1

Introdução à Programação *Multithreaded*: explorando arquiteturas heterogêneas e vetorização com OpenMP 4.0

Silvio Luiz Stanzani (NCC/UNESP - silvio@ncc.unesp.br) [1]
Raphael Mendes de Oliveira Cóbe (NCC/UNESP - rmcobe@ncc.unesp.br) [2]
Rogério Luiz Iope (NCC/UNESP - rogerio@ncc.unesp.br)[3]

Resumo:

O uso de modelos de programação paralela que combinam técnicas como *multithreading*, *offloading* e vetorização é essencial para explorar as possibilidades oferecidas por modernas arquiteturas paralelas, compostas por combinações de recursos incluindo processadores de múltiplos núcleos, aceleradores e coprocessadores. O objetivo deste minicurso é apresentar algumas das características oferecidas pela versão 4 do OpenMP que permitem explorar a heterogeneidade dos recursos computacionais oferecidos por tais arquiteturas.

[1] Silvio Luiz Stanzani é doutor em ciência pela Escola Politécnica da Universidade de São Paulo. Atua no projeto *Intel Modern Code Partner* do Núcleo de Computação Científica da Universidade Estadual Paulista, desenvolvido em colaboração com a Intel, ministrando treinamentos e realizando atividades de consultoria relacionadas à otimização e modernização de código. Possui experiência no desenvolvimento de software com foco em computação paralela e distribuída desde 2005.

[2] Raphael Mendes de Oliveira Cóbe é doutor em computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo com ênfase em Inteligência Artificial, possui mais de 10 anos de experiência em desenvolvimento de software. Atualmente trabalha no Núcleo de Computação Científica da Universidade Estadual Paulista em projetos relacionados a computação em nuvem e em grade.

[3] Rogério Luiz Iope é doutor em Engenharia Elétrica (ênfase em Computação e Sistemas Digitais) pela Escola Politécnica da Universidade de São Paulo. Atua desde 2001 no desenvolvimento de sistemas de processamento de alto desempenho e sistemas distribuídos. Trabalha como engenheiro de sistemas no Núcleo de Computação Científica da Universidade Estadual Paulista desde 2009, sendo atualmente responsável pelo desenvolvimento de projetos de P&D em parceria com a indústria.

1.1. Introdução

A contínua evolução dos microprocessadores nas últimas décadas proporcionou melhorias significativas no desempenho dos computadores, com o desenvolvimento de microarquiteturas cada vez mais complexas, incluindo técnicas de paralelismo em nível de instrução de máquina, aumento dos níveis de memória cache, além de consideráveis avanços na frequência do *clock* interno que rege a execução das microinstruções. No entanto, as condições que permitiram tais melhorias começaram a atingir seus limites por volta de 2005, de modo que hoje em dia não se conseguem grandes avanços apenas com aumentos nas frequências de *clock*, ou melhorias na microarquitetura e/ou na hierarquia de cache, ou ainda através de incrementos nas taxas de transferências de dados de/para a memória. Para explorar as novas arquiteturas de modo a se obter melhorias significativas de desempenho torna-se cada vez mais necessário aplicar técnicas de paralelismo explícito, pois o aumento no desempenho dos computadores atuais tem se baseado no desenvolvimento de arquiteturas computacionais paralelas.

As modernas arquiteturas de computadores podem ser compostas por diversos processadores, que por sua vez dispõem de múltiplos núcleos de processamento. Mais recentemente, tem se popularizado o desenvolvimento de sistemas computacionais compostos por coprocessadores e aceleradores que dispõem de muitos núcleos e grande capacidade de processamento vetorial, e atuam em conjunto com os processadores principais [HEI 2012]. Para que as aplicações possam explorar as oportunidades oferecidas por arquiteturas computacionais compostas por processadores multinúcleos, coprocessadores e/ou aceleradores, é necessário considerar dois aspectos: o código deve explorar ao máximo o poder computacional oferecido pelas diversas unidades de processamento e utilizar de modo eficiente os poderosos recursos de processamento vetorial disponíveis nas novas arquiteturas.

