Capítulo 1

Programação com Aceleradores Vetoriais

Matheus da Silva Serpa
msserpa@inf.ufrgs.br
Grupo de Processamento Paralelo e Distribuído (GPPD)
Universidade Federal do Rio Grande do Sul (UFRGS)
Sala 201, Av. Bento Gonçalves, 9500 - Campus do Vale
91.501-970 - Porto Alegre - RS - Brasil

Philippe Olivier Alexandre Navaux
navaux@inf.ufrgs.br
Grupo de Processamento Paralelo e Distribuído (GPPD)
Universidade Federal do Rio Grande do Sul (UFRGS)
Sala 210, Av. Bento Gonçalves, 9500 - Campus do Vale
91.501-970 - Porto Alegre - RS - Brasil

Jairo Panetta
jairo.panetta@gmail.com
Divisão de Ciência da Computação (IEC)
Instituto Tecnológico de Aeronáutica (ITA)
Sala 109, Praça Marechal Eduardo Gomes, 50 - Vila das Acácias
12228-900 - São José dos Campos - SP - Brasil

Resumo

Tradicionalmente, o aumento de desempenho das aplicações se dava de forma transparente aos programadores através do aumento do paralelismo a nível de instruções e do aumento de frequência dos processadores. Entretanto, esse modelo não se sustenta mais. Para se ganhar desempenho nas arquiteturas modernas, são necessários conhecimentos sobre programação paralela e vetorial. Ambos paradigmas são tratados de forma lateral em cursos de computação, sendo que muitas vezes nem são abordados. Nesse contexto, este minicurso objetiva propiciar um maior entendimento sobre os paradigmas de programação paralela e vetorial, de forma que os participantes aprendam a otimizar adequadamente suas aplicações para arquiteturas modernas. Como plataforma experimental, será utilizado o processador vetorial NEC SX-Aurora TSUBASA. Será enfatizada a importância do processamento vetorial e de matrizes, presente em várias aplicações, tais como de petróleo e na área de previsões climáticas.

1.1. 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, a elaboração e execução paralelos e vetoriais. Neste sentido, pretende-se inicialmente introduzir as arquiteturas de *hardware* que existem atualmente tais como a arquitetura vetorial NEC SX-Aurora. 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.

A estrutura do capítulo é dividida em 4 partes. Inicialmente, na Seção 1.2, será apresentada uma introdução sobre arquiteturas paralelas e vetoriais, focando na arquitetura vetorial NEC SX-Aurora. A Seção 1.3 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. Por fim, a Seção 1.4, apresenta estudos de casos, onde aplicações sequencias são aceleradas na arquitetura vetorial NEC utilizando técnicas de programação paralela e vetorial.

1.2. Arquitetura Paralelas

Algumas das primeiras perguntas a serem feitas em relação ao desenvolvimento de aplicações é: por que estudar programação paralela e vetorial?, 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 e vetorial 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. E em cada um desses núcleos, unidades vetoriais capazes de executar diversas operações por ciclo. 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).

Vários desafios devem ser levados em consideração para atingir o alto desempenho nessas arquiteturas. Um dos aspectos mais importantes é o comportamento do subsistema de memória, já que o acesso à memória desempenha um papel fundamental no desempenho (DIENER et al., 2015; SERPA et al., 2019). O consumo de energia e a eficiência energética também são pontos a considerar (SUBRAMANIAM et al., 2013; CASTRO et al., 2016; SOUZA et al., 2017).

1.2.1. Arquitetura Vetorial: NEC SX-Aurora TSUBASA

Uma das arquiteturas modernas projetada para lidar com esses problemas matemáticos, científicos ou de engenharia complexos é a arquitetura paralela e vetorial SX-Aurora TSUBASA da NEC. Os sistemas de computação proprietários da NEC fornecem soluções completas para todos os tipos de requisitos, combinando com uma variedade de produtos baseados em x86 e dispositivos de armazenamento.

