ser utilizado para monitoramento de um único evento a cada dado momento, a quantidade de eventos que podem ser monitorados simultaneamente é limitada. Para possibilitar o monitoramento de um número maior de eventos, a ferramenta perf realiza uma técnica de multiplexação do tempo de monitoramento, permitindo que cada evento possua uma parcela do tempo de utilização do contador. No entanto, o que esta abordagem possibilita é apenas uma estimativa do real comportamento da aplicação, uma vez que os eventos multiplexados não são monitorados o tempo todo com exclusividade. Dessa forma, é aconselhável que cada evento seja monitorado individualmente por meio da instrução `perf stat`, realizando o *profiling* da aplicação várias vezes.

1.6.2. NEC FTRACE

Desenvolvida pela empresa NEC, a ferramenta FTRACE pode ser utilizada para a obtenção de informações de contadores de *hardware* presentes na arquitetura vetorial da NEC. Para utilizar a ferramenta é necessário recompilar a aplicação utilizando o compilador de-

senvolvido pela empresa NEC. Para aplicações escritas em linguagem C usamos o `ncc`, para aplicações C++ usamos o `nc++` e para aplicações Fortran usamos o `nfort`. Além disso, é necessário adicionar a *flag* de compilação `-ftrace`, como exibido no exemplo: `ncc -ftrace source.c`. A partir disso, podemos executar a aplicação como faríamos com um executável comum. Ao fim da execução, o arquivo de informações `ftrace.out` é gerado, podendo ser lido por meio da mesma ferramenta (e direcionando a saída para o arquivo *output*):

```
1 ftrace -f ftrace.out >> output
```

O arquivo gerado possui diversas informações. A Figura 1.7 mostra todas as informações de saída que a ferramenta FTRACE proporciona. Dentre elas podemos destacar o tempo total de execução, o nome das funções executadas, o número de vezes que cada função foi chamada, porcentagem de utilização de core por cada função, informações a respeito do número de *misses* dos diversos níveis de *cache*, dentre outros. É possível ainda controlar o tipo de informações que se deseja por meio dos dois diferentes modos de *profiling* indicados por meio da variável de ambiente `VE_PERF_MODE`. Caso a variável possua valor `VECTOR-OP` ou esteja indefinida (assumindo o valor padrão), o FTRACE gera informações relacionadas principalmente às instruções vetoriais. Caso o valor da variável `VE_PERF_MODE` seja `VECTOR-MEM`, os dados levantados correspondem principalmente a acessos à memória. Dessa forma, pode ser proveitoso fazer uso dos dois modos e agregar seus resultados ao fim. Pode-se alterar o valor da variável de ambiente `VE_PERF_MODE`, para um dos dois modos, da seguinte forma:

```
1 export VE_PERF_MODE=VECTOR-OP
2 export VE_PERF_MODE=VECTOR-MEM
```

1.7. Aplicações de Inteligência Artificial

Com a evolução de arquiteturas de processadores e os sistemas agregados, e.g. memória e aceleradores, algoritmos cada vez mais complexos, com grande volume de dados e operações, acharam aplicações práticas em computadores. Uma subárea da computação tornou-se o expoente atual na demanda por sistemas mais poderosos: desenvolvida na estatística, o Aprendizado de Máquina, mais genericamente conhecido como Inteligência Artificial, é a área em voga nos dias de hoje (GADEPALLY et al., 2019).

A base estatística para aprendizado de máquina é a inferência estatística, um ramo da estatística a qual tem por base o estudo de modelos preditivos baseados em amostras de uma população, tendo como principais escolas a Inferência Frequentista e a Inferência Bayesiana (CASELLA; BERGER, 2021). Quanto melhor a qualidade dessa amostra, melhor a capacidade de predição do modelo, o que muitos pesquisadores interpretam também como uma amostra de tamanho maior. Portanto, estes modelos processam um grande volume de dados para ajustar seus parâmetros de modo a melhorar suas predições.

Boa parte destes modelos assume que existe uma função de “erro”, a qual depende dos parâmetros do modelo. O algoritmo para treinar um modelo, portanto, consiste em testar várias amostras contra o modelo atual, e ajustar os parâmetros conforme o erro do modelo em prever o comportamento das amostras. Ao chegar em um mínimo local desta função de erro, avalia-se a qualidade do modelo, potencialmente aplicando-se uma reran-

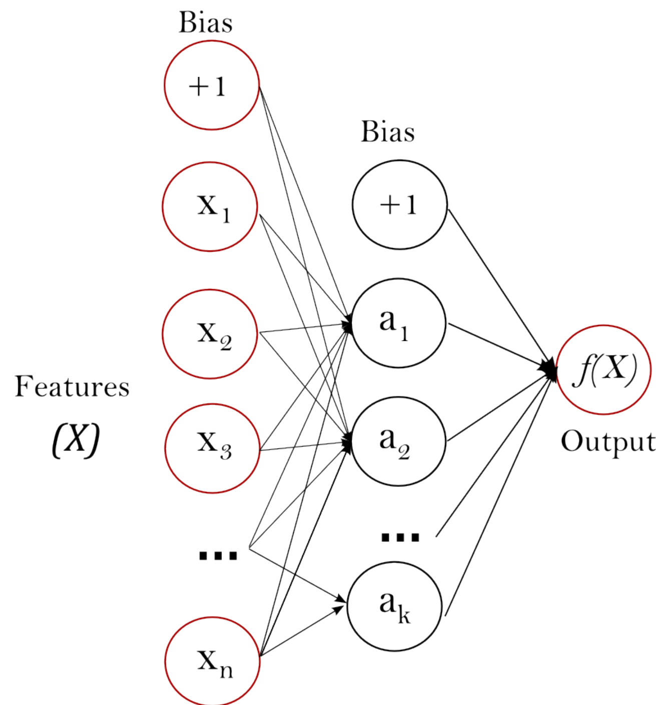


Figura 1.6. Rede Neural do tipo perceptron de múltiplas camadas.

domização de parâmetros para procurar um mínimo local melhor, embora não existam garantias de se achar um mínimo global.

Um dos modelos mais populares para vários problemas é o modelo de Redes Neurais (AGGARWAL et al., 2018). Neste modelo, temos camadas de "neurônios", os quais são representados por uma função de ativação de acordo com suas entradas. Na Figura 1.6, é ilustrado um exemplo de rede neural do tipo perceptron de múltiplas camadas com uma camada de entrada, a qual recebe características da amostra, uma camada oculta, a qual aproxima a função de erro, e uma camada de saída, a qual providencia um valor de predição. Neste exemplo, a rede neural é completamente conectada, onde cada neurônio representa uma soma $\sum(peso[k] * entrada[k])$, onde os valores de entrada vem da camada anterior, e os valores de peso são os parâmetros retidos no neurônio. Portanto, a representação do modelo é a matriz de neurônios e seus parâmetros.

Para atualizar os pesos (parâmetros) de uma rede neural, normalmente utiliza-se o algoritmo de retro-propagação (*Backpropagation*) (GOMEZ et al., 2017). No conjunto de benchmarks Rodinia (CHE et al., 2009), há um exemplo de rede neural onde é possível observar a implementação de *Backpropagation*. Abaixo, podemos observar a função de treino:

```
1 void bpnn_adjust_weights(delta, ndelta, ly, nly, w, oldw)
2 float *delta, *ly, **w, **oldw;
3 {
4   float new_dw;
```

```
5   int k, j;
6   ly[0] = 1.0;
7   //eta = 0.3;
8   //momentum = 0.3;
9
10 #ifdef OPEN
11   //omp_set_num_threads(NUM_THREAD);
12   #pragma omp parallel for \
13     shared(oldw, w, delta) \
14       private(j, k, new_dw) \
15       firstprivate(ndelta, nly)
16 #endif
17   for (j = 1; j <= ndelta; j++) {
18     for (k = 0; k <= nly; k++) {
19       new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM *
          oldw[k][j]));
20       w[k][j] += new_dw;
21       oldw[k][j] = new_dw;
22     }
23   }
24 }
```

A função recebe `delta` (o vetor com os erros da próxima camada), `ndelta` (o tamanho deste vetor), `ly` (o vetor com os valores atuais de ativação), `nly` (o tamanho deste vetor), `w` (o novo vetor de pesos a ser criado), e `oldw` (que retém o vetor de pesos anterior). Cada peso é atualizado de acordo com a fórmula `ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k][j])`, onde `ETA` e `MOMENTUM` são constantes para controlar a velocidade de treino e agressividade de adaptação dos parâmetros. Assim, esta pequena função é chave no funcionamento de redes neurais, e arquiteturas atuais propõe otimizações agressivas para melhorar sua performance. Neste minicurso, utilizamos a implementação do *Backpropagation* acima para ilustrar como podemos explorar a informação obtida por contadores de hardware e ferramentas de *profiling* para se obter desempenho em diversas arquiteturas, notadamente a arquitetura de CPU Cascade Lake e a arquitetura vetorial SX-Aurora TSUBASA.

1.8. Otimizando o Backpropagation com *profiling* e contadores de hardware

1.8.1. NEC SX-Aurora

Analisando o código da aplicação através do trecho abaixo, juntamente com a análise da `ftrace.out` gerado pelo FTRACE observado na Figura 1.7, percebe-se que a função `alloc_1d_dbl` corresponde a 63,7% do tempo de execução. Ao analisar o código abaixo, observa-se que a mesma realiza a alocação dos pesos de forma tradicional, por meio de uma matriz bidimensional. No entanto, essa alocação não favorece o acesso aos dados por conta da falta de localidade espacial dos dados, o que sugere que a alocação de memória pode ser otimizada para uma alocação de memória contígua. Modificar a alocação dos dados gera a necessidade de se alterar várias outras regiões do código, uma

```
*---------------------*
  FTRACE ANALYSIS LIST
*---------------------*

Execution Date : Sun Mar 28 08:41:41 2021 -03
Total CPU Time : 0:02'34"848 (154.848 sec.)


EXECUTION TERMINATED BUT NOT IN MAIN PROCEDURE.

 STOPPED  AT setup
 CALLED FROM main


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS      CONF HIT E.%

134217738    98.697( 63.7)     0.001    753.4      0.0   0.00   0.0     0.000  23.827    0.000    0.00 alloc_1d_dbl
        1    36.506( 23.6) 36506.441    744.5      0.0   0.00   0.0     0.000   8.087    0.000    0.00 bpnn_free
        4    13.218(  8.5)  3304.383   2853.4      0.0   0.00   0.0     0.000   0.000    0.000    0.00 alloc_2d_dbl
        1     5.524(  3.6)  5524.487    400.6      3.0   0.00   0.0     0.000   0.000    0.000    0.00 load
        1     0.636(  0.4)   636.077   3376.2      0.0   0.00   0.0     0.000   0.005    0.000    0.00 bpnn_read
        2     0.262(  0.2)   131.075   2879.9      0.0  17.78   1.0     0.262   0.000    0.000    0.00 bpnn_zero_weights
       16     0.004(  0.0)     0.267    531.9      0.0   0.01   1.0     0.000   0.000    0.000   93.75 bpnn_layerforward$1
        2     0.003(  0.0)     1.415    541.5      0.0   0.00   1.0     0.000   0.000    0.000   83.33  -thread0
        2     0.000(  0.0)     0.058    488.4      0.1   0.03   1.0     0.000   0.000    0.000  100.00  -thread1
        2     0.000(  0.0)     0.138    520.0      0.0   0.01   1.0     0.000   0.000    0.000  100.00  -thread2
        2     0.000(  0.0)     0.068    512.5      0.1   0.02   1.0     0.000   0.000    0.000  100.00  -thread3
        2     0.000(  0.0)     0.014    264.7      0.3   0.22   1.0     0.000   0.000    0.000  100.00  -thread4
        2     0.001(  0.0)     0.288    528.3      0.0   0.01   1.0     0.000   0.000    0.000  100.00  -thread5
        2     0.000(  0.0)     0.061    499.3      0.1   0.03   1.0     0.000   0.000    0.000  100.00  -thread6
        2     0.000(  0.0)     0.090    516.1      0.0   0.02   1.0     0.000   0.000    0.000   66.67  -thread7
       16     0.000(  0.0)     0.005    335.9      5.4   0.00   0.0     0.000   0.000    0.000    0.00 squash
        2     0.000(  0.0)     0.005    326.1      5.3   0.00   0.0     0.000   0.000    0.000    0.00  -thread0
        2     0.000(  0.0)     0.005    331.2      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread1
        2     0.000(  0.0)     0.005    337.2      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread2
        2     0.000(  0.0)     0.005    332.2      5.3   0.00   0.0     0.000   0.000    0.000    0.00  -thread3
        2     0.000(  0.0)     0.005    339.0      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread4
        2     0.000(  0.0)     0.005    336.7      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread5
        2     0.000(  0.0)     0.005    343.9      5.5   0.00   0.0     0.000   0.000    0.000    0.00  -thread6
        2     0.000(  0.0)     0.005    341.0      5.4   0.00   0.0     0.000   0.000    0.000    0.00  -thread7
        1     0.000(  0.0)     0.052    628.4      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_initialize
        1     0.000(  0.0)     0.043    141.1      0.0   0.00   0.0     0.000   0.000    0.000    0.00 backprop_face
        1     0.000(  0.0)     0.042    361.3      0.0   0.00   0.0     0.000   0.000    0.000    0.00 setup
       16     0.000(  0.0)     0.002    597.1      2.9   1.72   1.0     0.000   0.000    0.000  100.00 bpnn_adjust_weights$1
        2     0.000(  0.0)     0.002    574.2      2.3   1.42   1.0     0.000   0.000    0.000  100.00  -thread0
        2     0.000(  0.0)     0.002    584.5      3.2   1.96   1.0     0.000   0.000    0.000  100.00  -thread1
        2     0.000(  0.0)     0.002    602.7      3.1   1.86   1.0     0.000   0.000    0.000  100.00  -thread2
        2     0.000(  0.0)     0.002    607.9      3.1   1.82   1.0     0.000   0.000    0.000  100.00  -thread3
        2     0.000(  0.0)     0.002    589.3      2.8   1.73   1.0     0.000   0.000    0.000  100.00  -thread4
        2     0.000(  0.0)     0.002    596.6      2.8   1.68   1.0     0.000   0.000    0.000  100.00  -thread5
        2     0.000(  0.0)     0.002    622.5      3.0   1.75   1.0     0.000   0.000    0.000  100.00  -thread6
        2     0.000(  0.0)     0.002    606.8      2.8   1.68   1.0     0.000   0.000    0.000  100.00  -thread7
        2     0.000(  0.0)     0.009    322.5      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_layerforward
        1     0.000(  0.0)     0.013    503.1      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_internal_create
        1     0.000(  0.0)     0.011    304.0      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_train_kernel
        2     0.000(  0.0)     0.002    730.0      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_adjust_weights
        1     0.000(  0.0)     0.001    281.2      0.0   0.00   0.0     0.000   0.000    0.000    0.00 main
        1     0.000(  0.0)     0.000   1441.9    292.0  49.19  16.0     0.000   0.000    0.000   50.00 bpnn_output_error
        1     0.000(  0.0)     0.000   2461.5      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_hidden_error
---------------------------------------------------------------------------------------------------------------------
134217807   154.848(100.0)     0.001    932.3      0.1   0.09   1.0     0.262  31.920    0.000   88.12 total
```

Figura 1.7. Resultados do comando FTRACE para a versão original do *Backpropagation*.

vez que a forma de acesso e indexação dos dados agora deve ser feita de forma diferente. Dessa forma, o código abaixo é modificado de modo a substituir a alocação de uma matriz de tamanho *m x n* por uma alocação de um espaço contíguo de memória de tamanho *m x n*.

```
1 BPNN *bpnn_internal_create(n_in, n_hidden, n_out)
2 int n_in, n_hidden, n_out;
3 {
4   ...
5
6   newnet->input_weights = alloc_2d_dbl(n_in + 1, n_hidden
        + 1);
7   newnet->hidden_weights = alloc_2d_dbl(n_hidden + 1,
       n_out + 1);
8
9   newnet->input_prev_weights = alloc_2d_dbl(n_in + 1,
       n_hidden + 1);
10  newnet->hidden_prev_weights = alloc_2d_dbl(n_hidden +
       1, n_out + 1);
11
12  return (newnet);
13 }
```

A otimização é apresentada nos trechos de código abaixo. Na linha 2 do primeiro bloco, adicionamos `long unsigned` na declaração da variável, e das linha 8 à 12 substitui-se a função de alocação bidimensional pela função de alocação unidimensional contígua.

```
1 BPNN *bpnn_internal_create(n_in, n_hidden, n_out)
2 long unsigned int n_in, n_hidden, n_out;
3 {
4   ...
5
6   newnet->input_weights = alloc_1d_dbl( (n_in + 1) * (
       n_hidden + 1));
7   newnet->hidden_weights = alloc_1d_dbl((n_hidden + 1) *
        (n_out + 1));
8
9   newnet->input_prev_weights = alloc_1d_dbl((n_in + 1) *
        (n_hidden + 1));
10  newnet->hidden_prev_weights = alloc_1d_dbl((n_hidden +
       1) * (n_out + 1));
11
12  return (newnet);
13 }
```

```
*---------------------*
  FTRACE ANALYSIS LIST
*---------------------*

Execution Date : Sun Mar 28 04:56:48 2021 -03
Total CPU Time : 0:00'09"394 (9.394 sec.)


EXECUTION TERMINATED BUT NOT IN MAIN PROCEDURE.

 STOPPED  AT setup
 CALLED FROM main


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS     CONF HIT E.%

        1     6.578( 70.0)  6577.831   2611.8      0.0   0.00 138.5     0.000   0.263    0.000  100.00 bpnn_read
       16     1.638( 17.4)   102.395  10404.4   2662.5  75.59  16.0     1.638   0.000    0.000   33.35 bpnn_adjust_weights$1
        2     0.205(  2.2)   102.396  10404.3   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread0
        2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread1
        2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread2
        2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread3
        2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread4
        2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread5
        2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread6
        2     0.205(  2.2)   102.395  10404.4   2662.5  75.59  16.0     0.205   0.000    0.000   33.35  -thread7
        2     1.025( 10.9)   512.373   5042.6      0.0  44.16  17.0     1.025   0.000    0.000    0.00 bpnn_zero_weights
       16     0.152(  1.6)     9.528  28483.8  14086.6  98.91 256.0     0.150   0.000    0.000   69.87 bpnn_layerforward$1
        2     0.020(  0.2)    10.052  27026.0  13351.7  98.81 256.0     0.019   0.000    0.000    2.92  -thread0
        2     0.019(  0.2)     9.502  28559.4  14124.7  98.91 256.0     0.019   0.000    0.000   98.88  -thread1
        2     0.019(  0.2)     9.426  28787.5  14239.7  98.93 256.0     0.019   0.000    0.000   26.30  -thread2
        2     0.019(  0.2)     9.400  28865.9  14279.2  98.94 256.0     0.019   0.000    0.000   99.98  -thread3
        2     0.019(  0.2)     9.449  28717.9  14204.6  98.93 256.0     0.019   0.000    0.000   49.33  -thread4
        2     0.019(  0.2)     9.587  28312.9  14000.5  98.90 256.0     0.019   0.000    0.000   99.34  -thread5
        2     0.019(  0.2)     9.406  28846.3  14269.4  98.93 256.0     0.019   0.000    0.000   82.35  -thread6
        2     0.019(  0.2)     9.403  28856.0  14274.3  98.93 256.0     0.019   0.000    0.000   99.82  -thread7
       17     0.000(  0.0)     0.007    248.2      4.1   0.00   0.0     0.000   0.000    0.000    0.00 squash
        3     0.000(  0.0)     0.005    239.4      5.2   0.00   0.0     0.000   0.000    0.000    0.00  -thread0
        2     0.000(  0.0)     0.007    241.4      3.8   0.00   0.0     0.000   0.000    0.000    0.00  -thread1
        2     0.000(  0.0)     0.007    246.0      3.9   0.00   0.0     0.000   0.000    0.000    0.00  -thread2
        2     0.000(  0.0)     0.007    247.4      3.9   0.00   0.0     0.000   0.000    0.000    0.00  -thread3
        2     0.000(  0.0)     0.007    248.5      3.9   0.00   0.0     0.000   0.000    0.000    0.00  -thread4
        2     0.000(  0.0)     0.007    253.0      4.0   0.00   0.0     0.000   0.000    0.000    0.00  -thread5
        2     0.000(  0.0)     0.007    252.1      3.9   0.00   0.0     0.000   0.000    0.000    0.00  -thread6
        2     0.000(  0.0)     0.007    259.4      4.0   0.00   0.0     0.000   0.000    0.000    0.00  -thread7
        1     0.000(  0.0)     0.052    641.0      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_initialize
        1     0.000(  0.0)     0.042    374.2      0.0   0.00   0.0     0.000   0.000    0.000    0.00 setup
        1     0.000(  0.0)     0.038    150.1      0.0   0.00   0.0     0.000   0.000    0.000    0.00 backprop_face
       10     0.000(  0.0)     0.002    285.5      0.0   0.00   0.0     0.000   0.000    0.000    0.00 alloc_1d_dbl
        2     0.000(  0.0)     0.009    355.6      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_layerforward
        1     0.000(  0.0)     0.017    118.9      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_free
        1     0.000(  0.0)     0.013    513.3      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_internal_create
        1     0.000(  0.0)     0.010    311.0      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_train_kernel
        1     0.000(  0.0)     0.009    377.0     15.7  10.18   3.5     0.000   0.000    0.000  100.00 bpnn_hidden_error
        2     0.000(  0.0)     0.003    536.8      0.0   0.00   0.0     0.000   0.000    0.000    0.00 bpnn_adjust_weights
        1     0.000(  0.0)     0.001    336.4      0.0   0.00   0.0     0.000   0.000    0.000    0.00 main
        1     0.000(  0.0)     0.000    789.2     18.4   5.65   1.0     0.000   0.000    0.000   50.00 bpnn_output_error
----------------------------------------------------------------------------------------------------------------
       75     9.394(100.0)   125.249   4655.8    693.0  44.50  19.6     2.813   0.263    0.000   47.96 total
```

Figura 1.8. Resultados do FTRACE para a otimização de alocação de memória contígua.

Por vezes, o tamanho `m x n` ultrapassa a capacidade do tipo de dados *int*. Dessa forma, faz-se necessário alterar os tipos das variáveis `m` e `n` para `long unsigned` nas demais funções. Além disso, por agora termos um bloco contíguo de dados ao invés de uma matriz bidimensional, faz-se necessário também alterar a forma como esses dados são acessados. Já não se utiliza mais o comum `matriz[i][j]`. Agora é necessário calcular o `findex` que permite realizar o acesso correto aos dados armazenados continuamente na memória. O trecho de código abaixo exemplifica essas duas alterações, com a utilização de `long unsigned` nas linhas 3 e 5, e com o cálculo do `findex` e sua utilização nas linhas 7 e 9, respectivamente.

```
1 bpnn_zero_weights(w, m, n)
2 float *w;
3 long unsigned int m, n;
4 {
5   long unsigned int i, j, findex;
6   for (i = 0; i <= m; i++) {
7     findex = i*(n+1);
8     for (j = 0; j <= n; j++) {
9         w[findex + j] = 0.0;
10    }
11  }
12 }
```

Verificando os resultados dessa otimização através do novo `ftrace.out`, representado na Figura 1.8, temos que o tempo de execução diminui de 154 segundos para 9 segundos, e a função `alloc_1d_dbl` agora corresponde a uma parcela irrisória do tempo de execução.

A última otimização aplicada é relacionada à função `bppn_read`, que, após a primeira otimização, se tornou responsável por 70% do tempo de execução, como visto na Figura 1.8. Nessa seção do código é realizada a leitura dos pesos da Rede Neural a partir de um arquivo texto. No trecho de código abaixo, temos a implementação da função `fastcopy` (bastante utilizada pela função `bppn_read` para copiar os dados lidos de uma estrutura de dados para outra) e as operações de leitura realizadas pela função `bpnn_read` em suas versões originais.

```
1 #define fastcopy(to,from,len)
2 {
3   register char *_to,*_from;
4   register int _i,_l;
5   _to = (char *)(to);
6   _from = (char *)(from);
7   _l = (len);
8   for (_i = 0; _i < _l; _i++) *_to++ = *_from++;
9 }
10
11 BPNN *bpnn_read(filename)
12 char *filename;
```

```
13 {
14   ...
15   int fd, n1, n2, n3, i, j, memcnt;
16
17   if ((fd = open(filename, 0, 0644)) == -1) {
18     return (NULL);
19   }
20
21   printf("Reading '%s'\n", filename); //fflush(stdout);
22
23   // leituras com a funcao read
24   read(fd, (char *) &n1, sizeof(int));
25   read(fd, (char *) &n2, sizeof(int));
26   read(fd, (char *) &n3, sizeof(int));
27
28   ...
29
30   mem = (char *) malloc ((unsigned) ((n1+1) * (n2+1) *
       sizeof(float)));
31   // leituras com a funcao read e loop complexo com a
       funcao fastcopy
32   read(fd, mem, (n1+1) * (n2+1) * sizeof(float));
33   for (i = 0; i <= n1; i++) {
34     for (j = 0; j <= n2; j++) {
35       fastcopy(&(new->input_weights[i][j]), &mem[memcnt],
           sizeof(float));
36       memcnt += sizeof(float);
37     }
38   }
39   free(mem);
40
41   printf("Done\nReading hidden weights..."); //fflush(
       stdout);
42
43   memcnt = 0;
44   mem = (char *) malloc ((unsigned) ((n2+1) * (n3+1) *
       sizeof(float)));
45   // leituras com a funcao read e loop complexo com a
       funcao fastcopy
46   read(fd, mem, (n2+1) * (n3+1) * sizeof(float));
47   for (i = 0; i <= n2; i++) {
48     for (j = 0; j <= n3; j++) {
49       fastcopy(&(new->hidden_weights[i][j]), &mem[memcnt],
           sizeof(float));
50       memcnt += sizeof(float);
51     }
```

```
52   }
53   free(mem);
54   close(fd);
55
56   ...
57 }
```

Como otimização buscamos portanto modificar a leitura do arquivo, substituindo as chamadas à função `read` (da biblioteca `unistd.h`) pela função `fread` (da biblioteca `stdio.h`), e removemos as chamadas à função `fastcopy`. No trecho de código abaixo pode-se ver as modificações realizadas.

```
 1 BPNN *bpnn_read(filename)
 2 char *filename;
 3 {
 4   ...
 5   FILE *pFile;
 6   pFile = fopen( filename, "rb" );
 7
 8   if (pFile == NULL) {
 9     return (NULL);
10   }
11
12   printf("Reading '%s'\n", filename); //fflush(stdout);
13
14   // substituicao das chamadas a funcao read por chamadas
         a funcao fread
15   fread((char *) &n1, sizeof(char), sizeof(long unsigned
        int), pFile);
16   fread((char *) &n2, sizeof(char), sizeof(long unsigned
        int), pFile);
17   fread((char *) &n3, sizeof(char), sizeof(long unsigned
        int), pFile);
18
19   ...
20
21
22   // loop de leitura com fastcopy substituido
23   long unsigned int totalmem = (n1+1) * (n2+1) * sizeof(
        float);
24   fread((char*)new->input_weights, sizeof(char), totalmem
        , pFile);
25   float * reader = mem;
26
27   ...
28
29   // loop de leitura com fastcopy substituido
```

```
30   totalmem = (n2+1) * (n3+1) * sizeof(float);
31   fread((char*) new->hidden_weights, sizeof(char),
        totalmem, pFile);
32   fclose(pFile);
33
34   ...
35 }
```

Analisando o arquivo de saída `ftrace.out` da Figura 1.9, podemos ver que o tempo de execução diminui em quatro vezes, de 9 segundos para 2 segundos.

```
*--------------------*
  FTRACE ANALYSIS LIST
*--------------------*

Execution Date : Sun Mar 28 04:54:59 2021 -03
Total CPU Time : 0:00'02"162 (2.162 sec.)


EXECUTION TERMINATED BUT NOT IN MAIN PROCEDURE.

 STOPPED  AT setup
 CALLED FROM main


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS      CONF HIT E.%

       16     1.639( 75.8)   102.438  10400.1   2661.4  75.59  16.0     1.639   0.000     0.000   33.35 bpnn_adjust_weights$1
        2     0.205(  9.5)   102.439  10400.0   2661.4  75.59  16.0     0.205   0.000     0.000   33.35  -thread0
        2     0.205(  9.5)   102.437  10400.1   2661.4  75.59  16.0     0.205   0.000     0.000   33.35  -thread1
        2     0.205(  9.5)   102.437  10400.1   2661.4  75.59  16.0     0.205   0.000     0.000   33.35  -thread2
        2     0.205(  9.5)   102.438  10400.0   2661.4  75.59  16.0     0.205   0.000     0.000   33.35  -thread3
        2     0.205(  9.5)   102.438  10400.1   2661.4  75.59  16.0     0.205   0.000     0.000   33.35  -thread4
        2     0.205(  9.5)   102.438  10400.1   2661.4  75.59  16.0     0.205   0.000     0.000   33.35  -thread5
        2     0.205(  9.5)   102.437  10400.1   2661.4  75.59  16.0     0.205   0.000     0.000   33.35  -thread6
        2     0.205(  9.5)   102.438  10400.0   2661.4  75.59  16.0     0.205   0.000     0.000   33.35  -thread7
        2     0.369( 17.1)   184.693   8765.9      0.0  70.47  17.0     0.369   0.000     0.000    0.00 bpnn_zero_weights
       16     0.153(  7.1)     9.573  28354.0  14021.2  98.90 256.0     0.150   0.000     0.000   70.26 bpnn_layerforward$1
        2     0.021(  1.0)    10.286  26424.0  13048.2  98.76 256.0     0.019   0.000     0.000   55.43  -thread0
        2     0.019(  0.9)     9.472  28649.9  14170.4  98.92 256.0     0.019   0.000     0.000   38.17  -thread1
        2     0.019(  0.9)     9.401  28862.6  14277.6  98.93 256.0     0.019   0.000     0.000   99.52  -thread2
        2     0.019(  0.9)     9.498  28571.5  14130.9  98.92 256.0     0.019   0.000     0.000   80.13  -thread3
        2     0.019(  0.9)     9.532  28471.0  14080.2  98.91 256.0     0.019   0.000     0.000   98.41  -thread4
        2     0.019(  0.9)     9.522  28501.8  14095.7  98.91 256.0     0.019   0.000     0.000    0.03  -thread5
        2     0.019(  0.9)     9.416  28817.2  14254.8  98.93 256.0     0.019   0.000     0.000   99.98  -thread6
        2     0.019(  0.9)     9.453  28704.8  14198.1  98.92 256.0     0.019   0.000     0.000   90.40  -thread7
        1     0.000(  0.0)     0.171    278.7      0.0   2.33 138.5     0.000   0.000     0.000  100.00 bpnn_read
       17     0.000(  0.0)     0.007    239.6      4.0   0.00   0.0     0.000   0.000     0.000    0.00 squash
        3     0.000(  0.0)     0.005    239.1      5.2   0.00   0.0     0.000   0.000     0.000    0.00  -thread0
        2     0.000(  0.0)     0.007    231.2      3.7   0.00   0.0     0.000   0.000     0.000    0.00  -thread1
        2     0.000(  0.0)     0.007    239.7      3.8   0.00   0.0     0.000   0.000     0.000    0.00  -thread2
        2     0.000(  0.0)     0.007    233.7      3.7   0.00   0.0     0.000   0.000     0.000    0.00  -thread3
        2     0.000(  0.0)     0.007    240.9      3.8   0.00   0.0     0.000   0.000     0.000    0.00  -thread4
        2     0.000(  0.0)     0.007    245.0      3.8   0.00   0.0     0.000   0.000     0.000    0.00  -thread5
        2     0.000(  0.0)     0.007    242.5      3.8   0.00   0.0     0.000   0.000     0.000    0.00  -thread6
        2     0.000(  0.0)     0.007    245.3      3.8   0.00   0.0     0.000   0.000     0.000    0.00  -thread7
        1     0.000(  0.0)     0.053    628.2      0.0   0.00   0.0     0.000   0.000     0.000    0.00 bpnn_initialize
        1     0.000(  0.0)     0.042    374.2      0.0   0.00   0.0     0.000   0.000     0.000    0.00 setup
        1     0.000(  0.0)     0.039    148.8      0.0   0.00   0.0     0.000   0.000     0.000    0.00 backprop_face
        1     0.000(  0.0)     0.032    174.6      0.0   0.00   0.0     0.000   0.000     0.000    0.00 bpnn_free
       10     0.000(  0.0)     0.002    284.4      0.0   0.00   0.0     0.000   0.000     0.000    0.00 alloc_1d_dbl
        2     0.000(  0.0)     0.009    349.7      0.0   0.00   0.0     0.000   0.000     0.000    0.00 bpnn_layerforward
        1     0.000(  0.0)     0.013    514.4      0.0   0.00   0.0     0.000   0.000     0.000    0.00 bpnn_internal_create
        1     0.000(  0.0)     0.010    312.4      0.0   0.00   0.0     0.000   0.000     0.000    0.00 bpnn_train_kernel
        1     0.000(  0.0)     0.009    385.8     16.1  10.18   3.5     0.000   0.000     0.000  100.00 bpnn_hidden_error
        2     0.000(  0.0)     0.003    546.5      0.0   0.00   0.0     0.000   0.000     0.000    0.00 bpnn_adjust_weights
        1     0.000(  0.0)     0.001    325.9      0.0   0.00   0.0     0.000   0.000     0.000    0.00 main
        1     0.000(  0.0)     0.000    671.6     15.6   5.65   1.0     0.000   0.000     0.000   50.00 bpnn_output_error
----------------------------------------------------------------------------------------------------------
       75     2.162(100.0)    28.828  11390.2   3010.8  79.03  19.6     2.158   0.001     0.000   48.12 total
```

Figura 1.9. Resultados do FTRACE para a otimização de leitura

1.8.2. Intel Cascade Lake

Todas as otimizações propostas acima são válidas também para a microarquitetura Intel Cascade Lake. O que faremos nesta seção é demonstrar como podemos perceber quais são as modificações necessárias por meio de contadores de *hardware* e como atestar o ganho de desempenho após elas. Uma vez que tem-se uma ampla gama de contadores disponíveis na arquitetura da Intel, escolher por onde iniciar a análise de desempenho pode ser uma tarefa complicada. Um ponto de início pode ser a verificação do número de *cache misses* nos vários níveis de *cache* (FOG, 2012; INTEL, 2019), uma vez que encontrar o dado o mais próximo do processador possível é importante para se evitar grandes latências da hierarquia de memória (como mencionado na Seção 1.3) (PATTERSON; HENNESSY, 2004). Além disso, verificar o número de instruções de predição de desvio que foram feitas de forma incorreta (*branch mispredictions*) (FOG, 2012; INTEL, 2019) também é um fator importante, já que uma predição de salto feita da forma incorreta pode acarretar na necessidade de se limpar todas as instruções presentes no *pipeline* (PATTERSON; HENNESSY, 2004).

O código abaixo representa a execução da aplicação *Backpropagation* em sua versão original no processador Intel Xeon Gold 6226 utilizando a ferramenta Linux perf e com tamanho de entrada equivalente a 2^{26}. Os contadores de *hardware* escolhidos são o *LLC-loads* e o *LLC-load-misses*, que representam respectivamente o número total de instruções de requisição de dados que chegam à LLC e a parte dessas requisições que deram *miss* nesse nível de *cache*. Logo abaixo é possível ver a saída da execução com a ferramenta perf, com a saída da execução da própria aplicação, os números absolutos de acesso à LLC e a porcentagem desses acessos que resultaram em *miss*, bem como o tempo de execução da aplicação.

```
1 perf stat -e LLC-loads,LLC-load-misses ./backprop
      67108864
2
3 Random number generator seed: 7
4 Input layer size : 67108864
5 Starting training kernel
6 Performing CPU computation
7 Training done
8
9 Performance counter stats for './backprop 67108864':
10
11 2,984,760,563  LLC-loads
12   961,573,754  LLC-load-misses # 32.22% of all LL-cache
        hits
13
14  39.501156910 seconds time elapsed
```

Além disso, a execução é repetida usando contadores semelhantes referentes à predição de desvio, como visto no código abaixo:

```
1 perf stat -e branch-instructions,branch-misses ./backprop
     67108864
2 Random number generator seed: 7
3 Input layer size : 67108864
4 Starting training kernel
5 Performing CPU computation
6 Training done
7
8 Performance counter stats for './backprop 67108864':
9
10  43,232,691,964 branch-instructions
11     167,438,939  branch-misses # 0.39% of all branches
12
13   39.568886150 seconds time elapsed
```

Pode-se notar que a porcentagem de *branch mispredictions* é praticamente irrisória, o que não pode ser afirmado com relação aos *cache misses*, que chegam a 32,22% das requisições de leitura que chegam à LLC. Para analisar melhor o acesso à memória, pode-se utilizar contadores referentes a níveis de *cache* mais próximos do processador. Por fins de simplificação, o código abaixo mostra diretamente as informações obtidas da *cache* L1, novamente em termos de requisições totais e porcentagem que resulta em *miss*. Note que no que, diz respeito à *cache* L1, faz-se necessário indicar que quer-se informações referentes apenas à *cache* de dados (*dcache*), uma vez que o objetivo é justamente analisar a eficiência do acesso aos dados. Para referir-se à *cache* instruções faz-se uso do termo *icache*.

```
1 perf stat -e L1-dcache-loads,L1-dcache-load-misses ./
    backprop 67108864
2 ...
3
4 Performance counter stats for './backprop 67108864':
5
6 54,733,941,264 L1-dcache-loads
7  7,941,597,753 L1-dcache-load-misses # 14.51% of all L1-
     dcache hits
8
9  38.764941014 seconds time elapsed
```

Novamente é possível verificar que a grande porcentagem de *misses* na L1 (14,51%) abre espaço para otimizações. O acesso a dados que não estão contíguos em memória prejudica qualquer aplicação devido à falta de localidade espacial, como já mencionado anteriormente. Dessa forma, as alterações na alocação e na forma de acesso aos dados também se fazem importantes, bem como a aplicação de `long unsigned` no lugar de `int`. Após aplicar tais modificações como proposto na Seção 1.8.1, tem-se tais resultados na análise de desempenho da aplicação:

```
1 perf stat -e L1-dcache-loads,L1-dcache-load-misses ./
    backprop 67108864
2 ...
3
4 Performance counter stats for './backprop 67108864':
5
6 47,274,433,208 L1-dcache-loads
7  2,790,105,769  L1-dcache-load-misses # 5.90% of all L1-
     dcache hits
8
9  24.095061856 seconds time elapsed
```

Alterando a forma como a aplicação aloca e acessa os dados permitiu uma redução de 64% no número de misses relativos na *cache* L1 de dados, e reduziu em 37% o tempo de execução da aplicação. A próxima otimização possível é a substituição da função `fastcopy` utilizada pela função `bpnn_read`, alterando a leitura do arquivo de pesos com a função `fread` no lugar da função `read`, assim como feito na Seção 1.8.1. Abaixo pode-se observar os resultados de acesso à *cache* L1 de dados e o novo tempo de execução da aplicação.

```
1 perf stat -e L1-dcache-loads,L1-dcache-load-misses ./
    backprop 67108864
2 ...
3
4 Performance counter stats for './backprop 67108864':
5
6 22,440,730,866 L1-dcache-loads
7  2,417,265,599  L1-dcache-load-misses # 10.77% of all L1-
     dcache hits
8  14.668708825 seconds time elapsed
```

Nota-se que, apesar de a porcentagem de *misses* relativos ter aumentado em comparação ao resultado anterior, a quantidade total de requisições de leitura que chegam à L1 diminui para menos da metade por conta das modificações feitas no código. Dessa forma, foi possível reduzir ainda mais o tempo de execução da aplicação, que foi de 39 segundos em sua versão original para menos de 15 segundos na versão com algumas otimizações.

1.9. Conclusão

Nesse capítulo de minicurso, apresentamos conceitos de arquitetura de computadores, hierarquia de memória, vetorização e inteligência artificial. Utilizou-se exemplos e técnicas de otimização que podem ser aplicadas em ambas as arquiteturas da Intel e da NEC empregadas sobre uma implementação de retro-propagação da área de inteligência artificial. Com essas técnicas demonstramos ganhos de desempenho consideráveis, aprimorando em até 71 vezes o tempo de execução da aplicação no processador vetorial SX-Aurora, um ganho possível graças à possibilidade de vetorização obtida com as alteralções.

Já no processador da Intel, um Xeon Gold 6226, foi possível reduzir o tempo de execução em 2,6 vezes. Diversas outras otimizações poderiam ser aplicadas para apri-

morar a capacidade de se usar as instruções vetoriais do processador. Uma vez que este é um minicurso de nível básico, nos detemos a explicar as funcionalidades básicas dos contadores em ambas as microarquiteturas.