Tais aspectos promoveram o desenvolvimento de novas diretivas de compilação na especificação OpenMP (*Open Multi-Processing*) para prover suporte à utilização desses novos dispositivos de processamento e suporte à programação vetorial [Ope 2013]. O OpenMP é uma interface de programação de aplicativo (API) para C, C++ e Fortran que oferece um modelo de programação paralela portável e escalável para arquiteturas de memória compartilhada, disponibilizando aos programadores uma interface simples e flexível para o desenvolvimento de aplicações paralelas. A versão 4.0 do OpenMP, lançada em 2013, oferece suporte à programação de aceleradores e coprocessadores e permite explorar o modelo de programação SIMD (*Single Instruction Multiple Data*) [FLY 72]. Além de diversas melhorias, a versão 4.0 oferece um novo mecanismo para descrever regiões de código que podem ser transferidos para dispositivos como coprocessadores e aceleradores.

O objetivo deste minicurso é capacitar os estudantes quanto ao uso das novas diretivas da especificação OpenMP em arquiteturas heterogêneas, compostas por processadores multinúcleos que coexistem com coprocessadores e aceleradores com grande número de núcleos e poderosas unidades de processamento vetorial. Para ilustrar os conceitos, serão utilizados exemplos simples e um estudo de caso baseado em uma aplicação real [CIV 2015]. A infraestrutura computacional em que os exemplos foram executados é formada por servidores equipados com processadores Intel Xeon e coprocessadores Intel Xeon Phi, disponíveis no Núcleo de Computação Científica da UNESP. A estrutura geral do minicurso é detalhada a seguir.

1.1.1. Estrutura do Minicurso

A Seção 1.2. descreve os conceitos relacionados com arquiteturas paralelas e arquiteturas paralelas híbridas compostas por processador e coprocessador. A Seção 1.3. apresenta a interface de programação OpenMP, os elementos do padrão e os novos elementos presentes na versão 4.0.

As Seções 1.4., 1.5. e 1.6. apresentam algumas das principais diretivas do OpenMP: a Seção 1.4. descreve as novas diretivas para prover suporte à vetorização presentes no OpenMP 4.0, a Seção 1.5. apresenta as diretivas para prover suporte à execução de código em coprocessadores, e a Seção 1.6. descreve as diretivas para prover suporte à execução de liga de *threads*.

A Seção 1.7. apresenta um estudo de caso de paralelização de uma aplicação denominada simulação *N-Body* usando OpenMP 4.0. Na Seção 1.8. são apresentadas algumas conclusões.

1.2. Arquiteturas Computacionais Paralelas

Sistemas computacionais modernos são constituídos por uma combinação de recursos que incluem processadores multinúcleos, subsistemas de memória que podem apresentar múltiplos níveis de acesso, e subsistemas de entrada e saída. Tais sistemas são ditos heterogêneos quando dispõem de processadores auxiliares, como coprocessadores e/ou aceleradores (gráficos ou de uso geral), que podem acrescentar dezenas ou centenas de elementos de processamento extras, acessíveis ao programador. Diversos sistemas podem ainda ser agrupados formando agregados ou *clusters*. Podemos distinguir diversos níveis, como a seguir:

- nível do núcleo de processamento (*processing core*), constituído de registradores, unidades lógicas e aritméticas, unidades vetoriais, *caches* de instruções e de dados;
- nível do processador (*chip multiprocessor*), que pode conter múltiplos núcleos de processamento e um ou mais níveis de *cache*;
- nível do nó computacional (*computing node*), constituído por múltiplos processadores e os mecanismos de comunicação entre eles, o subsistema de memória (em geral, compartilhado entre os processadores) e os subsistemas de entrada e saída;
- nível de conjunto de nós computacionais (*computing cluster*): caracterizado por dispor de múltiplos nós computacionais e de mecanismos de comunicação entre nós (em geral, de grande largura de banda e baixa latência).