A NEC SX-Aurora Vector Engine (VE) explora a técnica de computação vetorial

comprovamente eficiente pela longa história de supercomputação da NEC, na qual uma aplicação completa é executado em VEs de alto desempenho, e apenas as tarefas do sistema são realizadas pelo Vector Host (VH), um servidor x86 padrão. Com o mecanismo de computação vetorial, grande largura de banda de memória e um pequeno número de núcleos poderosos, a arquitetura oferece uma base sólida para um alto desempenho.

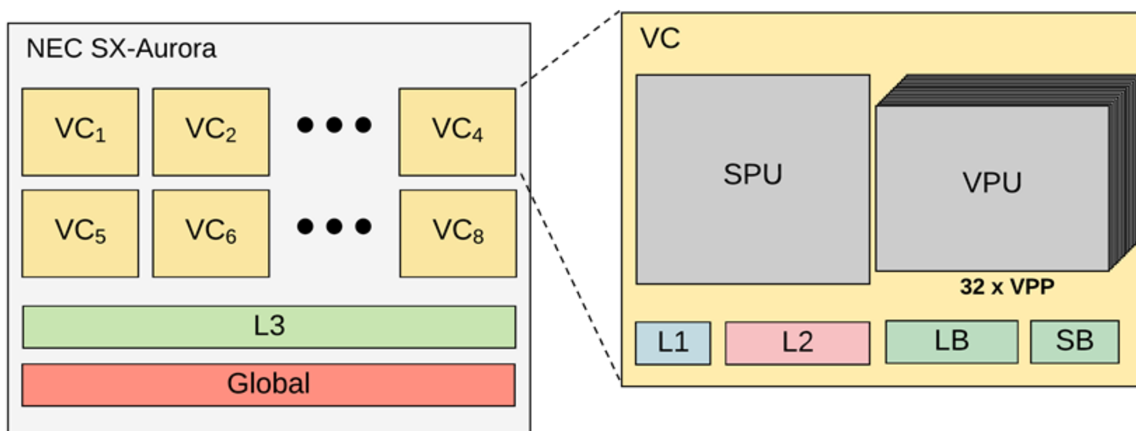

Figura 1.1. Arquitetura de um processador vetorial NEC SX-Aurora.

Os experimentos apresentados nesse capítulo e os realizados no dia do minicurso utilizaram o ambiente NEC SX-Aurora TSUBASA da infraestrutura PCAD, <http://gppd-hpc.inf.ufrgs.br>, no INF/UFRGS. Na Figura 1.1, temos uma VE com 8 *Vector Cores*, memória global de 48 GB. Em cada um dos *Vector Cores*, temos memórias *cache* L1 e L2, uma unidade de processamento escalar (SPU) e uma unidade de processamento vetorial (VPU), sendo que cada VPU contém *load buffer* (LB), *store buffer* (SB). Por fim, dentro das VPUs, temos um *pipeline* paralelo vetorial (VPP) de tamanho 32 (NEC, 2020).

1.3. Programação Paralela e Vetorial em OpenMP

Neste capítulo, optou-se por utilizar OpenMP (*Open Multi-Processing*) (OPENMP, 2008; CHAPMAN; JOST; PAS, 2008), focando em memória compartilhada. 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. 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.

1.3.1. 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.

```
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("%d of %d - Hello world!\n",
13                  myid, nthreads);
14    }
15    return 0;
16 }
```

Figura 1.2. Exemplo de um *Hello world* em OpenMP.

O primeiro exemplo é um *Olá Mundo*. A Figura 1.2 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 `ncc` (*NEC C Compiler*) necessita da opção `-fopenmp`, como no exemplo abaixo:

```
$ /opt/nec/ve/bin/ncc -O4 -finline-functions -proginf
   -report-all -fopenmp -o hello hello.c
```

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 NEC de 8 *vector cores*, a execução será:

```
$ ./hello
0 of 8 - hello world!
1 of 8 - hello world!
2 of 8 - hello world!
3 of 8 - hello world!
4 of 8 - hello world!
5 of 8 - hello world!
6 of 8 - hello world!
7 of 8 - hello world!
```