Referências

A, i. et al. Survey on artificial intelligence. *International Journal of Computer Sciences and Engineering*, v. 7, p. 1778–1790, 05 2019. páginas

ADHIANTO, L. et al. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, v. 22, 01 2009. páginas

AGGARWAL, C. C. et al. Neural networks and deep learning. *Springer*, Springer, v. 10, p. 978–3, 2018. páginas

BAER, J.-L.; CHEN, T.-F. An effective on-chip preloading scheme to reduce data access penalty. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery, 1991. (Supercomputing '91), p. 176–186. ISBN 0897914597. Disponível em: <https://doi.org/10.1145/125826.125932>. páginas

BAKHSHALIPOUR, M. et al. Bingo spatial data prefetcher. In: LOURI, A.; VENKATARAMANI, G.; GRATZ, P. (Ed.). *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. [S.l.], 2019. p. 399–411. páginas

BHATIA, E. et al. Perceptron-based prefetch filtering. In: *Proceedings of the 46th International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2019. (ISCA '19), p. 1–13. ISBN 9781450366694. Disponível em: <https://doi.org/10.1145/3307650.3322207>. páginas

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. A survey of multicore processors. *IEEE Signal Processing Magazine*, IEEE, v. 26, n. 6, p. 26–37, 2009. páginas

BOKADE, S.; DAKHOLE, P. Cla based 32-bit signed pipelined multiplier. In: IEEE. *2016 international conference on communication and signal processing (ICCSP)*. [S.l.], 2016. p. 0849–0852. páginas

CASELLA, G.; BERGER, R. L. *Statistical inference*. [S.l.]: Cengage Learning, 2021. páginas

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE. *2009 IEEE international symposium on workload characterization (IISWC)*. [S.l.], 2009. p. 44–54. páginas

CHEN, T.-F.; BAER, J.-L. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, IEEE, v. 44, n. 5, p. 609–623, 1995. páginas

CUTRESS, I. *The AMD Zen and Ryzen 7 Review: A Deep Dive on 1800X, 1700X and 1700*. 2017. Disponível em: <https://www.anandtech.com/show/11170/the-amd-zen-and-ryzen-7-review-a-deep-dive-on-1800x-1700x-and-1700>. páginas

FLOYD, T. L. *Digital Fundamentals, 10/e*. [S.l.]: Pearson Education India, 2010. páginas

FOG, A. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, p. 02–29, 2012. páginas

GADEPALLY, V. et al. Ai enabling technologies: A survey. *arXiv preprint arXiv:1905.03592*, 2019. páginas

GERNDT, M.; FÜRLINGER, K.; KEREKU, E. Periscope: Advanced techniques for performance analysis. In: *PARCO*. [S.l.: s.n.], 2005. páginas

GIRELLI, V. S. et al. Investigating memory prefetcher performance over parallel applications: From real to simulated. *Concurrency and Computation: Practice and Experience*, n/a, n. n/a, p. e6207. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6207>. páginas

GOMEZ, A. N. et al. The reversible residual network: Backpropagation without storing activations. *arXiv preprint arXiv:1707.04585*, 2017. páginas

Guttman, D. et al. Performance and energy evaluation of data prefetching on intel xeon phi. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. [S.l.: s.n.], 2015. p. 288–297. páginas

HENNESSY, J. et al. Mips: A microprocessor architecture. *ACM SIGMICRO Newsletter*, ACM New York, NY, USA, v. 13, n. 4, p. 17–22, 1982. páginas

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128119055. páginas

INTEL. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2019. <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf>. [Accesed in: 16 Jan. 2020]. páginas

ISEN, C.; JOHN, L. K.; JOHN, E. A tale of two processors: Revisiting the risc-cisc debate. In: SPRINGER. *Spec benchmark workshop*. [S.l.], 2009. p. 57–76. páginas

JACOB, B.; WANG, D.; NG, S. *Memory systems: cache, DRAM, disk*. [S.l.]: Morgan Kaufmann, 2010. páginas

KANG, H.; WONG, J. L. To hardware prefetch or not to prefetch? a virtualized environment study and core binding approach. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages*

and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2013. (ASPLOS '13), p. 357–368. ISBN 9781450318709. Disponível em: <https://doi.org/10.1145/2451116.2451155>. páginas

KNÜPFER, A. et al. The vampir performance analysis tool-set. In: *Parallel Tools Workshop*. [S.l.: s.n.], 2008. páginas

KOMATSU, K. et al. Performance evaluation of a vector supercomputer sx-aurora tsubasa. In: IEEE. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. [S.l.], 2018. p. 685–696. páginas

KSHEMKALYANI, A. *Vector Processors*. 2012. <https://www.cs.uic.edu/~ajayk/c566/VectorProcessors.pdf>. Accessed: 2021–02-21. páginas

LE, H. Q. et al. Ibm power6 microarchitecture. *IBM Journal of Research and Development*, IBM, v. 51, n. 6, p. 639–662, 2007. páginas

Liao, S. et al. Machine learning-based prefetch optimization for data center applications. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. [S.l.: s.n.], 2009. p. 1–10. ISSN 2167-4337. páginas

MARR, D. T. et al. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, v. 6, n. 1, 2002. páginas

MEY, D. et al. Score-p: A unified performance measurement system for petascale applications. In: ____. [S.l.: s.n.], 2012. p. 85–97. ISBN 9783642240249. páginas

MORGAN, T. P. *Drilling Xeon Skylake Architecture*. 2017. Disponível em: <https://www.nextplatform.com/2017/08/04/drilling-xeon-skylake-architecture/>. páginas

NEC. *How to Use C/C++ Compiler for Vector Engine*. 2020. <https://www.hpc.nec/api/v1/forum/file/download?id=pgNh9b>. Acessado em: 08/2020. páginas

NEC. *SX-Aurora TSUBASA A100-1 series user's guide*. 2020. <https://www.hpc.nec/documents/guide/pdfs/A100-1_series_users_guide.pdf>. Acessado em: 08/2020. páginas

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design (4nd Ed.): The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN 1558604286. páginas

Peled, L. et al. Semantic locality and context-based prefetching using reinforcement learning. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. [S.l.: s.n.], 2015. p. 285–297. páginas

PEREZ, A. F. et al. *Lower Numerical Precision Deep Learning Inference and Training*. 2018. <https://software.intel.com/content/www/us/en/develop/articles/lower-numerical-precision-deep-learning-inference-and-training.html>. Accessed: 2021–02-28. páginas

SHENDE, S. S.; MALONY, A. D. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, Sage Publications, Inc., USA, v. 20, n. 2, p. 287–311, maio 2006. ISSN 1094-3420. Disponível em: <https://doi.org/10.1177/1094342006064482>. páginas

THORNTON, J. E. The cdc 6600 project. *Annals of the History of Computing*, IEEE, v. 2, n. 4, p. 338–348, 1980. páginas

TULLSEN, D. M.; EGGERS, S. J.; LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In: *Proceedings of the 22nd annual international symposium on Computer architecture*. [S.l.: s.n.], 1995. p. 392–403. páginas

WULF, W.; MCKEE, S. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, v. 23, 01 1996. páginas

YEH, T.-Y.; PATT, Y. N. Two-level adaptive training branch prediction. In: *Proceedings of the 24th annual international symposium on Microarchitecture*. [S.l.: s.n.], 1991. p. 51–61. páginas

Zangeneh, S. et al. Branchnet: A convolutional neural network to predict hard-to-predict branches. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. [S.l.: s.n.], 2020. p. 118–130. páginas

Zhang, L. et al. A dynamic branch predictor based on parallel structure of srnn. *IEEE Access*, v. 8, p. 86230–86237, 2020. páginas

Capítulo 2

Otimização de Programas Paralelos com uso do OpenACC

Evaldo B. Costa – *IC/UFRJ – ebcosta@ic.ufrj.br*
Gabriel P. Silva – *IC/UFRJ – gabriel@ic.ufrj.br*

Resumo

Este minicurso tem por objetivo apresentar técnicas de otimização de programas paralelos que façam uso de diretivas do OpenACC. Para isso, serão utilizadas ferramentas que realizam uma análise completa de desempenho do código para identificação de regiões paralelizáveis e quais métodos podem ser aplicados. O OpenACC é um modelo de programação para computação paralela que pode ser executado em diversos tipos de arquiteturas: multicore, manycore e aceleradores. Assim, neste minicurso são avaliados os efeitos dos componentes de hardware sobre o desempenho de programas paralelos. Ressaltam-se as modificações que devem ser feitas no código para explorar com vantagem as características dos recursos computacionais, avaliando os seus respectivos impactos no desempenho de um programa.

2.1. Arquitetura dos Aceleradores Gráficos

As arquiteturas dos aceleradores gráficos (GPUs) são bem diferencidas das arquiteturas dos processadores convencionais. O paralelismo nos aceleradores gráficos é explorado através de um conjunto maçico de multiprocessadores de fluxo (*streaming multiprocessors (SM)*), executando em paralelo e de forma sincronizada trechos computacionalmente intensivos, chamados de *kernels*, das diversas aplicações.

Para o melhor entedimento dos aceleradores gráficos (GPUs) vamos estudar, sem perda de generalidade, a arquitetura de um tipo de acelerador gráfico desenvolvido pela NVIDIA, a arquitetura Kepler, observada na Figura 2.1.

Na Figura 2.1 verificamos que o acelerador gráfico possui uma arquitetura distinta, com diversos níveis de hierarquia de memória, algumas delas compartilhadas, outras exclusivas de cada processador de fluxo (SM). Analisamos esses e outros detalhes a seguir.

Figura 2.1: Arquitetura NVIDIA Kepler [NVIDIA 2014]

Principais características da arquitetura Kepler Cada unidade de multiprocessador de fluxo (Figura 2.2) possui 192 núcleos CUDA de precisão simples e 64 de precisão dupla, onde cada núcleo tem unidades lógicas de aritmética inteira de ponto flututante capazes de operar em modo “pipeline”, incluindo operações do tipo *fused multiply-add* (FMA). As 32 unidades de função especial (SFU) dentro de cada SM são utilizadas para aproximar operações transcendentais como raiz quadrada, seno, cosseno e recíproco (1/x). O projeto dessa arquitetura está focado no desempenho/consumo energético, fundamental na computação de alto desempenho moderna.

O escalonador do multiprocessador de fluxo (SM) dispara as *threads* em grupos de 32 *threads* chamadas de *warps*. Cada SM possui quatro escalonadores de *warp*, permitindo um máximo de quatro *warps* disparadas e executadas concorrentemente. O número de registradores pode chegar até 255 registradores utilizados simultaneamente por cada *thread*.

Para melhorar ainda mais o desempenho, a arquitetura Kepler apresenta uma instrução de *shuffle* que permite às *threads* dentro de uma mesma *warp* compartilhar dados. Anteriormente, o compartilhamento de dados entre *threads* demandava o acesso à memória compartilhada, com operações de *load* e *store*, impactando em muito o desempenho de aplicações como a transformada de Fourier (FFT).

Outro tipo de instrução disponível são operações atômicas em memória, permitindo que as *threads* realizem adequadamente operações de *read-modify-write*, como soma, máximo, mínimo e compare-e-troque em estruturas de dados compartilhadas. Operações atômicas são amplamente utilizadas para ordenação em paralelo e para o acesso em paralelo a estruturas de dados compartilhadas sem a necessidade de travas que serializam a execução do código.

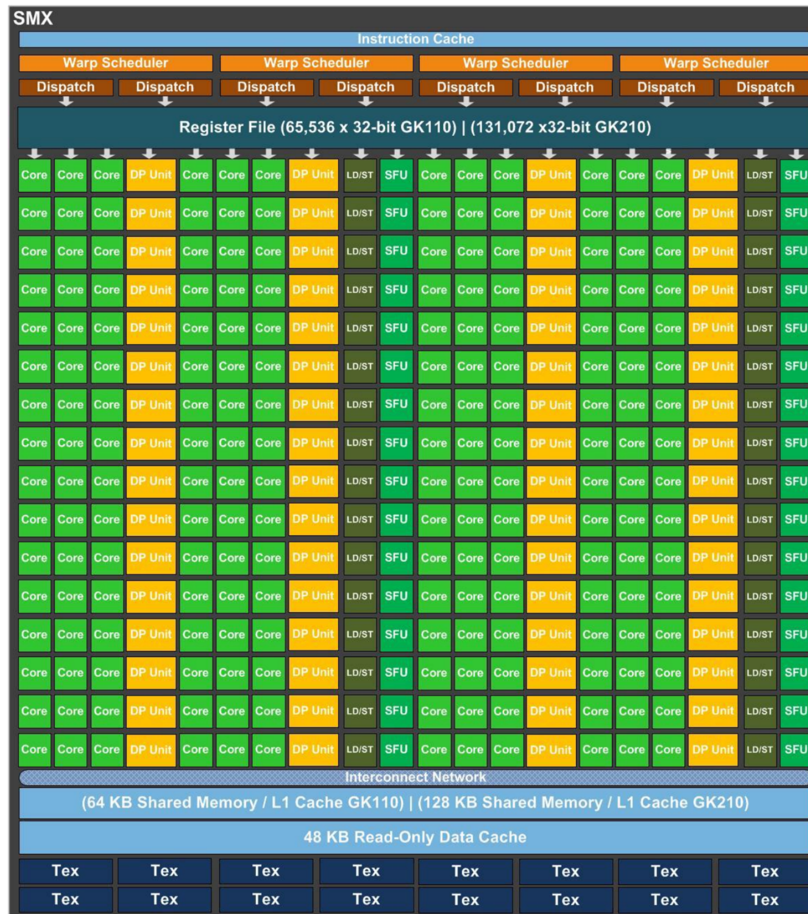


Figura 2.2: Multiprocessador de Fluxo (SM) [NVIDIA 2014]

A arquitetura de memória do acelerador está organizada em diversos níveis, possuindo uma cache L1 para cada SM, além de uma cache apenas de leitura, como visto na Figura 2.3.

A quantidade de memória de cada SM é configurável, por exemplo, a memória local (64 ou 128 KB) pode ser dividida nas seguintes proporções: 75% x 25%, 25% x 75% ou 50% x 50% entre uma memória compartilhada e uma cache L1.

Além da cache L1, a arquitetura Kepler introduz uma cache apenas de leitura de 48KB. O gerenciamento dessa cache pode ser feito automaticamente pelo compilador ou explicitamente pelo programador. O acesso a uma variável ou estrutura de dados que o programador identifica como apenas de leitura, pode ser declarada com a palavra chave *const __restrict*, permitindo ao compilador carregá-la na cache apenas de leitura.

Essa arquitetura possui também um cache de nivel 2 (L2) com 1,5 MB de capacidade. A cache L2 é o ponto primário de unificação de dados entre os diversos SMs, servindo operações de *load, store* e de textura, provendo um compartilhamento de dados eficiente e de alta velocidade.

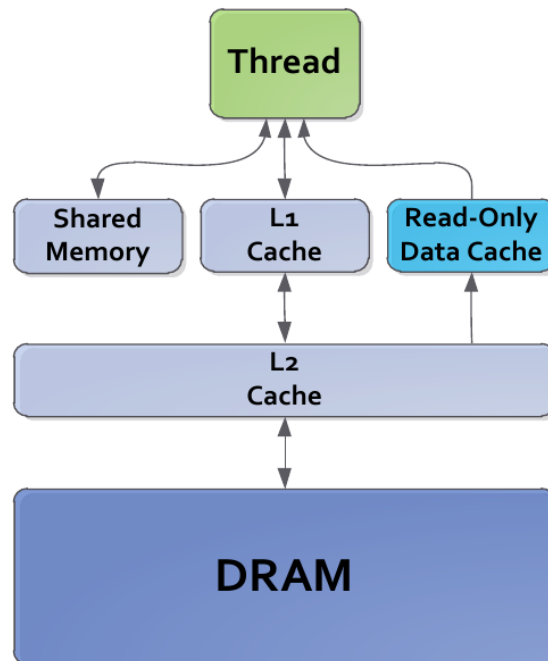


Figura 2.3: Hierarquia de Memória CUDA [NVIDIA 2014]

Algoritmos onde o endereço dos dados é conhecido previamente, tais como solucionadores de física, *ray tracing* e multiplicação esparsa de matrizes, se beneficiam especialmente das hierarquia de cache. Os *kernels* de filtro e convolução onde é necessário que diversos SMs leiam os mesmos dados, também se beneficiam dessa hierarquia.

A arquitetura possui uma série de outras facilidades como código de correção de erro, paralelismo dinâmico, gerenciamento de filas de trabalho e unidade de gerencimento de *grids*, que servem para melhorar o desempenho e a confiabilidade do acelerador. Maiores detalhes podem ser vistos na referência [NVIDIA 2014].

A arquitetura dos aceleradores está em constante evolução, na Tabela 2.1 apresentamos as características de alguns aceleradores NVIDIA lançados mais recentemente.

Para finalizar esta seção, gostaríamos de mostrar como se dá a execução das *threads* no acelerador, apresentando o modelo de programação paralela CUDA. O CUDA é uma combinação de plataforma de *hardware/software* que permite aos aceleradores gáficos executar programas escrito em C, C ++, Fortran e outras linguagens.

Um programa CUDA invoca funções paralelas chamadas *kernels*, que são executados em várias *threads* paralelas. O programador ou compilador organiza essas *threads* em blocos de *thread* e grades de blocos de *thread*, conforme mostrado na Figura 2.4.

Cada *thread* dentro de um *bloco de thread* executa uma instância do *kernel*. Cada *thread* também possui IDs de *thread* e de bloco dentro de seu *bloco de thread* e grade, um contador de programa, registradores, memória privada por *thread*, entradas e resultados de saída.

Características da GPU	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100	Kepler
GPU Codename	GP100	GV100	GA100	GK110
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere	NVIDIA Kepler
Compute Capability	6.0	7.0	8.0	3.5
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	32	32	32	16
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	65536	65536	65536
Max Registers / Thread	255	255	255	255
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	64	64	64	192
Ratio of SM Registers to FP32 Cores	1024	1024	1024	341
Shared Memory Size / SM	64 KB	up to 96 KB	up to 48 KB	up to 48 KB

Tabela 2.1: Comparação entre Arquiteturas NVIDIA

Um *bloco de thread* é um conjunto de *threads* em execução simultânea cooperando entre si por meio de sincronização de barreira e memória compartilhada. Um *bloco de thread* possui um ID de bloco em sua grade. Uma grade é uma matriz de *blocos de thread* que executam o mesmo *kernel*, leem as entradas da memória global, gravam os resultados para a memória global e fazem a sincronização entre chamadas de *kernel* dependentes.

No modelo de programação paralela CUDA, cada *thread* tem um espaço de memória privado por *thread* usado para salvamento de registradores, chamadas de função e variáveis

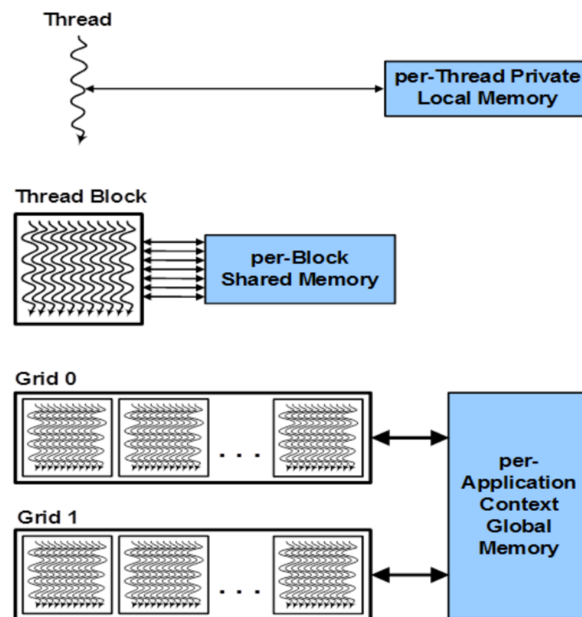


Figura 2.4: Modelo de programação CUDA [NVIDIA 2014]

de arranjo automáticas em "C". Cada *bloco de thread* tem um espaço de memória compartilhado por bloco usado para comunicação inter-thread, compartilhamento de dados e compartilhamento de resultados em algoritmos paralelos. Grades de *blocos de thread* compartilham resultados em um espaço de memória global após a sincronização global em todo o *kernel*.

A hierarquia de *threads* do CUDA é mapeada para uma hierarquia de processadores na GPU; uma GPU executa uma ou mais grades de *kernel*; um multiprocessador de streaming (SM) executa um ou mais *blocos de threads*; e os núcleos CUDA e outras unidades de execução de instruções no SM executam as *threads*.

O SM executa threads em grupos de 32 *threads* chamados *warps*. Embora os programadores possam ignorar os *warps* para uma execução funcionalmente correta, o desempenho das aplicações pode ser muito melhorado, mantendo-se as *threads* de um mesmo *warp* executando o mesmo código e realizando acessos de memória com endereços próximos.

2.2. Memória Unificada (Unified Memory)

As memórias da CPU e GPU são normalmente separadas, ou seja, as informações armazenadas em seus sistemas de memórias não são compartilhadas. A movimentação de dados entre os dois sistemas de memórias é realizada sempre que existe a necessidade de processamento de alguma informação.

Uma forma de solucionar esse problema é o uso de memória unificada. A memória unificada é um espaço de endereçamento único acessível tanto pela CPU como pela GPU. Com o uso dessa tecnologia de *hardware/software*, as aplicações alocam dados que podem ser lidos e escritos tanto pelas CPUs como pelas GPUS (Figura 2.5).

Quando o código em execução em uma CPU ou GPU acessa dados alocados dessa maneira (*CUDA managed memory*), o sistema de *software* CUDA e/ou *hardware* se encarrega de migrar as páginas para a memória do processador que estiver acessando esses dados.

Destacamos que a arquitetura de GPU Pascal é a primeira com suporte de *hardware* para falha e migração de página de memória virtual, por meio de seu mecanismo de migração de página. As GPUs mais antigas baseadas nas arquiteturas Kepler e Maxwell também suportam uma forma mais limitada de memória unificada.

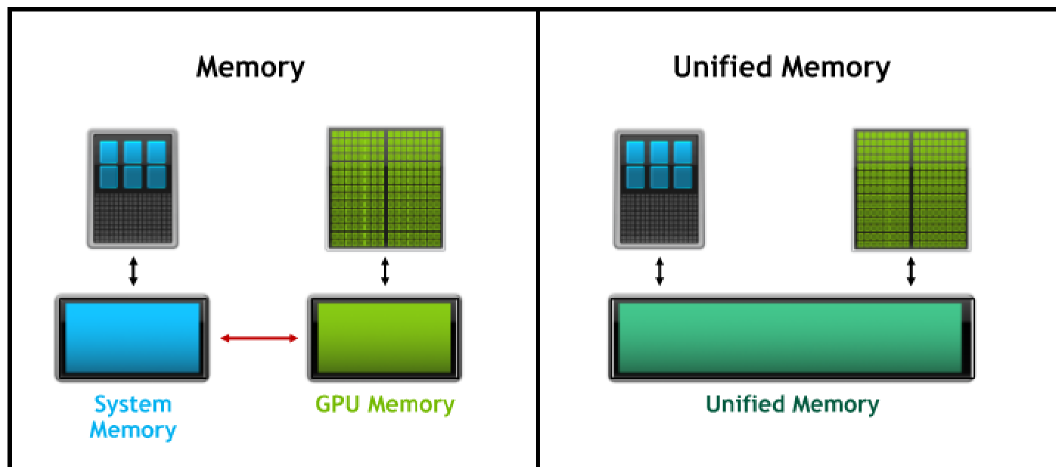


Figura 2.5: Modelo de memória tradicional e memória unificada [Harris 2017]

A alocação de memória unificada pode ser feita explicitamente em CUDA, substituindo-se as chamadas para a função **malloc()** por chamadas à função **cudaMallocManaged()**, uma função de alocação de memória que retorna um ponteiro acessível por qualquer processador.

Já no OpenACC essa substituição das chamadas é feita automaticamente pelo compilador. Para o uso da Memória Unificada no OpenACC com o compilador PGI, basta utilizar usar a opção *-ta=tesla:managed* na compilação do código do programa. Portanto, as cláusulas e diretivas de dados OpenACC não são necessárias para dados "gerenciados". Elas são essencialmente ignoradas e, de fato, podem ser omitidas.

O uso de memória gerenciada se aplica apenas aos dados alocados dinamicamente. Dados estáticos (variáveis externas e estáticas em C, módulos em Fortran, blocos comuns e variáveis salvas) e dados locais das funções são ainda manipulados pelo ambiente de execução do OpenACC.

Embora o uso da memória unificada ofereça uma grande simplificação da programação, o desempenho final é variável em função de cada aplicação e deve ser avaliado com cuidado antes de sua utilização em produção [Harris 2017].

2.3. Conceitos de gang, worker e vector

Como o OpenACC serve como uma linguagem para aceleradores genéricos existem três níveis de paralelismo disponíveis no OpenACC. Eles especificam o nível de paralelismo

contidos na rotina, são chamados de *gangue, trabalhador e vetor*. Uma *gangue* é composta por um ou vários *trabalhadores* e corresponde a um *bloco de threads* em CUDA. Todos os *trabalhadores* de uma *gangue* podem compartilhar os mesmos recursos, como memória cache ou processador.

Em OpenACC, um *trabalhador* é um grupo de *vetores*. A sua dimensão vertical é igual ao número de *trabalhadores* e a dimensão horizontal é o tamanho do *vetor*. A dimensão do *trabalhador* se estende até a altura da gangue (*bloco de threads*). Cada *vetor* OpenACC (um elemento do arranjo iterado) é uma *thread* CUDA. A dimensão do *vetor* se dá ao longo da largura do *bloco de threads*.

Cada *trabalhador* corresponde a um número de *threads* igual ao tamanho do vetor. Então um *trabalhador* corresponde a uma *warp* em CUDA apenas se o *vetor* tiver um comprimento igual a 32; já que um *trabalhador* não corresponde necessariamente a uma *warp*. Por exemplo, um *trabalhador* pode corresponder a duas *warps* se o *vetor* tiver tamanho 64. A característica principal de uma *warp* é que todas as suas *threads* executam concorrentemente. Uma *grade* CUDA é composta de vários *blocos de threads* ou gangues do OpenACC, os quais podem ser organizados em uma ou duas dimensões.

Os aceleradores podem ter limitações quanto aos valores que podem ser atribuídos a esses particionamentos. Por exemplo, para GPUS da NVIDIA, as seguintes limitações existem:

- O comprimento de um *vetor* deve ser um múltiplo de 32 (até 1024)
- O tamanho de uma *gangue* é dado pelo número de *trabalhadores* vezes o tamanho de um *vetor*, não podendo ser maior que 1024.

As diretivas do OpenACC que especificam nível de paralelismo são **gang, worker e vector**, respectivamente para os níveis *gangue, trabalhador e vetor*. Essas diretivas também podem ser combinadas em um laço específico. Por exemplo, um laço **gang vector** pode ser particionado entre *gangues*, cada uma delas com 1 *trabalhador* 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.

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 2.1: Cláusulas da diretiva parallel loop

Adicionalmente, o programador pode definir esses parâmetros dentro de uma região **parallel** ou **kernels** com o uso das cláusulas **num_gangs(N), num_workers(M), vector_ length(Q)**. Esses níveis serão empregados para todos os *kernels* disparados dentro da região.

#pragma acc parallel num_gangs(expr-inteira)

#pragma acc parallel num_workers(expr-inteira)

#pragma acc parallel vector_length(expr-inteira)

2.4. 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).

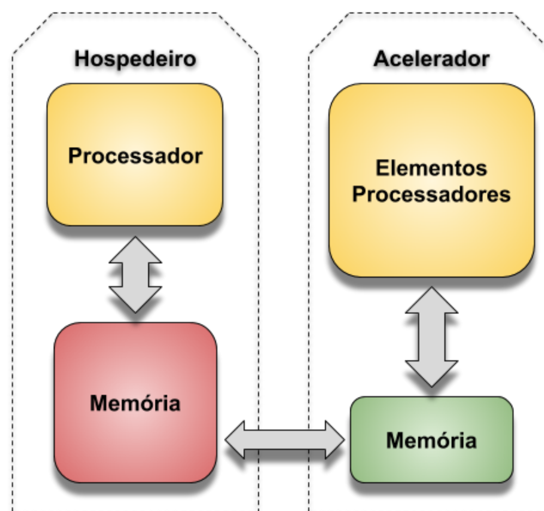


Figura 2.6: Modelo básico de movimentação de dados entre hospedeiro e o acelerador [Chen 2017]

A movimentação de dados entre o hospedeiro 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 hospedeiro 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 **data**, **kernels** ou **parallel**.

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 hospedeiro 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 hospedeiro.
copyout	Cria espaço para as variáveis listadas no dispositivo, mas não as inicia. No final da região, copia os resultados de volta para o hospedeiro 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 2.2: Cláusulas da Diretiva Data

#pragma acc data [cláusula]

2.5. Dicas para a paralelização de laços

A paralelização de estruturas iterativas pode disparar avisos de compilação e, às vezes, é necessário reexpressar o código do laço. Por exemplo, se o programador usa uma diretiva como **kernels ou parallel**, e se o compilador vê alguma dependência entre os laços, o compilador não paralelizará esse trecho do código. Expressando a mesma iteração de uma maneira diferente, pode ser possível evitar os avisos de compilação e fazer com que o compilador execute os laços em paralelo. Os exemplos de código abaixo ilustram laços que geram mensagens de erro do compilador.

```
#pragma acc kernels
{
   while (i < N && found == -1) {
      if (A[i] >= 102.0f) {
         found = i;
      }
      ++i;
   }
}
```

Compilando o código acima os seguintes avisos são gerados pelo compilador:

```
Accelerator restriction: loop has multiple exits
Accelerator region ignored
```

O problema aqui é que “i” poderia assumir valores diferentes quando o *loop while* é encerrado, dependendo se uma *thread* em execução encontra um valor de A [i] maior ou igual a 102,0. O valor de “i” vai variar de execução para execução e não produzirá o resultado que o programador pretendia.

Re-expressando o código com o laço “for” a seguir, com uma lógica de desvio, o compilador agora reconhece o primeiro laço como sendo paralelizável.

```
#pragma acc kernels
{
   for (i=0; i<N; ++i) {
      if (A[i] >= 102.0f) {
         found[i] = i;
      }
      else {
         found[i] = -1;
      }
   }
}
i=0;
while (i < N && found[i] < 0) {
   ++i;
}
```

Embora esse código seja um pouco maior, com dois laços, acelerar o primeiro laço compensa a separação de um laço em dois. Normalmente, separar um laço em dois é ruim para o desempenho, mas nesse caso ao expressaros laços paralelos temos um ganho de desempenho [Murphy 2016].

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 *restrict* 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 hospedeiro 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 hospedeiro e o dispositivo entre os dois laços, o que pode resultar em menor movimentação de dados por padrão.

2.6. Diretivas e Cláusulas Avançadas

2.6.1. 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.

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.

Esta cláusula é especialmente útil quando alguns laços não tem um número total de iterações suficientemente grande para fazer uso efetivo do acelerador. A sua sintaxe é vista a seguir:

#pragma acc loop collapse(n)

O exemplo a seguir apresenta um trecho de código com um laço com o uso desta cláusula, seguido de um laço que exemplifica o efeito do seu uso.

```
#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++)...
```

2.6.2. 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 2.2: Diretiva Routine

Quando a rotina *foo* é chamada a partir do código do hospedeiro, ele será executado no hospedeiro, incrementando os valores do hospedeiro. 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.

2.6.3. 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* executando a operação nela contida simultaneamente. O uso da diretiva **atomic** às vezes é 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 2.3: 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.

2.6.4. Cláusula tile

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 2.4: 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.

2.7. Cláusulas device_type e vector_length

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 determinadas arquiteturas.

Apenas as cláusulas **async, wait, num_gangs, num_workers e vector_length** podem

aparecer em seguida a uma cláusula **device_type**.

#pragma acc loop device_type(lista-tipo-dispositivo)

Um exemplo de cláusula que pode ser utilizada em associação com a cláusula **device_type** é a **vector_length**, que é utilizada para especificar o tamanho do vetor que será usado em um laço paralelo (com a diretiva **parallel loop**).

No Exemplo 2.5 é especificado um comprimento difernte de vetor, dependendo do tipo de acelerador que será usado. Nesse 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.

```
#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 2.5: Cláusula tile

2.8. PGI Profiler

O PGI Profiler é uma ferramenta utilizada para analisar o desempenho de programas paralelos escritos com OpenMP, MPI, OpenACC ou CUDA. O PGI Profiler permite a visualização e otimização do desempenho de uma aplicação através da análise da linha do tempo de execução da aplicação. Com isso é possível identificar regiões de gargalos que podem ser otimizadas, eliminando ou reduzindo esses gargalos para alcançar um melhor desempenho.

Para iniciar a análise da execução de uma aplicação é utilizamos primeiro o comando **pgprof**. Esse comando gera um arquivo de saída com as informações de uso dos recursos computacionais em todos os trechos da aplicação. Não é necessário nenhum tipo de alteração no código para criar o arquivo de saída, entretanto, existem alguns parâmetros do compilador PGI que podem ser usados para coletar mais informações de uso dos recursos. Veja mais detalhes na referência [PGI 2019].

Após a execução do comando **pgprof**, a análise da aplicação pode ser realizada com o uso de comandos em modo terminal ou gráfico. Para executar no modo terminal, use o comando:

 # pgprof [parametro] [aplicação]

Em modo gráfico executar somente o comando:

 # pgprof

2.9. Exemplos

Após a introdução dos conceitos e das principais diretivas que serão usadas neste minicurso, veremos alguns exemplos de aplicações dessas diretivas, como se comportam e quais as melhores opções do seu uso para melhorar o desempenho.

2.9.1. Cálculo de Pi

Iniciaremos com o exemplo o cálculo do número Pi. Em computação existem diversos algoritmos que podem ser utilizados para o cálculo aproximado do Pi. No Exemplo 2.6 é apresentada uma implementação do cálculo de Pi utilizando uma integral cujo resultado é aproximado com uso do método do trapézio.

```
#include <stdio.h>
#define N 1000000000
int main(int argc, char *argv[]) { /* calcpi_seq.c  */
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 2.6: Cálculo de Pi sequencial

Para execução do código em paralelo no acelerador podem ser utilizadas as diretivas **kernels ou parallel** do OpenACC. Para o uso da diretiva **kernels**, adicionaremos a linha:

#pragma acc kernels.

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

Para o uso da diretiva parallel adicionaremos, sempre antes do laço, a linha:

#pragma acc parallel loop reduction(+: pi)

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

Durante a compilação do código é possível observar o resultado do uso dessas diretivas (Figura 2.7).

```
$  pgcc -acc -ta=nvidia -Minfo=all piacc_kernels.c -o piacc_kernels

main:
     11, Loop is parallelizable
         Generating Tesla code
         11, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         14, Generating implicit reduction(+:pi)
$
$  pgcc -acc -ta=nvidia -Minfo=all piacc_parallel.c -o piacc_parallel

main:
     10, Generating Tesla code
         11, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:pi)
     10, Generating implicit copy(pi) [if not already present]
$
```

Figura 2.7: Saída do compilador PGI

```
$  pgcc -acc -ta=nvidia -Minfo=all piacc_kernels.c -o piacc_kernels
main:
     10, Generating copy(pi) [if not already present]
     13, Loop is parallelizable
         Generating Tesla code
         13, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         16, Generating implicit reduction(+:pi)
$
$  pgcc -acc -ta=nvidia -Minfo=all piacc_parallel.c -o piacc_parallel

main:
     10, Generating copy(pi) [if not already present]
     12, Generating Tesla code
         13, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:pi)
$
```

Figura 2.8: Saída do compilador PGI usando movimentação de dados

Na diretiva **kernels** a redução sempre é feita de forma implícita pelo compilador, mas a movimentação dos dados para o acelerador não ocorre. Por sua vez, com uso da diretiva **parallel**, a redução é realizada de forma explícita e a movimentação dos dados é feita de forma implícita pelo compilador.

É possível fazer a movimentação de dados explicitamente para o acelerador usando a diretiva **data**. Deve-se então adicionar a linha **#pragma acc data copy(pi)** antes do laço, de modo que seja feita a cópia da variável *pi* para a memória do acelerador. Desse modo, todo o cálculo da variável *pi* é realizado no acelerador (Figura 2.8).

Em alguns casos, o uso de movimentação de dados do hospedeiro para o acelerador pode ser mais lento por conta custo computacional. Neste exemplo, o tempo gasto em movimentar os dados para o acelerador não compensa devido ao baixo custo computacional do

kernel, sendo mais eficiente realizar esses cálculos no hospedeiro. Na Figura 2.9 e Figura 2.10 os tempos de execução usando a movimentação de dados são maiores tanto com a diretiva **kernels** como com a diretiva **parallel**.

```
$ ./piacc_kernels
O valor de pi é: 3.141593
O tempo de execução é 0.905411 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/piacc_kernels.c
  main  NVIDIA  devicenum=0
    time(us): 48
    10: compute region reached 1 time
        10: data copyin transfers: 1
             device time(us): total=25 max=25 min=25 avg=25
        11: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=644,773 max=644,773 min=644,773 avg=644,773
        11: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=121 max=121 min=121 avg=121
        11: data copyout transfers: 1
             device time(us): total=23 max=23 min=23 avg=23
$
$ ./piacc_kernelsdm
O valor de pi é: 3.141593
O tempo de execução é 0.945265 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/piacc_kernelsdm.c
  main  NVIDIA  devicenum=0
    time(us): 71
    10: data region reached 2 times
        10: data copyin transfers: 1
             device time(us): total=16 max=16 min=16 avg=16
        19: data copyout transfers: 1
             device time(us): total=55 max=55 min=55 avg=55
    12: compute region reached 1 time
        13: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=646,115 max=646,115 min=646,115 avg=646,115
        13: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=122 max=122 min=122 avg=122
$
```

Figura 2.9: Tempo de execução usando movimentação de dados com a diretiva kernels

```
$ ./piacc_parallel
O valor de pi é: 3.141593
O tempo de execução é 0.943007 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/piacc_parallel.c
  main  NVIDIA  devicenum=0
    time(us): 70
    10: compute region reached 1 time
        10: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=644,827 max=644,827 min=644,827 avg=644,827
        10: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=124 max=124 min=124 avg=124
    10: data region reached 2 times
        10: data copyin transfers: 1
             device time(us): total=16 max=16 min=16 avg=16
        16: data copyout transfers: 1
             device time(us): total=54 max=54 min=54 avg=54
$
$ ./piacc_paralleldm
O valor de pi é: 3.141593
O tempo de execução é 0.943742 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/piacc_paralleldm.c
  main  NVIDIA  devicenum=0
    time(us): 66
    10: data region reached 2 times
        10: data copyin transfers: 1
             device time(us): total=16 max=16 min=16 avg=16
        19: data copyout transfers: 1
             device time(us): total=50 max=50 min=50 avg=50
    12: compute region reached 1 time
        12: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=644,876 max=644,876 min=644,876 avg=644,876
        12: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=122 max=122 min=122 avg=122
$
```

Figura 2.10: Tempo de execução usando movimentação de dados com a diretiva parallel

Para medir os tempos de execução dos diferentes códigos, variou-se o valor de N. Os valores de N usados para as medições foram entre $1,0 \times 10^9$ a $1,5 \times 10^{10}$. Na Tabela 2.3, são apresentados os resultados obtidos para o tempo de processamento.

Tamanho de N	kernels	kernels+data	parallel	parallel+data
$1,0 \times 10^9$	$0,305284$	$0,344658$	$0,343664$	$0,343403$
$2,0 \times 10^9$	$0,346696$	$0,384993$	$0,384135$	$0,385037$
$1,0 \times 10^{10}$	$0,688010$	$0,723660$	$0,723909$	$0,728856$
$1,5 \times 10^{10}$	$0,905921$	$0,943896$	$0,944225$	$0,944746$

Tabela 2.3: Uso de recursos do acelerador

Nas Figura 2.11 e Figura 2.12, são apresentadas as análises do comportamento do cálculo

de Pi com uso do **pgprof** com o uso de ambas diretivas, com movimentação de dados (diretiva **data**). Como visto anteriormente, o comportamento é o mesmo independente da diretiva utilizada.