Uma tendência no desenho de novas arquiteturas computacionais é o desenvolvimento de arquiteturas paralelas híbridas, que combinam recursos heterogêneos em um mesmo nó computacional. Na seção 1.2.1. é descrito este caso especial.

1.2.1. Arquiteturas Paralelas Híbridas

As arquiteturas paralelas híbridas combinam processadores e coprocessadores ou aceleradores em um mesmo nó computacional. Nessas arquiteturas o conjunto de processadores é chamado de *host* e os coprocessadores e aceleradores são usualmente denominados *devices* [VAJ 2011].

Tanto o *host* quanto os *devices* possuem núcleos de processamento e memória próprios. Nas arquiteturas mais comuns, a comunicação entre *host* e *device* ocorre por meio de um barramento de interconexão, como por exemplo o PCI (*Peripheral Component Interconnect*). Nesse sentido, a execução de uma aplicação nessa arquitetura pode ser realizada somente no *host*, somente nos *devices* ou de modo colaborativo entre *host* e *devices*.

Para que a execução seja realizada de modo colaborativo entre *host* e *device*, um dos mecanismos utilizados é conhecido pelo termo *offloading*. Nesse mecanismo, a execução de determinados trechos do código é enviada do *host* para um ou mais *devices*. Algumas arquiteturas paralelas híbridas permitem que os *devices* sejam acessíveis por meio de um endereço de rede virtualizado; nesses casos, é possível executar uma aplicação entre *host* e *devices* utilizando mecanismos de passagem de mensagens como o MPI (*Message Passing Interface*) [FOR 94], em conjunto com mecanismos de memória compartilhada como o OpenMP.

A utilização de arquiteturas paralelas híbridas apresenta muitas possibilidades, e a utilização eficiente dessas arquiteturas deve levar em conta os seguintes fatores: o paralelismo presente nos diversos níveis do sistema computacional, a heterogeneidade dos recursos, e os custos de transferência entre *host* e *devices*.

1.2.2. Explorando Paralelismo em Múltiplos Níveis

Dois mecanismos básicos para explorar o paralelismo presente nos diversos níveis do sistema computacional são os seguintes: vetorização e programação *MultiThreaded*.

1.2.2.1. Vetorização

A vetorização é uma técnica que tem como objetivo explorar o paralelismo no nível de instruções. Cada núcleo de processamento é capaz de executar instruções escalares e vetoriais. As instruções escalares utilizam sempre dois operandos por vez, já as instruções vetoriais podem utilizar diversos operandos simultaneamente.

Para explorar o uso de instruções vetoriais as seguintes estratégias podem ser adotadas:

- vetorização automática: está presente na maioria dos compiladores e consiste em utilizar instruções vetoriais no lugar das instruções escalares tradicionais, sempre que nenhuma alteração no resultado final seja gerada;
- semi-vetorização automática: consiste em diretivas que podem ser colocadas no código para guiar o compilador, de maneira precisa, quanto ao uso de instruções vetoriais;
- vetorização explícita: é a utilização de chamadas de funções implementadas por meio de instruções vetoriais diretamente no código.

1.2.2.2. Multithreading

A maioria dos sistemas operacionais modernos controla a execução de programas por meio de processos e *threads*. Processos representam programas em execução e sob controle do sistema operacional. Os processos são compostos por uma ou mais *threads*,

que representam uma linha de execução formada por uma sequência de instruções que podem ser controladas de maneira independente pelo sistema operacional [SIL 2008].

Todo processo tem pelo menos uma *thread*, chamada de *thread* principal. A *thread* principal pode disparar novas *threads*, que podem ser executadas concorrentemente em diversos núcleos, e compartilharem os dados do processo.

O conjunto de *threads* criadas por um processo pode ser organizado de duas formas: time de *threads*, que é composto pela *thread* principal e outras *threads* para dividir o processamento, e liga de *threads*, que representa um conjunto de times de *threads*.

1.2.2.3. Programação *Multithreaded*

O uso de arquiteturas paralelas híbridas introduziu novos desafios à exploração de paralelismo intrísico de aplicações em arquiteturas multinúcleos. O primeiro é o controle de vetorização e o segundo é o controle de execução de código no *host* ou em *devices*.