Na linha de comando, pode-se alterar o número de *threads* com a variável de ambiente `OMP_NUM_THREADS`. Por exemplo, com 12 *threads*:

```
$ OMP_NUM_THREADS=12 ./hello
0  of 12 - hello world!
1  of 12 - hello world!
2  of 12 - hello world!
3  of 12 - hello world!
4  of 12 - hello world!
5  of 12 - hello world!
6  of 12 - hello world!
7  of 12 - hello world!
8  of 12 - hello world!
9  of 12 - hello world!
10 of 12 - hello world!
11 of 12 - hello world!
```

Por fim, em uma máquina com múltiplos processadores vetoriais da NEC, como é o caso do ambiente que estamos utilizando, é possível definir em qual das VEs a aplicação irá executar. Por exemplo, para executar na VE 1:

```
$ export VE_NODE_NUMBER=1
```

1.3.2. 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.

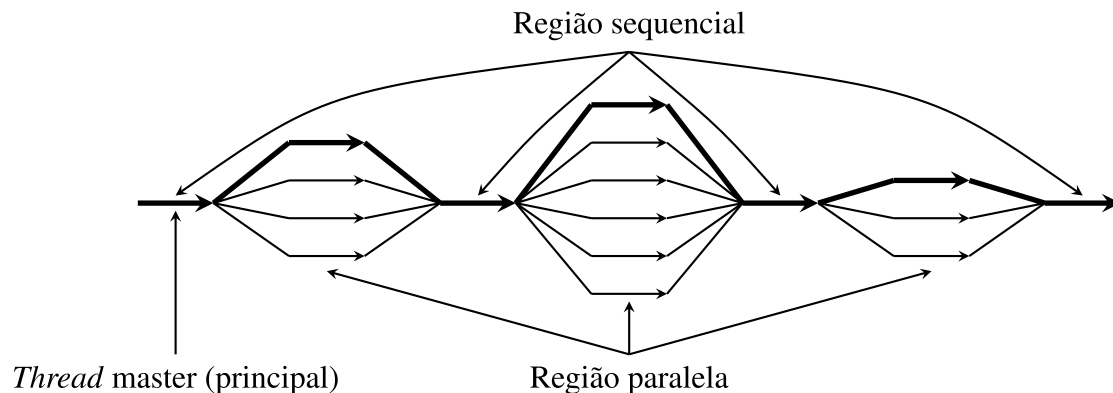

Figura 1.3. Modelo de execução *fork/join* do OpenMP.

A Figura 1.3 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 1.3) existem três regiões paralelas com quatro, seis e três *threads*, respectivamente.

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.

1.3.3. 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 1.2.

1.3.4. 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.

1.3.5. 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**.

```
1 long long int sum(int *v, long long int N){
2   long long int i, sum = 0;
3
4   for(i = 0; i < N; i++)
5     sum += v[i];
6
7   return sum;
8 }
```

Figura 1.4. Exemplo do cálculo de soma de vetor sequencial

A Figura 1.4, apresenta uma soma de todos elementos de um vetor, função implementada de forma sequencial, a qual será utilizada nos outros exemplos dessa seção.

Nesse código, nosso foco no momento da escrita da versão paralela será o laço da linha 4, sendo que é importante notar que é feita uma soma com incremento na variável sum, operação que deve ser considerada no momento da construção do programa paralelo e vetorial.

A Figura 1.5 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.

```
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 1.5. Laço paralelo com OpenMP.

1.3.6. 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 1.1. 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

Tabela 1.1. 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 1.5 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`.

1.3.7. 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 programa-

ção 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 1.6. Exemplo do cálculo de soma de vetor com redução em OpenMP.

Na Figura 1.6 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 `||`.

1.3.8. 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 1.7, 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 _NEC ivdep`.

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

```
void sum(int *a, int *b, int *c, long long int N){
   long long int i;

   #pragma _NEC ivdep
   for(i = 0; i < N; i++)
     a[i] = b[i] + c[i]
}
```

Figura 1.7. Exemplo do cálculo de soma de dois vetores com SIMD.

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.

As instruções vetoriais já estão presentes há muitos anos em processadores x86 e também no acelerador vetorial da NEC SX-Aurora. 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.

1.4. Estudos de Caso

Nesta seção, vamos apresentar um conjunto de aplicações e as respectivas alterações necessárias no código fonte com intuito de melhorar o seu desempenho com programação paralela e vetorial.