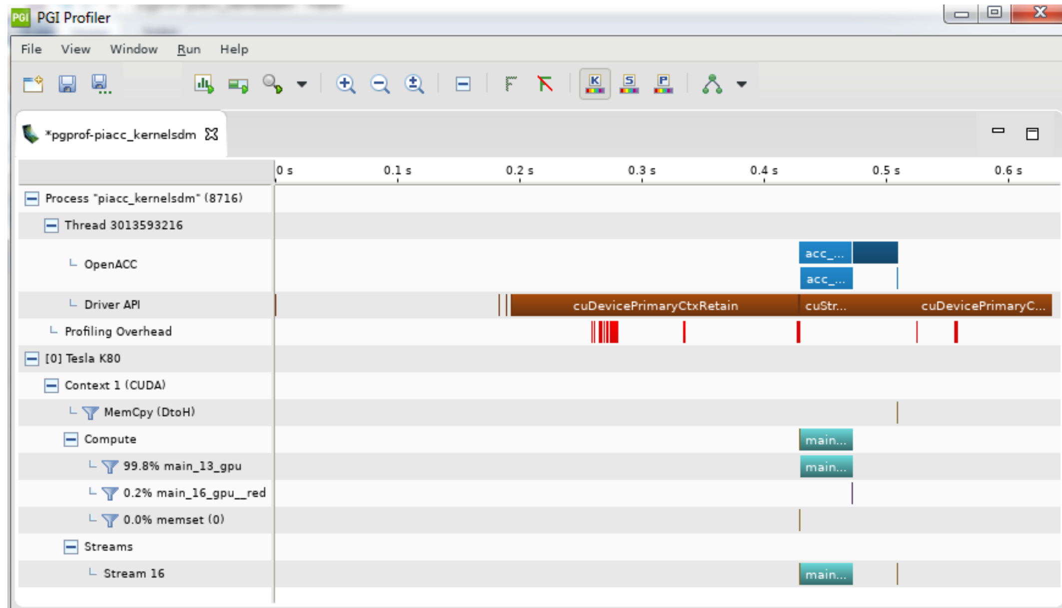


Figura 2.11: Análise de execução do código usando a diretiva kernels com movimentação de dados

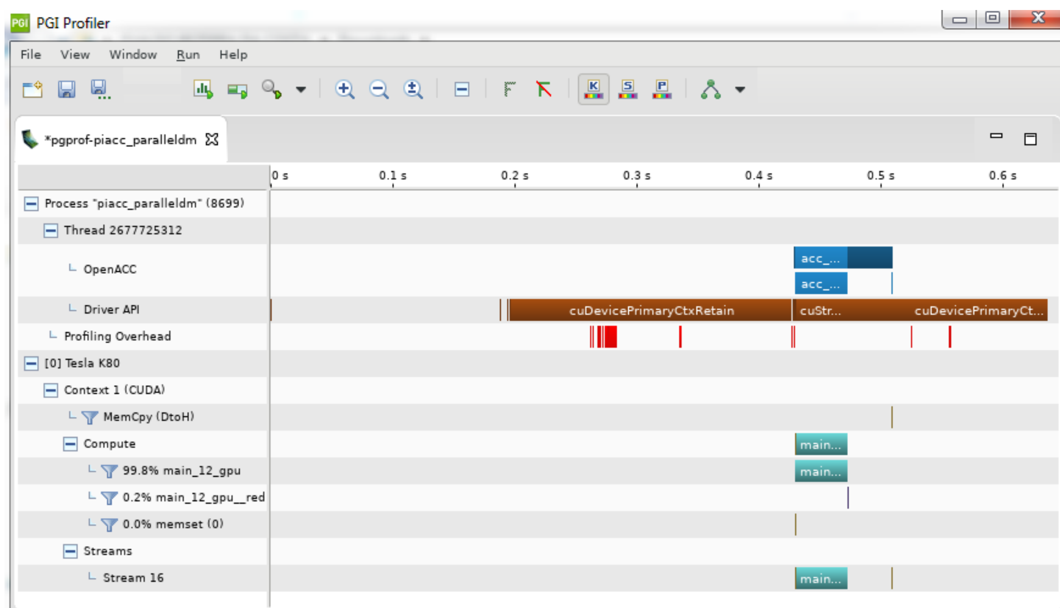
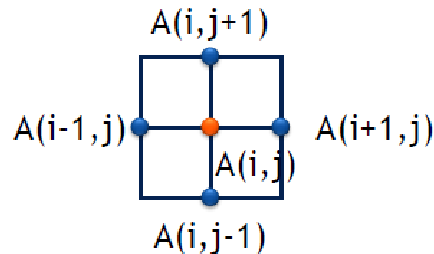


Figura 2.12: Análise de execução do código usando a diretiva parallel com movimentação de dados

2.9.2. Método Jacobi

O Método de Jacobi é um procedimento iterativo para a resolução de sistemas lineares. Converge 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: $\nabla^2 f(x,y) = 0$.



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

```
#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], A[ROWS+2][COLUMNS+2];
void initialize();
int main(int argc, char *argv[]) {
    int i, j, iteration=1, max_iterations=1000;
    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);
}
```

Exemplo 2.7: Método de Jacobi Sequencial

Como pode ser visto no código sequencial (Exemplo 2.7), o primeiro laço dentro do *while* de convergência calcula o novo valor para cada elemento com base nos valores atuais de seus vizinhos, cujo resultado é armazenado em uma matriz temporária **Anew**. Isso garante que todos os valores sejam calculados usando o estado atual de **A** antes que o conteúdo de **A** seja novamente atualizado. Como resultado, cada iteração do laço é completamente independente da outra.

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

Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Adicionar a linha antes do primeiro laço:

#pragma acc parallel loop

E a linha:

#pragma acc parallel loop reduction(max:dt)

Antes do segundo laço, como visto a seguir.

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

Na Figura 2.13 vemos a saída do compilador PGI. A movimentação dos dados para o acelerador é realizada de forma automática pelo compilador. Desse modo 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 hospedeiro. Esta operação tem um alto custo computacional, tornando a execução do código lenta (Figura 2.14).

```
$ pgcc -acc -ta=tesla -Minfo=all jacobiacc.c -o jacobiacc

main:
     29, Generating Tesla code
         30, #pragma acc loop gang /* blockIdx.x */
         31, #pragma acc loop vector(128) /* threadIdx.x */
     29, Generating implicit copyin(A[:][:]) [if not already present]
         Generating implicit copyout(Anew[1:1000][1:1000]) [if not already present]
     31, Loop is parallelizable
     39, Generating Tesla code
         40, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:dt)
         41, #pragma acc loop vector(128) /* threadIdx.x */
     39, Generating implicit copy(A[1:1000][1:1000],dt) [if not already present]
         Generating implicit copyin(Anew[1:1000][1:1000]) [if not already present]
     41, Loop is parallelizable
initialize:
     58, Memory zero idiom, loop replaced by call to __c_mzero8
$
```

Figura 2.13: Saída do compilador PGI sem uso de movimentação de dados

```
$ ./jacobiacc
Max error at iteration 1000 was 0.034767
O tempo de execução é 11.718717 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/jacobiacc.c
  main  NVIDIA  devicenum=0
    time(us): 3,339,276
    29: compute region reached 1000 times
        29: kernel launched 1000 times
            grid: [1000]  block: [128]
            elapsed time(us): total=152,516 max=181 min=147 avg=152
    29: data region reached 2000 times
        29: data copyin transfers: 1000
             device time(us): total=671,517 max=690 min=668 avg=671
        37: data copyout transfers: 1000
             device time(us): total=660,866 max=674 min=659 avg=660
    39: compute region reached 1000 times
        39: kernel launched 1000 times
            grid: [1000]  block: [128]
            elapsed time(us): total=275,134 max=421 min=268 avg=275
        39: reduction kernel launched 1000 times
            grid: [1]  block: [256]
            elapsed time(us): total=22,640 max=67 min=20 avg=22
    39: data region reached 2000 times
        39: data copyin transfers: 3000
             device time(us): total=1,336,393 max=683 min=5 avg=445
        47: data copyout transfers: 2000
             device time(us): total=670,500 max=678 min=7 avg=335
$
```

Figura 2.14: Tempo de execução sem uso de movimentação de dados

Para resolver esse problema é necessário informar ao compilador que uma cópia da matriz **A** deve ser feita para o acelerador no início da região paralela. Desse modo, não será

mais necessário gravar as informações no hospedeiro toda vez que for realizada uma operação de escrita na matriz pelo acelerador. Adicionalmente, a matriz **Anew** será criada exclusivamente no acelerador. Usaremos a diretiva **data** do OpenACC para realizar essas operações. Deve-se então adicionar a linha antes dos dois laços:

#pragma acc data copy(A) create (Anew)

```
#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];
            }
        }
        iteration++;
    }
}
```

Agora todas os acessos às matrizes **A** e **Anew** serão realizados exclusivamente no acelerador, com isto, não haverá custo adicional de acesso aos dados na memória do hospedeiro para o cálculo da matriz (Figura 2.15).

```
$ pgcc -acc -ta=tesla -Minfo=all jacobiaccdm.c -o jacobiaccdm

main:
     27, Generating create(Anew[:][:]) [if not already present]
         Generating copy(A[:][:]) [if not already present]
     30, Generating Tesla code
         31, #pragma acc loop gang /* blockIdx.x */
         32, #pragma acc loop vector(128) /* threadIdx.x */
     32, Loop is parallelizable
     40, Generating Tesla code
         41, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:dt)
         42, #pragma acc loop vector(128) /* threadIdx.x */
     40, Generating implicit copy(dt) [if not already present]
     42, Loop is parallelizable
initialize:
     59, Memory zero idiom, loop replaced by call to __c_mzero8
$
```

Figura 2.15: Saída com do compilador PGI usando movimentação de dados

O tempo total gasto para a execução sem a movimentação de dados foi de 11,71 segundos.

Usando a movimentação de dados o tempo total passou para 0,80 segundos, ou seja, 14,62 vezes mais rápido. Este é um exemplo onde o uso de movimentação de dados propicia ganhos consideráveis (Figura 2.16).

```
$ ./jacobiaccdm
Max error at iteration 1000 was 0.034767
O tempo de execução é 0.809362 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/jacobiaccdm.c
  main  NVIDIA  devicenum=0
    time(us): 15,135
    27: data region reached 2 times
        27: data copyin transfers: 1
             device time(us): total=689 max=689 min=689 avg=689
        50: data copyout transfers: 1
             device time(us): total=654 max=654 min=654 avg=654
    30: compute region reached 1000 times
        30: kernel launched 1000 times
            grid: [1000]  block: [128]
            elapsed time(us): total=149,177 max=182 min=144 avg=149
    40: compute region reached 1000 times
        40: kernel launched 1000 times
            grid: [1000]  block: [128]
            elapsed time(us): total=267,672 max=278 min=261 avg=267
        40: reduction kernel launched 1000 times
            grid: [1]  block: [256]
            elapsed time(us): total=20,539 max=37 min=19 avg=20
    40: data region reached 2000 times
        40: data copyin transfers: 1000
             device time(us): total=5,599 max=14 min=4 avg=5
        48: data copyout transfers: 1000
             device time(us): total=8,193 max=23 min=7 avg=8
$
```

Figura 2.16: Tempo de execução usando movimentação de dados

As Figuras 2.17 e 2.18 apresentam os resultados gerados usando o **pgprof**. Na Figura 2.17 a análise do código foi realizada sem movimentação de dados e a Figura 2.18 com a movimentação de dados para o acelerador.

Nota-se que o tempo e o custo computacional são maiores quando não é feita a movimentação de dados, visto que toda vez que se faz uma operação na matriz, existe a necessidade do acelerador fazer o acesso aos dados no hospedeiro. Quando a matriz é movimentada de forma definitiva para o acelerador, esse custo computacional é muito menor.

2.9.3. Cálculo do fractal de Mandelbrot

Os fractais são figuras geométricas complexas que apresentam como característica principal a autossimilaridade. O fractal de Mandelbrot é um fractal definido como o conjunto de pontos C no plano complexo. O conjunto de Mandelbrot é obtido quando submetemos

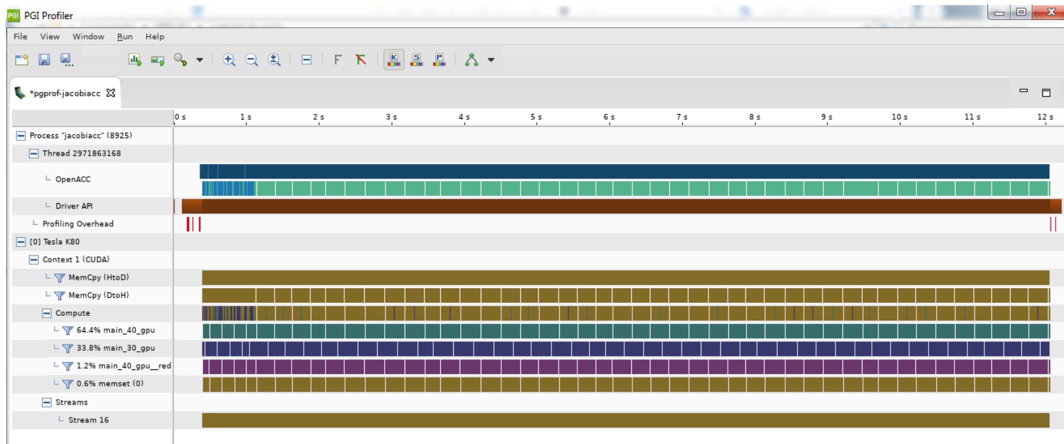


Figura 2.17: Análise de execução do código sem uso de movimentação de dados

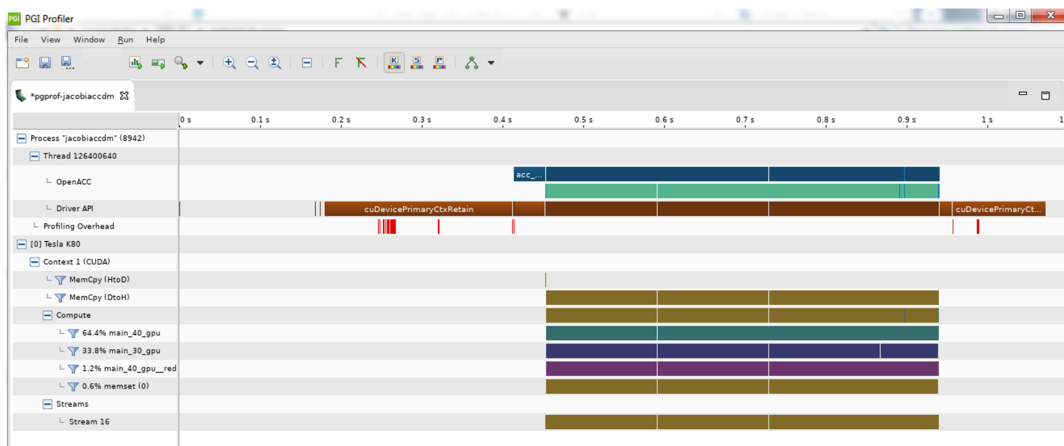


Figura 2.18: Análise de execução do código usando movimentação de dados

os números complexos a um processo iterativo e recursivo utilizando a fórmula:

$$Z_{n+1} = Z_n{}^2 + C \quad (1)$$

A execução do fractal de Mandelbrot dentro do laço na versão sequencial (Exemplo 2.8) será feito por uma *thread*, independente da quantidade de processadores que existam no sistema (Figura 2.19).

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <omp.h>
#include <stdlib.h>
#define X_RESN 800 /* x resolution */
#define Y_RESN 800 /* y resolution */
typedef struct complextype
{
  float real, imag;
} Compl;
void main(int argc, char *argv[]) {
  unsigned int width, height,
      x, y,
      border_width,
      display_width, display_height,
      screen;
  char *window_name = "Mandelbrot", *display_name = NULL;
  unsigned long valuemask = 0;
  FILE *fp, *fopen();
  char str[100];
  int i, j, k;
  Compl z, c;
  float lengthsq, temp;
  width = X_RESN;
  height = Y_RESN;
  x = 0;
  y = 0;
  border_width = 4;
  double t_inicio = omp_get_wtime();
  int counter = 0;
  for (i = 0; i < X_RESN; i++)
    for (j = 0; j < Y_RESN; j++)
    {
      z.real = z.imag = 0.0;
      c.real = ((float)j - 400.0) / 200.0;
      c.imag = ((float)i - 400.0) / 200.0;
      k = 0;
      do
      {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2.0 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        k++;
      } while (lengthsq < 4.0 && k < 100000);
      if (k == 100000){
          counter++;
      }
    }
  double t_fim = omp_get_wtime();
  printf("Imagem: %d e Y: %d \n", X_RESN, Y_RESN);
  printf("Tempo: \t %f\n", t_fim-t_inicio);
  printf("Counter %d\n", counter);
}
```

Exemplo 2.8: Mandelbrot Sequencial

```
$ pgcc -Minfo=all mandelbrotserial.c -o mandelbrotserial

main:
     84, FMA (fused multiply-add) instruction(s) generated
$
$ ./mandelbrotserial
Com tamanho de imagem X: 800 e Y: 800
Tempo para executar sequencial:          73.652713
Counter 60312
$
```

Figura 2.19: Compilação Mandelbrot sequencial e tempo de execução

O tempo total para a execução sequencial foi de 73,65 segundos. Para a versão em OpenACC (Figura 2.20), usaremos inicialmente a diretiva **parallel** adicionando a linha:

#pragma acc parallel loop copy

```
#pragma acc parallel loop copy(counter)
for (i = 0; i < X_RESN; i++)
  for (j = 0; j < Y_RESN; j++)
```

```
$ pgcc -acc -ta=tesla -Minfo=all mandelbrotacc.c -o mandelbrotacc
main:
     52, Generating copy(counter) [if not already present]
         Generating Tesla code
         53, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         54, #pragma acc loop seq
         74, Generating implicit reduction(+:counter)
     54, Loop carried scalar dependence for counter at line 74
     69, Accelerator restriction: induction variable live-out from loop: k
     70, Accelerator restriction: induction variable live-out from loop: k
$
$ ./mandelbrotacc
Com tamanho de imagem X: 800 e Y: 800
Tempo para executar paralelizado:        5.361158
Counter 60314

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/mandelbrotacc.c
  main  NVIDIA  devicenum=0
    time(us): 69
    52: compute region reached 1 time
        52: kernel launched 1 time
            grid: [7]  block: [128]
            elapsed time(us): total=5,056,435 max=5,056,435 min=5,056,435 avg=5,056,435
        52: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=76 max=76 min=76 avg=76
    52: data region reached 2 times
        52: data copyin transfers: 1
             device time(us): total=15 max=15 min=15 avg=15
        78: data copyout transfers: 1
             device time(us): total=54 max=54 min=54 avg=54
$
```

Figura 2.20: Compilação do código usando OpenACC e o tempo de execução

Embora esta versão do código usando OpenACC tenha um tempo de execução de 5,36

segundos, sendo 13,74 vezes mais rápido que a versão sequencial, podemos melhorar o código usando outras diretivas. O tempo de execução pode ainda ser diminuído com o uso da diretiva **atomic**:

#pragma acc atomic update

```
        if (k == 100000){
          #pragma acc atomic update
            counter++;
        }
      }
```

Com o uso dessa diretiva o tempo de execução passou para 0,40 segundos. Nessa nova versão do código o tempo de execução foi 13,40 vezes mais rápido em comparação de primeira versão do OpenACC e 184,12 mais rápido vezes em comparação a versão sequencial (Figura 2.21).

```
$ pgcc -acc -ta=tesla -Minfo=all mandelbrotacc.c -o mandelbrotacc2

main:
     52, Generating copy(counter) [if not already present]
         Generating Tesla code
         53, #pragma acc loop gang /* blockIdx.x */
         54, #pragma acc loop vector(128) /* threadIdx.x */
     54, Loop is parallelizable
     69, Accelerator restriction: induction variable live-out from loop: k
     70, Accelerator restriction: induction variable live-out from loop: k
$
$ ./mandelbrotacc2
Com tamanho de imagem X: 800 e Y: 800
Tempo para executar paralelizado:        0.406515
Counter 60314

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/mandelbrotacc.c
  main  NVIDIA  devicenum=0
    time(us): 72
    52: compute region reached 1 time
        52: kernel launched 1 time
            grid: [800]  block: [128]
            elapsed time(us): total=103,340 max=103,340 min=103,340 avg=103,340
    52: data region reached 2 times
        52: data copyin transfers: 1
             device time(us): total=14 max=14 min=14 avg=14
        78: data copyout transfers: 1
             device time(us): total=58 max=58 min=58 avg=58
$
```

Figura 2.21: Compilação do código usando a diretiva atomic e o tempo de execução

O uso da diretiva **parallel** e a movimentação de dados obteve um ganho significativo em relação a versão sequencial (Figura 2.22). Porém, quando foi feito o uso da diretiva **atomic**, esse ganho foi muito superior a versão sequencial (Figura 2.23).

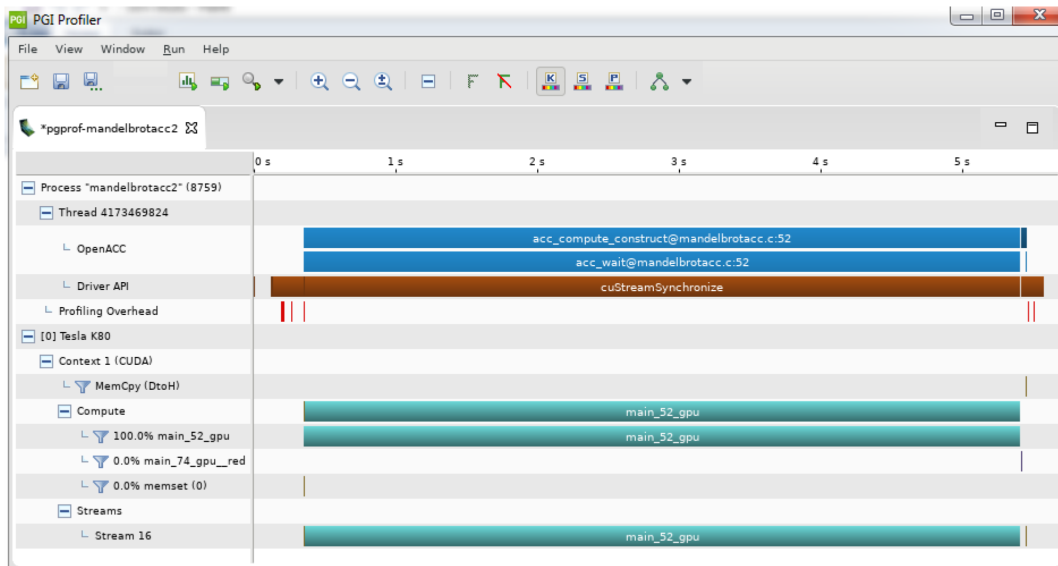


Figura 2.22: Análise de execução do código usando movimentação de dados

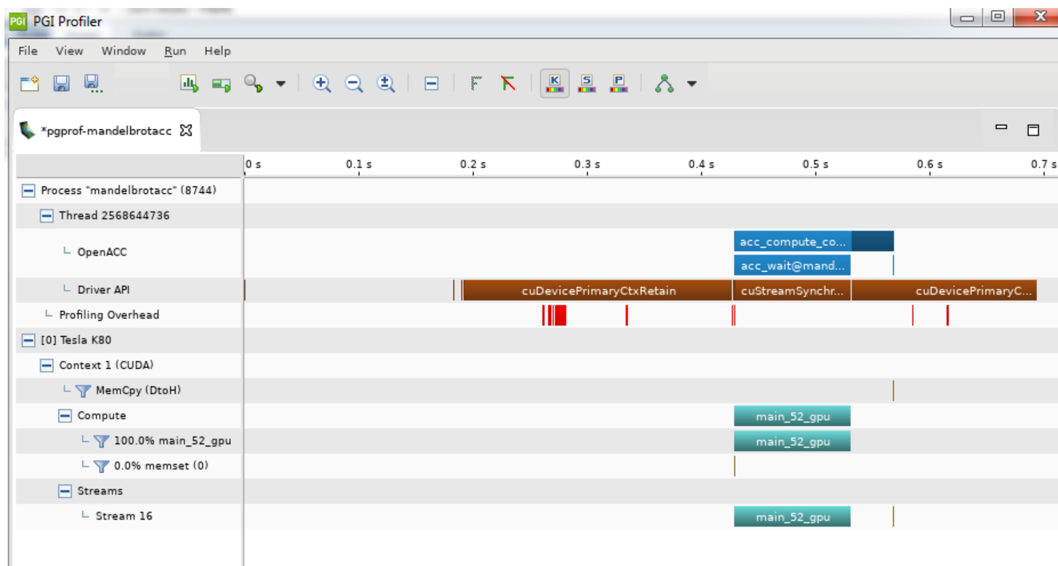


Figura 2.23: Análise de execução do código usando movimentação de dados e a diretiva atomic

2.9.4. Cálculo de números primos

O código do Exemplo 2.9 calcula a quantidade de números primos entre 0 e um determinado valor inteiro N. Este programa basicamente verifica se N é divisível por algum número ímpar entre 0 e a raiz quadrada de N, sendo que os números pares são descartados de imediato.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
int primo (long int n) {
        for (long int i = 3; i < (long int)(sqrt(n) + 1); i+=2)
             if (n%i == 0)
                 return 0;
        return 1;
}
int main(int argc, char *argv[]) {
    long long int i, n, quantidadePrimos = 0;
    if (argc < 2) {
        printf("Valor invalido! Entre com o valor do maior inteiro
    \n");
        return 0;
    }
    else {
        n = strtol(argv[1], (char **) NULL, 10);
    }
    double inicio = omp_get_wtime();
    for (i = 3; i <= n; i += 2)
         if(primo(i) == 1) quantidadePrimos++;
    quantidadePrimos += 1;
    double fim = omp_get_wtime();
    printf("Quantidade de numeros primos entre 1 e %ld e : %ld \n"
    , n, quantidadePrimos);
    printf("O tempo de execucao foi de : %f \n", fim-inicio);
    return(0);
}
```

Exemplo 2.9: Primos Sequencial

Na Figura 2.24 pode ser vista a compilação do código e o tempo total para a execução sequencial, que foi de 16,12 segundos. Para a versão em OpenACC, primeiramente usaremos as diretivas **routine** e **parallel** dentro da rotina **primo()**. Para o uso da diretiva **routine** adicionar a linha:

#pragma acc routine

Antes de iniciar a rotina e a linha:

#pragma acc loop

Antes do primeiro laço para calcular a raiz quadrada.

```
#pragma acc routine
int primo (long int n) {
    #pragma acc loop
        for (long int i = 3; i < (long int)(sqrt(n) + 1); i+=2)
             if (n%i == 0)
                 return 0;
        return 1;
}
```

Para o programa principal adicionaremos a linha:

#pragma acc parallel loop reduction(+:quantidadePrimos)

Antes do segundo laço para calcular a quantidade de números primos existentes no intervalo. O resultados podem ser vistos na Figura 2.25.

```
#pragma acc parallel loop reduction(+:quantidadePrimos)
for (i = 3; i <= n; i += 2)
     if(primo(i) == 1) quantidadePrimos++;
quantidadePrimos += 1;
```

```
$ pgcc -Minfo=all primosserial.c -o primosserial
$
$ ./primosserial 10000000
Quantidade de números primos entre 1 e 10000000 é : 664579
O tempo de execução foi de : 16.123722
$
```

Figura 2.24: Compilação do código sequencial e tempo de execução

```
$ pgcc -acc -ta=tesla -Minfo=all primosacc.c -o primosacc

primo:
      7, Generating acc routine seq
         Generating Tesla code
main:
     33, Generating Tesla code
         34, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:quantidadePrimos)
     33, Generating implicit copy(quantidadePrimos) [if not already present]
$
$ ./primosacc 10000000
Quantidade de números primos entre 1 e 10000000 é : 664579
O tempo de execução foi de : 0.437250

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/primosacc.c
  main  NVIDIA  devicenum=0
    time(us): 68
    33: compute region reached 1 time
        33: kernel launched 1 time
            grid: [39063]  block: [128]
            elapsed time(us): total=137,373 max=137,373 min=137,373 avg=137,373
        33: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=89 max=89 min=89 avg=89
    33: data region reached 2 times
        33: data copyin transfers: 1
             device time(us): total=14 max=14 min=14 avg=14
        37: data copyout transfers: 1
             device time(us): total=54 max=54 min=54 avg=54
$
```

Figura 2.25: Compilação do código usando OpenACC e o tempo de execução

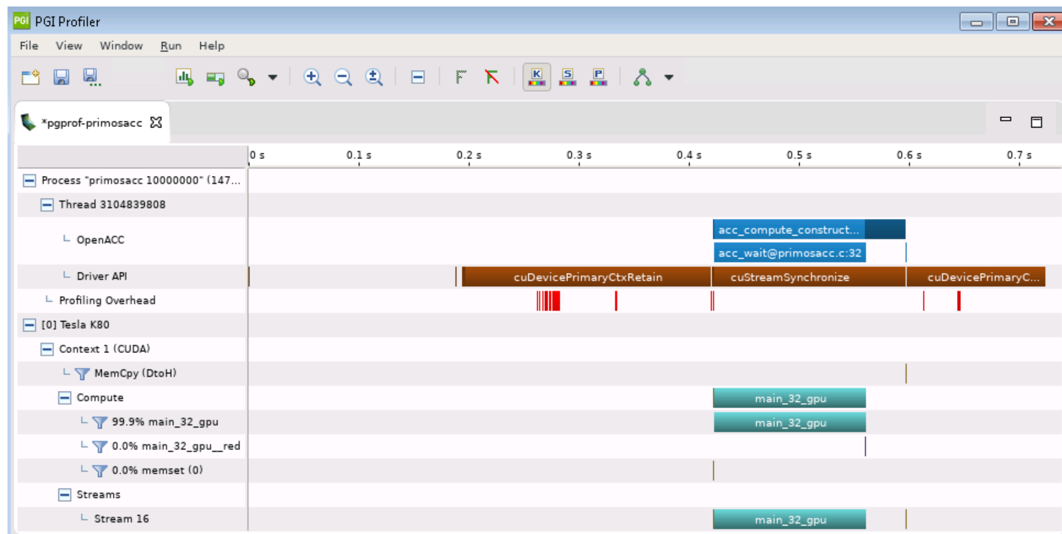


Figura 2.26: Análise de execução do código usando as diretiva Routine e parallel

A Figura 2.26 apresenta a análise da execução do código para o cálculo de números primos usando as diretivas **routine** e **parallel** utilizando o programa **pgprof**.

2.9.5. Cálculo de multplicação de matrizes

A multiplicação de matrizes corresponde ao produto entre duas matrizes. O produto de duas matrizes só é possível somente quando o número de colunas da primeira matriz é igual ao número de linhas da segunda matriz. O algoritmo que iremos apresentar, conhecido como “ingênuo”, apresenta complexidade computacional $O(mnp)$, para a multiplicação de uma matriz $m\ x\ n$ por outra $n\ x\ p$, se todas as dimensões forem iguais a “n”, diz-se que a complexidade é $O(n^3)$.

Utilizando-se álgebra linear, podem-se obter algoritmos que podem alcançar complexidades melhores, tais como o algoritmo do alemão Volker Strassen, que consegue uma complexidade de $O(n^{2,807})$ pela redução do número de multiplicações necessárias necessárias para cada sub-matriz 2x2 de 8 para 7.

Um outro algoritmo conhecido de multiplicação de matrizes é o Coppersmith-Winograd, com uma complexidade de $O(n^{2,3737})$. Contudo, a menos que as matrizes sejam enormes, esses algoritmos não resultam em reduções significativas no tempo de computação.

Assim sendo, na prática, o melhor método para acelerar a multiplicação de matrizes é o uso de algoritmos paralelos, como os que iremos apresentar a seguir. Contudo, vamos apresentar primeiramente o código sequencial “ingênuo”, qque pode ser visto no Exemplo 2.10.

```
#include <stdio.h>
#include <omp.h>
#define SIZE 5000
float a[SIZE][SIZE];
float b[SIZE][SIZE];
float c[SIZE][SIZE];
int main() {
int i,j,k;
double tIni, tFinal;
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          a[i][j] = (float)i + j;
          b[i][j] = (float)i - j;
          c[i][j] = 0.0f;
      }
    }
    tIni = omp_get_wtime();
    for (i = 0; i < SIZE; ++i) {
       for (j = 0; j < SIZE; ++j) {
          for(k=0; k < SIZE; ++k)
                c[i][j] = a[i][k] * b[k][j];
       }
    }
    tFinal = omp_get_wtime();
    printf("tempo: %3.6f\n", tFinal - tIni);
    return 0;
}
```

Exemplo 2.10: Multiplicação de matrizes sequencial

Esse código sequencial apresenta diversos problemas para uma implementação em OpenACC. Em primeiro lugar, deve ser feita a linearização das matrizes, para que o acesso aos dados no acelerador seja feito de uma forma mais otimizada, ou seja, em endereços sequenciais na memória.

Em segundo lugar, deve-se declarar uma variável temporária para armazenar o valor de **c[i][j]** que está sendo calculado no laço mais interno. Da forma que está colocada no código original, serão gerados acessos desnecessários à matriz **c**, independentemente de ela estar armazenada na memória do hospedeiro ou do acelerador. Logo, no Exemplo 2.11 chegamos a uma versão sequencial mais adequada para trabalharmos a conversão para OpenACC.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5000
int main() {
float *a = (float*) malloc (sizeof(float)*SIZE*SIZE);
float *b = (float*) malloc (sizeof(float)*SIZE*SIZE);
float *c = (float*) malloc (sizeof(float)*SIZE*SIZE);
int   i, j, k;
float temp;
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          a[i*SIZE+j] = (float)i + j;
          b[i*SIZE+j] = (float)i - j;
          c[i*SIZE+j] = 0.0f;
      }
    }
    for (i = 0; i < SIZE; ++i) {
       for (j = 0; j < SIZE; ++j) {
     temp  = 0.0;
          for (k = 0; k < SIZE; ++k) {
           temp += a[i*SIZE+k] + b[k*SIZE + j];
           }
           c[i*SIZE+j] = temp;
      }
    }
    return 0;
}
```

Exemplo 2.11: Multiplicação de matrizes otimizada

Para uma primeira versão em OpenACC utilizaremos a diretiva **kernels** e **data** e veremos como o compilador se comporta para realizar essa paralelização. Essa primeira versão é apresentada no Exemplo 2.12.

```
#pragma acc data copyin (a[0:SIZE*SIZE], b[0:SIZE*SIZE]), copy(c
    [0:SIZE*SIZE])
{
int   i, j, k;
float temp;
#pragma acc kernels loop
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          a[i*SIZE+j] = (float)i + j;
          b[i*SIZE+j] = (float)i - j;
          c[i*SIZE+j] = 0.0f;
      }
    }
#pragma acc kernels loop
    for (i = 0; i < SIZE; ++i) {
       for (j = 0; j < SIZE; ++j) {
     temp  = 0.0;
          for (k = 0; k < SIZE; ++k) {
           temp += a[i*SIZE+k] + b[k*SIZE + j];
```

```
            }
            c[i*SIZE+j] = temp;
        }
    }
}
```

Exemplo 2.12: Multiplicação de matrizes - versão inicial OpenACC

Como podemos observar, o ganho de desempenho obtido não é tão relevante para essa versão, mesmo com os cuidados para a movimentação de dados para o acelerador e otimizações realizadas (Figura 2.27).

```
$ pgcc -acc -ta=tesla -Minfo=all matmulacc.c -o matmulacc

main:
     13, Generating copyin(a[:25000000]) [if not already present]
         Generating copy(c[:25000000]) [if not already present]
         Generating copyin(b[:25000000]) [if not already present]
     16, Complex loop carried dependence of a->,b-> prevents parallelization
         Loop carried dependence of a->,b-> prevents parallelization
         Loop carried backward dependence of b->,a-> prevents vectorization
         Complex loop carried dependence of c-> prevents parallelization
         Accelerator serial kernel generated
         Generating Tesla code
         16, #pragma acc loop seq
         17, #pragma acc loop seq
     17, Complex loop carried dependence of a->,b->,c-> prevents parallelization
     24, Complex loop carried dependence of c->,b-> prevents parallelization
         Loop carried dependence of c-> prevents parallelization
         Loop carried backward dependence of c-> prevents vectorization
         Complex loop carried dependence of a-> prevents parallelization
     25, Complex loop carried dependence of c->,b->,a-> prevents parallelization
         Generating Tesla code
         24, #pragma acc loop seq
         25, #pragma acc loop seq
         27, #pragma acc loop vector(128) /* threadIdx.x */
         28, Generating implicit reduction(+:temp)
     27, Loop is parallelizable
$
$ ./matmulacc
Tempo de execução: 485.34

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/matmulacc.c
  main  NVIDIA  devicenum=0
    time(us): 33,054
    13: data region reached 2 times
        13: data copyin transfers: 18
             device time(us): total=24,876 max=1,398 min=1,337 avg=1,382
        34: data copyout transfers: 6
             device time(us): total=8,178 max=1,376 min=1,333 avg=1,363
    15: compute region reached 1 time
        16: kernel launched 1 time
            grid: [1]  block: [1]
            elapsed time(us): total=1,667,599 max=1,667,599 min=1,667,599 avg=1,667,599
    23: compute region reached 1 time
        25: kernel launched 1 time
            grid: [1]  block: [128]
            elapsed time(us): total=483,247,625 max=483,247,625 min=483,247,625 avg=483,247,625
$
```

Figura 2.27: Compilação do código usando a diretiva kernels

No Exemplo 2.13 iremos fazer uso da diretiva **parallel** e das cláusulas **reduction**, **collapse** e **tile**. A diretiva que tem maior impacto é **tile** pelo fato de permitir o acesso otimizado

a sub-blocos das matrizes. Esse acesso é realizado nos níveis mais altos da hierarquia do acelerador, a partir cache de nível 2, resultando em tempos de computação muito mais otimizados. Note que essa otimização só é eficiente porque o laço aninhado possui três níveis e matriz foi linearizada (Figura 2.28).