Uma das especificações mais tradicionais de programação *multithreaded* em arquiteturas multinúcleos é o OpenMP. Para atender os novos desafios presentes nas arquiteturas paralelas híbridas, a versão 4.0 do OpenMP foi lançada, disponibilizando mecanismos para semi-vetorização automática e *offloading*. A Seção 1.3. descreve em detalhes tais mecanismos.

1.3. OpenMP

OpenMP é uma especificação que define um conjunto de diretivas de compilador, bibliotecas e variáveis de ambiente, com o objetivo de prover recursos para explorar o paralelismo intrínseco de programas por meio de *threads*. A especificação OpenMP é mantida por um grupo chamado *OpenMP Architecture Review Board*[4], que é composto por diversos fornecedores de *hardware* e *software* e usuários de computação paralela. Atualmente, a especificação OpenMP é implementada em diversos compiladores C/C++ e Fortran, e portável entre diversas arquiteturas computacionais.

A estrutura de um programa que explora o paralelismo intrínseco utilizando diretivas OpenMP é composto por regiões de código sequenciais e regiões de código paralelas. As regiões sequenciais são executadas apenas pela *thread* principal. As regiões paralelas são executadas por um time de *threads* ou por uma liga de times de *threads*. Nesse sentido, a execução é iniciada em uma região sequencial e regiões paralelas são iniciadas e finalizadas ao longo da execução do programa conforme mostrado na Figura 1.1.

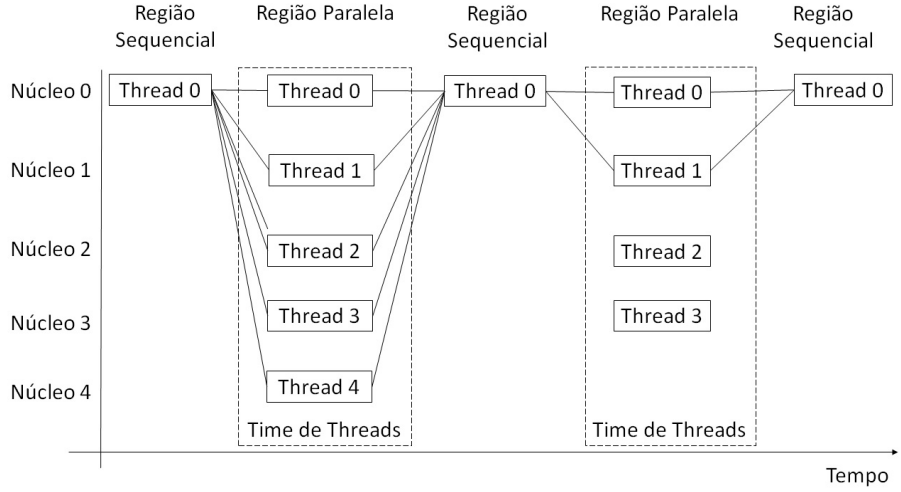

Figura 1.1: Criação de Times de *Threads* e Alocação das *Threads* nos Recursos Computacionais.

A Seção 1.3.1. descreve o formato e os grupos de diretivas da especificação OpenMP.

1.3.1. Diretivas da Especificação OpenMP

As diretivas do OpenMP são implementadas nos compiladores C/C++ e Fortran. Nos compiladores C/C++ a implementação é feita por meio de um recurso da linguagem de programação denominado diretivas "pragma". Tais diretivas têm como objetivo definir extensões da linguagem e são incluídas no código por meio da palavra reservada #pragma. Caso o compilador não reconheça a diretiva a linha é ignorada e o programa é compilado normalmente. Nesse sentido, as diretivas OpenMP são interpretadas pelos compiladores utilizando uma opção de linha de comando. Caso a opção não seja habilitada ou caso o compilador não possua suporte ao OpenMP, tais diretivas são ignoradas e o programa é compilado de modo sequencial.

