Capítulo

3

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

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

Claudio Schepke
Universidade Federal do Pampa
Alegrete, Brasil

Resumo

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

3.1. Introdução

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

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

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

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

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

3.2. Arquiteturas paralelas

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

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

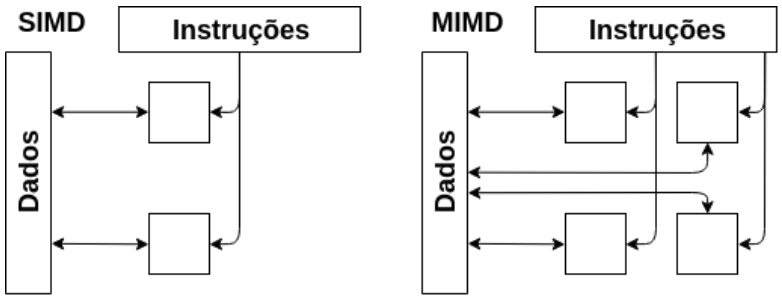
Nesse sentido, a computação de alto desempenho tem sido responsável por uma revolução científica. A evolução das arquiteturas de computadores melhorou o poder computacional, aumentando a gama de problemas e a qualidade das soluções que poderiam ser resolvidas no tempo requerido como, por exemplo, a previsão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

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

3.2.1. Classificação de arquiteturas paralelas

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

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



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

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

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

3.2.2. Arquiteturas multicore e manycore

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

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

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

3.3. Modelagem de aplicações paralelas

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

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

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

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

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

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

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

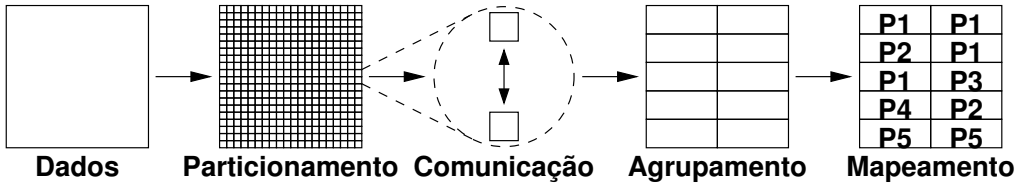


Figura 3.2. Principais etapas na paralelização de um algoritmo.

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

3.4. Programação paralela e vetorial em OpenMP

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

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

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

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

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

3.4.1. Programando com OpenMP

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

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

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

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

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

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

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

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

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

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

3.4.2. Modelo de execução

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

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

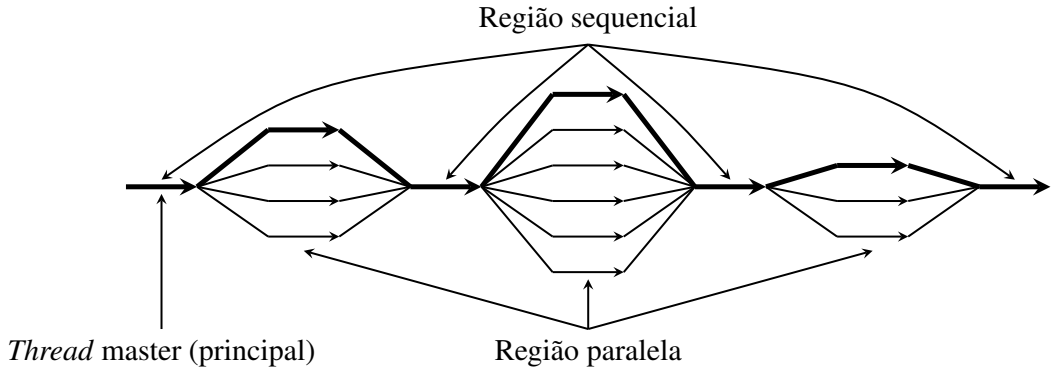


Figura 3.4. Modelo de execução *fork/join* do OpenMP.

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

3.4.3. Métodos de biblioteca

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

```
void omp_set_num_threads(int N)
```

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

```
int omp_get_num_threads()
```

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

```
int omp_get_thread_num()
```

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

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

3.4.4. Cláusulas de dados

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

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

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

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

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

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

3.4.5. Laços paralelos

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

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

3.4.6. Exclusão mútua

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

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

```
1  long long int sum(int *v, long long int N){
2    long long int i, sum_local, sum = 0;
3
4    #pragma omp parallel private(i, sum_local)
5    {
6      sum_local = 0;
7
8      #pragma omp for
9      for(i = 0; i < N; i++)
10       sum_local += v[i];
11
12     #pragma omp atomic
13     sum += sum_local;
14   }
15   return sum;
16 }
```

Figura 3.5. Laço paralelo com OpenMP.

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

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

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

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

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

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

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

3.4.7. Redução

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

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

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

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

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

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

3.4.8. Vetorização

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

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

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

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

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

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

3.5. Avaliação de desempenho com contadores de *hardware*

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

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

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

3.5.1. Linux perf

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

Tabela 3.2. Alguns eventos medidos pelo Perf

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

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

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

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

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

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

```
 Performance counter stats for mult 2048:

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

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

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

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

3.5.2. Performance application programming interface (PAPI)

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

Tabela 3.3. Alguns eventos medidos pelo PAPI

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

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

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

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

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

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

Figura 3.8. Exemplo de como medir desempenho utilizando PAPI.

3.6. Estudo de caso: aplicação geofísica

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

3.6.1. Modelagem Fletcher

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

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

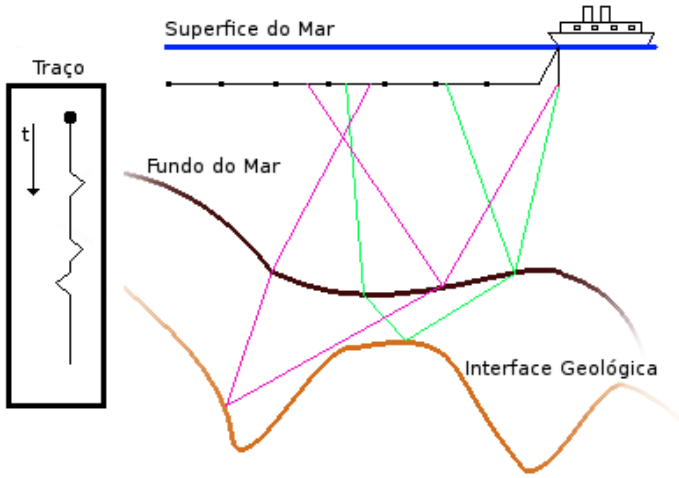


Figura 3.9. Coleta de dados em levantamento sísmico marítimo.

3.6.2. Avaliação e otimização de desempenho

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

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

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

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

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

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

Tabela 3.4. Taxa de Acerto da Aplicação Geofísica

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

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

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

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

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

Tabela 3.5. Desempenho da Aplicação Geofísica

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

3.7. Conclusão

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

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

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

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

Exercícios e soluções deste capítulo estão disponíveis em:

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

Referências

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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