```
#pragma acc data copyin (a[0:SIZE*SIZE], b[0:SIZE*SIZE]), copy(c
    [0:SIZE*SIZE])
{
    #pragma acc parallel loop gang vector collapse(2)
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          a[i*SIZE+j] = (float)i + j;
          b[i*SIZE+j] = (float)i - j;
          c[i*SIZE+j] = 0.0f;
      }
    }
    #pragma acc parallel
    #pragma acc loop tile(256,256) independent
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          temp  = 0.0;
                 #pragma acc loop reduction(+:temp)
                 for (k = 0; k < SIZE; ++k) {
                     temp += a[i*SIZE+k] + b[k*SIZE + j];
                 }
                 c[i*SIZE+j] = temp;
      }
    }
}
```

Exemplo 2.13: Multiplicação de matrizes - versão final OpenACC

```
$ pgcc -acc -ta=tesla -Minfo=all matmulacc2.c -o matmulacc2

main:
     13, Generating copyin(a[:25000000]) [if not already present]
         Generating copy(c[:25000000]) [if not already present]
         Generating copyin(b[:25000000]) [if not already present]
     15, Generating Tesla code
         16, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         17,   /* blockIdx.x threadIdx.x collapsed */
     23, Generating Tesla code
         25, #pragma acc loop gang, vector tile(256,256) /* blockIdx.x threadIdx.x */
         26,   /* blockIdx.x threadIdx.x tiled */
         29, #pragma acc loop seq
     29, Loop is parallelizable
$
$ ./matmulacc2
Tempo de execução: 0.47

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/matmulacc2.c
  main  NVIDIA  devicenum=0
    time(us): 33,014
    13: data region reached 2 times
        13: data copyin transfers: 18
             device time(us): total=24,873 max=1,402 min=1,336 avg=1,381
        36: data copyout transfers: 6
             device time(us): total=8,141 max=1,365 min=1,319 avg=1,356
    15: compute region reached 1 time
        15: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=1,887 max=1,887 min=1,887 avg=1,887
    23: compute region reached 1 time
        23: kernel launched 1 time
            grid: [25600]  block: [1024]
            elapsed time(us): total=40,997 max=40,997 min=40,997 avg=40,997
$
```

Figura 2.28: Compilação do código usando a diretiva parallel

A Figura 2.29 apresenta a análise da execução do código de multiplicação de matrizes usando a diretivas **kernels**, como dito anteriormente mesmo com a movimentação de dados o ganho de desempenho obtido não é tão relevante.

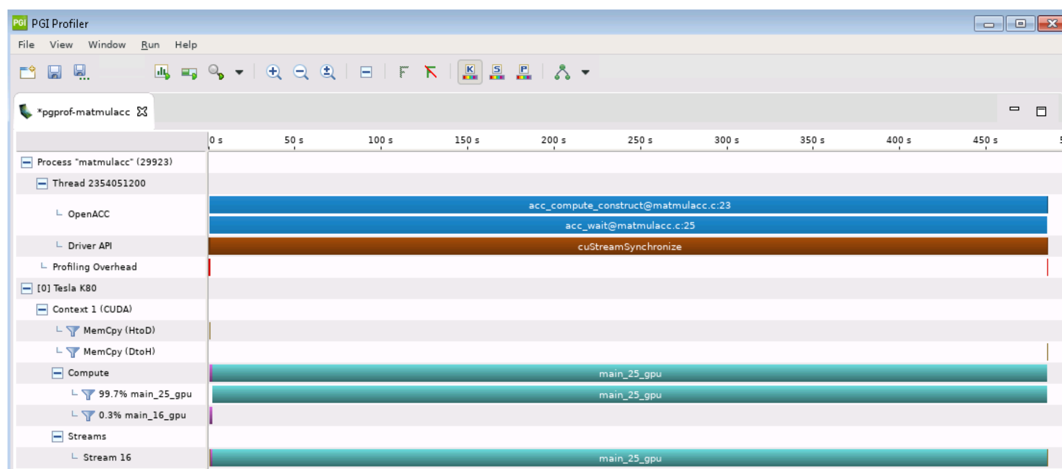


Figura 2.29: Análise de execução do código usando as diretiva data e parallel

A Figura 2.30 apresenta a análise da execução do código de multiplicação de matrizes usando a diretivas **parallel** com as cláusulas **collapse** e **tile**, observa-se que o desempenho foi muito superior a versão anterior utilizando a diretiva **kernels**.

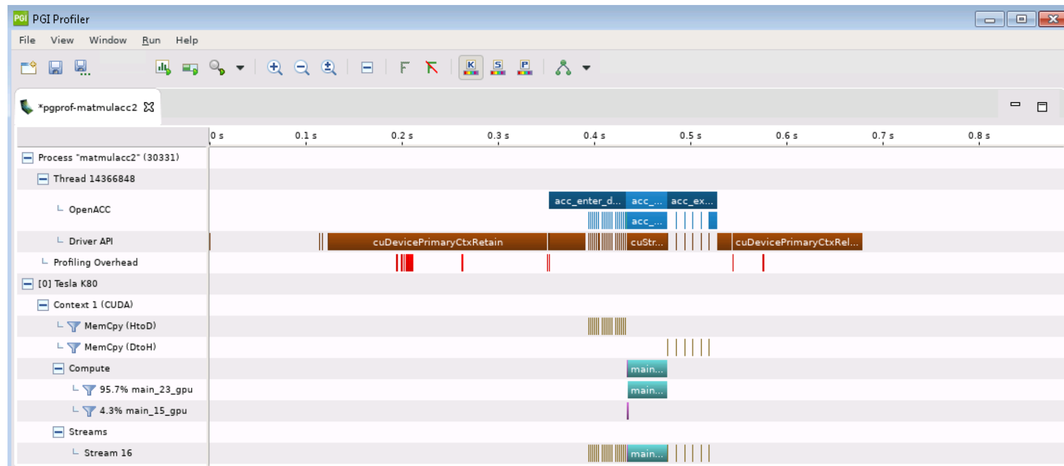


Figura 2.30: Análise de execução do código usando as diretiva data e parallel

Referências

[Chen 2017] Chen, S. (2017). *Introduction to OpenACC*. Research Computing Services Information Services and Technology Boston University.

[Harris 2017] Harris, M. (2017). *Unified Memory for CUDA Beginners*. NVIDIA Corporation.

[Murphy 2016] Murphy, J. (2016). *More Tips on OpenACC Acceleration*. Microway Corporation.

[NVIDIA 2014] NVIDIA (2014). *NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110/210*. NVIDIA Corporation.

[PGI 2019] PGI (2019). *PROFILER USER'S GUIDE*. NVIDIA Corporation.

Capítulo 3

Are you root? Experimentos Reprodutíveis em Espaço de Usuário

Jessica Imlau Dagostini, Vinicius Garcia Pinto,
Lucas Leandro Nesi, Lucas Mello Schnorr
Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Resumo

O minicurso aborda o gerenciamento de pacotes de software e a reprodutibilidade de experimentos. Gerir pacotes em ambiente de usuário pode ser desafiador, caso o mesmo não possua devidos conhecimentos do tema. Todavia, tendo tal conhecimento, é possível não só corretamente utilizar ambiente de supercomputadores como também criar e gerenciar ambientes de forma a torná-lo reprodutível. O presente minicurso tem como objetivo apresentar técnicas e comandos para criar ambientes reprodutíveis utilizando o gerenciador de pacotes Spack e criando contêineres com Docker e Singularity.

3.1. Introdução

O método científico, pilar da ciência moderna, se baseia na observação controlada de eventos de maneira que os fatos sejam verificáveis e hipóteses possam ser testadas. Esse processo culmina em um entendimento da realidade, uma teoria científica. Tal teoria, uma vez exposta, gera implicações e conclusões sobre o mundo que por sua vez leva a novos experimentos a serem observados de maneira controlada. Esse ciclo gerador de conhecimento toma por base a necessidade de que todos os experimentos sejam suficientemente reprodutíveis. De fato, qualquer teoria científica deve ser constantemente confirmada ou refutada com novas observações.

A Ciência da Computação é parte integral do mundo atual e deve seguir os preceitos do método científico no que diz respeito à investigação, à descoberta de novos algoritmos, etc. Na área de Processamento de Alto Desempenho (PAD), por exemplo, novos algoritmos e estratégias para melhorar o desempenho de aplicações paralelas e sistemas computacionais são frequentementes apresentados. É de extrema importância que os ganhos computacionais observados a partir de um algoritmo sejam verificáveis de maneira reprodutível. Neste sentido, a computação, de uma maneira geral, necessita de um

ferramental que permita a instalação de ambientes de testes controlados. Para se obter um ambiente controlado, é necessário que toda a pilha de *software* (sw) e *hardware* (hw) seja configurável. Assim, aumentam-se as chances que as observação neste ambiente sejam mais fáceis de serem reproduzidas ou repetidas por outros pesquisadores.

Ter um controle completo sobre toda a pilha de sw/hw é desafiador pois frequentemente o ambiente computacional de alto desempenho – um cluster ou supercomputador – é suficientemente específico do ponto de vista de hw e limita a capacidade de customização da pilha de sw (difícil acesso aos direitos de superusuário). Técnicas de virtualização para alto desempenho (NUSSBAUM et al., 2009) aportam um caminho possível para controlar o ambiente, embora nem sempre possíveis em plataformas de PAD. Alternativas em nível de usuário são preferíveis pois, além de permitir independência da configuração, são facilmente postas em prática sem envolver os administradores das plataformas.

Historicamente, NIX (DOLSTRA; JONGE; VISSER, 2004) foi uma das primeiras ferramentas a empregar assinaturas *hash* para ter um controle de versões de sw e suas variantes. Mais recentemente, GNU Guix (COURTÈS; WURMUS, 2015) (<https://guix.gnu.org/>) provê gerenciamento de pacotes utilizando transações, tal qual o conceito visto em banco de dados, para a construção de ambientes reprodutíveis. Tanto NIX quanto GNU Guix exigem que a ferramenta em si seja instalada com permissões de superusuário. Outras ferramentas, tais como Homebrew (HOWELL, 2017) e Spack (GAMBLIN et al., 2015) se diferenciam por ter uma instalação puramente em nível de usuário, tanto para a ferramenta em si quanto para os sw instalados. Ambientes de processamento de alto desempenho com múltiplos usuários normalmente tem demanda de sw e versões de sw diversos. Nesses ambientes, Spack se destaca por ter sido concebido especificamente às demandas de plataformas de alto desempenho.

Este minicurso apresenta a ferramenta Spack, que é um gerenciador de pacotes de sw multi-plataforma que permite compilar e instalar múltiplas versões e configurações de sw. Dentre outras soluções possíveis (discutidas na Seção 3.5), Spack tem a vantagem de oferecer uma sintaxe dita de "especificação" suficientemente simples mas capaz de capturar as diferentes possibilidades de instalação dos pacotes. Escrito em Python, Spack é extensível na medida que receitas para instalação de outras ferramentas possam ser criadas.

3.1.1. Instalação do Spack

Para instalar Spack, na linha de comando *bash*:

SH

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

3.1.2. Organização do documento

O texto deste minicurso está organizado em três partes, seguindo como base o tutorial da equipe mantenedora do Spack no SC'20 (GAMBLIN et al., 2015) e disponível em

inglês no site <https://spack-tutorial.readthedocs.io/en/latest/>. A Seção 3.2 apresenta o processo de configuração de ambientes isolados de instalação, para um uso inicial da ferramenta spack. A Seção 3.3 lista os comandos necessários para a configuração e utilização de ambientes reprodutíveis através da possibilidade de criação de containers. A Seção 3.4 detalha como se emprega a linguagem de programação Python para se definir receitas de instalação. Utilizaremos como exemplo a ferramenta `pajeng` (SCHNORR, 2021). Enfim, a Seção 3.5 discute outras ferramentas equivalentes ou semelhantes à Spack e finaliza o minicurso com a principal mensagem a ser levada como conhecimento.

3.2. Configuração de ambientes Spack isolados

Após a instalação do spack demonstrada na introdução, todos os comandos estão disponíveis pela invocação do executável `spack`. Veremos os comandos para a criação de ambientes isolados, para a instalação de pacotes, para a configuração de compiladores. Veremos no final como funcionam os arquivos de configuração e como estes podem ser compartilhados de maneira que outras pessoas possam reproduzir a pilha de software em outra máquina.

3.2.1. Criação de ambientes para isolamento da instalação de pacotes

Os softwares instalados com spack podem ser isolados em ambientes. Cada ambiente possui seus pacotes, configurações, enfim, os comandos serão executados internamente. A principal vantagem de sua utilização é a possibilidade de ter vários ambientes distintos, independentes, que podem ser compartilhados. Mesmo assim, seu uso é opcional, já que pacotes podem ser instalados no ambiente padrão de cada usuário. A diretiva `env` permite gerenciar ambientes. Por exemplo, para a criação de um novo ambiente chamado *meuambiente* o seguinte comando pode ser utilizado:

SH

```sh
spack env create meuambiente
```

Informações de ambientes spack podem ser gravadas e compartilhadas com arquivos `.lock` ou `.yaml`. Para a criação de um ambiente descrito por um arquivo no formato `yaml`, o seguinte comando pode ser utilizado:

SH

```sh
spack env create meuambiente arquivo.yaml
spack install
```

Quando for necessário trocar ou acessar um ambiente nomeado, a diretiva `env activate` é utilizada. Por exemplo, para ativar o ambiente chamado *experimentos* pode-se utilizar o seguinte comando:

SH

```sh
spack env activate experimentos
```

Para desativar e sair de um ambiente, utiliza-se a diretiva `deactivate`:

SH
```
spack env deactivate
```

3.2.2. Instalação de pacotes e suas dependências

A grande vantagem da utilização do spack é a instalação de pacotes em nível de usuário com múltiplas opções por pacote e a instalação automática de dependências. Para a instalação de um pacote com spack, utiliza-se a diretiva `install` com o nome do pacote. Por exemplo, para instalar o pacote `zlib`:

SH
```
spack install zlib
```

O pacote será baixado, configurado e compilado utilizando o compilador padrão detectado pelo spack. Caso o sistema tenha múltiplos compiladores pode-se adicionar `%compilador` após o nome do pacote para escolher o compilador a ser utilizado. Por exemplo, caso seja desejado utilizar o compilador `clang`:

SH
```
spack install zlib %clang
```

Os pacotes ainda podem possuir várias opções de compilação e instalação. Cada possibilidade de instalação é chamada de variante. Cada variante diferente possui um *hash* diferente. As variantes para cada pacote podem ser listadas com `info`. O seguinte comando pode ser utilizado para verificar as variantes do `zlib`:

SH
```
spack info zlib
```

Por exemplo, o pacote `zlib` pode ser compilado como uma biblioteca compartilhada ou estática. O padrão é compartilhada. Para ativar uma variante utiliza-se + (o símbolo mais) e o nome da variante. Para negar utiliza-se ~ (o símbolo til). Caso deseja-se que o pacote `zlib` seja compilado como uma biblioteca estática podemos negar a opção `shared` utilizando portanto o símbolo ~:

SH
```
spack install zlib~shared
```

Ainda é possível alterar variáveis de ambiente para a instalação dos pacotes. Para tal, deve-se passar o nome da variável após o pacote com a declaração desejada. Por

exemplo, no comando a seguir, pode-se adicionar o parâmetro `-fPIC` na variável `CFLAG` fazendo que o pacote e toda sua pilha de dependências seja compilada com essa variável de ambiente.

SH

```
spack install zlib cflags="-fPIC"
```

As versões dos pacotes disponíveis nos repositórios base do spack podem ser visualizadas utilizando a diretiva `versions` seguido do nome do pacote. Para instalar uma versão específica utilizamos `@` com a versão. Assim, para mostrar as versões do pacote `zlib` e para instalar a versão `1.2.10`, utiliza-se os seguintes comandos:

SH

```
spack versions zlib
spack install zlib@1.2.10
```

Os pacotes podem ter várias dependências. O comando `spec` é utilizado para mostrar todas as dependências de um pacote com as opções a serem concretizadas. Durante a instalação de um pacote, pode-se realizar as mesmas customizações de versões e variantes na instalação de dependências. Para isto, basta especificar a dependência com ^ e as opções desejadas. Os comandos abaixo demonstram a verificação das dependências do pacote `tcl` e, em seguida, a instalação utilizando a versão `1.2.8` de sua dependência `zlib`. Ambos os pacotes serão compilados utilizando o compilador `clang`:

SH

```
spack spec tcl
spack install tcl ^zlib @1.2.8 %clang
```

Para mostrar todos os pacotes instalados pode-se utilizar a diretiva `find`. Esta ainda tem as opções: `-d`, para mostrar as dependências; `-v`, para mostrar as opções de variantes; e `-l`, para mostrar a identificação única (*hash*) de cada instalação. Para ver todas as instalações do `zlib` é utilizado, por exemplo, o seguinte comando:

SH

```
spack find -d -v -l zlib
```

Existem várias formas para utilizar um pacote instalado. Em uma primeira possibilidade, pode-se utilizar a opção `load` para carregar e atualizar as variáveis de ambiente (`PATH`, `LD_LIBRARY_PATH` e outras que possam ser considerada necessárias) com os locais de instalação do pacote e suas dependências. Outra possibilidade é criar um diretório com a estrutura tradicional de instalação de pacotes e as dependências necessárias (`bin/`, `lib/`, `include/`) utilizando a diretiva `view` e a opção `soft`. Os comandos a seguir mostram o emprego destas duas maneiras com o pacote `zlib`, sendo que o segundo o faz no diretório *pasta*.

SH

```
spack load zlib@1.2.8
spack view soft pasta zlib@1.2.8
```

Para remover um pacote utiliza-se a diretiva `uninstall`. Por exemplo, para remover o pacote `tcl`:

SH

```
spack uninstall tcl
```

Caso o pacote seja uma dependência de outros pacotes, deve-se desinstalá-lo utilizando a opção `--dependents`. Assim, todos as dependências também serão desinstaladas em cascata. Caso múltiplas variantes estejam instaladas, pode-se remover todos as instalações de um pacote com a opção `--all`. No caso para remover todas as variantes do `zlib` e todos os pacotes que são dependentes dele pode-se utilizar o seguinte comando:

SH

```
spack uninstall --all --dependents zlib
```

3.2.3. Controlando a coexistência de diversos compiladores

Uma das possibilidades do spack é o emprego de diversos compiladores. Todos os comandos de instalação permanecem os mesmos, entretanto, diferentes compiladores e versões podem ser utilizados para a instalação de toda a pilha de *software*. Para verificar os compiladores disponíveis no spack, pode-se utilizar a diretiva `compilers`:

SH

```
spack compilers
```

Caso seu sistema possua compiladores que ainda não foram encontrados, pode-se utilizar a diretiva `compiler find` para o spack tentar localizá-los automaticamente.

SH

```
spack compiler find
```

Quando a localização automática falhar, provavelmente porque os compiladores não estão presentes no `PATH`, pode-se adicioná-los manualmente com a diretiva `compiler add` e o caminho absoluto para o compilador:

SH

```
spack compiler add /local/compilador
```

Após os compiladores serem adicionados, pode-se fazer uso na instalação dos pacotes com a opção `%` e a especificação do compilador. Esta especificação pode conter a versão do compilador caso mais de uma versão esteja disponível.

3.2.4. Arquivos de configuração e opções

O spack possui diversos arquivos de configurações. Estes arquivos podem estar localizados em diretórios diferentes para escopos de trabalho distintos. Quando diferentes configurações são encontradas, a ordem de precedência é a listagem da seguinte tabela:

Escopo de Trabalho	Local ou Diretório
Linha de Comando	Informado diretamente pelo usuário
Diretório	Especificado com `--config-scope`
Usuário	`~/.spack/`
Site	`$SPACK_ROOT/etc/spack/`
Sistema	`/etc/spack/`
Padrão	`$SPACK_ROOT/etc/spack/defaults/`

Os vários arquivos de configuração do spack estão todos no formato `.yaml` e são brevemente apresentados na tabela a seguir.

Arquivo	Objetivo e opções de configuração
compilers.yaml	Compiladores, locais, flags padrões de compilação
packages.yaml	Pacotes e variantes, instalações locais do sistema
config.yaml	Funcionamento do spack
mirrors.yaml	Espelhos de onde se baixam os fontes
modules.yaml	Módulos do spack
repos.yaml	Listagem de repositórios de softwares alternativos ao oficial

3.2.5. Compartilhando configurações

Um dos recursos disponíveis no spack é o compartilhamento de configurações ou ambientes com todas as informações sobre os pacotes instalados. Para conseguir o arquivo que contenha toda estas descrições, pode-se utilizar a diretiva `cd -e meuambiente` para se deslocar ao diretório do ambiente. Neste diretório encontram-se os arquivos `spack.yaml` e `spack.lock`. O arquivo `spack.yaml` registra os pacotes a serem instalados e suas opções, entretando tal listagem não encontra-se finalmente concretizadas (opções como versão podem mudar em plataformas diferentes ou versões do spack diferentes). O arquivo `spack.lock` descreve toda a pilha de *software* efetivamente concretizada, versões utilizadas, configurações e inclui o ambiente utilizado. Desta forma, o arquivo `spack.lock` só poderá ser utilizado em ambientes que possuem as mesmas configurações (CPU, compiladores, etc). Estes arquivos podem ser compartilhados para outros usuários e sistemas que terão uma réplica da pilha de *software*. Esta opção auxilia na reprodutibilidade dos experimentos. O comando a seguir ilustra como encontrar estes arquivos para o ambiente `meuambiente`:

SH

```
spack cd -e meuambiente
ls
```

A leitura destes arquivos ocorre com o comando `env create` como discutindo anteriormente.

3.3. Utilização de ambientes reprodutíveis

Além da possibiliadde de criar ambientes com as configurações específicas de pacotes necessários para a execução de alguma aplicação – facilitando assim a reprodutibilidade de experimentos – também é possível criar containers a partir do Spack. Containers vêm ganhando espaço com a comunidade científica em HPC, uma vez que eles provêm ambientes portáteis para a reprodução mais fiel de experimentos, e podem ser executados em qualquer recurso computacional a partir de um único arquivo de imagem (KURTZER; SOCHAT; BAUER, 2017). Alguns estudos (TORREZ; RANDLES; PRIEDHORSKY, 2019) (ALLES; CARISSIMI; SCHNORR, 2018) demonstram que containers tem uma interferência muito pequena (e por vezes nula) no tempo de execução de aplicações paralelas que são computacionalmente intensivas. Apresentamos a seguir a criação, a partir de ambientes do spack, de containers Docker e Singularity, seguido de um detalhamento de configurações adicionais que influenciam nestes processos.

3.3.1. Criando containers Docker

Uma vez criado um ambiente, pode-se empregar a diretiva `containerize` para transformá-lo em um container. Com esta diretiva, o Spack cria o arquivo de configuração inicial para a construção da imagem base do container. O Spack suporta a criação de arquivos de configuração para containers Docker (MERKEL, 2014) e Singularity (KURTZER; SOCHAT; BAUER, 2017), sendo este com maior foco para uso em processamento de alto desempenho. Mesmo assim, como podemos criar um container Singularity a partir de uma imagem Docker, este material se inicia com a geração de um Dockerfile.

Para criar o Dockerfile, devemos acessar o diretório do ambiente que pretendemos transformar em um container. Os ambientes spack ficam normalmente no diretório `./var/spack/environments/` a partir de onde o spack foi instalado. Assumindo que temos o ambiente nomeado `erad` e queremos transformá-lo em um container, basta nos deslocarmos ao diretório do ambiente e lançar a diretiva `containerize`, assim:

SH

```
cd ~/spack/var/spack/environments/erad
spack containerize > Dockerfile
```

A partir do arquivo `Dockerfile` gerado, é possível criar uma imagem Docker seguindo os comando padrões da plataforma:

SH

```
docker build -t erad:1.0 .
docker image list
```

Feito isso, a imagem deve estar criada e disponível para utilização.

3.3.2. Criando containers Singularity

Containers Docker acabam não sendo os mais recomendados para uso em processamento de alto desempenho (PAD). Outros orquestradores de container, como Singularity, são mais indicados para executar aplicações em supercomputação por se integrarem mais facilmente aos gerenciadores de trabalhos (GAMBLIN et al., 2015).

Podemos criar containers Singularity (representados normalmente por arquivos com a extensão `.sif`) tanto a partir de uma imagem Docker pré-existente, quanto diretamente de um ambiente existente no spack. Para ambas as opções, devemos nos certificar que o Singularity está um uma versão igual ou maior a `3.7.7`. Para converter uma imagem Docker em Singularity, podemos executar o comando:

SH

```
sudo singularity build erad.sif docker-daemon://erad:1.0
```

Já para criar uma imagem Singularity diretamente do Spack, precisamos alterar o arquivo `spack.yaml` que contém as configurações do ambiente a ser transformado em container. No marcador `format`, devemos especificar Singularity, assim:

YAML

```
spack:
  specs: [zlib]

  container:
    format: singularity
```

Em seguida, gerar o arquivo de configurações `.def` do singularity e enfim gerar a imagem `.sif` do container com os comandos:

SH

```
spack containerize > erad.def
sudo singularity build erad.sif erad.def
```

3.3.3. Configurações adicionais para containers

É possível também adicionar configurações extras ao arquivo `spack.yaml` do ambiente a ser transformado em container. Por exemplo, é possível especificar a imagem base a ser utilizada, rótulos e outras informações. No exemplo de código abaixo, a subárvore `container:` traz as especificações a serem consideradas na criação do container.

YAML

```
spack:
  specs: [zlib]

  container:
    format: docker

    images:
      os: "ubuntu:18.04"
      spack: develop

    strip: true

    os_packages:
      final:
      - gcc
      - libgomp

    labels:
      mpi: "openmpi"
```

No exemplo do código acima, determinamos com o marcador `format` se o container será do tipo Docker ou Singularity. Já no marcador `image` podemos definir qual versão de sistema operacional desejamos usar como base, e qual versão do repositório spack também desejamos usar. Neste exemplo, definimos a versão 18.04 do Ubuntu, com o spack em sua versão de desenvolvimento. Também é possível usar imagens personalizadas como base, como veremos no exemplo a seguir. Por fim, no marcador `os_packages` podemos definir pacotes de sistema que desejamos que sejam instalados no sistema operacional da imagem para serem utilizados na execução do container. Neste exemplo, estamos adicionando a instalação da biblioteca `libgomp` (para execução de aplicações OpenMP). O marcador `labels` permite definir rótulos para a imagem final diretamente pela criação do Dockerfile. Com esse novo arquivo no formato `yaml`, basta reexecutar os comando de criação do Dockerfile e da imagem, como previamente demonstrado.

Para criar uma imagem a partir de uma imagem base externa, deve-se usar a tag `images:build` e `images:final`, com o identificador da imagem a ser utilizada, como no exemplo abaixo:

YAML

```
spack:
  specs:
  - gromacs@2019.4+cuda build_type=Release
  - mpich
  - fftw precision=float
  packages:
    cuda:
      buildable: False
      externals:
      - spec: cuda%gcc
        prefix: /usr/local/cuda

  # Criando o container com imagens base externas.
  container:
    images:
      build: custom/cuda-10.1-ubuntu18.04:latest
      final: nvidia/cuda:10.1-base-ubuntu18.04
```

3.4. Criação de pacotes

Os pacotes Spack são escritos em Python. A criação de um novo pacote resume-se a criação de um arquivo chamado `package.py` contendo as informações básicas do pacote tais como descrição, versões, dependências e sistema de construção como Autotools ou CMake, por exemplo.

3.4.1. Passos iniciais

O passo inicial para criação de um novo pacote deve especificar a descrição, URL para *download* e dependências obrigatórias. A diretiva `create` através do comando `spack create` facilita a criação de um novo pacote preparando um diretório para o mesmo e um esqueleto do arquivo `package.py` a ser preenchido pelo usuário. Aqui usaremos como exemplo a criação de um pacote Spack para a ferramenta `pajeng`:

SH

```
spack create -n pajeng -t cmake https://github.com/schnorr/
   pajeng/archive/1.3.6.tar.gz
```

Em seguida, podemos preencher o `package.py` adicionando as informações básicas para que o `pajeng` possa ser construído e instalado. A opção `-t cmake` utilizada no comando acima permite criar um esqueleto preparado para *softwares* construídos com o CMake, como é o caso do `pajeng`.

As linhas 4 a 8 do código abaixo descrevem informações básicas como descrição e página do `pajeng`, tais informações serão exibidas através da diretiva `info` ao executar o comando `spack info pajeng`. A linha 10 aponta os mantenedores do pacote enquanto a linha 12 define uma versão que instalará o `pajeng` no *release* 1.3.6. Nesta

mesma linha, utilizamos o parâmetro `preferred` para designá-la como versão preferencial. Por fim, as linhas 18 a 20 declaram as dependências requeridas para compilação do `pajeng`. Estas dependências serão automaticamente instaladas pelo spack e seus respectivos locais de instalação serão fornecidos de maneira transparente ao CMake do `pajeng`. Também é possível instalar um pacote a partir do código fonte disponibilizado em um repositório git, svn ou hg conforme ilustrado na linha 15. Parâmetros adicionais como `branch`, `commit` e `tag` permitem especificar versões diferentes do código fonte no escopo do repositório informado.

PYTHON

```python
from spack import *

class Pajeng(CMakePackage):
  """PajeNG is a re-implementation of the well-known Paje
   visualization tool for the analysis of execution traces."""

  homepage = "https://github.com/schnorr/pajeng"
  git = "https://github.com/schnorr/pajeng.git"
  url = "https://github.com/schnorr/pajeng/archive/1.3.6.tar.gz"

  maintainers = ['viniciusvgp', 'schnorr']

  version('1.3.6',
          sha256 = '1a2722bfaeb0c6437fb9e8efc2592edbf14ba01172f9
    7e01c7839ffea8b9d0b3',
          preferred = True)
  version('develop',
          git = 'https://github.com/schnorr/pajeng.git')

  depends_on('boost')
  depends_on('flex')
  depends_on('bison')
```

O pacote recém criado pode ser instalado com:

SH

```sh
spack install pajeng
```

3.4.2. Alterações em pacotes existentes

Alterações em pacotes criados previamente podem ser feitas com a diretiva `edit` como no comando `spack edit pajeng`. A título de exemplo, adicionaremos as demais versões do `pajeng`. Uma maneira simples de se obter todas as versões disponíveis para um dado pacote é executar o comando `spack checksum` e adicionar sua saída ao conteúdo do `package.py`. Por meio da `url` base informada com `spack create` quando da criação do pacote, será feita uma busca pelos números de versões lançadas e das respectivas somas de verificação SHA-256.

SH

```
spack checksum -b pajeng
```

Como resultado do comando acima, obtemos as seguintes linhas a serem adicionadas ao arquivo do pacote:

PYTHON

```
version('1.3.6', sha256 = '1a2722bfaeb0c6437fb9e8efc2592edbf14ba
    01172f97e01c7839ffea8b9d0b3')
version('1.3.5', sha256 = 'ea8ca02484de4091dcf57289724876ec17dd9
    8e3a032dc609b7ea020ca2629eb')
version('1.3.4', sha256 = '284e9a590a2861251e808542663bf1b77bc2c
    99650a1fbf945cd5bab65402f9e')
version('1.3.3', sha256 = '42cf44003d238fd5c4ab512bdeb445fc12f7e
    3bd3f0526b389f080c84b83b19f')
version('1.3.2', sha256 = '97154415a22f9b7f83516e988ea664b399037
    7d69fca859275ca48d7bfad0932')
version('1.3.1', sha256 = '4bc3764aaa7e79da9a81f40c0593b646007b6
    89e4ac20886d06f271ce0fa0a60')
version('1.3',   sha256 = '781b8be935e10b65470207f4f179bb1196aa6
    740547f9f1af0cb1c0193f11c6f')
version('1.1',   sha256 = '986d03e6deed20a3b9d0e076b1be9053c1bc8
    6c8b41ca36cce3ba3b22dc6abca')
version('1.0',   sha256 = '4d98d1a78669290d0a2e6bfe07a1eb4ab96bd
    05e5ef78da96d2c3cf03b023aa0')
```

3.4.3. Dependências e versões

Na seção 3.4.1, demostramos como adicionar as dependências mínimas para que se possa construir o `pajeng`. Entretanto, diferentes versões de um dado pacote podem ter dependências distintas e bastante específicas. Como exemplos, usaremos tanto versões mais antigas do `pajeng` que dependem das bibliotecas `qt` na versão 4.x e `glut` quanto a versão em desenvolvimento que depende da biblioteca `fmt`. Para tratar o primeiro caso, podemos especificar novas dependências que serão aplicadas apenas a versões do `pajeng` iguais ou anteriores a 1.3.2. Note que é possível selecionar versões específicas de uma dependência. Neste exemplo, é requerida uma versão do `qt` inferior a 4.999 e com a variante `opengl` habilitada. No segundo caso, adicionamos a dependência `fmt` que será aplicada apenas quando solicitada a versão `develop`. As linhas a serem adicionadas são:

PYTHON

```
depends_on('qt@:4.999+opengl', when='@:1.3.2')
depends_on('freeglut', when='@:1.3.2')
depends_on('fmt', when='@develop')
```

3.4.4. Variantes

Pacotes spack podem ter variantes que permitem que o usuário possa habilitar ou desabilitar funcionalidades. Para ilustrar este recurso, vejamos um exemplo de variantes para

habilitar a ligação estática e a documentação e para desabilitar a construção da biblioteca `libpaje` e das ferramentas auxiliares:

PYTHON

```
variant('static',
        default = False,
        description = "Build as static library")
variant('doc',
        default = False,
        description = "The Paje Trace File documentation")
variant('lib',
        default = True,
        description = "Build libpaje")
variant('tools',
        default = True,
        description = "Build auxiliary tools")

def cmake_args(self):
  args = [
    self.define_from_variant('STATIC_LINKING', 'static'),
    self.define_from_variant('PAJE_DOC', 'doc'),
    self.define_from_variant('PAJE_LIBRARY', 'lib'),
    self.define_from_variant('PAJE_TOOLS', 'tools')
  ]
  return args
```

Note que as variantes podem ser marcadas como habilitadas ou desabilitadas por padrão através do parâmetro `default`. Como as novas variantes implicam alterar a configuração padão de construção do `pajeng`, também se faz necessário sobreescrever o método `cmake_args` para levar em conta a ativação ou desativação das variáveis relacionadas às funcionalidades ativadas ou não pelo usuário no momento da instalação.

3.4.5. Conflitos

Em alguns casos pode ser interessante declarar conflitos entre as diversas opções a serem aplicadas quando da construção de um pacote. Estes conflitos podem refletir incompatibilidades com dependências ou compiladores, *bugs*, ou simplesmente configurações contraditórias do próprio pacote. No caso do `pajeng`, definimos anteriormente duas variantes `lib` e `tools`. Enquanto requisitar a instalação `pajeng+lib~tools` é perfeitamente válido, fazer o contrário, isto é `pajeng~lib+tools`, é inconsitente visto que as ferramentas auxiliares (`tools`) dependem da biblioteca (`lib`). Para lidar com estes casos, podemos definir conflitos de forma a impedir tal tentativa de instalação:

PYTHON