[4]http://openmp.org

As diretivas OpenMP nas compiladores C/C++ [5] seguem o seguinte formato:
#pragma omp diretiva [cláusulas]
Cada linha deve conter ao menos uma diretiva e pode conter uma ou mais cláusulas.

As diretivas do OpenMP são divididas nos seguintes grupos:

- Construtor paralelo: forma um time de *threads* e executa um bloco de código em todas as *threads* disponíveis, sem realizar divisão de trabalho automática.
  - **#pragma omp parallel**
- Compartilhamento de trabalho: distribuem a carga de trabalho de um programa que pode ser: passos de um laço ou sub-blocos de código, entre as *threads* de um time.
  - **#pragma omp [parallel] for**
  - **#pragma omp [parallel] sections**
  - **#pragma omp [parallel] workshare**
  - **#pragma omp single**
- Tarefas: são usadas para definir regiões de código independentes chamadas de tarefas, que podem ser executadas por qualquer *thread* de um time, e também para definir dependências entre as tarefas como em um workflow. Dessa forma, a partir de uma região de código, podem ser geradas diversas tarefas, que serão escalonadas e executadas pelas *threads* disponíveis.
  - **#pragma omp task**
  - **#pragma omp taskwait**
- Sincronização: são usadas para sincronizar a execução de comandos pelas *threads*. Um exemplo é, a diretiva #pagma critical, que define um bloco de código dentro de uma região paralela que deve ser executado por apenas uma *thread* por vez.
  - **#pragma omp atomic**
  - **#pragma omp barrier**
  - **#pragma omp critical**
  - **#pragma omp flush**
  - **#pragma omp master**
  - **#pragma omp ordered**

A paralelização de passos de um laço é um exemplo de uso de diretivas do OpenMP para explorar paralelismo intrínseco, e consiste em dividir passos de um laço em *threads* de um time conforme mostrado na Figura 1.2.

Na Seção 1.3.2. serão detalhadas as inovações introduzidas pela versão 4.0.

[5] Nesse trabalho apresentaremos o uso de OpenMP apenas para C/C++

```
N=25;
#pragma omp parallel for
for (i=0; i<N; i++)
  a[i] = a[i] + b;
```

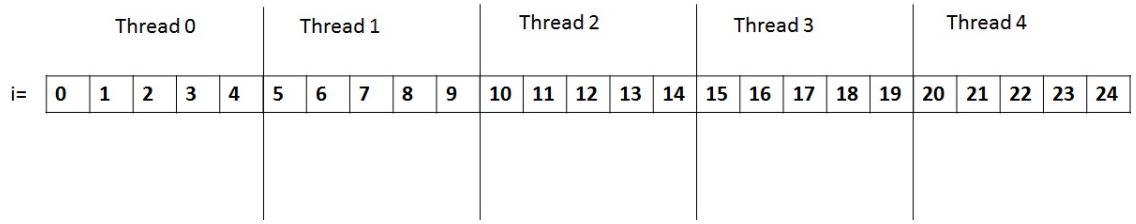

Figura 1.2: Divisão dos Passos de um Laço Pelas *Threads* de um Time.

1.3.2. OpenMP 4.0

A principal inovação presente na versão 4.0 da especificação OpenMP [Ope 2013], é o aparecimento das diretivas para prover suporte à execução de tarefas com paralelismo intrínseco em arquiteturas paralelas híbridas.

O desenvolvimento dessas novas diretivas foi motivado pelo trabalho realizado em diversas iniciativas, com o objetivo de explorar a execução de tarefas com paralelismo intrínseco, em arquiteturas paralelas híbridas.

No contexto da arquitura FPGA (*Field Programmable Gate Array*), uma iniciativa [CAB 2009] foi responsável pelo desenvolvimento de uma extensão do OpenMP 3.0 com o objetivo de realizar *offloading* para *devices* FPGA.

Outra iniciativa foi desenvolvida para criar um ambiente de execução OpenMP para um processador de sinais digitais (DSP) desenvolvido utilizando arquitetura ARM (*Advanced RISC Machine*) [STO 2013] [MIT 2013].