```
double dot_product(double *a, double *b, int n){
    int i;
    double dot = 0;

    for(i = 0; i < n; i++)
        dot += a[i] * b[i];

    return dot;
}
```

Figura 1.8. Produto interno sequencial.

1.4.1. Produto Interno

O primeiro exemplo é uma operação de produto interno entre dois vetores. Para esse problema, temos como entrada dois vetores e a saída um número real que relaciona o módulo desses dois vetores.

Na Figura 1.8, podemos ver a versão sequencial da função que resolve esse problema. Nesse código, o laço da linha 5 pode ser tanto paralelizado quanto vetorizado. Nesses casos, o ideal é testar separadamente cada uma das técnicas e depois em conjunto buscando verificar qual das versões apresenta o melhor desempenho.

A versão paralela e vetorial dessa aplicação é apresentada na Figura 1.9. Nessa versão, a linha 5 paraleliza o laço entre os *vector cores* do processador vetorial e, as linhas 6 e 7 indicam o envio dessas operações para as unidades vetoriais. Uma vez que a linha 9 possui uma soma com incremento, a qual resulta em uma seção crítica, uma redução é necessária.

```
1  double dot_product(double *a, double *b, int n){
2      int i;
3      double dot = 0;
4
5      #pragma omp parallel for private(i) reduction(+ :  dot)
6      #pragma _NEC vector
7      #pragma _NEC ivdep
8      for(i = 0; i < n; i++)
9          dot += a[i] * b[i];
10
11     return dot;
12 }
```

Figura 1.9. Produto Interno Paralelo e Vetorial.

1.4.2. Multiplicação de Matrizes

Operações sobre matrizes são fundamentais para diferentes campos do conhecimento, sendo que um algoritmo eficiente para essa operação é necessário. Uma implementação eficiente da multiplicação de matrizes leva em consideração o uso da memoria *cache*, do paralelismo e também das unidades vetoriais de um processador.

```
1  void matrix_mult(double *A, double *B, double *C, int N){
2      int i, j, k;
3
4      for(i = 0; i < N; i++)
5          for(j = 0; j < N; j++)
6              for(k = 0; k < N; k++)
7                  C[i * N + j] += A[i * N + k] * B[k * N + j];
8  }
```

Figura 1.10. Multiplicação de Matrizes sequencial.

Na Figura 1.10, podemos ver a versão sequencial da operação de multiplicação de

matrizes sobre duas matrizes (A) e (B), armazenando seu resultado em uma matriz (C). Para tanto, são necessários três laços de repetição, sendo que quando estamos fazendo a versão paralela nosso foco é sempre o laço mais externo possível. Por outro lado, na construção da versão vetorial, nosso foco é o laço mais interno.