```
conflicts('+tools',
          when = '~lib',
          msg = "Enable libpaje to compile tools.")
```

Ao solicitar qualquer fórmula de instalação que case, ainda que implicitamente, com a regra especificada como conflito o usuário receberá a mensagem de erro descrita no parâmetro `msg`.

3.4.6. Publicação de pacotes

Existem duas maneiras de tornar público um novo pacote Spack. A primeira, e mais abrangente, é submetê-lo ao repositório oficial. Submissões de novos pacotes podem ser feitas por meio de *pull-requests*. Uma vez que o pacote passe nos testes automatizados, ele estará disponível para instalação por qualquer usuário. Uma versão completa do pacote `pajeng` foi submetida ao repositório oficial do Spack e pode ser encontrada em <https://github.com/spack/spack/blob/develop/var/spack/repos/builtin/packages/pajeng/package.py>.

A segunda maneira de publicar um novo pacote é por meio da criação de repositórios adicionais quem podem ser públicos ou privados. Repositórios adicionais podem ser criados com `spack repo create` e posteriormente compartilhados com demais usuários que poderão adicioná-los em suas instâncias locais executando `spack repo add`. Repositórios adicionais são úteis para *softwares* cujo acesso é restrito ou para pacotes Spack em desenvolvimento que ainda não estão suficientemente maduros para serem submetidos ao repositório oficial. Algumas instituições procuram manter repositórios Spack externos e públicos para divulgar e facilitar a instalação dos *softwares* por elas desenvolvido. Outro caso de uso para repositórios adicionais é quando, por algum motivo, é necessário sobrepor um pacote do repositório oficial.

3.5. Conclusão e Discussão

Este minicurso abordou os conceitos básicos fundamentais para se gerenciar pacotes de *software* em nível de usuário. Facilitando a reprodutibilidade dos experimentos, apresentamos a ferramenta Spack que tem sido bastante utilizando em parques computacionais de alto desempenho. Para saber mais sobre o Spack, referenciamos ao tutorial Spack (GAMBLIN et al., 2015), que porta uma enorme quantidade de diretivas auxiliares para tratar casos mais específicos não abordados neste minicurso.

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) com a bolsa 141971/2020-7 para o terceiro autor, 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

ALLES, G. R.; CARISSIMI, A.; SCHNORR, L. M. Assessing the computation and communication overhead of linux containers for hpc applications. In: IEEE. *2018 Symposium on High Performance Computing Systems (WSCAD)*. [S.l.], 2018. p. 116–123. páginas 8

COURTÈS, L.; WURMUS, R. Reproducible and user-controlled software environments in hpc with guix. In: HUNOLD, S. et al. (Ed.). *Euro-Par 2015: Parallel Processing Workshops*. Cham: Springer International Publishing, 2015. p. 579–591. ISBN 978-3-319-27308-2. páginas 2

DOLSTRA, E.; JONGE, M. de; VISSER, E. Nix: A safe and policy-free system for software deployment. In: *Proceedings of the 18th USENIX Conference on System Administration*. USA: USENIX Association, 2004. (LISA '04), p. 79–92. páginas 2

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 2, 9, 15

HOWELL, M. *Homebrew, the missing package manager for OS X*. 2017. Disponível em: <http://brew.sh>. páginas 2

KURTZER, G. M.; SOCHAT, V.; BAUER, M. W. Singularity: Scientific containers for mobility of compute. *PloS one*, Public Library of Science San Francisco, CA USA, v. 12, n. 5, p. e0177459, 2017. páginas 8

MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, Belltown Media, Houston, TX, v. 2014, n. 239, mar. 2014. ISSN 1075-3583. Disponível em: <http://dl.acm.org/citation.cfm?id=2600239.2600241>. páginas 8

NUSSBAUM, L. et al. Linux-based virtualization for hpc clusters. In: *Montreal Linux Symposium*. [S.l.: s.n.], 2009. páginas 2

SCHNORR, L. M. *PajeNG – Paje Next Generation*. 2021. Disponível em: <https://github.com/schnorr/pajeng>. páginas 3

TORREZ, A.; RANDLES, T.; PRIEDHORSKY, R. Hpc container runtimes have minimal or no performance impact. In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. [S.l.: s.n.], 2019. p. 37–42. páginas 8

Capítulo 4

Além de Simplesmente: #pragma omp parallel for

João Vicente Ferreira Lima - jvlima@inf.ufsm.br[1]
Claudio Schepke - claudioschepke@unipampa.edu.br[2]
Natiele Lucca - natielelucca@gmail.com[3]

Resumo

OpenMP tem sido o padrão de fato para a programação em memória compartilhada. No entanto, a maioria dos programadores explora apenas o paralelismo de laços, deixando de usar novos e outros recursos disponíveis nas versões mais recentes da especificação de OpenMP (3, 4 e 5). Com isso, outras abordagens paralelas não tem sido tão difundidas. Além disso, disparar tarefas em CPU e GPU usando uma única interface de programação é um grande atrativo para a paralelização de aplicações. Neste contexto, este capítulo tem como objetivo aprofundar a programação paralela em aplicações, com recursos considerados avançados de OpenMP, geralmente não adotados ou vistos nas disciplinas introdutórias de programação paralela. Para tanto, são apresentadas técnicas de exploração do paralelismo disponibilizado pelas diretivas de execução concorrente de OpenMP em diferentes trechos de código de duas aplicações científicas.

[1]João Lima possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2006), mestrado em Computação pela Universidade Federal do Rio Grande do Sul (2009) e doutorado em Computação em co-tutela entre a Université de Grenoble e Universidade Federal do Rio Grande do Sul (2014). Atualmente é Professor Adjunto do Departamento de Linguagens e Sistemas de Computação da Universidade Federal de Santa Maria. Tem experiência na área de Ciência da Computação, com ênfase em Processamento Paralelo de Alto Desempenho, atuando principalmente nos seguintes temas: programação paralela e linguagens de programação.

[2]Claudio Schepke possui graduação em Ciência da Computação pela Universidade Federal de Santa Maria (2005) e mestrado (2007) e doutorado (2012) em Computação pela Universidade Federal do Rio Grande do Sul, sendo este feito na modalidade sanduíche na Technische Universität Berlin, Alemanha (2010-2011). É professor adjunto da Universidade Federal do Pampa (UNIPAMPA), campus Alegrete/RS desde 2012. Tem experiência na área de Ciência da Computação, com ênfase em Processamento Paralelo e Distribuído, atuando principalmente nos seguintes temas: processamento de alto desempenho, programação paralela, aplicações científicas e computação em nuvem.

[3]Natiele Lucca é graduada em Ciência da Computação pela Universidade Federal do Pampa (UNIPAMPA) (2020). Atualmente é mestranda do Programa de Pós-Graduação em Engenharia de Software (PPGES) da UNIPAMPA/Alegrete. Tem experiência em programação paralela e algoritmos bio-inspirados.

4.1. Introdução

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 de código, 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 de maneira simples e eficiente a partir de um código-fonte é inserir diretivas (pragmas). Diretivas de pré-compilação possibilitam a geração de código específico e automatizado pela conversão das instruções. Desta forma, as instruções paralelas podem ser incluídas no código antes da compilação de fato do mesmo.

Diretivas paralelas geram fluxos concorrentes de código, que podem ser executados tanto em arquiteturas multi-core como many-core. Este capítulo aborda a prática de programação com diretivas paralelas e tem como objetivo apresentar técnicas de exploração de paralelismo em diferentes trechos de código para um conjunto de aplicações científicas usando a interface de programação OpenMP. Neste sentido, são 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.

4.2. Arquiteturas Paralelas

A execução de aplicações pode ser feita em diferentes tipos e níveis de paralelismo. Multi-core com unidades vetoriais expressivas, processadores vetoriais, coprocessadores e GPUs tem-se destacado na composição de computadores e combinados na formação de *clusters* de alta performance, conforme ilustrado na Figura 4.1. Nesta seção são discutidas as diferentes formas de paralelismo oferecidas pelas arquiteturas disponíveis atualmente.

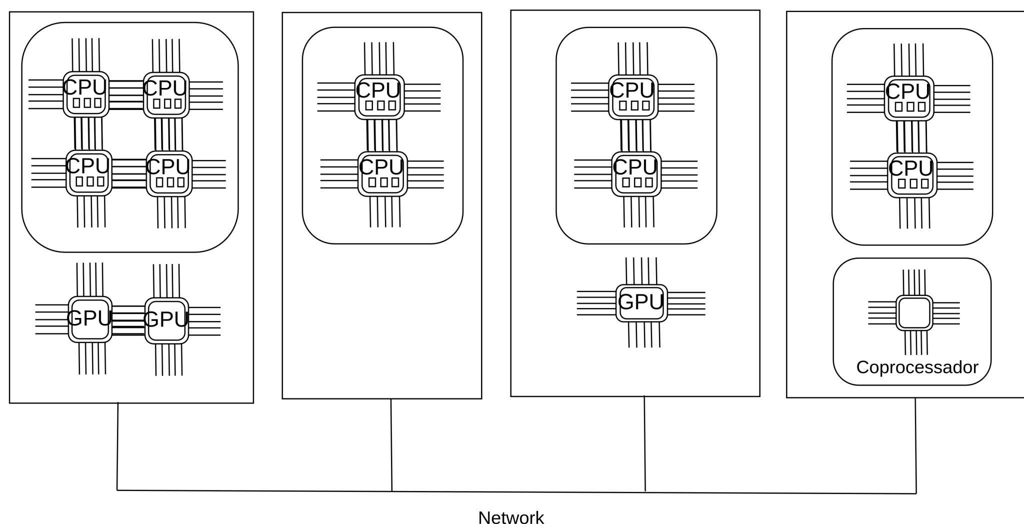


Figura 4.1. Exemplo de arquitetura de *cluster* com composição heterogênea de nós.

A concorrência das instruções, além da concepção clássica de processadores (pipeline e superescalar), visto tradicionalmente em disciplinas de arquitetura e organização

de computadores, pode ainda ser feito através de instruções vetoriais. Embora existam arquiteturas com processadores especificamente vetoriais (como é o caso dos processadores NEC SX-Aurora TSUBASA - Vector Engine (NEC Corporation, 2021)), qualquer core de um processador de propósito geral também possui uma unidade para a execução de instruções vetoriais. Tem-se visto inclusive, com o passar dos anos, o aumento do número de operações que podem ser realizadas simultaneamente. Por exemplo, em `AVX512` pode-se executar 32 operações de ponto flutuante de precisão dupla ou 64 operações de ponto flutuante de precisão simples por ciclo de clock ou oito inteiros de 64 bits ou dezesseis inteiros de 32 bits com até duas unidades de operação fundida multiplicação-adição - FMA (CORPORATION, 2021a). Neste sentido, basta ativar, no momento da compilação do código-fonte, *flags* que orientam o compilador a gerar instruções para essas unidades vetoriais (`AVX512, AVX2, AVX, SSE4_1, SSE4_2, SSE, MMX`). Uma dica de GCC é a opção `-O3` ou `-ftree-vectorize` mais `mavx`.

Especificamente em relação à arquitetura multicore, uma grande variedade de modelos e quantidades de core estão à disposição para a implementação da concorrência em nível de processo ou *thread*. Estes processadores possuem também diferentes quantidades de memória *cache* e frequência de *clock* e de acesso ao barramento. Não é tão comum em computadores pessoais, mas um computador pode também ser composto por mais de um processador multicore, aumentando ainda mais o número de unidades de computação presentes em uma única máquina (placa-mãe). Atualmente, como exemplos, há processadores com até 28 cores / 56 *threads* para processadores Intel (Xeon Platinum 8380H, 2.90 GHz / 4.30 GHz, 38.5 MB de Cache, TDP 250 W e 14 nm de litografia (CORPORATION, 2021b)) e 64 cores / 128 *threads* para processadores AMD (EPYC Embedded 7H12, 2.6GHz / 3.3GHz, 256 MB de Cache, TDP 280 W e 7/14 nm de litografia (Advanced Micro Devices, Inc, 2021)). Além dos processadores multi-core tradicionais, existem os conhecidos aceleradores de hardware, como cooprecessadores e GPUs.

A ideia do uso de coprocessadores ressurgiu nos meados de 2012 e aparentemente já foi descartada, embora a arquitetura ainda encontra-se presente em diversas máquinas em produção. Os coprocessadores Intel Xeon Phi foram criados para otimizar a relação performance por watt para aplicações com cargas de trabalho altamente paralelizáveis (Gonçalves; Girardi; Schepke, 2018). Foram também disponibilizados um conjunto de ferramentas de software para utilizar esta arquitetura *manycore* com suporte a instruções vetoriais (SIMD). Nos modelos *Knights Corner* há versões com até 61 cores físicos de 1.2GHz e com 4 *threads* de hardware por core, o que possibilita a execução simultânea em até 244 *threads* físicas. Os coprocessadores disponibilizam instruções SIMD com tamanho de 512 *bits*, com 32 registradores nativos, que suportam um desempenho de pico com precisão dupla de até 1 teraFLOPS/s Nos modelos *Knights Landing* há versões com até 72 cores, também com 4 *threads* de hardware por core.

Já a computação de propósito geral em Unidades de Processamento Gráfico (GPU), abordagem conhecida como GPGPU, superou as expectativas iniciais, e mantém-se como uma alternativa aos processadores convencionais, em parte devido à facilidade com que esses processadores podem explorar o paralelismo em grande escala. O desempenho de GPUs contemporâneas cresceu mais rapidamente do que o provido pelas arquiteturas multicores. Isso beneficiou aplicações bem estabelecidas ou clássicas da computação científica, como por exemplo as operações que envolvem álgebra linear, que demandam

historicamente de muito tempo de processamento para as simulações computacionais. Mais recentemente, com o ressurgimento da área de Inteligência Artificial com a noção de *Deep Learning*, GPUs mostraram-se uma ótima alternativa para acelerar o treinamento de algoritmos desenvolvidos nesta área, o que inclusive modificou os tipos e tamanho das operações suportadas por novas versões de GPUs. Um modelo de GPU como o A100 da NVIDIA, por exemplo, tem condições de processar até 9,7 TFlops de ponto flutuante de precisão dupla ou 9,5 TFlops usando *Tensor Cores*. Para este modelo há um total de 6.912 núcleos CUDA, 40 GB de memória e 1.6 TB de largura de banda (NVIDIA, 2021).

Diante de tudo isso, tem-se atualmente uma composição bastante heterogênea em termos de arquitetura de computadores disponíveis. Por outro lado, não há uma interface de padrão estabelecida que possibilite programar todos os modelos de arquitetura existentes. Excluindo-se a computação inter-computadores (*clusters*), que necessitam de uma biblioteca de troca de mensagens como *Message-Passing Interface* - MPI, OpenMP é uma das alternativas para a geração de código do tipo SIMD (instruções vetoriais), multicore e aceleradores de hardware.

4.3. A Interface de Programação OpenMP

OpenMP é uma API para programação paralela de memória compartilhada e multiplataforma disponível em C/C++ e Fortran (OPENMP, 2021). 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). OpenMP aplica o modelo em segmentos do código que são informados pelo programador. Dessa forma, um código sequencial é executado pela *thread* mestre até um bloco ou área de execução paralela, conforme apresentado na Figura 4.2.

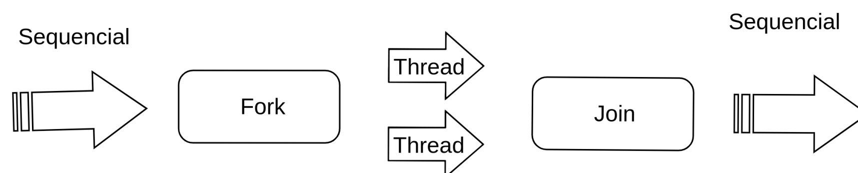


Figura 4.2. Instanciação de novas *threads* (*fork*) e término da execução (*join*).

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 `!%OMP END` 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, uma biblioteca de rotinas de tempo de execução e variáveis de ambiente para a programação paralela (TORELLI; BRUNO, 2004). Uma diretiva é precedida obrigatoriamente por `#pragma omp` (em C) ou `!$omp` (em FORTRAN) e seguida por `[atributos]`, sendo que os atributos são opcionais. Seguem algumas diretivas que compõem a API OpenMP e que

tradicionalmente são usadas pelos desenvolvedores (OPENMP, 2021):

- `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, chamada de função 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.
- `target`: Mapeia variáveis para um ambiente de dados do dispositivo e executa a construção nesse dispositivo.
- `task`: Define uma tarefa explicitamente.

Também há construções de compartilhamento de trabalho, como:

- `section`: A construção de seções é uma construção de compartilhamento de trabalho não iterativo que contém um conjunto de blocos estruturados que devem ser distribuídos e executados pelos *threads* em uma equipe. Cada bloco estruturado é executado uma vez por uma das *threads* da equipe no contexto de sua tarefa implícita.
- `single`: a construção especifica que o bloco estruturado associado é executado por apenas uma das *threads* na equipe (não necessariamente a *thread* principal), no contexto de sua tarefa implícita. As outras *threads* na equipe, que não executam o bloco, esperam em uma barreira implícita no final de uma única região, a menos que uma cláusula `nowait` seja especificada. `workshare`: A construção de compartilhamento de trabalho divide a execução do bloco estruturado fechado em unidades de trabalho separadas e faz com que as *threads* da equipe compartilhem o trabalho de modo que cada unidade seja executada apenas uma vez por uma *thread*, no contexto de sua tarefa implícita.

Na sequência são apresentados alguns atributos da API OpenMP (OPENMP, 2021). 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, 2021):

- `OMP_NUM_THREADS`: Especifica o número *n* de *threads* utilizados nos blocos paralelos do algoritmo.
- `OMP_SCHEDULE`: A variável de ambiente controla o tipo de programação e o tamanho do bloco de todas as diretivas de *loop* do tipo `runtime` com as opções `static, dynamic, guided, or auto`.
- `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*.

4.3.1. Paralelismo de Tarefas

O desenvolvimento de algoritmos paralelos em geral necessita da divisão de trabalho entre as unidades de processamento (PU), processos ou threads, onde cada PU recebe aproximadamente a mesma quantia de trabalho. Idealmente, também precisa-se coordenar os PUs para sincronizar e comunicar entre si. Foster (FOSTER, 1995) descreve um roteiro de quatro passos para desenvolver um programa paralelo: particionamento, comunicação, aglomeração e mapeamento. *Paralelismo de dados* e *paralelismo de tarefas* estão diretamente relacionados com a fase de particionamento, onde expõe-se as oportunidades de concorrência. As duas estratégias dividem o problema em pedaços pequenos baseados nos dados ou na computação, respectivamente.

Paralelismo de dados, também *decomposição de dados* ou *decomposição de domínio*, é um método recorrente para expressar concorrência em algoritmos. Nesse modelo de programação, os dados associados com o problema são particionados e então mapeados para tarefas. Os dados dessa decomposição podem ser dados de entrada, saída, ou intermediários, ou seguem *owner-computes rule* (OCR).

Paralelismo de tarefas, também *paralelismo funcional* or *paralelismo de controle*, representa uma forma diferente e complementar de expressar paralelismo. Essa estratégia decompõe a computação ao invés dos dados manipulados. Esse modelo de programação pode ser utilizado em tarefas que realizam computações diferentes e são independentes. Todavia, o paralelismo de tarefas é utilizado em algoritmos onde as tarefas podem ter dependências que resultam em um grafo acíclico direcionado (DAG). Quando as dependências entre tarefas são associadas aos dados, o algoritmo gera um grafo de fluxo de dados ou *data flow graph* (DFG) (GAUTIER; BESSERON; PIGEON, 2007).

Por exemplo, algoritmos recursivos são um exemplo direto de paralelismo de tarefas em que a chamada recursiva é substituída por uma tarefa e por uma sincronização para esperar os resultados se necessário. Outro exemplo pode ser descrito por laços paralelos em que cada iteração é mapeada para uma tarefa sem dependências. A Figura 4.3 ilustra ambos os exemplos em que pode-se substituir cada chamada de função pela criação de uma tarefa concorrente.

```
1 int fibo( int n ){
2  if( n < 2 ) return n;
3  int x = fibo(n-1);
4  int y = fibo(n-2);
5  return x + y;
6 }
```

```
1 for( i = 0; i < n; i++ )
2  compute_job(i);
```

Figura 4.3. Exemplos em que o paralelismo de tarefas pode ser aplicado para algoritmos recursivos (esquerda) e laços paralelos (direita).

A partir de sua versão 3.0, o OpenMP suporta o paralelismo de tarefas através da construção `task` para tarefas explícitas e `taskwait` para sincronização. A Figura 4.4 ilustra uma função para percorrer listas com criação de tarefas OpenMP. Note que em relação aos outros programas OpenMP, as tarefas são criadas dentro de uma construção `single` na linha 3. Isso se deve ao fato da região paralela executar o mesmo código em todas as threads, o que não é desejado nesse caso. Aqui quero que a execução inicie com uma única thread apenas para que novas tarefas sejam criadas em seguida. Na linha 7 uma nova tarefa é criada para a função `process`. Note que usei a cláusula `firstprivate` pois a variável `p` é modificada na próxima linha. Caso contrário, haveria uma condição de corrida entre a thread que cria tarefas e a nova tarefa.

4.3.2. Dependências de Dados

O paralelismo de tarefas desenrola sua execução em um DAG onde as dependências são descritas pela estrutura recursiva do programa. Esse modo de execução, denominado *fully strict mode*, define que as relações de dependências ocorrem somente entre nós raízes e folhas com ligação direta. Por outro lado, o paralelismo com dependências de dados controla a execução por meio de um grafo de fluxo de dados ou *data flow graph* (DFG) (GAUTIER; BESSERON; PIGEON, 2007). O controle de execução é feito exclusivamente pelo fluxo de dados da aplicação e depende do modo de acesso descrito pela tarefa.

```
 1 #pragma omp parallel
 2 {
 3 #pragma omp single
 4  {
 5   node* p = head;
 6   while(p) {
 7 #pragma omp task firstprivate(p)
 8    process(p);
 9    p = p->next;
10   }
11 #pragma omp taskwait
12  }
13 }
```

Figura 4.4. Exemplo de tarefas OpenMP para visitar elementos de uma lista encadeada.

Os modos de acesso que podem ser listados, de uma forma genérica, são:

- **Read only (RO ou R)** - somente leitura, sem permissão para modificar.
- **Write only (WO ou W)** - somente escrita, sem leitura de dados de entrada.
- **Read and write (RW)** - ou modo exclusivo, com leitura e escrita.

O OpenMP versão 4.0 incluiu o uso de diretivas para expressar dependências de dados em tarefas. A diretiva `depend` de uma construção `task` lista as dependências de dados que podem ser:

- **in** – somente leitura.
- **out** – somente escrita.
- **inout** – leitura e escrita.

Além disso, a API inclui a construção de sincronização **taskgroup** que permite a sincronização implícita ao final do bloco de código a fim de esperar por todas as tarefas criadas recursivamente, o que não era possível com a diretiva **taskwait**.

A Figura 4.5 demonstra um exemplo simples da criação de tarefas OpenMP com dependências de dados de entrada (`in`) e saída (`out`), além da sincronização recursiva para este bloco de código (`taskgroup`).

4.3.3. Aceleradores

O padrão OpenMP 4.5 inclui diretivas de execução de trechos de código em aceleradores por meio do modelo de execução *host-centric* onde a CPU principal , ou *host*, é o lugar

```
1 #pragma omp taskgroup
2 {
3 #pragma omp task depend(in:data) depend(out:result)
4  foo(data, result);
5 }
```

Figura 4.5. Exemplo simples de tarefas OpenMP com dependências de dados.

onde a execução do programa inicia e o *device* seria o acelerador para execução de trechos de código. O acelerador pode executar iterações de laços paralelos por meio de grupos de threads chamados *teams* que cooperam a fim de executar o trabalho.

A construção `target` muda o controle de execução do *host* para o acelerador e a construção `teams` cria um grupo de threads semelhante à construção `parallel`. Apenas algumas construções podem estar aninhadas a um `teams` como `distribute` e `parallel`. A construção `distribute` distribui as iterações de um laço entre as threads do grupo no acelerador. Outros atributos do `distribute` podem determinar o escalonamento e o grão de trabalho a cada thread (`dist_schedule`).

A diretiva `map` descreve o mapeamento explícito de variáveis ao ambiente de dados do acelerador. O tipo de mapeamento de dados com a diretiva `map` pode ser:

- `alloc` - aloca memória para a variável correspondente;
- `to` - aloca memória e copia o valor original para esta variável na entrada;
- `from` - aloca memória e copia o valor dela para a variável original na saída;
- `tofrom` - é o padrão, onde copia o valor na entrada e saída da região.

A construção `target` pode ser acompanhada da diretiva `nowait` indicando que a CPU não espera o término do código na região `target`. A diretiva `depend` também pode ser utilizada a fim de sincronizar trechos de código assíncronos com `nowait`.

A Figura 4.6 demonstra um exemplo simples de programa SAXY de um laço executado em um acelerador com a construção `target`. Primeiramente a construção `target` (linha 5) seguida da construção `teams` define a região a ser acelerada com um grupo de threads juntamente com o mapeamento dos vetores `x` e `y`. O vetor `x` é um dado de entrada e o vetor `y` é entrada e saída. Cada vetor tem o atributo `[0:n]` que define o tamanho do dado mapeado sendo o vetor inteiro em nosso exemplo. Em seguida, a construção `distribute parallel for` permite que o compilador execute um laço paralelo dentro da região acelerada no grupo de threads definido anteriormente.

4.4. Exemplos de Aplicações Científicas

Duas aplicações científicas foram consideradas para a inserção de diretivas paralelas. As próximas subseções descrevem as aplicações que e apresentam as formas como os trechos

```
1 int n = 1024;
2 float a = 32.0f, b = 17.0f;
3 float x[1024], y[1024];
4
5 #pragma omp target teams map (to:x[0:n]) map(tofrom:y[0:n])
6 #pragma omp distribute parallel for
7 for(int i= 0; i < n; i++) {
8   y[i] = a*x[i] + b*y[i];
9 }
```

Figura 4.6. Exemplo simples de uso de OpenMP para aceleradores.

de código foram paralelizados.

4.4.1. Aplicação de Meios Porosos

A alta eficiência do controle de secagem pode superar as perdas de grãos. Evitar a secagem excessiva ou insuficiente é o começo para evitar perdas por secagem. O gasto de muitos recursos para prever a temperatura ideal do grão, a taxa de secagem ao ar, a umidade do grão e o tempo de secagem são necessários para se obter uma melhor eficiência do processo, conforme ilustrado na Figura 4.7. Esses recursos, como experimentos, equipamentos e mão de obra, tornam o estudo experimental mais caro. Uma forma de reduzir esses investimentos é por meio de simulações numéricas. Soluções numéricas de equações da mecânica dos fluidos são utilizadas para representar, com alguns pressupostos de acordo com as condições iniciais e de contorno, fenômenos naturais e artificiais.

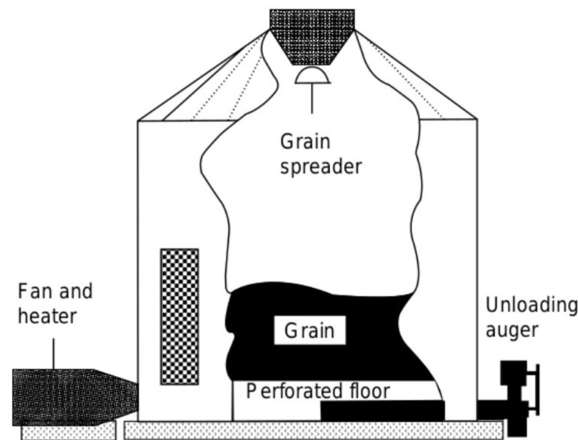


Figura 4.7. Processamento de grãos.

A secagem de grãos é um processo de transferência de calor e massa entre o grão e o ar. Há um movimento de energia do fluxo de ar quente através dos granéis, pelo processo de convecção, que se distribui rapidamente na massa do grão, vaporizando parte da água

do grão. Enquanto isso, a água dentro do grão é transferida pelo processo de difusão como um movimento fluido e como um processo de convecção na superfície úmida.

Considerando que a massa do grão é uma quantidade de espaços sólidos e vazios (orifícios) pelos quais um fluido pode passar, pode-se assumir a secagem do grão como um problema de meio aberto-poroso acoplado. A modelagem matemática e simulação computacional são amplamente utilizadas para descrever a convecção em um fluxo livre com um obstáculo poroso. A previsão da taxa de fluxo que passa através e ao redor de um meio poroso pode ser encontrada em muitos estudos na literatura. Ele usa a formulação da lei de Darcy e suas modificações atuais na parte porosa e a formulação de Navier-Stokes na parte aberta. No entanto, deve-se levar em consideração a mudança abrupta do fluxo livre e do meio poroso, criando uma zona de transição, conforme Figura 4.8.

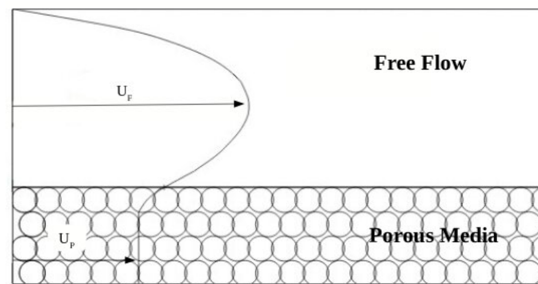


Figura 4.8. O fluxo em meio livre e em meio poroso

A aplicação aqui apresentada modela um problema de convecção através de um meio poroso cilíndrico, adotando uma abordagem de domínio único (OLIVEIRA, 2020). O problema clássico de um fluxo ao redor de um obstáculo de cilindro circular é amplamente estudado em engenharia, principalmente na forma de suprimir o derramamento de vórtices. Assim, um esquema de dinâmica dos fluidos computacional é implementado em FORTRAN usando Volume Finitos para simular e computar as soluções numéricas. O fluxograma do algoritmo é apresentado na Figura 4.9.

Atualmente, o trabalho em desenvolvimento, denominado projeto *Poros*, realiza esta resolução de forma sequencial, utilizando como base as equações de Navier-Stokes (CONSTANTIN; FOIAS, 1988). No entanto, o desempenho obtido fica aquém das necessidades de tempo e poder de processamento existentes, sendo desejada a otimização da execução na busca por ganhos de eficiência na resolução do problema. Dessa forma, este trabalho apresenta melhorias aplicadas ao código, com a utilização de paralelismo através da biblioteca OpenMP (CHANDRASEKARAN; JUCKELAND, 2017).

4.4.2. Método de Lattice-Boltzmann

O MLB é um método numérico iterativo discreto utilizado para a modelagem e simulação mesoscópica de fluxos de fluidos (SUCCI, 2001). A estrutura principal do algoritmo do MLB é constituido de um laço de repetição no qual são feitas operações de movimentação dos dados, simulando um fluxo de fluido. Tais operações consistem na propagação e relação das partículas, além de cálculos de valores macroscópicos e tratamento das condições

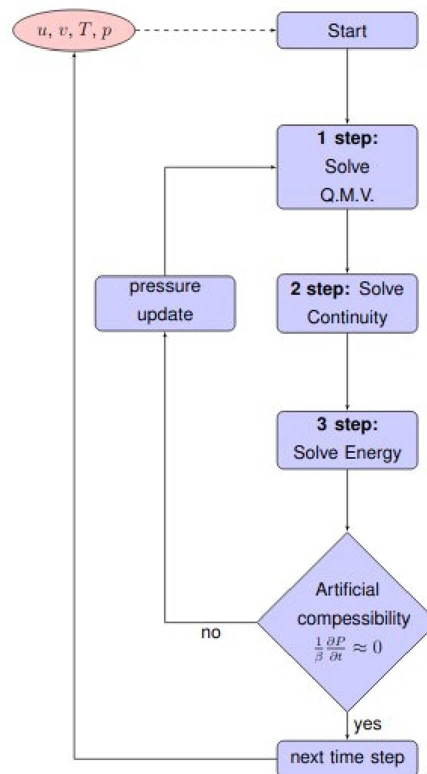


Figura 4.9. Fluxograma do algoritmo de simulação do meio poroso

de contorno (*Bounce Back*, conforme apresentado na Figura 4.10. O critério de parada do laço principal pode ser determinado pelo número de iterações ou por outro fator, como a estabilização do fluxo.

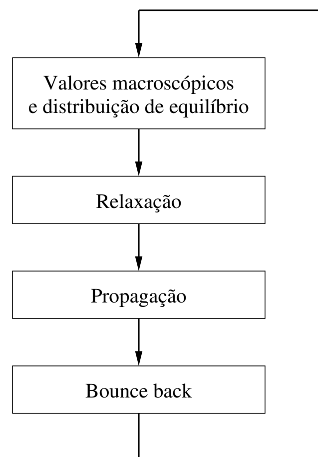


Figura 4.10. Algoritmo do MLB

Para a implementação foi adotado o modelo de reticulado mais utilizado do mé-

todo para o modo bidimensional, com 9 direções de propagação das partículas (CHEN; DOOLEN, 1998). Tal modelo é conhecido por D2Q9, sendo este ilustrado na Figura 4.11.

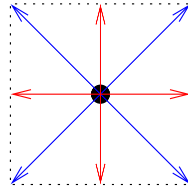


Figura 4.11. Modelos de reticulado bidimencional e tridimencional

A implementação feita consistiu em obter valores macroscópicos, tais como velocidade e pressão, para um fluxo de fluido através de um canal com obstáculos. Como parâmetro de entrada são repassadas algumas informações ao programa através de dois arquivos: um que contém informações genéricas tais como as propriedades macroscópicas e parâmetros de configuração, e o segundo que contém a estrutura de pontos do reticulado (limites e barreiras). Essas informações são armazenadas em duas estruturas de dados independentes.

O estudo de caso escolhido para avaliar a implementação paralela consiste em uma simulação de um fluxo de fluido cruzando canais com obstáculos. A distribuição dos obstáculos é composta de 5 barreiras dispostas ciclicamente ao longo do eixo x, conforme indicado na ilustração da esquerda da Figura 4.12. O tamanho de cada barreira é igual a metade do número de pontos que compõem os elementos da dimensão y. Já a distância entre cada uma das barreiras também é fixa, tendo-se utilizado para isso o valor de $1/5$ do tamanho total de pontos da dimensão x. Nos testes, variou-se o tamanho do reticulado, para avaliar a performance paralela.

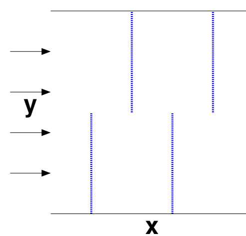


Figura 4.12. Disposição das barreiras no reticulado bidimensional

4.5. Paralelização de Aplicações

Antes de iniciar propriamente a paralelização de qualquer aplicação é necessário um estudo prévio, a fim de identificar os trechos de código que podem ser paralelizados. Um código possui trechos que não são paralelizáveis. Este é o caso de etapas de pré e pós processamento, que incluem a leitura ou escrita de arquivos, a alocação de memória e a inicialização de variáveis. Em outras situações, há trechos de códigos em que o custo de instanciação da execução paralela não compensa o pouco tempo de execução, devido a natureza das operações ou a baixa carga de computação.

Muitos algoritmos tem uma característica iterativa, onde uma etapa depende da computação da etapa anterior e internamente a cada iteração há computações que podem ocorrer concorrentemente. A dependência entre os dados é um fator importante na paralelização de aplicações uma vez que, se não tratado adequadamente, pode gerar resultados incorretos devido ao acesso de posições 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.

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 cópias podem inviabilizar algumas paralelizações, pois a análise não deve considerar apenas a execução das instruções, mas também a sincronização das memórias.

Uma abordagem objetiva para identificar trechos de código paralelizáveis é fazer uso de ferramentas de perfilamento de aplicações. A ferramenta gprof (GRAHAM; KESSLER; MCKUSICK, 1982), por exemplo, faz coletas estatísticas do tempo de execução demandado por cada rotina que compõe o código e tem sido usado por muitos programadores para identificar inicialmente as funções mais custosas do código.

A performance de um código OpenMP 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.

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

4.6. Paralelização de Poros

O algoritmo Poros é composto por um grande laço iterativo que percorre a transição de um tempo discretizado. Para cada tempo discreto há um número máximo de iterações até que se chegue a convergência dos valores de resíduo das propriedades de continuidade e momentos (P, U e V) do código. Nesta etapa iterativa são calculados as equações de Momento de *Quick Scheme* (`solve_U`, `solve_V`), equação de continuidade (`solve_P`)

e equação de Energia (`solve_Z`). Todas as rotinas dessas equações estão implementadas no arquivo `equations.f90`. As rotinas `solve_U` e `solve_V` representam o tempo respectivamente de 43% e 41% do tempo de execução total do código, restando 7% para `solve_Z` e 1% para `solve_P`. Existem ainda as funções `upwind_U` e `upwind_W` que demandam 2% do tempo de execução para cada uma.

As quatro funções executadas na etapa iterativa possuem trechos de código que podem ser paralelizados com OpenMP. Essencialmente há um padrão em cada função: há 4 laços aninhados que percorrem a representação bidimensional do domínio do problema. Assim, cada trecho (`do`) pode ser paralelizado com `!$omp parallel do`, com as devidas variáveis específicas indicadas com `private`. Como cada elemento a ser computado é o mesmo, o paralelismo de laços tende a ser uma abordagem eficiente para garantir um bom desempenho paralelo. A Figura 4.13 demonstra como o paralelismo de laços foi aplicado um trecho de código da rotina `solve_U`.

```
1 !$omp parallel do private(i,j)
2 DO i=3,imax-1
3   DO j=2,jmax-1
4     call solve_res_u(i,j,um,um_tau,RU,res_u)
5     ui(i,j) = ( um_tau(i,j) + res_u(i,j))
6   ENDDO
7 ENDDO
8 !$omp end parallel do
```

Figura 4.13. Implementação de laços paralelos usando a diretiva `parallel do` na função `solve_U`.

Uma outra abordagem de paralelização possível é fazer uso da diretiva `!$omp task`, uma vez que existem trechos de computação que podem ser executados concorrentemente. Isto é, existem rotinas que a cada iteração do tempo discreto do código são executados sequencialmente, mas que poderiam ser executados concorrentemente, pois operam sobre conjuntos de dados distintos. Como exemplo tem-se as operações que são feitas em `solve_U` e `solve_V`, pois são de dimensões distintas (em x e em y).

4.7. Paralelização do Método de Lattice Boltzmann

A paralelização do Método de Lattice Boltzmann apresentada neste capítulo engloba essencialmente a utilização de diretivas do tipo `target` Cada uma das operações da etapa iterativa do método foi implementada como uma função específica. Desta forma, tem-se as funções `redistribute()`, `propagate()`, `bounceback()` e `relaxation()` que operam sobre os elementos do reticulado.

As quatro funções podem ser implementadas de maneira semelhante, fazendo uso da computação em GPU, através da diretiva `target`, variando apenas quais são as variáveis privadas em cada função. A Figura 4.15 apresenta a paralelização da função `bounceback()` usando a opção `target` de paralelismo. O opção `collapse(2)`

indica a junção dos 2 primeiros laços aninhados. Especificamente, esta função realiza ainda uma operação de redução.

```
1  #pragma omp target map(to: lx, ly, n, x, y)
2  #pragma omp teams distribute parallel for private(x, y) /
      collapse(2)
3  for (x = 1; x < lx - 1; x++) {
4   for(y = 1; y < ly - 1; y++) {
5    if (obst[x * ly + y] == true) {
6     int base = (x * ly + y) * n;
7     node[base + 1] = temp[base + 3];
8     node[base + 2] = temp[base + 4];
9     node[base + 3] = temp[base + 1];
10    node[base + 4] = temp[base + 2];
11    node[base + 5] = temp[base + 7];
12    node[base + 6] = temp[base + 8];
13    node[base + 7] = temp[base + 5];
14    node[base + 8] = temp[base + 6];
15   }
16  }
17 }
```

Figura 4.14. Implementação de tarefas usando a diretiva `target` para a função `bounceback()`.

Além das 4 funções executadas na etapa iterativa, uma função de verificação da solução numérica (`check_density()`) também pode ser invocada a qualquer momento da execução da etapa iterativa ou somente ao final da execução. A Figura 4.15 apresenta a paralelização da função `check_density()` usando a opção `teams` de paralelismo.

```
1  #pragma omp teams distribute parallel for private(x, y, _n)/
       reduction(+: n_sum) collapse(2)
2  for (x = 1; x < lx - 1; x++) {
3   for (y = 1; y < ly - 1; y++) {
4     int base = (x * ly + y) * n;
5    for (_n = 0; _n < n; _n++) {
6     n_sum = n_sum + node[base + _n];
7    }
8   }
9  }
```

Figura 4.15. Implementação de tarefas usando `teams` para a função `check_density()`.

Além da paralelização das chamadas, antes da etapa iterativa, é necessário co-

piar os dados para a GPU, conforme apresentado na Figura 4.16. Estas 3 estruturas de dados são manipuladas pelas funções chamadas a cada iteração e não é necessário fazer sincronização a cada iteração.

```
1 #pragma omp target data map(tofrom: temp[0:node_sz], node/
     [0:node_sz], obst[0:obst_sz])
2 {
3   for (time = 0; time < properties->t_max; time++) {
4     ...
5   }
6 }
```

Figura 4.16. Copia das 3 estruturas de dados utilizadas na etapa iterativa para a GPU.

4.8. Conclusão

Paralelizar uma aplicação possui desafios. Muitas vezes é necessário reescrever ou realizar adaptações no código sequencial, para que o mesmo possa ser executado concorrentemente, ou seja, sem dependência de dados ou de operações. Posteriormente, deve-se partir para a paralelização do código. Algumas técnicas de paralelismo podem ser escolhidas por serem mais adequadas para uma determinada classe de problemas ou devido as características que o domínio do problema possui. Também é preciso garantir a equivalência numérica dos resultados, ou seja, uma versão paralela não pode resultar em valores inconsistentes da solução do programa sequencial.

Neste capítulo foram descritas duas aplicações e apresentadas formas de paralelização que puderam ser aplicadas usando a interface de programação OpenMP. As especificações mais recentes de OpenMP permitem tanto a criação de tarefas paralelas quando o uso de GPUs através de diretivas `target`. Desta forma, OpenMP aparece com uma alternativa para que que uma aplicação possa ser paralelizada tanto em um ambiente multicore, quanto many-core, deixando de ser utilizado somente o tradicional paralelismo de laços através da combinação das diretivas `parallel` e `for`.

Referências

Advanced Micro Devices, Inc. *AMD EPYC 7H12*. 2021. páginas 3

CHANDRASEKARAN, S.; JUCKELAND, G. *OpenACC for Programmers: Concepts and Strategies*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2017. ISBN 0134694287. páginas 11

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 4

CHEN, S.; DOOLEN, G. D. Lattice Boltzmann Method for Fluid Flows. *Annual Review of Fluid Mechanics*, v. 30, p. 329–364, 1998. páginas 13

CONSTANTIN, P.; FOIAS, C. *Navier-Stokes Equations*. [S.l.]: University of Chicago Press, 1988. páginas 11

CORPORATION, I. *Intel Advanced Vector Extensions 512 (Intel AVX-512)*. 2021. páginas 3

CORPORATION, I. *Processador Intel Xeon Platinum 8380H*. 2021. páginas 3

FOSTER, I. *Designing and Building Parallel Programs: Concepts and tools for Parallel Software Engineering*. Reading, MA: Addison Wesley, 1995. páginas 6

GAUTIER, T.; BESSERON, X.; PIGEON, L. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: *2007 international workshop on Parallel symbolic computation*. Waterloo, Canada: ACM, 2007. p. 15–23. Disponível em: <https://hal.inria.fr/hal-00684843>. páginas 7

Gonçalves, R.; Girardi, A.; Schepke, C. Performance and energy consumption analysis of coprocessors using different programming models. In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. [S.l.: s.n.], 2018. p. 508–512. páginas 3

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 14

NEC Corporation. *NEC SX-Aurora TSUBASA - Vector Engine*. 2021. páginas 3

NVIDIA. *GPU NVIDIA A100*. 2021. páginas 4

OLIVEIRA, D. P. de. *Fluid Flow Through Porous Media With The One Domain Approach: A Simple Model For Grains Drying*. 49 p. Dissertação (Mestrado) — Universidade Federal do Pampa, Alegrete, 2020. páginas 11

OPENMP. *The OpenMP API specification for parallel programming*. 2021. Disponível em: https://www.openmp.org. Disponível em: <https://www.openmp.org/>. páginas 4, 5, 6

SUCCI, S. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. New York, USA: Oxford University Press, 2001. ISBN 0-19-850398-9. páginas 11

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

Capítulo 5

Ambiente de Nuvem Computacional Privada para Teste e Desenvolvimento de Programas Paralelos

Anderson M. Maliszewski[1], Adriano Vogel[2], Dalvan Griebler[3], Claudio Schepke[4], Philippe O. A. Navaux[5]

Resumo

A computação de alto desempenho costuma utilizar agregados de computadores para a execução de aplicações paralelas. Alternativamente, a computação em nuvem oferece recursos computacionais distribuídos para processamento com um nível de abstração além do tradicional, dinâmico e sob-demanda. Este capítulo tem como objetivo introduzir conceitos básicos, apresentar noções básicas para implantar uma nuvem privada e demonstrar os benefícios para o desenvolvimento e teste de programas paralelos em nuvem.

5.1. Introdução

Com o aumento da complexidade e do número de problemas computacionais, assim como, do valor de aquisição de infraestruturas particulares, percebeu-se um aumento significativo na utilização de ambientes computacionais que proveem recursos de forma

[1]Grupo de Processamento Paralelo e Distribuído - GPPD, Instituto de Informática - INF, Universidade Federal do Rio Grande do Sul - UFRGS - Brasil, email: ammaliszewski@inf.ufrgs.br, ORCID: 0000-0001-5585-3471

[2]Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS - Brasil e Laboratório de Pesquisas Avançadas para Computação em Nuvem - LARCC Faculdade Três de Maio - SETREM - Brasil, email: adriano.vogel@acad.pucrs.br, ORCID: 0000-0003-3299-2641

[3]Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS - Brasil e Laboratório de Pesquisas Avançadas para Computação em Nuvem - LARCC Faculdade Três de Maio - SETREM - Brasil, email: dalvan.griebler@pucrs.br, ORCID: 0000-0002-4690-3964

[4]Laboratório de Estudos Avançados em Computação - LEA Universidade Federal do Pampa - UNIPAMPA - Campus Alegrete – Brasil, email: claudioschepke@unipampa.edu.br, ORCID: 0000-0003-4118-8831

[5]Grupo de Processamento Paralelo e Distribuído - GPPD, Instituto de Informática - INF, Universidade Federal do Rio Grande do Sul - UFRGS- Porto Alegre - RS - Brasil, email:navaux@inf.ufrgs.br, ORCID: 0000-0002-9957-5861

rápida, escalável e com pagamento pelo uso, como a computação em nuvem (MELL; GRANCE et al., 2011; BHOWMIK, 2017). A nuvem, por sua vez, vem sendo desenvolvida e disponibilizada de forma prática desde o inicio da década de 2010, levando a uma migração considerada agressiva de ambientes tradicionais para seu recinto. Entretanto, de acordo com previsões e pesquisas da Gartner[6], essa migração que já era considerada significativa antes da pandemia, tende a aumentar em aproximadamente 18% em 2021, chegando a impressionante marca de 304 bilhões de dólares investidos nesta tecnologia. Essa estatística leva em consideração principalmente a completa "validação" do ambiente de nuvem, uma vez que durante a crise do COVID-19, vários trabalhos tornaram-se remotos ou mesmo necessitaram maior flexibilidade, fazendo constante uso desta.

Consequentemente, essa crescente demanda por computação em nuvem no mercado e seus desafios no gerenciamento de recursos, beneficia a pesquisa e desenvolvimento da mesma. Apesar de que melhorias e novas solução são necessárias, ferramentas que criam as chamadas nuvens privadas (ambientes isolados para o gerenciamento de recursos dentro de instituições), como por exemplo, OpenNebula, OpenStack e CloudStack, já vem sendo estudadas e abordadas em diversos artigos científicos (VOGEL et al., 2016; ROVEDA et al., 2015; MALISZEWSKI et al., 2019). Desta forma, o objetivo deste capítulo é apresentar de forma útil e prática, a criação e implantação de uma nuvem computacional privada. Este tipo de nuvem pode ser uma alternativa para o gerenciamento de recursos computacionais, tornando-a um ambiente eficiente, flexível e de baixo custo para desenvolvimento ou teste de programas paralelos.

A estrutura do capítulo está dividida da seguinte forma. As primeiras seções explicam aspectos conceituais básicos. Na Seção 5.2 são discutidas as formas em que a computação pode ocorrer em ambientes paralelos e distribuídos. Na Seção 5.3 são apresentadas as formas de virtualização em hardware e em sistema operacional. Já na Seção 5.4 tem-se toda a formulação necessária para compreender como é feita a computação em nuvem. A Seção 5.5 mostra como é feita a implementação da nuvem privada com OpenNebula, configuração e realização de testes MPI. Por fim, a Seção 5.6, fecha o capítulo com uma conclusão.

5.2. Computação Paralela e Distribuída

Os computadores paralelos evoluíram de máquinas proprietárias e dedicadas para se tornaram as ferramentas diárias dos cientistas de diversas áreas de conhecimento, que precisam poder de processamento computacional para resolver seus problemas. A computação paralela e distribuída é imprescindível para que as aplicações computacionais possam usar os recursos de hardware disponíveis.

Apesar de terem diferentes significados e implicações, os termos computação paralela e computação distribuída são muitas vezes considerados como sinônimos (BUYYA; VECCHIOLA; SELVI, 2013). Porém, computação paralela corresponde a um modelo onde é possível separar as computações e processá-las individualmente em processadores usualmente homogêneos e se comunicando com memória compartilhada. Por outro lado, o termo computação distribuída é mais genérico, voltado para computações que po-

[6]<https://www.gartner.com/en/newsroom/press-releases/2020-11-17-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-18-percent-in-2021>

dem ser executadas em múltiplos processadores, bem como em múltiplas máquinas, como execuções em *clusters*, onde a comunicação ocorre por trocas de mensagens.

O processamento paralelo ocorre quando múltiplas tarefas são executadas ao mesmo tempo em múltiplos processadores, onde diferentes técnicas, abordagens e modelos de exploração de paralelismo podem ser usados, como mestre-escravo e divisão e conquista. Ainda, chama-se programação paralela a ação de modelar e codificar uma determinada computação para executar em paralelo.

5.2.1. Modelos de Computação Paralela

Executar um programa de forma paralela demanda uma modelagem para explorar apropriadamente os recursos paralelos do hardware, criando um modelo computacional para execução paralela. Um modelo computacional pode ser visto como modelo conceitual, representando as operações e seus tipos disponíveis em um determinado programa, sem considerar sintaxes específicas e é usualmente pouco relacionado com a arquitetura do hardware (GROPP; LUSK; SKJELLUM, 2014). Portanto, um modelo computacional pode ser visto como uma estrutura de alto nível do programa. Porém, o desempenho e funcionalidade do programa paralelo é fortemente relacionado com a máquina a ser executado.

Os modelos computacionais podem ser classificados de diferentes formas e considerando diferentes aspectos, como a memória (compartilhada ou distribuída), a forma de comunicação, a demanda por comunicação, entre outros.

5.2.1.1. Memória Compartilhada

Um modelo computacional com controle simples do paralelismo e da comunicação é o de memória compartilhada. Nesse caso, cada processo ou *thread* tem acesso a toda memória da máquina que é representada como um único espaço compartilhado de endereçamento (GROPP; LUSK; SKJELLUM, 2014). É importante notar que as execuções paralelas demandam mecanismos de controle de acesso à memória, como *locks*, para coordenar o acesso à endereços modificados por múltiplos processos ou *threads*. Um exemplo do modelo de memória compartilhada são os programas executando em sistemas multicores, presentes em computadores pessoais, servidores e até em *smartphones*.

5.2.2. Memória Distribuída

O modelo de memória distribuída usa a troca de mensagens (*message-passing*) para comunicação entre processos que possuem memória local. Nesse caso, a rede de interconexão é usada para envio e recebimento de mensagens. Cada processo pode estar sendo executado em uma máquina diferente, caracterizando uma computação distribuída (GROPP; LUSK; SKJELLUM, 2014).

A computação distribuída é possível a partir da ligação de múltiplos sistemas e computadores independentes que através de abstrações são usados por usuários e gerenciados por programadores, como se fosse um único sistema. Tal abstração é possível graças a uma arquitetura bem definida e diversas interfaces entre as camadas de abstrações e entidades. A Figura 5.1 apresenta um exemplo de arquitetura de um sistema distribuído. No

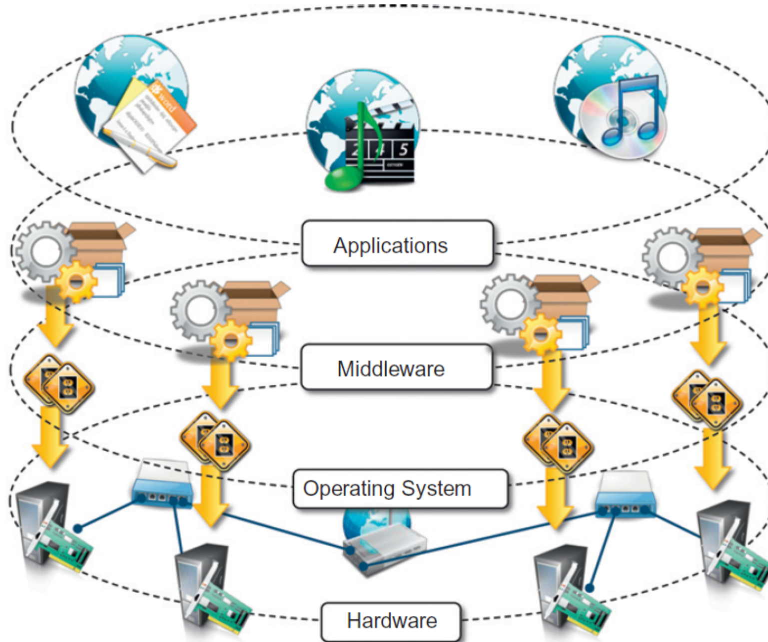


Figura 5.1. Visão geral de um sistema distribuído. Extraído de (BUYYA; VECCHIOLA; SELVI, 2013).

nível mais baixo tem-se o hardware, que é gerenciado pela camada superior: o sistema operacional. O hardware e sistemas operacionais se comunicam usando a infraestrutura de rede (VOGEL et al., 2017; MALISZEWSKI et al., 2019). Acima, o *middleware* é uma camada de abstração que controla os recursos subjacentes e oferece (*Application Programming Interface*) APIs para programar aplicações bem como mecanismos para gerenciar tais aplicações (BUYYA; VECCHIOLA; SELVI, 2013).

Na computação distribuída, uma interface amplamente aceita e utilizada para a programação é o MPI (SNIR et al., 1998), que pode ser definido como uma interface padrão com especificações e rotinas para implementar programas com execuções distribuídas, implementada nas linguagens C, C++ e Fortran. No Algoritmo 5.1 é mostrado um exemplo de código MPI, onde um vetor é computado em paralelo por `size` processos. Primeiramente, o processo identificado pelo `rank` 0 inicializa um vetor e os demais processos recebem esse vetor através de uma operação de *broadcast* (`MPI_Bcast`). Na sequência, cada processo realiza uma soma parcial de elementos de um trecho do vetor. Através de uma operação de redução, o processo de `rank` 0 recebe o resultado da computação parcial de cada processo e exibe o resultado final. Por fim, deve-se lembrar que toda a computação paralela de MPI deve estar limitada entre as chamadas `MPI_Init` e `MPI_Finalize`.

```
1 #include <mpi.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define TAM 100
5
6 int main(int argc, char** argv) {
```

```
7     int myrank, size;
8     int i, local = 0, total;
9     int vet[TAM];
10
11    MPI_Init(&argc, &argv);
12    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); //Quem sou?
13    MPI_Comm_size(MPI_COMM_WORLD, &size); //Quantos somos?
14
15    if (myrank == 0)
16      for(i = 0; i < TAM; i++)
17        vet[i] = 1;
18    MPI_Bcast(vet, TAM, MPI_INT, 0, MPI_COMM_WORLD);
19
20    for(i=(TAM/size)*myrank; i<(TAM/size)*(myrank+1); i++)
21      local += vet[i]; // Realiza as somas parciais
22
23    MPI_Reduce(&local,&total,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
24
25    if (myrank == 0)
26      printf("Soma = %d\n", total);
27    MPI_Finalize();
28    return 0;
29  }
```

Listing 5.1. Um exemplo de computação paralela usando MPI.

5.3. Virtualização

A virtualização permite abstrair os recursos físicos como memória, processadores, armazenamento e rede, de tal forma que é possível criar ambientes virtuais e fornecer recursos usando máquinas virtuais (VMs). Portanto, a virtualização é um paradigma que revolucionou a forma como recursos computacionais são fornecidos para os usuários e é considerada a tecnologia principal para criação da computação em nuvem. Com a virtualização, é possível um melhor uso e gerenciamento dos recursos, instalando vários sistemas operacionais em diferentes máquinas virtuais no mesmo *hardware* (CHANDRASEKARAN, 2014), como representado na Figura 5.2.

Existem diferentes tipos e abordagens de virtualização. Neste documento, é relevante definir dois tipos: virtualização a nível de hardware usando virtualizadores representada pela solução (KVM) e virtualização de sistema operacional representada pelo LXC.

5.3.1. Virtualização de Hardware com KVM

A virtualização no nível de hardware é uma forma de virtualização que abstrai o hardware de computador, sendo possível executar nesse hardware múltiplos sistemas operacionais independentes e separados. Tais sistemas operacionais são executados dentro de VMs, hospedados pelo hardware físico do computador e gerenciados pelo virtualizador (*Hypervisor*). O virtualizador pode ser visto como uma camada que é um programa ou uma combinação de software e hardware que efetivamente abstrai o hardware (BUYYA; VECCHIOLA; SELVI, 2013).

KVM é uma solução de código aberto que cria um ambiente de virtualização com-

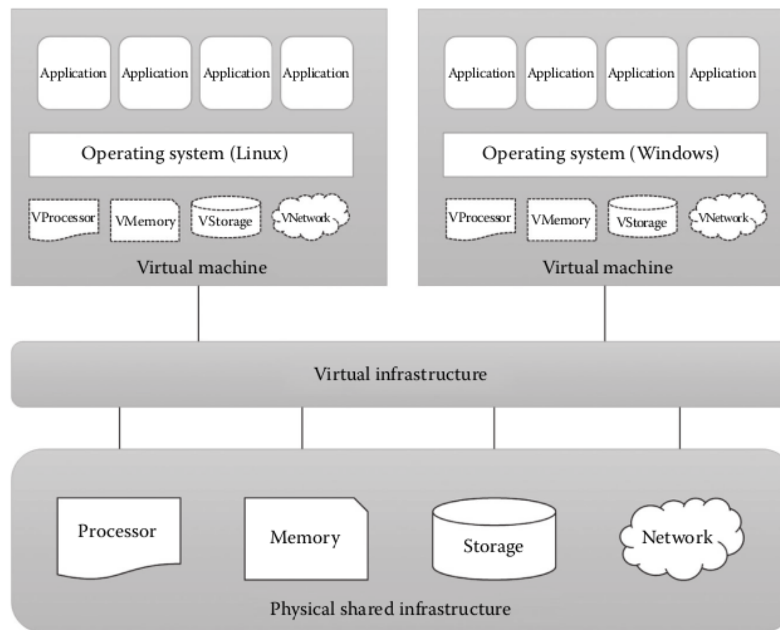


Figura 5.2. Visão geral de virtualização. Extraído de (CHANDRASEKARAN, 2014).

pleta de hardware. No KVM, cada máquina virtual criada é tratada com um processo regular do Linux e recebe um cotas de recursos e camadas de isolamento. Nesse cenário, cada VM tem seu próprio *kernel* separado.

5.3.2. Virtualização de Sistema Operacional LXC

Na virtualização no nível do sistema operacional, diferentemente da virtualização de hardware, não há gerenciador de máquina virtual ou virtualizador. Na virtualização de sistema operacional, o *kernel* do sistema operacional é uma das partes mais importantes pois este que permite várias instâncias isoladas seja executadas. Nesse cenário, o *kernel* do sistema operacional pode ser compartilhado e disponibiliza recursos do para as instâncias. Ainda, o *kernel* define e aplica cotas que limitam a quantidade de recursos que cada instância pode utilizar, como recursos de processador e memória.

O LXC é uma virtualização em nível de Sistema Operacional (SO), que abstrai os recursos computacionais por meio de Grupos de Controle (*cgroups*). No LXC, a criação e limitação do uso de recuros dos contêineres (LXC, 2019)é feita através de *namespaces*. Portanto, ps contêineres compartilham o mesmo *kernel* do sistema operacional nativo, onde seus processos e sistema de arquivos são acessíveis a partir do *host* hospedeiro possibilitando a execução de instruções nativas sem demandar mecanismos adicionais de interpretação. Nos contêineres, apesar do compartilhamento do *kernel* e de recursos, o sistema operacional fornece para as aplicações a ilusão de estarem executando em máquinas separadas.

A Figura 5.3 mostra uma comparação entre LXC (virtualização no nível do SO) e KVM (virtualização de hardware). Como pode ser visto, o LXC requer menos camadas de

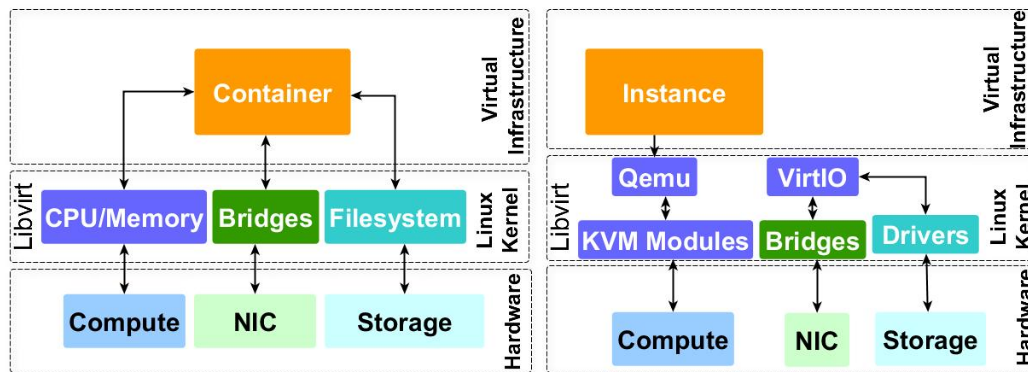


Figura 5.3. Visão geral do LXC (direita) e KVM (esquerda). Extraído de (VOGEL et al., 2017)

software, pois o KVM usa *drivers* VirtIO e bibliotecas para fornecer recursos e gerenciar as VMs.

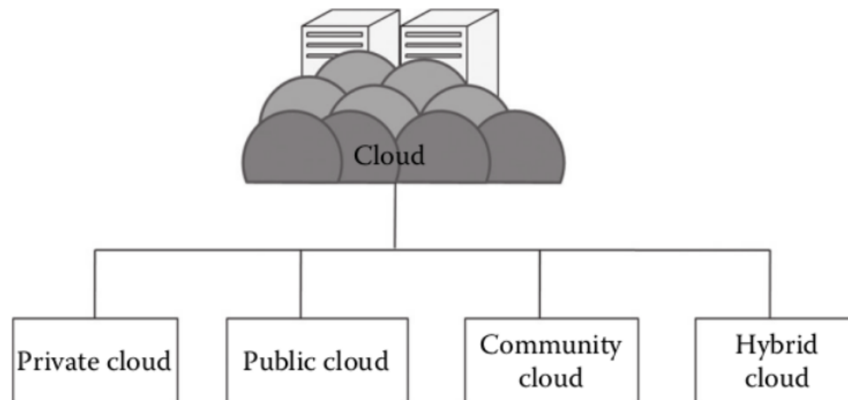
5.4. Computação em Nuvem

O paradigma de Computação em Nuvem surgiu para oferecer recursos computacionais na forma de serviços, facilitando o acesso e aumentando a disponibilidade de recursos computacionais (ROLOFF et al., 2012; VOGEL et al., 2016; GRIEBLER et al., 2018). Na infraestrutura, as aplicações e serviços são executados em ambientes virtualizados e oferecidos de forma simplificada usando conhecidos padrões e protocolos de rede. Conforme (BUYYA; VECCHIOLA; SELVI, 2013; CHANDRASEKARAN, 2014; BHOWMIK, 2017), as principais vantagens oferecidas pela Computação em Nuvem são: diminuição de custos de uso e investimento inicial, acesso amplo a recursos computacionais, potencial de escalabilidade e alta disponibilidade dos serviços. O provisionamento de serviços é dividido em IaaS (*Infrastructure as a Service*), PaaS (*Platform as a Service*), e SaaS (*Software as a Service*).

Computação em Nuvem é um conceito em consolidação de uma tecnologia que pode servir tanto para usuários finais, quanto para pesquisadores e/ou empresas, que através desta podem disponibilizar serviços com uma grande gama de opções e vantagens para seus clientes. Ela é constituída principalmente sobre os pilares de outras tecnologias (cluster, grid, redes), oferecendo serviços, que é basicamente definido pelo compartilhamento de recursos configuráveis. Sobre as tecnologias que compõem a nuvem, destaca-se a virtualização, que dentre vários benefícios, busca aproveitar e obter o máximo desempenho possível do hardware (BUYYA; VECCHIOLA; SELVI, 2013).

Dentre as características e vantagens que a Nuvem possui, pode-se destacar. A alta disponibilidade, na qual o usuário pode acessar sua infraestrutura de qualquer local apenas sendo necessária a conexão a internet. O provisionamento de recursos sob demanda, significa que a utilização dos recursos é disposta de acordo com o uso necessário do ambiente/aplicação, desta forma, evitando desperdício de utilização computacional que resulta em fatores positivos como a economia energética e compromisso ambiental. A rápida elasticidade, diferentemente de uma infraestrutura local, com a utilização da computação

Figura 5.4. Representação dos modelos de implantação de Nuvem. Figura extraída de (CHANDRASEKARAN, 2014)



em nuvem, pode-se aumentar ou diminuir a quantidade de recursos que se está utilizando. Para o usuário final, esta quantidade parece ser infinita e pode ser modificada em qualquer quantidade e em qualquer tempo.

Também é importante destacar o pagamento pelo uso. Apenas é necessário efetuar o pagamento em relação ao tempo de uso/quantidade de recursos que foram "locados" na nuvem, acarretando em economia financeira se comparada a uma infraestrutura convencional. Os serviços mensuráveis, qualquer serviço que é oferecido a partir da computação em nuvem é automaticamente controlado e otimizado no nível de abstração para sua demanda específica (ex. armazenamento, processamento, tráfego de rede, entre outros). Além disso, a utilização dos serviços pode ser monitorada e reportada, oferecendo transparência tanto para quem provisiona o serviço, quanto para quem o utiliza (VACCA, 2016; MELL; GRANCE et al., 2011; CHANDRASEKARAN, 2014).

A composição da nuvem é disposta em modelos de serviço que constituem uma pilha de três camadas, sendo elas o IaaS (*Infrastructure as a Service*) como base, PaaS (*Platform as a Service*) logo acima e SaaS (*Software as a Service*) no topo da pilha. Além disso, a nuvem inclui quatro modelos de implantação, descritos como Nuvem Privada, Nuvem Comunitária, Nuvem Pública e Nuvem Híbrida (BUYYA; VECCHIOLA; SELVI, 2013). Os modelos de implantação e os modelos de serviço serão descritos a seguir.

Visando uma melhor forma de disponibilização de serviços e objetivo final de sua utilização, a computação em nuvem divide-se em quatro principais modelos de implantação, cada qual com ênfase em uma determinada área. Na Figura 5.4 estão representados os 4 modelos de implantação, juntamente com suas características principais, a seguir descritos.

- **Nuvem Pública:** caracteriza-se pela utilização aberta ao público em geral. Neste modelo, é necessário que haja um provedor de nuvem. Os recursos oferecidos normalmente são gratuitos e passíveis de aquisição para realização de upgrades (ex. Maior capacidade de armazenamento). Esse é o modelo de nuvem mais conhecido entre os usuários. Destaca-se principalmente como provedores as empresas gigante da área de TI, que oferecem suas respectivas soluções em nuvem (ex. Amazon com

o AWS, Microsoft com o Azure, Google com o Google Drive).

- **Nuvem Privada:** este modelo é criado/utilizado dentro de organizações empresariais, as quais gerenciam, criam e disponibilizam seus próprios recursos dentro da nuvem. Além disso, este modelo necessita de uma plataforma de criação de nuvem, sendo que a partir desta todos os recursos são gerenciados. Como exemplo temos o CloudStack, OpenNebula e o OpenStack, este último sendo a plataforma de cloud mais popular e utilizada.

- **Nuvem Híbrida:** este modelo tem como característica única a utilização de duas ou mais infraestruturas de nuvem distintas (ex. Nuvens privada, comunitária ou pública) que de forma isolada, continuam tendo a mesma designação, porém, utilizam recursos umas das outras (ex. Cloud bursting para balanceamento de carga entre nuvens).

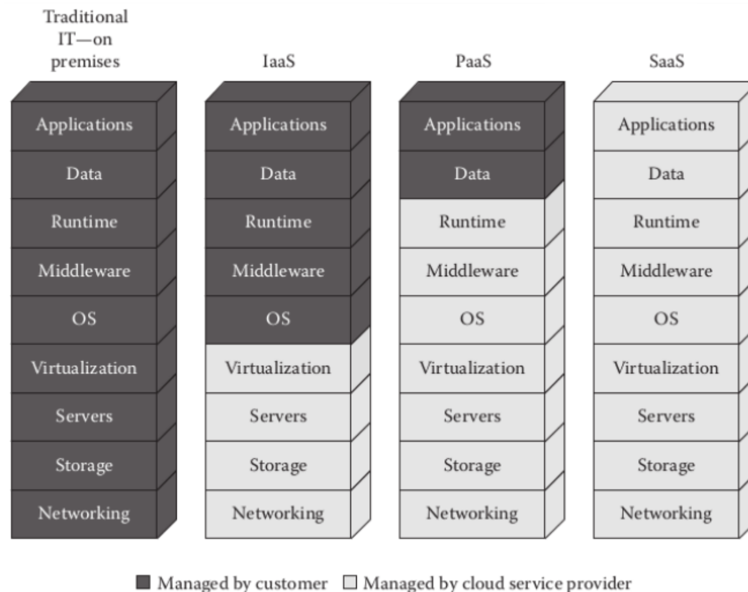
- **Nuvem Comunitária:** é aquela em que os recursos são compartilhados entre várias organizações de uma área específica. Pode ser tanto gerenciada internamente, quanto terceirizada. Este é o modelo de nuvem menos conhecido e também o menos utilizado (MELL; GRANCE et al., 2011).

Através de três modelos de serviço, a nuvem busca atender a todo o tipo de demanda requisitada pelo usuário e/ou empresa. Na Figura 5.5 é ilustrado a representação em camadas de como os tipos de serviço se situam e qual é o campo de trabalho de cada uma. Na primeira camada, tem-se o IaaS (Infrastructure as a Service). Esta permite que os usuários realizem modificações de baixo nível, em comparação com as demais camadas. Nela são efetuados o provisionamento de recursos (ex. Armazenamento e processamento) a disposição do usuário. Além disso, este é capaz de implantar e executar qualquer software que inclui desde sistemas operacionais (SO) até aplicações propriamente ditas (BUYYA; VECCHIOLA; SELVI, 2013; CHANDRASEKARAN, 2014). Entretanto, mesmo esta sendo o camada de menor nível, o usuário que contrata um ambiente de nuvem IaaS não pode efetuar modificações ou controles “abaixo” de sua infraestrutura (ex. Modificações físicas no hardware). Esta camada é a que fornece infraestrutura para as demais (ex. instâncias), além de gerenciar questões como a virtualização.

Logo a seguir, temos o PaaS (Platform as a Service) que está diretamente associado a utilização e desenvolvimento de software através de uma infraestrutura já criada. O usuário pode configurar e realizar modificações ao nível de software, porém, não controla questões relacionadas a infraestrutura (ex. Armazenamento, rede ou servidores). Portanto, neste modelo de serviço é provisionado ao usuário a capacidade de realizar a implantação e criação de programas, além do compartilhamento deste para com os demais usuários (VACCA, 2016).

No topo da pilha de camadas temos o SaaS (Software as a Service), que como o próprio nome sugere, provisiona ao usuário o uso de aplicações executadas diretamente na nuvem. Além disso, as aplicações ficam disponíveis, em relação ao acesso, de várias maneiras (ex. Thin Client, navegador de internet ou mesmo uma interface da aplicação). Esta é a camada de maior nível, ou seja, mais distante do hardware. Com isso, o usuário fica limitado a realizar modificações apenas de acesso a aplicação por exemplo, excluindo qualquer modificação na infraestrutura (MELL; GRANCE et al., 2011).

Figura 5.5. Representação das camadas e serviços na nuvem. Figura Extraída de (VACCA, 2016).



5.4.1. Ferramentas para Nuvem Privada

Existem diversas ferramentas de código aberto para gerenciamento de infraestrutura de nuvem e criação de ambientes de nuvem pública, privada e comunitária. A virtualização que é uma camada adicionado no *hardware* é controlada e gerenciado por tais ferramentas. Logo a seguir, são apresentadas as principais ferramentas para implantação de ambientes de nuvem privada.

5.4.1.1. OpenStack

OpenStack[7] é uma ferramenta *open source* muito usada para criação de ambientes de nuvens privadas e públicas e para IaaS como um todo. O projeto OpenStack foi lançado pelas respeitadas Rackspace e NASA, e posteriormente diversas gigantes do mundo da tecnologia uniram-se ao projeto.

O OpenStack pode ser visto com uma combinação de diversos componentes e serviços independentes que funcionam de forma interoperável (VOGEL et al., 2016), o que aumenta a flexibilidade de implantações de nuvem. A escolha de qual componente usar é customizável de acordo com a demanda de cada arquiteto de nuvem (CHOWDHURY, 2017), sendo apenas necessária adoção de alguns componentes imprescindíveis do *core* da ferramenta.

Alguns componentes se destacam por serem imprescindíveis para uma ambiente de nuvem. Por exemplo, o *Keystone* que controla a autorização e autenticação de todo o ambiente de nuvem (componentes, serviços, usuários, etc) e precisa ser configurado e

[7] <https://www.openstack.org/>

conectado a cada componentes configurado no ambiente de nuvem. Ainda, o *Glance* é um componente que provê um repositório de imagens para a criação de instâncias na nuvem e componente *Neutron* oferece interconectividade. Diversos outros componentes podem ser adicionados a nuvem de forma customizável e a comunicação e sincronização entre os componentes ocorre por trocas de mensagens, sendo o *RabbitMQ* o mensageiro mais popular.

5.4.1.2. OpenNebula

OpenNebula[8] é uma ferramenta que surgiu na academia em um projeto que buscava propor novas soluções para gerenciamento de recursos virtuais em *data centers*. Após isso se tornou uma ferramenta para provisionamento de serviços de nuvem. O OpenNebula tem uma característica de buscar uma implantação simplificada, intuitiva, eficiente, e customizável para usuários e desenvolvedores (MILOJIČIĆ; LLORENTE; MONTERO, 2011).

A arquitetura do OpenNebula também buscar ser modular, iniciando com os *drivers* básicos que implementar controle sob a virtualização e no núcleo da ferramenta são implementadas as rotinas mais importantes como o controle de serviços, usuários, e escalonamento.

5.4.1.3. CloudStack

CloudStack[9] foi inicialmente desenvolvido pela empresa estadunidense de software *Cloud.com* que mais tarde foi adquirida pela *Citrix Systems*. Porém, a *Citrix Systems* doou o CloudStack à *Apache Software Foundation*. Dessa forma, o CloudStack passou a se chamar Apache CloudStack, que se tornou uma ferramenta estável principalmente para implantação de ambientes de nuvem privada. O Apache CloudStack tem APIs próprias e também suporta APIs compatíveis com a AWS (*Amazon Web Services*) que permitem integrar a nuvem privada com nuvens públicas tornando-se, portanto, uma nuvem híbrida (BHOWMIK, 2017; VOGEL et al., 2016).

O Apache CloudStack é focado para alta disponibilidade e flexibilidade e a sua instalação tem 2 componentes importantes: o gerente e o agente de nuvem. O gerente controla a infraestrutura virtual e é instalado no servidor principal, enquanto o agente é instalado nos demais nodos formandos clusters de disponibilidade de recursos.

5.4.2. Gerenciamento de Infraestrutura e Recursos

Em ambientes de nuvem é possível otimizar o gerenciamento de recursos através da flexibilidade da camada de virtualização e do controle oferecido por ferramentas de IaaS (BHOWMIK, 2017; VOGEL et al., 2016). Da perspectiva de gerenciamento da infraestrutura, exemplos de funcionalidades relevantes são:

- Cotas de uso de recursos: permite ao administrador do ambiente definir a quantidade que cada usuário pode alocar e utilizar. Por exemplo, no IaaS a flexibilidade

[8] <https://opennebula.io/>
[9] <https://cloudstack.apache.org/>

é maior pois é possível um usuário alocar uma determinada quantidade de recursos como processadores e memória ao invés de receber alguma máquina física. Tal cota terá um impacto na disponibilidade de recursos para as aplicações executadas no ambiente de nuvem, o que demanda um controle do uso re recursos de processamento (VOGEL; GRIEBLER; FERNANDES, 2021).

- Nodos dedicados para determinados usuários: para usuários que demandam um maior controle sobre o ambiente é possível criar máquinas dedicadas para usuários. Por exemplo, um usuário com requisitos rígidos de segurança.

- Otimizar a alocação de instâncias nos nodos físicos: a flexibilidade da virtualização também permite otimizar quais instâncias são executadas em quais nodos físicos, podendo oferecer ganhos de desempenho ou maior eficiência energética. Por exemplo, sob menor demanda as VMs podem ser movidas para alguns nodos específicos enquanto outros nodos podem ser desligados, potencialmente reduzindo o consumo energético. Em outros casos, VMs com maior demanda de desempenho podem ser alocadas de forma dedicada em nodos adequados.

Considerando as evidentes vantagens oferecidas por ambientes de nuvem, a tendência é que cada vez mais aplicações e cargas de trabalho sejam migradas para a nuvem. Dessa forma, existem diversas demandas de melhorias para o processo de migração, pois costuma ser mais complexo gerenciar tais ambientes robustos, o que demanda automações e abstrações adicionais para se ter produtividade. Uma potencial solução é o uso de contêineres que facilitam a execução de aplicações e o gerenciamento do ambiente. Quanto a programabilidade, se acredita que os *frameworks* de programação serão cada vez mais integrados e flexíveis para ambientes e aplicações da nuvem.

5.5. Ambiente de Desenvolvimento e Execução

Para este curso, escolheu-se utilizar a ferramenta de nuvem privada OpenNebula. Essa escolha baseou-se em fatores como uma maior facilidade para a implantação e configuração, o desenvolvimento da ferramenta ser de código aberto, além de já estar sendo utilizada em produção para realização de pesquisas no LARCC (Laboratório de Pesquisas Avançadas para Computação em Nuvem). A seguir serão descritos todos os procedimentos para instalar, configurar e gerenciar a nuvem privada OpenNebula[10].

5.5.1. Implantação do OpenNebula

A ferramenta OpenNebula fornece um ambiente intuitivo e simples, mas ao mesmo tempo oferece várias funcionalidades e soluções flexíveis para a implantação de nuvens e ambientes de virtualização privados ou híbridos. Primeiramente, serão descritos os processos de definição da arquitetura e projeto da nuvem a ser implantada.

[10] Os itens a seguir estão descritos igualmente como na documentação oficial da ferramenta, localizada em <https://docs.opennebula.io/5.12/index.html>

5.5.1.1. Arquitetura da Nuvem

Para criar um ambiente confiável é necessário criar um plano de implantação. Neste projeto, devem ser alinhados funcionalidades esperadas e também quais componentes de hardware serão adicionados e abstraídos para a nuvem. Isso engloba:

- A infraestrutura - *hardware* (rede, *storages* e servidores).
- O dimensionamento de recursos da nuvem, baseando-se principalmente em características como número de usuários e cargas de trabalho que serão utilizadas.
- Fluxo de provisionamento, o qual inclui a forma de como os usuários serão isolados, utilização da nuvem e qual será a forma de seu acesso.

É necessário criar um plano que inclua ferramentas, desempenho, escalabilidade e características de alta disponibilidade. Desta forma, a arquitetura de nuvem é definida em três componentes, sendo eles, o armazenamento, rede e a virtualização. O OpenNebula presume que o ambiente ao qual será instalado utiliza a arquitetura clássica de um *cluster beowulf*, com um *Frontend* e demais *hosts* que serão utilizados para hospedar as VMs[11]. Além disso, também é necessário de uma rede física para que os nodos se comuniquem e sejam adicionados no *frontend*. Os componentes previamente descritos da arquitetura, tem funções específicas definidas pelo OpenNebula, sendo elas:

- *Frontend*: Executar os serviços do OpenNebula.
- *Hosts*: Provisionam os recursos necessários para as VMs. (Necessário que esteja habilitado o suporte à virtualização.)
- *Storage*: Responsável por armazenar as imagens de *templates* e discos das VMS.
- Redes Físicas: Usadas para realizar a comunicação entre o *storage*, *frontend*, e *hosts* com as VMs.

Na Figura 5.6 temos a representação da arquitetura descrita e a interconexão entre os componentes. Somado a isso, uma ampla gama de recursos pode ser incorporada juntamente ao OpenNebula.

5.5.1.2. Dimensionando a Nuvem

O dimensionamento de recursos da infraestrutura de nuvem é um dos pontos chave para garantir o bom funcionando do sistema. Existem requisitos mínimos disponibilizados pelo OpenNebula para o *frontend*, nodos KVM e nodos LXC. A seguir eles serão listados na Tabela 5.1.

[11]Embora a utilização do OpenNebula possa ser realizada em apenas um *host*, onde serão instalados ambas soluções de *frontend* e nodo, essa forma de implantação terá várias limitações, incluindo desempenho e escalonamento, por exemplo. Assim, recomenda-se a utilização de um *cluster*

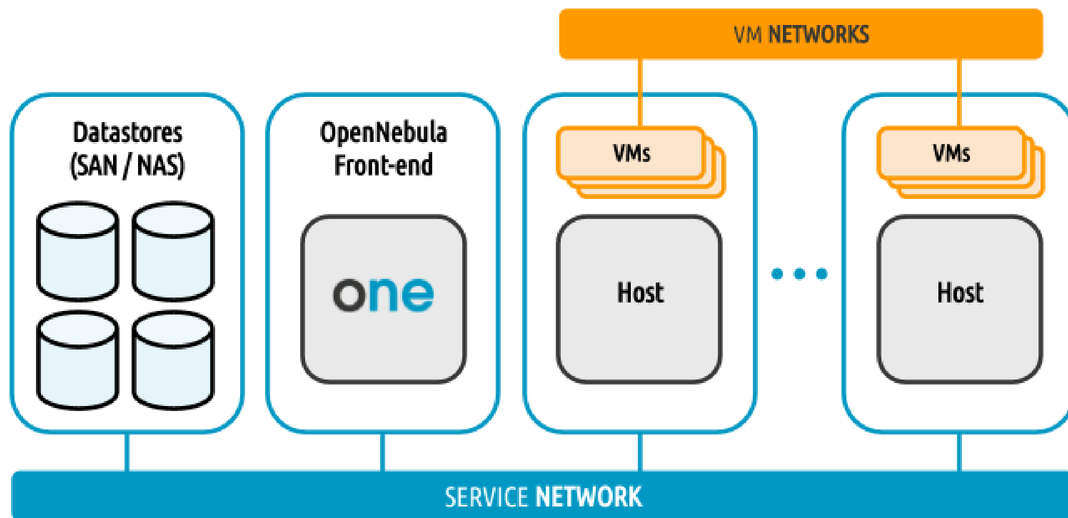


Figura 5.6. Representação da arquitetura da nuvem. Figura extraída de (OPENNEBULA, 2021)

.

	Frontend	Nodos KVM	Nodos LXD
Recursos	Mínimo	Mínimo	Mínimo
Memória	8 GB	1 GB para cada CPU core	>Nodos KVM
CPU	2 CPU (4 cores)	Varia de acordo com a quantidade de *hosts* e utilização de *overcommitment*.	>Nodos KVM
Disco	200 GB		
Rede	2 NICs		

Tabela 5.1. Requisitos Mínimos de Hardware, incluindo o *frontend*, nodos que utilizam o virtualizador KVM e LXD.

Os requisitos de *hardware* dos nodos KVM variam de acordo a utilização de premissas como *overcommitment* de CPU. Caso esta opção não esteja sendo utilizada, define-se que cada núcleo de CPU delegado para uma VM deve existir fisicamente e 1 GB de memória RAM disponível para cada núcleo de CPU. Por outro lado, com a utilização de *overcommitment* de CPU, o dimensionamento de CPU pode ser realizado com antecedência, usando os atributos de CPU e vCPU. Por outro lado, os nodos LXD não definem precisamente uma quantia mínima de recursos necessários, uma vez que este não emula/-virtualiza o hardware e seu *overhead* de virtualização é menor se comparado ao KVM. Portanto, os recursos mínimos para os nodos LXD são menores que os requisitos mínimos para nodos KVM.

Para dimensionar o armazenamento no OpenNebula, é necessário primeiro entender como o armazenamento é organizado. Primeiramente, ele é dividido em forma de três *datastores*: o de imagem, sistema e de arquivos. O primeiro é onde a ferramenta guarda todas as imagens que podem ser utilizadas para criar VMs (imagens de sistemas operacionais). As imagens são movidas, clonadas, de/para o sistema de *datastores* quando as VMs são implantadas, desligadas, adicionados discos ou criados *snapshots*. Desta forma, é necessário que o armazenamento criado tenha pelo menos o tamanho para guardar o número de imagens a serem utilizadas. O *datastore* de sistemas é representado por onde as VMs em execução armazenam seus dados. Dependendo da tecnologia utilizada no armazenamento, essas imagens podem ser copias completas da imagem original `QCOW` ou simplesmente os links do sistema de arquivos. Estimar o tamanho mínimo das imagens é mais complexo, uma vez que podem ser utilizadas discos voláteise varia conforme o sistema operacional utilizado nas VMs. Por fim, o *datastore* de arquivos é usado para armazenar arquivos comuns, como documentos ou anotações.

Em relação a rede, é necessário que esta seja criada com cuidado a fim de garantir a confiabilidade da infraestrutura de nuvem. Recomenda-se 2 interfaces físicas de rede (NICs) no *frontend*, ou 3 dependendo da forma de armazenamento utilizada. Nos *hosts* são recomendadas 4 interfaces (IP privado, IP público, serviços e armazenamento). No entanto, para ambientes com necessidades menores, um número menor de interfaces pode ser adotada.

5.5.1.3. Frontend

A máquina física que hospeda a instalação do OpenNebula (ONE) é chamada de *frontend*. Ela necessita de conexão de rede com todos os *hosts* e o *storage* de *datastores*. Os serviços básicos do OpenNebula incluem o *daemon* de gerenciamento (`oned`), escalonador (`mm_sched`), servidor de interface web (`sunstone-server`). Outros serviços podem ser instalados e habilitados em instalações regulares do OpenNebula. Além disso, o banco de dados padrão utilizado é o *sqllite*. Caso o ambiente seja de produção pode-se considerar a utilização do *MySQL* que é uma solução mais robusta.

5.5.1.4. Monitoramento

O monitoramento reúne informações relacionadas aos *hosts* e máquinas virtuais, como por exemplo, status do *host*, indicadores de desempenho, status das VMs, quantidade de recursos consumidos. Tais informações são coletadas através da execução de rotinas de coleta de dados disponibilizadas pelo ONE. No processo para a coleta de dados, cada *host* envia os dados monitorados para o *frontend* e este os processa em um modulo dedicado. Este modulo é altamente escalável e é limitado pelo desempenho do servidor executando o *oned* e o servidor de banco de dados.

5.5.1.5. Hosts Virtualizados

Os *hosts* são as máquinas físicas que hospedam as VMs em um ambiente de nuvem usando o OpenNebula. O subsistema de virtualização é responsável por intermediar os virtualizares instalados nos *hosts* e as ações necessárias em cada passo do ciclo de vida de uma VM. Nativamente, o OpenNebula suporta três virtualizadores de código aberto, o KVM, LXD e Firecracker. Idealmente, as configurações dos *hosts* devem ser homogêneas em termos de softwares instalados, usuário administrador *oneadmin*, acessibilidade ao armazenamento e conectividade de rede. Por outro lado, é natural que existam diferentes tipos de *hosts* (com configurações e propósitos diferentes), desta forma, estes podem ser agrupados em aglomerados computacionais. Por exemplos, *cluster* KVM, *cluster* LXD.

5.5.1.6. Armazenamento

A ferramenta OpenNebula utiliza *datastores* para armazenar imagens de disco das VMs. Um *datastore* pode ser um tipo de servidor de armazenamento, por exemplo, um servidores NAS (*Network Attached Storage*) ou SAN (*Storage Area Network*). No geral, os *datastores* precisam estar accessíveis através do *frontend* usando qualquer tecnologia (NAS, SAN ou armazenamento local). Quando uma VM é implantada, sua imagem é copiada para o *host*. Baseado na tecnologia de armazenamento utilizada, isso pode significar uma cópia real, um link simbólico ou a configuração de um volume LVM (*Logical Volume Manager*. O OpenNebula é disponibilizado com 3 classes de *datastores*, descritos anteriormente (de imagens, sistemas e arquivos).

Os tipos de *datastores* de imagem podem ser diferentes, dependendo da tecnologia de armazenamento utilizada. São eles: *Filesystem*, que armazena as imagens em forma de arquivo, divididos em trés tipos (SSH, compartilhados e QCOW), LVM e Ceph. Se implantado de forma usual (sem customizações em relação ao armazenamento), o *datastore* de imagens fica localizado no *frontend* usando o tipo *filesystem*, e no momento em que as VMs são implantadas, elas são copiadas para os *hosts* que hospedam os discos da VM criada, usando SSH.

5.5.1.7. Rede

O ONE possui um subsistema de rede que é facilmente customizável e adaptável para integrar os requerimentos do sistema ao *cluster* existente. Recomenda-se ao menos duas redes físicas diferentes, sendo elas:

- Rede de Serviços: é utilizada pelo *frontend* para acessar os *hosts* para gerenciá-los, monitorar os virtualizadores e mover imagens. É recomendável que esta seja uma rede separada da rede das instâncias.
- Rede de Instância: Provisiona a conectividade de rede para as VMs entre diferentes *hosts*.

Quando uma VM é criada, o OpenNebula conecta suas interfaces de rede até o virtualizador com as definições estabelecidas nas redes virtuais. Isso irá permitir que a VM tenha acesso a diferentes redes, sejam elas públicas ou privadas. O gerenciador de nuvens privada suporta quatro modos de redes para criação da conexões virtuais, sendo elas:

- *Bridge*: a máquina virtual é diretamente anexada a uma *bridge* Linux existente no virtualizador.
- VLAN: as redes virtuais são implementadas através das TAGs 802.1Q.
- VXLAN: as redes virtuais implementam VLANs usando o protocolo VXLAN.
- Open vSwitch: Similar ao modo VLAN mas usando Openvswitch ao invés de *bridges* Linux. Também pode ser utilizado com VXLAN.

5.5.1.8. Autenticação

A autenticação na interface de linha de comando do OpenNebula pode ser realizada de quatro formas diferentes. Usuário/Senha criados durante a instalação da ferramenta, chaves SSH, certificados x509 (sendo possível utilizar tanto para acessar a CLI como o Sunstone) e o LDAP, permitindo a utilização da autenticação por meio de um gerenciador de contas centralizado externo ao ONE.

A autenticação do usuário nas suas VMs geralmente é feita através de chaves privadas. Esse recursos deve ser configurado pelo administrador do sistema durante a criação do usuário, no qual ele adiciona a chave pública deste. Além disso, nas *templates* compartilhadas com os usuários, deve estar selecionado a opção de contextualização SSH. Desta forma, quando o usuário criar uma VM, o OpenNebula automaticamente irá inserir a chave publica e este poderá acessar suas VM com segurança.

5.5.1.9. Gerenciamento de recursos

Uma das estratégias mais concisas dentro do OpenNebula em relação a recursos computacionais entregues aos usuários/grupos é a criação de cotas. O sistema de cotas rastreia as informações de utilização de recursos por parte dos usuários/grupos e permite que os administradores limitem o uso destes. Além disso, elas podem ser modificadas constantemente de acordo com a necessidade.

Esse sistema de cotas pode ser feito para usuários, criando a limitação de cotas individuais, ou para grupos, onde a limitação atua como média de uso por todos os usuários do grupo. Finalmente, os recursos que podem ser limitados vão desde *datastores* (quantidade de recursos alocados para cada usuário/grupo em cada *datastore*), computação (incluindo memória, CPU), rede (número de IPs) e imagens (número de imagens). Por padrão o número de recursos provisionado aos usuários é infinito, causando a impressão de que existem recursos ilimitados.

5.5.1.10. Gerenciamento de *templates*

Umas das vantagens que se destacam na utilização de ambientes como nuvens privadas, são as chamadas *templates*. Estes discos pré-configurados facilitam a implantação com configurações homogênea em escala e também o tempo de configuração necessário para novas VMs é significativamente reduzido. No OpenNebula, primeiramente deve-se instanciar ou criar uma VM. Depois disso, a VM pode ser configurada conforme as necessidades de utilização, e por fim, podem ser criadas as *templates* baseadas no disco da VM. Uma vez criadas, elas podem ser instanciadas em escala, possibilitando a criação de um ambiente pré-configurado.

Existem atributos que podem ser configurados pelos administradores em relação as escolhas de recursos das *templates*. Por exemplo, a criação de uma *template* para usuários finais. O dono da *template* pode configurar para que atributos da capacidade de hardware (CPU, memória e discos) sejam usados em determinadas quantidades e se podem ou não ser modificados. Outro ponto importante é a possibilidade de inserir a funcionalidade de sugerir o usuário a inserir certos atributos, ou seja, tornar os atributos pré-estabelecidos como dinâmicos. Desta forma, é necessário modificá-los todas as vezes que a *template* é instanciada. Este cenário torna-se interessante para ambientes com poucos recursos, ou mesmo para o correto dimensionamento das VMs, sem alocar *hardware* físico de forma demasiada.

Além disso, o gerenciamento das *templates* torna o ambiente altamente dinâmico e flexível (premissas clássicas da computação em nuvem), através de suas rotinas, como por exemplo criar, deletar, clonar, atualizar e compartilhar entre usuários. Outra técnica que quando combinada com o uso de *templates* torna o utilização da nuvem ainda melhor é a criação de *snapshots* de discos. Com isso, durante a utilização da VM, o usuário pode realizar um *snapshot*, sendo salvo o atual estado do disco e memória da mesma, possibilitando retornar no futuro a esse estado salvo. Também é possível exportar esse disco e utilizá-lo para novas *templates*.

5.5.2. Instalação

A instalação da ferramenta OpenNebula se inicia por sua parte central, o *frontend*. Nesta máquina será instalada o servidor do software que irá gerenciar todos os recursos computacionais e provisioná-los. A primeira etapa da instalação parte da escolha entre as edições "*Enterprise*"e "*Community*", sendo necessário na primeira destas, a aquisição de uma licença para utilização. (Utilizamos neste minicurso a versão sem assinatura). Uma vez realizada a escolha entre as edições, é necessário adicionar o repositório da ferramenta e atualizar a lista de pacotes disponíveis. Após, é necessário realizar a instalação propriamente dita, utilizando pacotes disponibilizados pelo repositório previamente adicionado. Por fim, é possível iniciar o OpenNebula, sendo disponibilizado o acesso a sua interface gráfica.

A próxima etapa a ser realizada é a instalação e configuração do nodo, que será utilizado como hospedeiro das VMs criadas. Primeiro deve-se escolher quais tipos de virtualização serão utilizados. As tecnologias mais comuns dentro do ambiente OpenNebula são o KVM e LXD. A instalação de ambos é semelhante e parte do mesmo ponto que o *frontend*, onde são adicionados os respectivos repositórios da ferramenta e atualizados a lista de pacotes disponível. A seguir são instalados os pacotes da ferramenta. A próxima etapa é a configuração SSH. Nela, o serviço SSH deve ser configurado para realizar conexões entre todos os *hosts* que compõe o sistema, sejam eles *frontend* ou nodos. Após isso, é necessário configurar a rede e como ela será utilizada. Como descrito anteriormente, a rede pode ser utilizada de várias formas. Aqui, usaremos a rede em forma de *bridge*, criando uma *bridge* Linux da interface de rede dos *hosts* que possui acesso a Internet. Por fim, é necessário registrar o nodo no OpenNebula, para que este possa utilizar os recursos computacionais do *host*. Esta última etapa pode ser realizada pela interface gráfica do ONE, conhecida como *Sunstone*, ou via linha de comando. É importante ressaltar que também é necessário configurar o arquivo "*host*". Este é utilizado pelo sistema operacional para relacionar *hostnames* e endereços IP. O processo de instalação e configuração do OpenNebula, tanto do *frontend* como dos *hosts* nodos está descrito no Github[12].

Uma vez instalados ambos *frontend* e nodo(s), deve-se verificar se os *hosts* encontram-se em status "ON". Após, pode-se realizar o download de imagens KVM/LXD pré-configuradas pela loja da ferramenta. Quando finalizado, as VMs podem ser instanciadas e customizadas de acordo com as preferências. Entretanto, as VMs não possuem uma rede configurada ainda, sendo necessário criar uma rede virtual no OpenNebula.

5.5.3. Executando programas MPI no OpenNebula

Após a configuração e instalação do OpenNebula, o usuário pode logar na interface Web deste, chamada de *Sunstone*. O painel de entrada está dividido nas Figuras 5.7 e 5.8, visualizado logo após o login do usuário, onde são mostrados as VMs, os recursos disponíveis e a porcentagem de utilização de cotas disponibilizadas para o usuário.

Como pode ser visto, o usuário possui 13 VMs criadas no total, 7 em execução e 6 desligadas. Além disso, na parte das cotas, pode-se visualizar a quantidade de recursos utilizada de um total "infinito", que na verdade apenas não esta delimitado pelo

[12] <https://github.com/larcc-group/opennebula-erad2021>

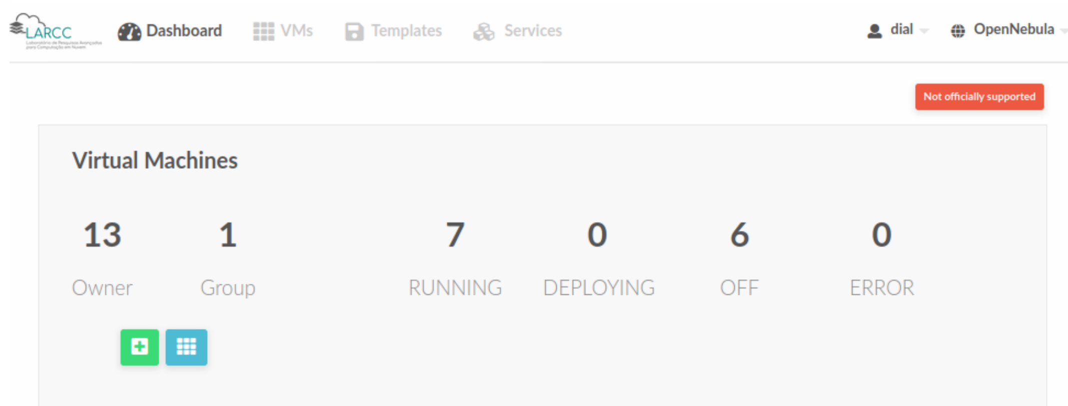


Figura 5.7. Painel de entrada dos usuários OpenNebula.

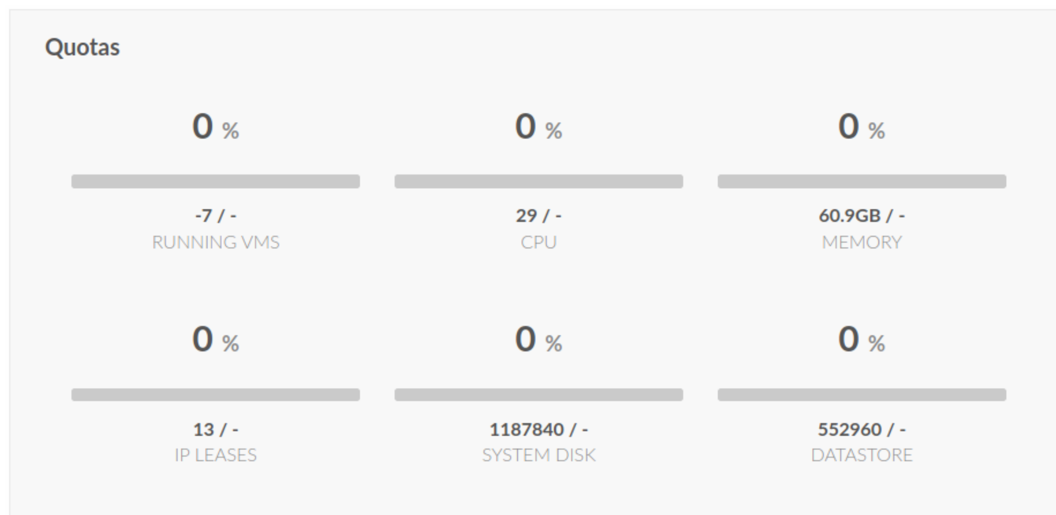


Figura 5.8. Cotas delimitadas pelo administrador da nuvem.

administrador. Agora realizaremos a criação de duas instâncias para demonstrar uma experimentação usando *benchmarks* paralelos que utilizam MPI. O conjunto de *benchmark* utilizado é o conhecido *NAS Parallel Benchmarks* (BAILEY et al., 1991). Para esta experimentação utilizaremos instâncias criadas pelo OpenNebula com o virtualizador KVM, sendo a primeira totalmente configurada e a segunda sendo criada em forma de *template* com base na primeira instância.

O processo para criação de uma VM pode ser feita pelo Sunstone ou por linha de comando. Utilizaremos a interface Web como forma mais intuitiva e simples. Basta clicar no sinal de "+" em verde, visto na Figura 5.7 e selecionar uma *template* que foi disponibilizada pelo administrador durante o processo de instalação (veja a Seção 5.5.2). Feita a criação da primeira VM, podemos fazer o acesso SSH até esta usando o IP disponibilizado. Após, faremos a atualização do sistema e instalação de alguns pacotes.