```
1 void matrix_mult(double *A, double *B, double *C, int N){
2     int i, j, k;
3
4     #pragma omp parallel for private(i, j, k)
5     for(i = 0; i < N; i++)
6         for(k = 0; k < N; k++)
7             #pragma _NEC vector
8             #pragma _NEC ivdep
9             for(j = 0; j < N; j++)
10                C[i * N + j] += A[i * N + k] * B[k * N + j];
11 }
```

Figura 1.11. Multiplicação de Matrizes sequencial.

No momento da construção da versão paralela, a qual apresentamos na Figura 1.11, é necessário alterar a ordem dos laços de maneira a diminuir o tamanho do salto nas matrizes. Quanto menor for o salto nos acessos mais eficiente é a *cache* e também a vetorização. Nesse sentido, os laços foram alterados de i-j-k para i-k-j. Além disso, o paralelismo foi adicionado na linha 4, no laço mais externo, e a vetorização nas linhas 7 e 8, próximo ao laço mais interno.

1.4.3. Reverse Time Migration (RTM)

A *Reverse Time Migration* (RTM) (FLETCHER; DU; FOWLER, 2009; ZHOU et al., 2018; CLAPP; FU; LINDTJORN, 2010; CLAPP, 2015) 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[1] foi escrito em linguagem *C* e a discretização foi feita utilizando diferenças finitas. Simulações sísmica desse tipo podem ser encontradas em diferentes áreas como mencionadas nos seguintes artigos científicos (GUO et al., 2017; ZHU, 2017; CONFAL et al., 2018; GUO et al., 2018).

A modelagem simula a coleta de dados em um levantamento sísmico, como na Figura 1.12. 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.

[1] Esse código é parcialmente financiado por recursos do projeto Petrobras 2016/00133-9.

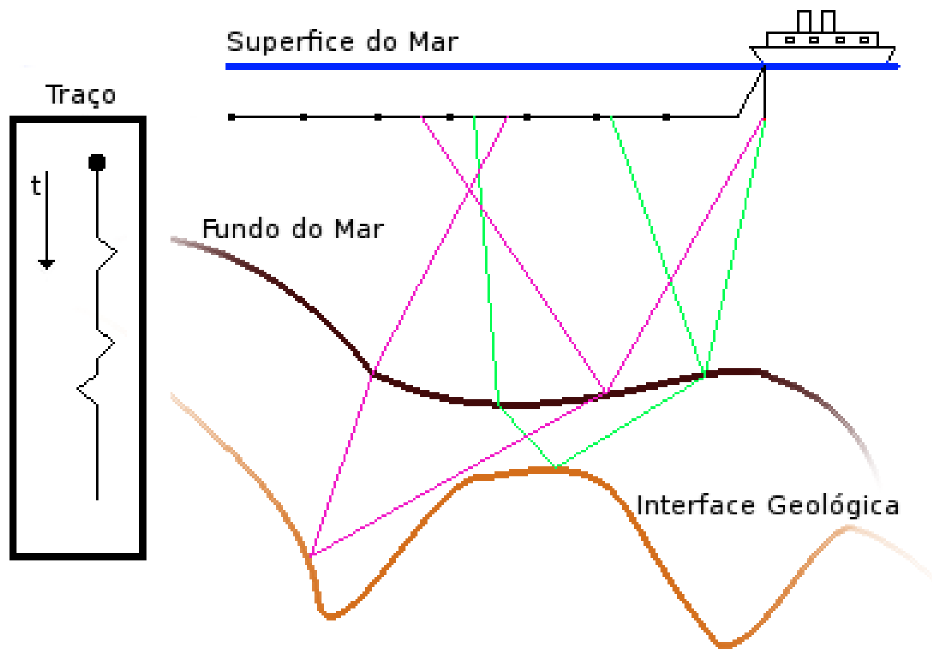

Figura 1.12. Coleta de dados em levantamento sísmico marítimo.

Na Figura 1.13 apresentamos a versão sequencial de uma das funções dessa aplicação, a qual possui três laços nas linhas 9, 10 e 11. A linha 16, a qual foi simplificada com a chamada da função cálculos de derivadas, está disponível no repositório `git` desse capítulo. Essas derivadas fazem parte do modelo proposto por Fletcher (FLETCHER; DU; FOWLER, 2009). A equação 1 representa seu modelo de propagação de onda.

$$
\begin{aligned}
\frac{\partial^2 p}{\partial t^2} &= v_{px}^2 H_2 p + \alpha v_{pz}^2 H_1 q + v_{sz}^2 H_1 (p - \alpha q) \\
\frac{\partial^2 q}{\partial t^2} &= \frac{v_{pn}^2}{\alpha} H_2 p + v_{pz}^2 H_1 q - v_{sz}^2 H_2 (\frac{1}{\alpha} p - q)
\end{aligned}
\tag{1}
$$

Onde $p(x,y,z)$ é a pressão da onda, q é uma variável auxiliar, v_{pz} é a velocidade da onda P na direção normal ao plano de simetria, v_{pn} é a velocidade normal de movimento da onda P, v_{px} é a velocidade da onda P no plano de simetria. Todas as velocidades da onda P são limitadas pelas equações $v_{pn} = v_{pz}\sqrt{1+2\delta}$ e $v_{px} = v_{pz}\sqrt{1+2\varepsilon}$, onde δ e ε são parâmetros anisotrópicos definidos por Thomsen (THOMSEN, 1986). v_{sz} é a velocidade da onda SV normal ao plano de simetria e α vem da solução da relação de dispersão P-SV DPE (FOWLER; DU; FLETCHER, 2010). Normalmente, α assume o valor de 1.

Além disso, sobre as equações, H_1 e H_2 são operadores lineares definidos como:

$$
\begin{aligned}
H_1 =& sin^2\theta cos^2\phi \frac{\partial^2}{\partial x^2} + sin^2\theta sin^2\phi \frac{\partial^2}{\partial y^2} + cos^2\theta \frac{\partial^2}{\partial z^2} + \\
& sin^2\theta sin2\phi \frac{\partial^2}{\partial x \partial y} + sin2\theta sin\phi \frac{\partial^2}{\partial y \partial z} + sin2\theta cos\phi \frac{\partial^2}{\partial x \partial z}
\end{aligned} \tag{2}
$$

$$
H_2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} - H_1
$$

onde θ é o ângulo de mergulho e ϕ o ângulo de azimute.