```
sudo apt update -y sudo apt upgrade -y
sudo apt install make gfortran openmpi-bin libopenmpi-dev \
g++ openssh-server -y
```

Preferencialmente, edita-se o nome do *hostname*, o que facilitará o acesso entre as instâncias. Isso é feito usando os seguintes comandos.

```
vim /etc/hostname
```

Dentro do arquivo *hostname* digitamos o nome `inst1`, e reiniciamos a instância. Após a reinicialização, o sistema já terá o nome que foi definido, e será necessário modificar o arquivo *hosts*, incluindo o IP e nome do *host*.

```
vim /etc/hosts
```

Para realizar a execução de programas paralelos é indicado que se utilize um usuário comum, sendo assim, realizamos a criação do mesmo a seguir.

```
sudo adduser erad
su erad
cd
```

Os campos que serão solicitados pós utilização do primeiro comando acima serão ignorados, apenas prosseguindo usando a tecla *Enter* e por fim `Y` para confirmar a criação do usuário. O segundo comando realizara login e o terceiro irá para a *home* no *user* erad. Agora, é necessário configurar o SSH sem senha entre as instâncias, realizando a criação da chave RSA através do comando:

```
ssh-keygen -t rsa
```

Os campos que serão solicitados pós utilização do comando não necessitam de modificações, apenas prosseguir usando a tecla *Enter* para cria a chave. Como criaremos

uma *template* pronta do ambiente já configurado, podemos liberar o acesso SSH da instância para com ela mesma, inserindo a chave pública no documento de chaves autorizadas com o seguinte comando:

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Apenas no primeiro acesso será necessário aceitar a conexão, a partir da próxima vez, ela será feita de forma automática e sem senha. Agora vamos realizar o download do *benchmark* NAS, logo após a de-compressão, sendo feita da seguinte maneira:

```
cd $HOME
wget https://www.nas.nasa.gov/assets/npb/NPB3.4.1.tar.gz
tar -zxv NPB3.4.1.tar.gz
cd NPB3.4.1/NPB3.4-MPI/
```

Os *benchmarks* serão compilados usando um arquivo de configuração do compilador e outro da suíte, ambos localizados no diretório *config*. Por primeiro realizaremos a copia dos arquivo de configuração.

```
cp config/make.def.template config/make.def
cp config/suite.def.template config/suite.def
```

Após realizaremos uma alterações no arquivo de compilação, sendo a troca do compilador Fortran para `mpifort`. Para realizar essas alteração utilize o comando abaixo:

```
sed -i "s/MPIFC = mpif90/MPIFC = mpifort/g" config/make.def
```

O arquivo da suíte descreve os *benchmarks* e tamanhos de execução (definidos em letras). Por padrão todos os programas estão listados para serem compilados com a classe S, mas iremos modificá-los para uma classe com maior entrada de dados. Para realizar isso, execute os comandos a seguir que realizam a inserção das aplicações junto com a classe A no arquivo `suite.def`.

```
for bench in bt cg ep ft is lu mg sp; do
    for size in A; do
        echo "$bench    $size" >> config/suite.def;
    done;
done
```

Agora realizamos a compilação dos programas, utilizando o comando:

```
make suite
```

Por fim, podemos testar rapidamente um dos programas compilados:

```
mpiexec -np 1 bin/bt.S.x
```

Ao final da execução, que deve acontecer em poucos segundos, será exibido um relatório onde está localizado o tempo de execução da aplicação em questão. Agora podemos criar uma *template* da instância. Para criar, primeiro precisamos nos logar na interface web do OpenNebula, clicar na representação da VM e desligá-la. Após ir no ícone de salvar em verde e aguardar o processo terminar. A última parte deste processo está ilustrada na Figura 5.9. Na próxima tela será necessário nomear a *template* e concluir esta etapa.

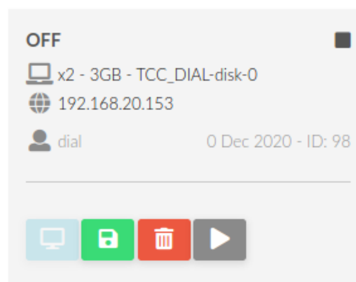


Figura 5.9. Criando *templates* do VMs.

Por fim, podemos religar a instância normalmente, clicando no ícone "*Play*" em cinza na Figura 5.9 e criar uma nova VM pré-configurada usando a *template*. Para realizar a criação da VM, deve-se utilizar o processo já descrito, clicar no ícone "+" da Figura 5.7 e selecionar o nome da *template* salva. Com a nova VM criada, precisamos apenas modificar o nome desta no arquivo *hostname* para inst2, e adicionamos o IP e *hostname* no arquivo *hosts*. É necessário reiniciar a VM.

```
vim /etc/hostname
vim /etc/hosts
```

Ainda, na primeira VM configurada, também é necessário adicionar o IP e *hostname* no arquivo *hosts* da VM criada a partir da *template*.

```
vim /etc/hosts
```

Agora, é necessário realizar o primeiro acesso entre as VMs e após esse acesso, esse processo será realizado sem senha. Assim, através da utilização da *template* para criar a segunda VM (poderiam ser N VMs), criou-se um ambiente computacional totalmente funcional em menos de 5 minutos. Agora podemos executar uma aplicação usando o seguinte comando:

```
mpiexec -np 4 --host inst1,inst1,inst2,inst2 \
NPB3.4.1/NPB3.4-MPI/bin/ft.A.x
```

Desta forma, estamos executando a aplicação FT classe A com 4 processos localizados 2 em cada instância. Ainda poderíamos fazer essa delimitação do numero de processos usando um arquivo:

```
echo "inst1 slots=2" > hostsmpi
echo "ints2 slots=2" >> hostsmpi
```

Assim, podemos modificar o comando de execução para:

```
mpiexec -np 4 --machinefile hostsmpi \
NPB3.4.1/NPB3.4-MPI/bin/ft.A.x
```

5.6. Conclusão

Este minicurso abordou aspectos relacionados a utilização de uma nuvem privada de forma útil e prática. Este ambiente torna-se uma alternativa para o gerenciamento e disponibilização de recursos computacionais de forma rápida e flexível, por meio de ferramentas como as *templates*, que por sua vez, podem criar aglomerados computacionais em poucos minutos. Foram elucidados os principais conceitos metodológicos que regem um ambiente de nuvem criado com a ferramenta OpenNebula, assim como realização da parte prática de instalação, configuração e teste de aplicações paralelas na nuvem. Espera-se que através dos conceitos e passos mostrados neste documento, os participantes possam realizar a implantação da nuvem privada e aplicação dos conceitos abordados.

Agradecimentos

Este trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001, o Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), e dos projetos “GREENCLOUD: Computação em Cloud com Computação Sustentável"(Nº 16/2551-0000 488-9), da FAPERGS e CNPq Brasil, programa PRONEX 12/2014 e o projeto SPARCLOUD (Nº 17/2551-0000871-5) do Edital Universal MCTIC/CNPq 28/2018. Por fim, os autores agradecem ao Laboratório de Pesquisa Avançada em Computação em Nuvem (LARCC / SETREM, Brasil) por fornecer recursos de computação em nuvem, que contribuíram para a realização deste minicurso. URL: <https://larcc.setrem.com.br>

Referências

BAILEY, D. H. et al. The NAS Parallel Benchmarks; Summary and Preliminary Results. In: *ACM/IEEE Conference on Supercomputing (SC)*. [S.l.: s.n.], 1991.

BHOWMIK, S. *Cloud Computing*. [S.l.]: Cambridge University Press, 2017. ISBN 9781316638101.

BUYYA, R.; VECCHIOLA, C.; SELVI, S. T. *Mastering Cloud Computing: Foundations and Applications Programming*. [S.l.]: Newnes, 2013.

CHANDRASEKARAN, K. *Essentials of Cloud Computing*. [S.l.]: Taylor & Francis, 2014. ISBN 9781482205435.

CHOWDHURY, O. K. C. D. *Mastering OpenStack*. [S.l.]: Packt Publishing Ltd, 2017.

GRIEBLER, D. et al. Performance of Data Mining, Media, and Financial Applications under Private Cloud Conditions. In: *IEEE Symposium on Computers and Communications (ISCC)*. Natal, Brazil: IEEE, 2018.

GROPP, W. D.; LUSK, E.; SKJELLUM, A. *Using MPI: portable parallel programming with the message-passing interface*. [S.l.]: MIT press, 2014.

LXC. *Linux Containers (LXC)*. 2019. Último acesso em dezembro de 2020. Disponível em: <http://linuxcontainers.org/>.

MALISZEWSKI, A. M. et al. Minimizing Communication Overheads in Container-based Clouds for HPC Applications. In: IEEE. *IEEE Symposium on Computers and Communications (ISCC)*. Barcelona, Spain, 2019.

MELL, P.; GRANCE, T. et al. The NIST Definition of Cloud Computing. *National Institute of Standards and Technology (NIST)*, Gaithersburg, United States, 2011.

MILOJIČIĆ, D.; LLORENTE, I.; MONTERO, R. Opennebula: A cloud management tool. *IEEE Internet Computing*, IEEE, 2011.

OPENNEBULA. *OpenNebula 5.12 Documentation*. 2021. Available on: <https://docs.opennebula.io/5.12/index.html>. Access date: 20 March.

ROLOFF, E. et al. High Performance Computing in the Cloud: Deployment, Performance and Cost Efficiency. In: *International Conference on Cloud Computing Technology and Science Proceedings (CloudCom)*. [S.l.: s.n.], 2012.

ROVEDA, D. et al. Analisando a Camada de Gerenciamento das Ferramentas CloudStack e OpenStack para Nuvens Privadas. In: *Escola Regional de Redes de Computadores (ERRC)*. Passo Fundo, Brazil: [s.n.], 2015.

SNIR, M. et al. *MPI-the Complete Reference: the MPI core*. [S.l.]: MIT press, 1998.

VACCA, J. R. *Cloud Computing Security: Foundations and Challenges*. [S.l.]: CRC Press, 2016.

VOGEL, A.; GRIEBLER, D.; FERNANDES, L. G. Providing High-level Self-adaptive Abstractions for Stream Parallelism on Multicores. *Softw Pract Exp*, 2021.

VOGEL, A. et al. Private IaaS Clouds: A Comparative Analysis of OpenNebula, CloudStack and OpenStack. In: *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. Heraklion Crete, Greece: IEEE, 2016.

VOGEL, A. et al. An Intra-Cloud Networking Performance Evaluation on CloudStack Environment. In: *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. St. Petersburg, Russia: IEEE, 2017.

Capítulo 6

Desenvolvimento de Aplicações Baseadas em Tarefas com OpenMP Tasks

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

Resumo

O minicurso enquadra-se no contexto de programação paralela utilizando diretivas de programação para facilitar o desenvolvimento de aplicações. O paradigma orientado a tarefas aporta facilidades na programação paralela porque transfere para um runtime muitas responsabilidades que seriam anteriormente realizadas pelo programador nos paradigmas tradicionais. Por exemplo, é responsabilidade do runtime escalonar as tarefas nas unidades de processamento, gerenciar a memória e o balanceamento de carga. Para isso, basta o programador definir as tarefas e suas dependências de dados. Neste minicurso, será apresentado o paradigma de programação paralela orientado a tarefas, e como construir programas com diretivas de programação utilizando tarefas OpenMP. O minicurso será conduzido de forma prática, com exemplos e exercícios de programas básicos e naturalmente adaptáveis ao paradigma orientado a tarefa. Serão empregados exemplos de aplicações como Mergesort, suavização de Gauss-Seidel e fatoração Cholesky. Por fim, abordaremos rapidamente as ferramentas e métodos de como analisar o desempenho destes programas.

6.1. Introdução

O panorama do Processamento de Alto Desempenho (PAD) passou por uma mudança de paradigma nos últimos anos. A inerente estagnação no aumento da frequência do processador levou à adoção de outras maneiras de atender à necessidade cada vez maior de poder de computação das aplicações paralelas. Atualmente, as plataformas de PAD envolvem nós com processadores equipados com múltiplos *cores* aprimorados com várias placas aceleradoras.

Essa mudança de paradigma no *hardware* revela limitações nas ferramentas tradicionais para programação de aplicações paralelas para PAD. Programar com eficiência essas máquinas, com desempenho portável e escalável, mantém-se desafiador. Por exemplo, o uso de modelos de programação explícitos, tais como a interface de programação MPI e versões antigas do OpenMP, exigem se preocupar com o balanceamento de carga, equilíbrio entre comunicações e computação. Frequentemente, tais preocupações imputadas ao desenvolvedor tornam a aplicação fortemente acoplada a uma plataforma alvo. Por consequência, esse modelo de programação explícito se torna inviável considerando as plataformas para PAD evoluem rapidamente ao longo do tempo.

Enquanto o paradigma de programação paralela tradicional depende de abstrações de baixo nível, como fluxos de execução e sincronizações explícitas, o modelo baseado em tarefas descreve a aplicação paralela com tarefas sequenciais dependentes. As sincronizações explícitas são substituídas por dependências de tarefas que podem ser, em vários casos, inferidas automaticamente do acesso aos dados pelo próprio ambiente de execução. Outra evidência é que semanticamente alguns algoritmos são melhor expressados em tarefas. Um exemplo são algoritmos do tipo "dividir para conquistar", como o *merge sort*, onde uma implementação recursiva com tarefas é natural. Existem outros benefícios da programação baseada em tarefas [Rico et al. 2019], tais como dependências finas entre tarefas no lugar barreiras de sincronização, e o uso de algoritmos de escalonamento sofisticados. O modelo baseado em tarefas é implementado por vários modelos de programação: OpenMP 5 [OpenMP 2020], OmpSs [Duran et al. 2011], Parsec [Bosilca et al. 2012], StarPU [Augonnet et al. 2011], entre outras. A crescente disponibilidade de ferramentas para programar e executar aplicações baseados em tarefas em plataformas híbridas demostra a crescente importância do paradigma baseado em tarefas.

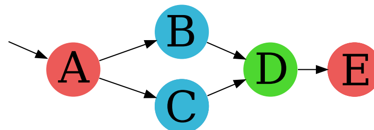


Figura 6.1. Um grafo com três tipos de tarefas (vermelhas, azuis e verdes), sendo que as dependências de dados são representadas pelas arestas.

A interface de programação OpenMP [OpenMP 2020] suporta a programação paralela em máquinas com memória compartilhada, tipicamente um nó computacional equipado com múltiplos processadores *multicore*. Desde as especificações 3.0 e 4.0, OpenMP especifica diretivas para a programação com tarefas. Com estas diretivas, o programador especifica a submissão de tarefas e a relação de dependência entre elas. Um grafo é frequentemente utilizado para ilustrar a aplicação como um todo, como representado na Figura 6.1. Nesta figura, o grafo representa uma aplicação com cinco tarefas (nós identificados de A até E) de três tipos (cores). Tarefas de diferentes tipos implementam funcionalidades diversas, mas que cooperam entre si através das dependências de dados (arestas). Cada tarefa é composta por um trecho de código, frequentemente sequencial, que implementa uma funcionalidade baseada em seus dados de entrada, e gerando na saída um outro conjunto de dados processados. Após a compilação, o ambiente de execução escalonará a tarefa A para execução, que uma vez terminada permitirá o escalonamento para execução das tarefas B e C, e assim por diante.

Este minicurso aborda a forma de se construir programas paralelos com diretivas de programação utilizando tarefas OpenMP. O minicurso traz exemplos e exercícios de programas básicos e naturalmente adaptáveis ao paradigma de tarefas. O texto do minicurso está organizado da seguinte forma. A Seção 6.2 apresenta as diretivas OpenMP relacionadas à criação do grafo de tarefas e geração das dependências. A Seção 6.3 apresenta exemplos de aplicações desenvolvidas em OpenMP de maneira a ilustrar boas práticas de programação. A Seção 6.4 apresenta um método para avaliar o desempenho das aplicações utilizando técnicas de rastreamento da execução das tarefas. Enfim, a Seção 6.5 apresenta um sumário e recomendações de como dar continuidade no estudo sobre o assunto. O repositório[1] contém códigos, apresentação e outros materiais deste minicurso.

6.2. Programação Paralela com Tarefas em OpenMP

A utilização do OpenMP Tasks segue inicialmente os mesmos princípios da utilização de outras diretivas do OpenMP. Anotações de código com `#pragma omp` são inseridas no código em lugares estratégicos. Um compilador que suporta OpenMP (como `gcc`, `clang`, `icc`) é utilizado para compilar o código com as `flags` adequadas[2]. O gerenciamento de *threads* é realizado pelo OpenMP em regiões paralelas anotadas com a diretiva `#pragma omp parallel`. Usualmente precisamos declarar regiões que devem ser executadas por uma única *thread* dentro de regiões `parallel`, para isso, usamos `#pragma omp single` nestes blocos. O OpenMP também dispõe de uma interface de funções, no cabeçalho `omp.h`, para gerenciar e acessar dados disponíveis. Um exemplo dessas funções, utilizada para se obter o identificador da *thread* executando o fluxo de execução, é `omp_get_thread_num`. O exemplo de código abaixo mostra a estrutura básica de um programa OpenMP.

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main(){
4    #pragma omp parallel
5    {//OpenMP inicia várias threads
6      printf("Bloco paralelo executado por:%d\n", omp_get_thread_num());
7      #pragma omp single
8      {// Este Bloco é executado unicamente por uma única thread
9        printf("Bloco único executado por:%d\n", omp_get_thread_num());
10     }
11   }
12 }
```

Para a compilação deste programa utilizando o `gcc`, pode-se utilizar:

```
gcc estrutura_openmp.c -fopenmp
```

O controle do acesso de variáveis em regiões paralelas com tasks é igual a outras diretivas tradicionais como `parallel for`, sendo dada pelas anotações `shared`, `private`, `firstprivate` e `lastprivate`. Por exemplo, em um laço anotado com `parallel for`, onde deseja-se que a variável `vec_c` seja compartilhada por todas as

[1] `https://gitlab.com/lnesi/companion-minicurso-openmp-tasks`

[2] As flags de compilação variam por compilador: -fopenmp (gcc e clang), -qopenmp (icc).

threads, e a variável `vec_p` seja privada (instanciada, lida e modificada apenas dentro de cada *thread*) utiliza-se o código abaixo:

```
int vec_c = 5;
int vec_p;
#pragma omp parallel for shared(vec_c) private(vec_p)
for(int i=0; i<10; i++){
   // vec_c é compartilhada por todas as threads
   // Existe uma vec_p para cada thread
}
```

Depois de relembrar os conceitos básicos do OpenMP, podemos iniciar com as diretivas orientadas a tarefas. Começamos pela principal diretiva para definição de uma tarefa. Ela é dada por `#pragma omp task` imediatamente antes de um bloco de código (que pode ser uma chamada de função) que fará ofício de tarefa, tal como ilustrado neste exemplo:

```
void minha_tarefa(char* s){
    printf("%s ", s);
}
int main(){
  #pragma omp parallel
  {//OpenMP inicia várias threads
    #pragma omp single
    {// Este Bloco é executado unicamente por uma thread
       #pragma omp task
       minha_tarefa("Olá");
       //#pragma omp taskwait
       #pragma omp task
       minha_tarefa("Mundo");
    }
  }
}
```

Cada invocação da função `minha_tarefa` será uma tarefa que pode ser executada por qualquer *thread* do grupo de *threads* do OpenMP. Como não existem dependências entre elas, e elas podem ser executadas paralelamente, não existe garantia que a tarefa que imprime “Olá” será executada antes que a tarefa que imprime “Mundo”. Isso pode ser verificado executando o programa algumas vezes. Em algumas situações o resultado será “Olá Mundo” enquanto outras “Mundo Olá”. Para garantir a execução de “Olá” antes de “Mundo”, podemos adicionar a diretiva `#pragma omp taskwait` entre a submissão de ambas as tarefas, na linha 11. Isso significa que quando a *thread* que esta submetendo as tarefas chegar no `taskwait` ela irá esperar pela conclusão de todas as tarefas submetidas até então. Isso garante que antes da submissão da tarefa “Mundo” a tarefa “Olá” já acabou. Entretanto, neste caso, não existe paralelismo. A Figura 6.2 apresenta o DAG desta simples aplicação.

Pode-se lançar várias tarefas com argumentos diferentes que são escalonadas pelo OpenMP. No exemplo abaixo, limitamos o número de *threads* para cinco utilizando `num_threads(5)`. Uma *thread* vai executar o `for` submetendo 10 tarefas com argumentos diferentes de zero à nove. A tarefa é simples, ela recebe `i` como argumento, e dorme `i` segundos. A ordem de execução é estocástica, mas o escalonamento será dado pela ordem de submissão. A

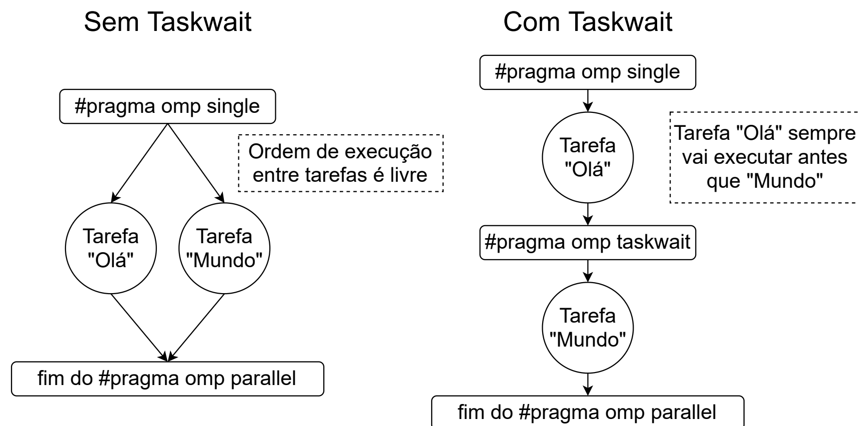


Figura 6.2. DAG para os códigos de Olá Mundo sem e com `taskwait`.

Figura 6.3 mostra uma provável execução do programa abaixo. Esta provável execução leva 13 segundos, onde apenas uma *thread* levou mais tempo que as demais. A situação ideal seria quando a tarefa de tempo 9 fosse executada junto com a tarefa 0, a tarefa 8 com a tarefa 1, a tarefa 7 com a tarefa 2, a tarefa 6 com a tarefa 3, e a tarefa 5 com a tarefa 4. A situação com tempo 13 acontece porque a tarefa mais custosa foi escalonada por último, e não será executada com a tarefa 0. Para tentar melhorar o desempenho pode-se dar dicas ao OpenMP em como escalonar melhor as tarefas utilizando prioridades. Isto é uma responsabilidade do programador, já que o OpenMP não tem como saber previamente o tempo de duração das tarefas ou outras situações.

```
1  void task(int i){
2     sleep(i);
3  }
4  int main(){
5    #pragma omp parallel num_threads(5)
6    {//OpenMP inicia várias threads
7      #pragma omp single
8      {// Este Bloco é executado unicamente por uma thread
9         for(int i = 0; i < 10; i++){
10           #pragma omp task
11           task(i);//Each task will sleep diferently
12        }
13     }
14   }
15 }
```

A utilização de prioridades nas tarefas do OpenMP se dá pela adição da cláusula `priority(valor)` em um `#pragma omp task`, onde o parâmetro valor é a prioridade da tarefa. Tarefas com prioridades maiores serão escalonadas primeiro. As prioridades no OpenMP assumem valores de zero até um valor máximo determinado pela variável de ambiente `OMP_MAX_TASK_PRIORITY`. Entretanto, o padrão desta variável de ambiente é zero. Desta forma, para qualquer utilização de prioridades em um programa com tarefas OpenMP, deve-se definir esta variável de ambiente.

O fragmento de código abaixo apresenta uma primeira variação do código anterior

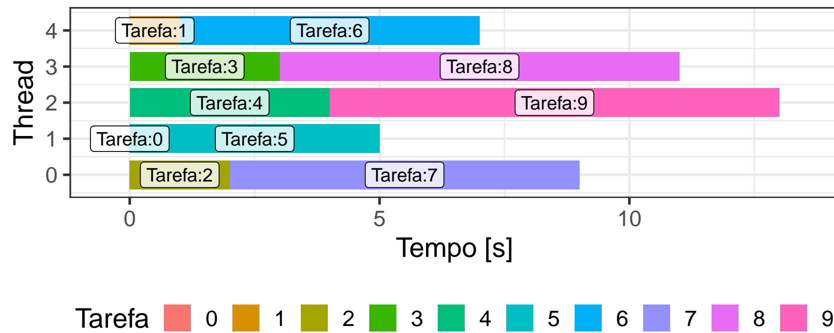


Figura 6.3. Uma provável execução do programa sem prioridades.

adicionando prioridades equivalentes ao tempo de execução das tarefas. Assim, escalonando as tarefas mais custosas primeiro, podemos preencher melhor o tempo. Já que com cinco *threads*, as tarefas 9, 8, 7, 6, 5 seriam escalonadas em um primeiro momento, e quando a primeira tarefa acabar (5), a tarefa 4 seria escalonada para a mesma *thread*. A Figura 6.4 ilustra uma provável execução desta variação. A execução não aconteceu da forma como se imaginava, apesar de reduzir o tempo de execução em um segundo. Isso aconteceu porque o escalonamento das tarefas acontece assim que a tarefa é submetida. Então como as tarefas 0 à 4 são submetidas primeiro, e existem recursos livres, elas são imediatamente escalonadas, quebrando assim a ordem que desejávamos estabelecer. Neste caso, para ficar na ordem desejada pode-se alterar a ordem de submissão ou alterar as prioridades.

```
1    #pragma omp single
2    {// Este Bloco é executado unicamente por uma thread
3       for(int i = 0; i < 10; i++){
4          int prio = i;
5          #pragma omp task priority(prio)
6          task(i);//Each task will sleep diferently
7       }
8    }
```

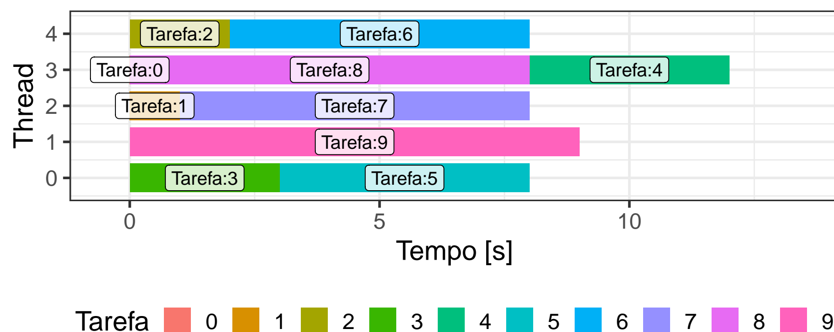


Figura 6.4. Provável execução do programa com prioridades igual ao tempo de execução.

As prioridades que auxiliam o OpenMP para este programa ter o melhor tempo de execução estão no fragmento de código abaixo. Considerando a ordem de submissão, a execução das tarefas 0 à 4 primeiro é inevitável. Desta forma, pode-se aplicar a prioridade máxima para elas, e após isso segue a prioridade do exemplo anterior. Quando a tarefa 0 termina, a tarefa 9 é escalonada para a mesma *thread* e assim por diante. A execução desta versão está na Figura 6.5. Esta versão atinge o menor tempo de execução possível de nove segundos. A Figura 6.6 mostra o DAG desta aplicação com a última versão das prioridades.

```
1    #pragma omp single
2    {// Este Bloco é executado unicamente por uma thread
3       for(int i = 0; i < 10; i++){
4          int prio = i < 5 ? 10 : i;
5          #pragma omp task priority(prio)
6          task(i);//Each task will sleep diferently
7       }
8    }
```

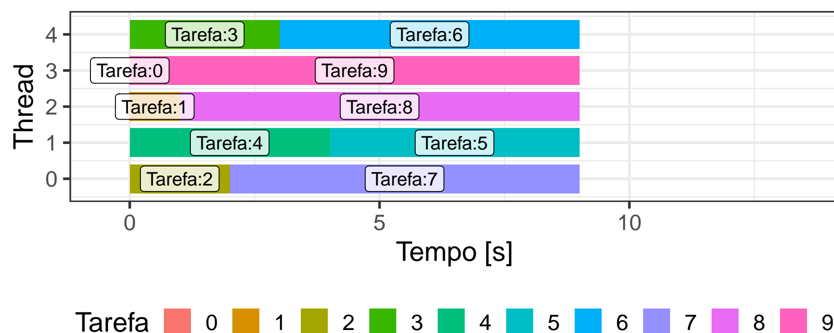


Figura 6.5. Provável execução do programa com novas prioridades.