```
1 void kernel_CPU_06_mod_3DRhoCte(float* gU0, float* gU1, ...
     float* gVorg,
2   int nnoi, int nnoj, int k0, int k1,
3   float FATMDFX, float FATMDFY, float FATMDFZ, float *W){
4
5   int index_X, index_Y;
6   int stride  = nnoi * nnoj;
7   int index, k;
8
9   for(index_X = 0; index_X < nnoi; index_X++)
10    for(index_Y = 0; index_Y < nnoj; index_Y++)
11      for(k = 0; k < k1 - k0; k++){
12
13        index = (index_Y * nnoi + index_X) + (k0 + k) * ...
              stride;
14
15        if(gVorg[index] > 0.0f)
16          gU1[index] = calculos_de_derivadas();
17
18      }
19 }
```

Figura 1.13. Algoritmo RTM sequencial.

Após analisar o código fonte da aplicação, mesmo ele sendo complexo, por se tratar de uma aplicação real, é possível ver que os laços na ordem original dificultam a vetorização da computação. Devido a isso, os laços são alterados de x - y - k para y - k - x, dessa maneira é implementar sua versão paralela e vetorial.

A versão otimizada é apresentada na Figura 1.14. Na linha 9, adicionamos a codificação para dividir o trabalho entre as *vector cores*. Após, nas linhas 12 e 13, a vetorização é feita dividindo o trabalho para as VPPs. Essas e outras propostas de otimização

para essa aplicação de geofísica podem ser vistas em diversos trabalhos (SERPA et al., 2017, 2018b, 2018a, 2019).

```
1 void kernel_CPU_06_mod_3DRhoCte(float* gU0, float* gU1, ...
      float* gVorg,
2   int nnoi, int nnoj, int k0, int k1,
3   float FATMDFX, float FATMDFY, float FATMDFZ, float *W){
4
5   int index_X, index_Y;
6   int stride  = nnoi * nnoj;
7   int index, k;
8
9   #pragma omp parallel for private(index_X, index_Y, index, k)
10  for(index_Y = 0; index_Y < nnoj; index_Y++)
11    for(k = 0; k < k1 - k0; k++)
12      #pragma _NEC vector
13      #pragma _NEC ivdep
14      for(index_X = 0; index_X < nnoi; index_X++){
15
16        index = (index_Y * nnoi + index_X) + (k0 + k) * ...
              stride;
17
18        if(gVorg[index] > 0.0f)
19          gU1[index] = calculos_de_derivadas();
20
21      }
22 }
```

Figura 1.14. Algoritmo RTM utilizando programação paralela e vetorial.

1.5. Conclusão

Neste capítulo, foram apresentados conceitos sobre programação paralela e vetorial, exemplificando técnicas que podem ser utilizadas em arquiteturas como a NEC SX-Aurora. Essas técnicas são importantes para arquiteturas atuais e também para as que estão por vir. No futuro, planeja-se abordar outras arquiteturas paralelas e vetoriais.

Exercícios e soluções deste capítulo estão disponíveis em: <https://gitlab.com/msserpa/minicurso-prog-paralela-vetorial>.

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 4

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

CASTRO, M. et al. Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, v. 54, 2016. ISSN 0167-8191. páginas 3

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. [S.l.]: MIT press, 2008. v. 10. páginas 4

CLAPP, R. G. Seismic Processing and the Computer Revolution(s). In: SCHNEIDER, R. V. (Ed.). *Society of Exploration Geophysicists (SEG) Technical Program Expanded Abstracts 2015*. [S.l.], 2015. p. 4832–4837. páginas 14

CLAPP, R. G.; FU, H.; LINDTJORN, O. Selecting the right hardware for reverse time migration. *The Leading Edge*, v. 29, n. 1, 2010. páginas 14

CONFAL, J. M. et al. Numerical simulation of 3-d mantle flow evolution in subduction zone environments in relation to seismic anisotropy beneath the eastern mediterranean region. *Earth and Planetary Science Letters*, Elsevier, v. 497, p. 50–61, 2018. páginas 14

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

DIENER, M. et al. Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation*, Elsevier, v. 88, p. 18–36, 2015. 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 14, 15

FOWLER, P. J.; DU, X.; FLETCHER, R. P. Coupled equations for reverse time migration in transversely isotropic media. *Geophysics*, Society of Exploration Geophysicists, v. 75, n. 1, p. S11–S22, 2010. páginas 15

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 3

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 3

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

GUO, J. et al. Seismic dispersion and attenuation in saturated porous rocks with aligned fractures of finite thickness: Theory and numerical simulations—part 2: Frequency-dependent anisotropy. *Geophysics*, Society of Exploration Geophysicists, v. 83, n. 1, p. WA63–WA71, 2018. páginas 14

GUO, J. et al. Effects of fracture intersections on seismic dispersion: theoretical predictions versus numerical simulations. *Geophysical Prospecting*, European Association of Geoscientists & Engineers, v. 65, n. 5, p. 1264–1276, 2017. páginas 14

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

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 4

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

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 4

OPENMP, A. Openmp application program interface v3. 0. *OpenMP Architecture Review Board*, 2008. [Online; accessed 15-January-2020]. páginas 4

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 11

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

SERPA, M. S. et al. Memory performance and bottlenecks in multicore and gpu architectures. In: D'AGOSTINO, D. (Ed.). *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. [S.l.], 2019. p. 233–236. páginas 3

SOUZA, M. A. et al. Cap bench: a benchmark suite for performance and energy evaluation of low-power many-core processors. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 29, n. 4, p. e3892, 2017. páginas 3

SUBRAMANIAM, B. et al. Trends in energy-efficient computing: A perspective from the green500. In: DONGARRA, J.; MATSOUKA, S. (Ed.). *2013 International Green Computing Conference Proceedings*. [S.l.], 2013. p. 1–8. páginas 3

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 5

THOMSEN, L. Weak elastic anisotropy. *Geophysics*, v. 51, p. 1954–1966, 10 1986. páginas 15

ZHOU, H.-W. et al. Reverse time migration: A prospect of seismic imaging methodology. *Earth-Science Reviews*, Elsevier, v. 179, p. 207–227, 2018. páginas 14

ZHU, T. Numerical simulation of seismic wave propagation in viscoelastic-anisotropic media using frequency-independent q wave equation. *Geophysics*, Society of Exploration Geophysicists, v. 82, n. 4, p. WA1–WA10, 2017. páginas 14