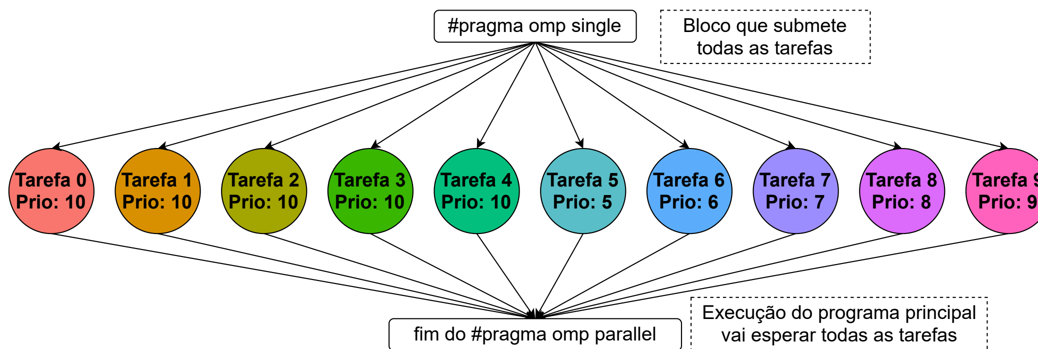


Figura 6.6. DAG da aplicação com 10 tarefas independentes de tempos diferentes.

OpenMP permite lançar tarefas dentro de tarefas. As novas tarefas podem começar sem necessariamente a tarefa que as submeteu terminar. Entretanto, a tarefa que as submete pode esperar por todas as tarefas, chamadas neste contexto de subtarefas, que ela submeteu com `taskwait`. Isto vai gerar a suspensão dela até que todas as subtarefas

estejam terminadas. Deste modo, a tarefa será dividida em etapas que serão executados em momentos distintos. O exemplo do segmento de código abaixo possui uma tarefa (`tarefa_complexa`) que submete outras tarefas (`tarefa_simples`), com e sem `taskwait` dentro da `tarefa_complexa`. O padrão do OpenMP é deixar apenas a *thread* que executou a tarefa antes da suspensão continuar sua execução. Entretanto, podemos utilizar a opção `untied`. Desta maneira, quando uma tarefa que entrou em suspensão voltar a ser executada, qualquer *thread* poderá fazê-lo. Tarefas podem anotar pontos de suspensão com `taskyield`, onde o OpenMP vai decidir se continua a tarefa ou a suspende em favor de outra tarefa (com prioridade maior por exemplo). A Figura 6.7 mostra o DAG do programa, com e sem o `taskwait` da linha 9.

```
1  void tarefa_simples(){
2     sleep(1);
3  }
4  void tarefa_complexa(){
5    #pragma omp task
6    tarefa_simples();
7    #pragma omp task
8    tarefa_simples();
9    //#pragma omp taskwait
10 }
11 int main(){
12   #pragma omp parallel
13   {//OpenMP inicia várias threads
14     #pragma omp single
15     {// Este Bloco é executado unicamente por uma thread
16        #pragma omp task //untied
17        tarefa_complexa();
18        #pragma omp task
19        tarefa_simples();
20     }
21   }
22 }
```

Tarefas podem ter dependências de dados, isto é, uma tarefa pode necessitar de um dado computado por outra tarefa e gerar dados que serão utilizados por outras tarefas. O conjunto de dependências entre tarefas formam a estrutura do DAG e guiam a ordem de execução das tarefas em aplicações mais complexas. Para informar uma dependência de uma variável (de entrada ou saída) em uma tarefa, deve-se utilizar a cláusula `depend(modo: variável)`. Primeiramente informa-se o modo: `in` para variáveis de entrada, `out` para variáveis de saída ou `inout` para ambos os casos. Em seguida temos `:` (dois pontos) e a lista de variáveis. Para vetores, uma das notações possíveis para explicitar a dependência de apenas algumas posições é a notação `vetor[inicio:tamanho]`. A especificação do OpenMP obriga o uso de declarações de posição nos vetores iguais ou disjuntas. Por exemplo, vamos assumir um vetor `A` de tamanho total 4 e uma tarefa que escreve em todas as posições deste vetor, seguido por uma outra tarefa que depende apenas das posições 2 e 3. A notação para a primeira tarefa deverá ser `depend(out: A[0:2], A[2:2])`. A razão disso é que a segunda tarefa deverá ter a notação `depend(in: A[2:2])`. Ou seja, não pode-se utilizar `A[0:4]` na primeira tarefa já que o conjunto de posições não é igual nem disjunto a `A[0:2]`.

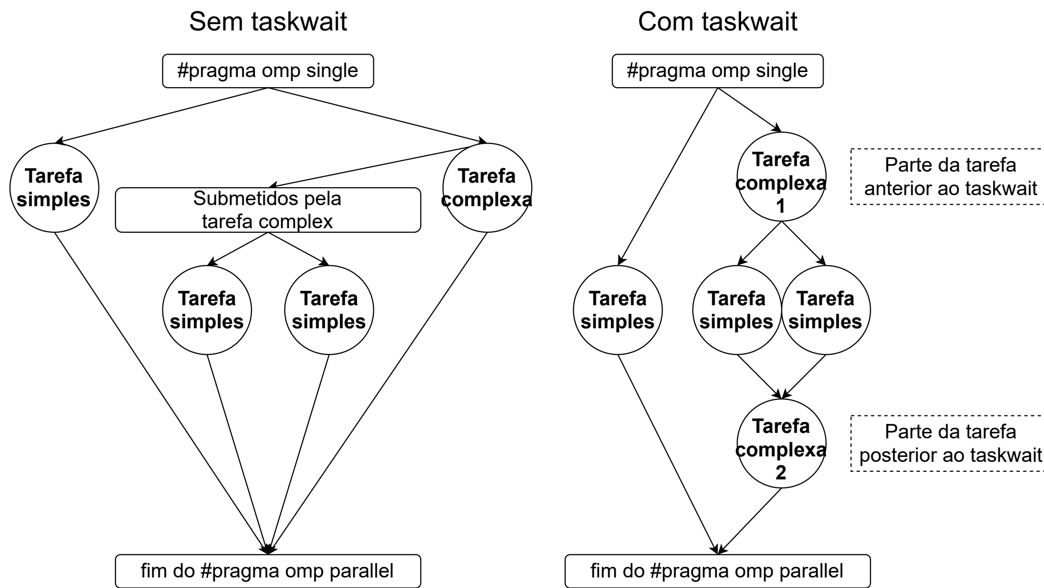


Figura 6.7. DAG da aplicação (tarefa_complexa) sem e com taskwait.

O exemplo do código abaixo emprega tarefas para demonstrar a utilização de dependências para o cálculo da seguinte equação utilizando o vetor `A`:

$$R = A[0] \times A[1] + A[2] \times A[3] \quad (1)$$

Neste caso, sabemos que as operações $A[0] \times A[1]$ e $A[2] \times A[3]$ podem acontecer paralelamente. Define-se cada uma delas como uma tarefa utilizando as variáveis temporárias c e d. Ainda, o exemplo tem uma tarefa de inicialização que define os valores do vetor A. A última tarefa realiza então a operação final $R = c + d$. A tarefa de inicialização tem como dependência todo o vetor. As tarefas de multiplicação tem como dependência duas células do vetor (posições [0,1] e [2,3] respectivamente). Como precisamos definir dependências iguais ou disjuntas, as dependências da tarefa de inicialização são `A[0:2]` e `A[2:2]` como saida (`out`). As dependências da tarefa $c = A[0] \times A[1]$ são `in: A[0:2]` e `out: c`, gerando a dependência com a tarefa anterior. A tarefa $d = A[2] \times A[3]$ tem dependências `in: A[2:2]` e `out: d`, causando novamente a dependência com a tarefa de inicialização. Estas dependências permitem a execução em paralelo das tarefas de multiplicação. Por último, a tarefa $R = c + d$ tem as dependências `in: c, d`, causando a espera de ambas as tarefas de multiplicação. A Figura 6.8 mostra o DAG deste exemplo.

```
1  int main(){
2    int A[4], c=0, d=0, result=0;
3    #pragma omp parallel
4    {
5      #pragma omp single
6      {
7         #pragma omp task depend(inout: A[0:2], A[2:2])
8         {
9            A[0]=1; A[1]=1; A[2]=2; A[3]=3;
```

```
10          }
11          #pragma omp task depend(in: A[0:2]) depend(out: c)
12          {
13             c = A[0] * A[1];
14          }
15          #pragma omp task depend(in: A[2:2]) depend(out: d)
16          {
17             d = A[2] * A[3];
18          }
19          #pragma omp task depend(in: c, d)
20          {
21             result = c + d;
22          }
23       }
24    }
25    printf("Resultado:%d\n", result);
26 }
```

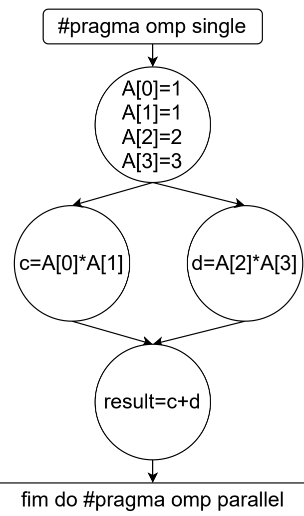


Figura 6.8. DAG da aplicação de exemplo da cláusula depend.

6.3. Exemplos de Aplicações com tarefas OpenMP

Três exemplos são utilizados para ilustrar o emprego das diretivas OpenMP orientadas a tarefas: *Merge Sort*, suavização de Gauss-Seidel e fatoração Cholesky.

6.3.1. Ordenação por mistura (*MergeSort*)

Algoritmos recursivos no modelo divisão e conquista como a ordenação por mistura (*MergeSort*) são diretamente adaptáveis ao modelo baseado em tarefas. As duas chamadas recursivas utilizadas para tratar os subproblemas (linhas 5 e 7 do código abaixo) podem ser transformadas em duas tarefas independentes com a adição das diretivas (`#pragma omp task`). Estas duas tarefas podem ser executadas concorrentemente em qualquer ordem visto que não há dependências de dados entre elas, i.e., ambas acessam partes distintas do vetor.

A adição da diretiva `#pragma omp task` altera o comportamento do código sequencial original (i.e., sem as diretivas OpenMP) pois a execução das funções `merge_sort` das linhas 5 e 7 passa a ser assíncrona. Dessa forma, não há garantias de que as chamadas à `merge_sort` imediatamente anteriores já tenham retornado ao executarmos a chamada à função `merge` na linha 9. Para garantir o correto funcionamento do algoritmo é necessário então impor um ponto de sincronização antes da execução desta função. Esta sincronização é obtida com a diretiva `#pragma omp taskwait` na linha 8, forçando o avanço da execução somente após o fim de todas as tarefas criadas anteriormente.

```
1  void merge_sort(int vetor[], int tam) {
2    int metade = tam / 2;
3    if (tam > 1) {
4      #pragma omp task
5      merge_sort(vetor, metade);
6      #pragma omp task
7      merge_sort(vetor + metade, tam - metade);
8      #pragma omp taskwait
9      merge(vetor, tam);
10   }
11   return;
12 }
```

Ao paralelizar algoritmos recursivos, como o *MergeSort*, é hábito estabelecer limiares para evitar a criação de novas tarefas que tratem de problemas demasiadamente pequenos. No exemplo a seguir, é utilizado o condicionante `if` junto à diretiva `#pragma omp task` para evitar que novas tarefas sejam criadas quando o subproblema for menor ou igual a `MIN_PAR`. Já a adição da cláusula `untied` permite que partes diferentes da uma mesma tarefa sejam executadas por *threads* diferentes. No caso do *MergeSort*, por exemplo, a *thread* 0 poderia executar a tarefa desde início até atingir o `taskwait`, colocando-a em espera. Quando a sincronização fosse concluída (i.e., no final das tarefas criadas anteriormente) a continuação da tarefa e a subsequente chamada a função `merge` poderiam ser executadas pela *thread* 1. Caso a cláusula `untied` seja omitida, o comportamento padrão `tied` é assumido e a continuação da tarefa teria que ser executada na mesma *thread* que a iniciou. Isso pode limitar o paralelismo se tal *thread* estiver ocupada executando outras tarefas. Outras otimizações ainda seriam possíveis neste código, como o emprego de outros algoritmos de ordenação para limitar a recursividade. Entretanto, tais otimizações são independentes do OpenMP e podem ser aplicadas já no algoritmo sequencial, estando portanto fora do escopo deste texto.

```
1  void merge_sort(int vetor[], int tam) {
2    int metade = tam / 2;
3    if (tam > 1) {
4      #pragma omp task if(metade > MIN_PAR) untied
5      merge_sort(vetor, metade);
6      #pragma omp task if(metade > MIN_PAR) untied
7      merge_sort(vetor + metade, tam - metade);
8      #pragma omp taskwait
9      merge(vetor, tam);
10   }
11   return;
12 }
```

6.3.2. Suavização de Gauss-Seidel

Algumas estratégias de se obter desempenho em aplicações consistem em alterar a forma com a qual os laços do programa são percorridos. Técnicas de otimização de laços aninhados [McKinley et al. 1996] transformam os laços a fim de se explorar melhor a hierarquia de cache através da localidade dos dados como nas técnicas de `loop tiling` e `loop interchange`, ou permitem paralelizá-los como a técnica de `loop skewing`. No entanto, a implementação de algumas dessas técnicas, como a de `loop skewing`, não parece óbvia em um primeiro olhar. Também, sua implementação pode ser complexa, reduzindo a legibilidade do código portanto mais sujeita à erros. A programação baseada em tarefas simplifica esse tipo de problema.

O algoritmo de suavização de Gauss-Seidel representa um passo essencial para os métodos `multigrid` [Flannery et al. 1992], suavizando os erros de alta frequência e acelerando a convergência dos resultados. No entanto, tal algoritmo de suavização é considerado de difícil paralelização, pois o cálculo de um valor na posição i,j na iteração k de uma matriz, conforme pode ser visto no código a seguir, depende dos valores previamente calculados para as posições vizinhas à esquerda $(i-1,j)$ e acima $(i,j-1)$ pela iteração k, e dos valores atuais abaixo $(i+1,j)$ e à direita $(i,j+1)$ calculados na iteração $k-1$. A Figura 6.9 ilustra graficamente as dependências para o cálculo de um ponto i,j.

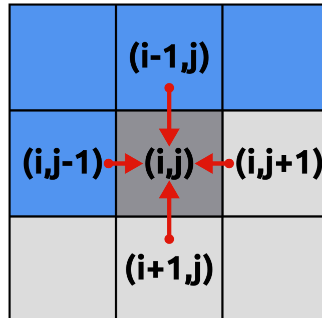


Figura 6.9. Dependências para o cálculo de um ponto i,j na suavização de Gauss-Seidel. Escrevemos em i,j, e lemos dos pontos vizinhos. Os pontos em azul já foram computados pela iteração atual, e os em cinza representam os valores calculados na iteração prévia ou os valores iniciais.

O comportamento da execução sequencial e as suas dependências são ilustradas na Figura 6.10, onde mostramos um exemplo com duas iterações do algoritmo, a primeira em azul e a segunda em amarelo. O percorrimento da matriz por linhas e colunas geram dependências que não permitem uma paralelização direta do algoritmo, tanto dos seus laços internos quanto o laço externo que controla o número de iterações do processo na matriz. Mesmo que algumas dependências sejam liberadas logo no começo, como após a execução da posição 1 que libera a posição 2 e 4, e estas posições liberam a posição 1 da segunda iteração em amarelo, expressar o paralelismo entre iterações parece bastante complexo.

Como mencionado anteriormente, existem algumas técnicas para permitir a exploração do paralelismo em casos específicos como a técnica de `loop skewing`. Usando essa técnica é possível parallelizar o laço interno de atualização de todas as posições de

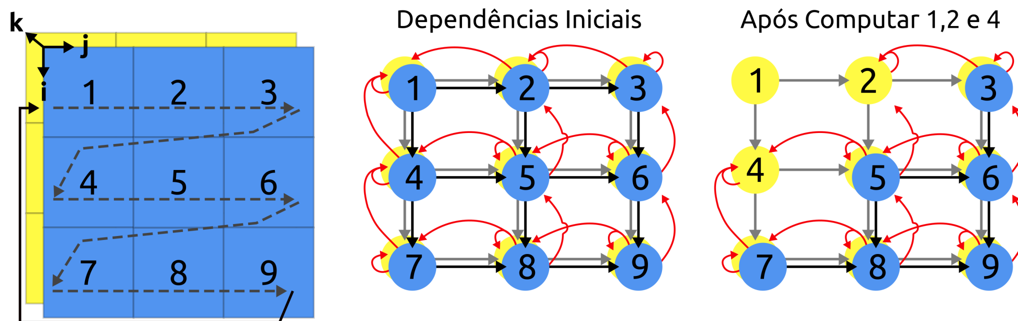


Figura 6.10. Ordem da execução sequencial e as dependências de dados entre as posições da matriz e duas iterações. A primeira iteração é representada em azul e a segunda em amarelo. Dependências entre a primeira e segunda iteração são representadas em vermelho.

uma diagonal percorrendo a matriz nesse sentido. Isso cria um nível de paralelismo dentro das diagonais que pode ser expressado usando diretivas clássicas como o `parallel for`, ilustrado pela Figura 6.11. O nível de paralelismo aumenta e diminui de acordo com a quantidade de elementos de uma diagonal, sendo bem limitado no início e final de cada iteração. Além disso, também é necessária uma sincronização após a computação de cada diagonal, e esta técnica não permite sobrebor as duas iterações (amarela e azul) de uma forma simples, o que também limita o paralelismo. O código para esta técnica é representado a seguir e mostra como pode ser complexo expressar o percorrimento da matriz diagonalmente. Isso torna a a programação mais difícil, menos eficiente, e mais propensa a erros, devido a complexidade das dependências impostas pelo problema.

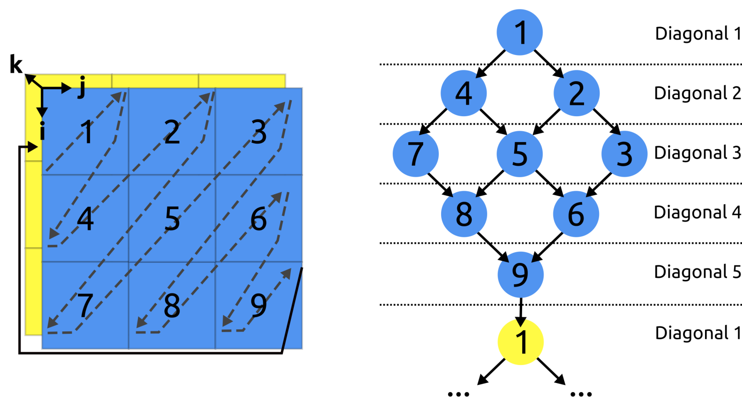


Figura 6.11. Método de Gauss-seidel usando a técnica de *loop skewing*.

```
1  void gauss_seidel_skewed(double **A, double* b, int N, int Niter)
2  {
3    double  h, h2;
4    h = 1.0/(N-1);
5    h2 = h*h;
6
7    for(int k=0; k<Niter; ++k) {
8      // primeiras N diagonais (paralelismo aumentando)
```

```
9      for (int diagonal=1; diagonal <= N; ++diagonal) {
10       #pragma omp parallel for shared(A, b, h2)
11       for (int j=1; j<=diagonal; ++j) {
12         int diag = diagonal+1-j;
13         // Fórmula para suavização de Gauss-Seidel em 2D
14         A[diag][jj] = 0.25 * ( A[diag+1][j] + // vizinho abaixo
15                                A[diag-1][j] + // vizinho acima
16                                A[diag][j+1] + // vizinho direita
17                                A[diag][j-1] - // vizinho esquerda
18                                h2*b[diag]
19                              );
20       } // sincronização entre diagonais
21     }
22
23     // resto das diagonais (paralelismo diminuindo)
24     for (int diagonal=1; diagonal <= N-1; ++diagonal) {
25       #pragma omp parallel for shared(A, b, h2)
26       for (int j=1; j<=N-diagonal; ++j) {
27         int jj = diagonal+j;
28         int diag = N+1-j;
29         // Fórmula para suavização de Gauss-Seidel em 2D
30         A[diag][jj] = 0.25 * ( A[diag+1][jj] + // vizinho abaixo
31                                A[diag-1][jj] + // vizinho acima
32                                A[diag][jj+1] + // vizinho direita
33                                A[diag][jj-1] - // vizinho esquerda
34                                h2*b[diag]
35                              );
36       } // sincronização entre diagonais
37     }
38   }
39 }
```

Dada a maior complexidade de se paralelizar este problema usando técnicas clássicas como a paralelização direta de laços, mostraremos como a programação baseada em tarefas pode ser bastante útil tanto em termos de expressividade do paralelismo, quanto desempenho comparado à solução anterior. O código abaixo já utiliza a otimização de `loop tiling`, percorrendo a matriz em blocos menores no mesmo sentido pela Figura 6.10. Para termos a execução baseada em tarefas, usamos exatamente o mesmo código da versão sequential com as seguintes modificações: 1) adiciona-se as diretivas das linhas 7 e 9 para termos uma região paralela executada por uma única *thread* (aquela que submeterá as tarefas); e 2) define-se a criação de uma tarefa por bloco da matriz usando a construção de tarefas do OpenMP nas linhas 15 a 19, especificando as dependências das posições vizinhas com a cláusula `depend`. A complexidade das dependências do problema é totalmente gerenciada pelo sistema de runtime. Isto simplificou muito a programação neste caso pois a paralelização é feita com uma mínima alteração no programa original, e ainda nos permite explorar o paralelismo em um nível maior, incluindo a execução de múltiplas iterações em paralelo.

```
1 void gauss_seidel_blocos(double **A, double* b, int N, int BS, int
     Niter) {
2   int NB = N / TS; // Número de blocos a partir do Block Size (BS)
3   double  h, h2;
4   h = 1.0/(N-1);
```

```
5     h2 = h*h;
6
7     #pragma omp parallel
8     {
9       #pragma omp single
10      {
11        for(int k=0; k<Niter; ++k) { // iterações de Gauss-Seidel
12          for (int ii=1; ii<N-BS; ii+=BS) { // laços sobre os blocos
13            for (int jj=1; jj<N-BS; jj+=BS) {
14              // cria uma tarefa para cada bloco da matriz
15              #pragma omp task depend(out: A[ii:BS][jj:BS]) depend(in: A[
      ii+BS:1][jj:BS], A[ii-1:BS][jj:BS], A[ii:BS][jj-1:BS], A[ii:BS][jj+
      BS:1]) firstprivate(ii, jj)
16              {
17                for(int i=ii; i<ii+BS; ++i) { // laços sobre um bloco
18                  for(int j=jj; j<jj+BS; ++j) {
19                    // Fórmula para suavização de Gauss-Seidel em 2D
20                    A[i][j] = 0.25 * ( A[i+1][j] + // vizinho abaixo
21                                       A[i-1][j] + // vizinho acima
22                                       A[i][j+1] + // vizinho direita
23                                       A[i][j-1] - // vizinho esquerda
24                                       h2*b[i]
25                                     );
26                  }
27                }
28              }
29            }
30          }
31        }
32      }
33    }
34  }
```

6.3.3. Fatoração de Cholesky

Algoritmos para fatoração de matrizes permitem decompor uma matriz no produto de duas outras matrizes. A fatoração serve de base para outros algoritmos, auxiliando, por exemplo, na resolução de sistemas de equações lineares. Nesta seção, abordamos o algoritmo de Cholesky para fatorar uma matriz simétrica positiva-definida A em uma matriz triangular L e sua transposta L^T ($A = LL^T$).

Dentre as várias maneiras possíveis de se implementar esta fatoração, escolhemos aqui um algoritmo em blocos [Buttari et al. 2009, Agullo et al. 2010]. Tal algoritmo tem por base quatro operações do padrão BLAS: `potrf`, `trsm`, `syrk` e `gemm`. Estas quatro operações são aplicadas em blocos (submatrizes) da matriz original. Para paralelizar esta aplicação, criaremos uma nova tarefa para executar cada uma das operações BLAS citadas anteriormente, conforme ilustrado nas linhas 13, 17, 23, 29 do código abaixo. Note que utilizamos a anotação `firstprivate` para os ponteiros `Akk`, `Aik`, `Aii`, `Aij` e `Ajk`, uma vez que cada tarefa trabalha sobre blocos diferentes da matriz. Cada operação acessa os blocos da matriz em modos de leitura, escrita ou leitura/escrita. A operação `gemm`, por exemplo, acessa três blocos A, B, C. Os blocos A e B são acessados em modo leitura enquanto o bloco C é acessado em modo leitura/escrita. Por meio da diretiva `depend`

podemos indicar como cada tarefa acessa cada bloco da matriz. Utilizando a sequência de desenrolamento dos laços e o modo de acesso em cada bloco da matriz, é possível construir um grafo de dependências (DAG). A Figura 6.12 ilustra um DAG para esta aplicação quando a matriz A é particionada em quatro blocos por linha/coluna.

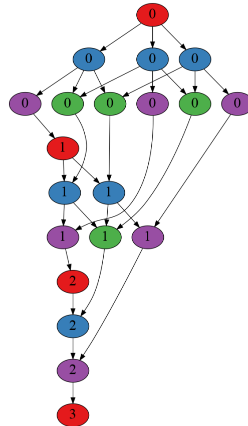


Figura 6.12. DAG da fatoração de cholesky por blocos quando número de blocos por linha/coluna (N) é 4. As cores dos nós representam o tipo da tarefa. Vermelho indica tarefas `potrf`, azul tarefas `trsm`, roxo tarefas `syrk` e verde tarefas `gemm`.

```
1  void cholesky(double *A, int OrdemMatriz, int OrdemBloco){
2    int i, j, k;
3    int NumBlocos = OrdemMatriz / OrdemBloco;
4    int TamBloco = OrdemBloco*OrdemBloco;
5    double *Akk, *Aik, *Aii, *Aij, *Ajk;
6
7    #pragma omp parallel
8    {
9      #pragma omp single
10     {
11       for(k = 0; k < NumBlocos; k++) {
12         Akk = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, k, k);
13         #pragma omp task depend(inout: Akk[0:TamBloco]) firstprivate(
     Akk)
14         potrf(Akk, OrdemBloco);
15         for(i = k+1; i < NumBlocos; i++) {
16           Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
17           #pragma omp task depend(in: Akk[0:TamBloco]) depend(inout:
     Aik[0:TamBloco]) firstprivate(Akk, Aik)
18           trsm(Akk, Aik, OrdemBloco);
19         }
20         for(i = k+1; i < NumBlocos; i++) {
21           Aii = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, i);
22           Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
23           #pragma omp task depend(in: Aik[0:TamBloco]) depend(inout:
     Aii[0:TamBloco]) firstprivate(Aii, Aik)
24           syrk(Aik, Aii, OrdemBloco);
25           for(j = k+1; j < i; j++) {
26             Aij = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, j);
27             Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
```

```
28            Ajk = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, j, k);
29            #pragma omp task depend(in: Aik[0:TamBloco]) depend(in: Ajk
    [0:TamBloco]) depend(inout: Aij[0:TamBloco]) firstprivate(Aij, Aik,
     Ajk)
30            gemm(Aik, Ajk, Aij, OrdemBloco);
31          }
32        }
33      }
34    }
35  }
36 }
```

Analisando o grafo da Figura 6.12, percebemos que alguns tipos de tarefas mais arestas de dependências que outros. O número de arestas que saem de um dado nó indica quantas outras tarefas dependem da execução desta. Tarefas do tipo `syrk` (em roxo) ou `gemm` (em verde) sempre liberam a execução de apenas uma outra tarefa enquanto tarefas do tipo `potrf` (em vermelho) liberam $(NumBlocos - k - 1)$ tarefas `trsm` (em azul), onde *NumBlocos* é o número de blocos por linha/coluna da matriz original e k é iteração do laço mais externo do algoritmo por blocos. A execução de cada uma das tarefas `trsm` por sua vez, habilita a execução de até $(NumBlocos - k - 1)$ tarefas dos tipos `syrk` e `gemm`. Logo, se priorizarmos a execução das tarefas `potrf` sobre tarefas `gemm` ou `syrk` habilitaremos mais rapidamente um maior número de tarefas, expondo maior nível de paralelismo ao escalonador.

Na Seção 6.2, nós vimos que prioridades é um bom meio para repassar dicas ao *runtime* do OpenMP sobre quais tarefas escalonar primeiro. No código abaixo, adicionamos prioridades a cada uma das tarefas do algoritmo seguindo a lógica discutida acima. A fórmula utilizada no cálculo destes valores de prioridade para cada tipo de tarefa é livremente inspirada naquela utilizado pela biblioteca de álgebra linear Chameleon [Agullo et al. 2010].

```
1  void cholesky(double *A, int OrdemMatriz, int OrdemBloco){
2    int i, j, k, prio;
3    int NumBlocos = OrdemMatriz / OrdemBloco, TamBloco = OrdemBloco*
      OrdemBloco;
4    double *Akk, *Aik, *Aii, *Aij, *Ajk;
5
6    #pragma omp parallel
7    {
8      #pragma omp single
9      {
10       for(k = 0; k < NumBlocos; k++) {
11         Akk = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, k, k);
12         prio = MAX_PRIO - 2*NumBlocos - 2*k;
13         #pragma omp task depend(inout: Akk[0:TamBloco]) firstprivate(
    Akk) priority(prio)
14         potrf(Akk, OrdemBloco);
15         for(i = k+1; i < NumBlocos; i++) {
16           Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
17           prio = MAX_PRIO -  2*NumBlocos - 2*k - i;
18           #pragma omp task depend(in: Akk[0:TamBloco]) depend(inout:
    Aik[0:TamBloco]) firstprivate(Akk, Aik) priority(prio)
19           trsm(Akk, Aik, OrdemBloco);
```

```
20          }
21          for(i = k+1; i < NumBlocos; i++) {
22            Aii = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, i);
23            Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
24            prio = MAX_PRIO - 2*NumBlocos - 2*k - i;
25            #pragma omp task depend(in: Aik[0:TamBloco]) depend(inout:
       Aii[0:TamBloco]) firstprivate(Aii, Aik) priority(prio)
26            syrk(Aik, Aii, OrdemBloco);
27            for(j = k+1; j < i; j++) {
28              Aij = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, j);
29              Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
30              Ajk = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, j, k);
31              prio = MAX_PRIO - 2*NumBlocos - 2*k - i - j;
32              #pragma omp task depend(in: Aik[0:TamBloco]) depend(in: Ajk
       [0:TamBloco]) depend(inout: Aij[0:TamBloco]) firstprivate(Aij, Aik,
        Ajk) priority(prio)
33              gemm(Aik, Ajk, Aij, OrdemBloco);
34            }
35          }
36        }
37      }
38    }
39  }
```

6.4. Análise de Desempenho de Tarefas OpenMP

O desenvolvimento de aplicações paralelas está ligado ao constante uso de técnicas de análise de desempenho. Nesse sentido, uma das funcionalidades associada às especificações do OpenMP é a chamada *OpenMP Tools Interface* (OMPT). Estas especificações atualmente ainda não são suportadas pelo *runtime* OpenMP do compilador GCC `libgomp`, mas é suportada por outros compiladores tal como o `LLVM/clang` com o runtime `libomp`. OMPT objetiva especificar uma interface de *callbacks* para rastrear o comportamento de aplicações OpenMP. A vantagem dessa API é que ela roda no mesmo espaço de endereçamento que o *runtime* OpenMP, facilitando o acesso a diversas informações sobre o estado do *runtime*.

Os *callbacks* permitem acessar dados de uma *thread* tais como seu estado atual (ex: *Idle*, *Work*, *Barrier wait*, *Overhead*, etc). Estes *callbacks* também nos permitem interceptar eventos, algo útil para a análise de programas baseados em tarefas. Destacamos dois: a criação de tarefas (`ompt_callback_task_create`) e ações de escalonamento sobre as tarefas (`ompt_callback_task_schedule`), definidos no código abaixo. Ambos são associados aos eventos correspondentes na função `ompt_initialize` de inicialização. Além disso, nossa ferramenta deve implementar o método `ompt_start_tool` que realizará a sua inicialização. Com a implementação dos *callbacks* podemos especificar o que a nossa ferramenta fará quando uma tarefa for criada ou um ponto de escalonamento for atingido.

O código abaixo mostra a definição do comportamento da ferramenta para o *callback* do ponto de escalonamento. A variável `prior_task_status` define o estado da tarefa `first_task_data` que chegou no ponto de escalonamento. O último parâmetro, `second_task_data`, contém dados da tarefa que irá assumir a execução após o

ponto de escalonamento. Neste exemplo, apenas escrevemos na tela o identificador da *thread* que estava executando aquela tarefa, o identificador da tarefa, o tipo de evento que ocorreu e uma marca de tempo de quando o evento ocorreu. A ferramenta é finalizada quando a função `ompt_finalize` é chamada.

```
#include <stdio.h>
#include <ompt.h>
#include <omp.h>
double INIT_TIME;
unsigned int TASK_ID;
// Define o comportamento para este callback
static void on_ompt_callback_task_schedule(ompt_data_t *first_task_data
    , ompt_task_status_t prior_task_status, ompt_data_t *
    second_task_data){
  switch (prior_task_status) {
    case ompt_task_complete:
      printf("%d %ld tarefa_terminou %lf\n", omp_get_thread_num(),
    first_task_data->value, omp_get_wtime()-INIT_TIME);
      break;
    case ompt_task_switch: // task second_task_data começou
      printf("%d %ld tarefa_iniciou %lf\n", omp_get_thread_num(),
    second_task_data->value, omp_get_wtime()-INIT_TIME);
      break;
    default:
      break;
  }
}
static void on_ompt_callback_task_create( ompt_data_t *parent_task_data
    ,
      const ompt_frame_t *parent_frame, ompt_data_t* new_task_data,
      int flag, int has_dependences, const void *codeptr_ra)
{
  new_task_data->value = TASK_ID++;
  printf("%d %ld tarefa_criada %lf\n", omp_get_thread_num(),
    new_task_data->value, omp_get_wtime()-INIT_TIME);
}
int ompt_initialize(ompt_function_lookup_t lookup, ompt_data_t* data) {
  printf("%d 0 OMPT_LIB_INICIO %lf\n", omp_get_thread_num(),
    omp_get_wtime()-INIT_TIME);
  // Registra os callbacks que queremos usar
  ompt_set_callback_t ompt_set_callback = (ompt_set_callback_t) lookup(
    "ompt_set_callback");
  ompt_set_callback(ompt_callback_task_schedule, (ompt_callback_t)
    on_ompt_callback_task_schedule);
  ompt_set_callback(ompt_callback_task_create, (ompt_callback_t)
    on_ompt_callback_task_create);
  return 1; //successo
}
void ompt_finalize(ompt_data_t* data) {
  printf("%d 0 OMPT_LIB_FIM %lf\n", omp_get_thread_num(), omp_get_wtime
    ()-INIT_TIME);
}
ompt_start_tool_result_t* ompt_start_tool( unsigned int omp_version,
    const char *runtime_version) {
  INIT_TIME = omp_get_wtime();
```

```
  TASK_ID = 0;
  static ompt_start_tool_result_t ompt_start_tool_result = {&
   ompt_initialize, &ompt_finalize};
  return &ompt_start_tool_result;
}
```

A ferramenta de rastreamento acima pode ser compilada separadamente, criando uma biblioteca que podemos linkar a qualquer programa OpenMP que desejarmos rastrear como exemplificado no processo de compilação a seguir, ou também pode ser adicionada diretamente ao programa. Caso queiramos compilar o código usando o `gcc` devemos especificar o caminho para um *runtime* que suporte as especificações do OMPT, ou podemos simplemente compilar a aplicação usando o compilador `clang`, que usa o runtime `libomp`.

```
# criamos nossa biblioteca para rastrear códigos OpenMP usando OMPT
clang-10 -fopenmp minha_ferramenta_ompt.c -shared -fPIC -o libompt.so
# compilamos a nossa aplicação OpenMP
clang-10 -o GaussSeidel Gauss-Seidel.c -fopenmp -O3
# especificamos a variavél de ambiente OMP_TOOL_LIBRARIES com a nossa
   biblioteca para gerar o rastro da execução
env OMP_TOOL_LIBRARIES=$(pwd)/libompt.so ./GaussSeidel < input
```

Com isto, podemos gerar rastros da execução de diferentes aplicações, o que enriquece o processo de análise de desempenho, permitindo compreender melhor o comportamento de uma aplicação. Como exemplo, a Figura 6.13 representa a execução para a aplicação da suavização de Gauss-Seidel apresentada anteriormente. Podemos ver como o paralelismo de tarefas permite explorar bem os recursos mesmo em um cenário onde temos um problema com dependências bastante complexas.

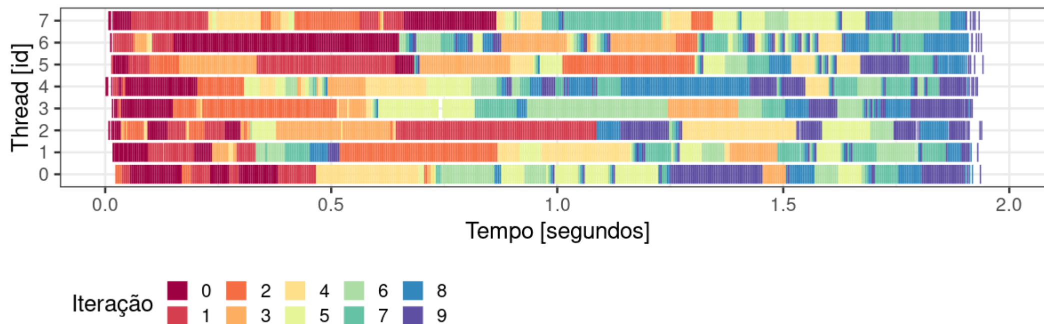


Figura 6.13. Rastro da execução paralela de diferentes iterações para o método de Gauss Seidel usando o paralelismo em tarefas.

6.5. Conclusão

Este minicurso abordou os conceitos básicos fundamentais para se construir programas paralelos com diretivas de programação utilizando tarefas OpenMP. O enfoque no grafo de tarefas (DAG) permite o estabelecimento de dependência finas entre as tarefas permitindo a construção de programas cujo grão de paralelismo pode facilmente ser explorado

para criar uma grande quantidade de tarefas capazes de ocupar todas as *threads* de execução. Esperamos que com os exemplos fornecidos os leitores possam incorporar este paradigma de programação, claramente mais fácil tendo em vista que uma boa parte do trabalho (balanceamento de carga, escalonamento) é relegado ao ambiente de execução. Os conceitos apresentados aqui já permitem começar a programação com tarefas. Referenciamos o leitor à especificação OpenMP [OpenMP 2020], que porta uma enorme quantidade de diretivas auxiliares para tratar casos mais específicos.

6.6. 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) com a bolsa 141971/2020-7 para o primeiro autor, 131347/2019-5 para o segundo autor, 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 et al. 2010] Agullo, E. et al. (2010). Faster, cheaper, better – a hybridization methodology to develop linear algebra software for GPUs. In mei W. Hwu, W., editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann. páginas

[Augonnet et al. 2011] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Conc. and Comp.: Pract. and Exp.*, 23(2). páginas

[Bosilca et al. 2012] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. (2012). DAGuE: A generic distributed {DAG} engine for high performance computing. *Parallel Computing*, 38(1–2):37 – 51. Extensions for Next-Generation Parallel Programming Models. páginas

[Buttari et al. 2009] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53. páginas

[Duran et al. 2011] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). OmpSs: a proposal for programming heterogeneous multi-core architectures. *Par. Proc. Letters*, 21(02). páginas

[Flannery et al. 1992] Flannery, B. P., Press, W. H., Teukolsky, S. A., and Vetterling, W. (1992). Numerical recipes in c. *Press Syndicate of the University of Cambridge, New York*, 24(78):36. páginas

[McKinley et al. 1996] McKinley, K. S., Carr, S., and Tseng, C.-W. (1996). Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453. páginas

[OpenMP 2020] OpenMP (2020). OpenMP application program interface version 5.1. Disponível em: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf. páginas

[Rico et al. 2019] Rico, A., Sánchez Barrera, I., Joao, J. A., Randall, J., Casas, M., and Moretó, M. (2019). On the benefits of tasking with openmp. In Fan, X., de Supinski, B. R., Sinnen, O., and Giacaman, N., editors, *OpenMP: Conquering the Full Hardware Spectrum*, pages 217–230, Cham. Springer International Publishing. páginas