A tendência foi adotada como política oficial da equipe que especifica o OpenMP, sendo agora possível realizar *offloading* para diversos dispositivos, tais como, GPU (*Graphics Processing Unit*) [BER 2014], Xeon Phi [HAR 2014] e ARM [MIT 2014]. Nesse sentido, os seguintes conjuntos de diretivas foram disponibilizados:

- SIMD: é utilizada para instruir o compilador a gerar versões vetoriais de laços;
- *Offloading*: permite enviar blocos de código para execução em um *device*;
- Liga de *threads*: permite criar uma liga de *threads*, e dividir a execução de regiões de código entre os times.

A Seção 1.4. descreve as diretivas SIMD, a Seção 1.5. descreve as diretivas para *offloading* e a Seção 1.6. descreve as diretivas de criação e uso liga de time de *threads*.

1.4. Vetorização

As diretivas do OpenMP 4.0 para vetorização são chamadas de diretivas SIMD e são divididas em três categorias: vetorização de laço **#pragma omp simd** descrita na Seção 1.4.1., vetorização de método **#pragma omp declare simd** descrita na Seção 1.4.2. e na Seção 1.4.3. é descrita aparalelização e vetorização em conjunto.

1.4.1. Vetorização de Laço

A diretiva **#pragma omp simd** define que o laço marcado deve ser compilado usando instruções SIMD. Nesse sentido, o laço é executado em apenas uma *thread*, porém para cada operação do laço diversos dados são utilizados concorrentemente conforme mostrado na Figura 1.3.

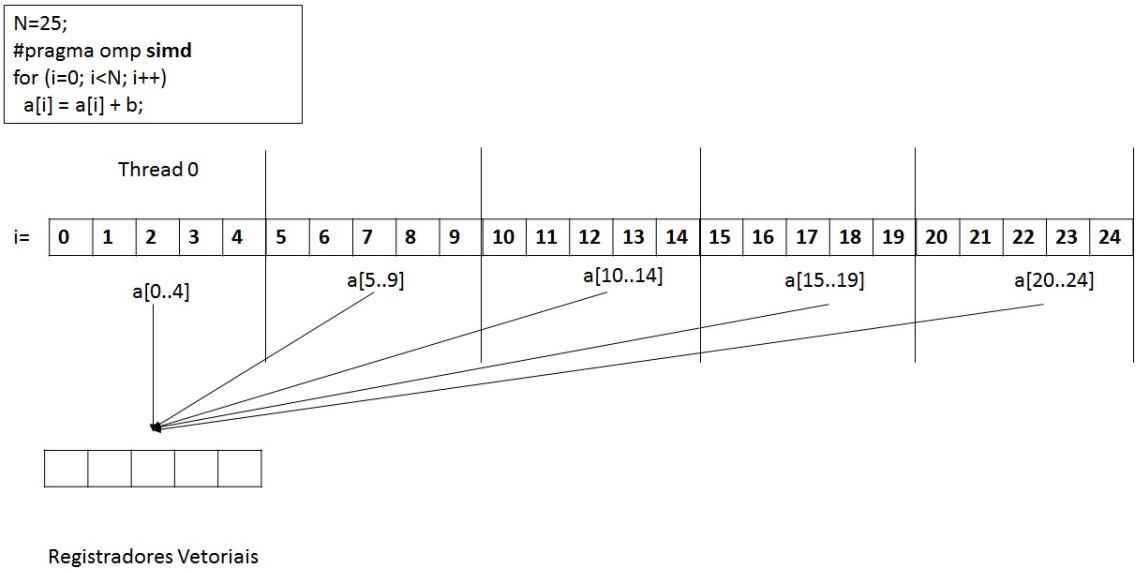

Figura 1.3: Execução concorrente de um Conjunto de Passos de um Laço.

Sintaxe:

```
1 #pragma omp simd [clausula[[,] clausula] ...]
2 laço for
```

A seguir são descritas as cláusulas que podem ser usadas em conjunto com as diretivas pragma omp simd:

- **safelen (tamanho)** : o parâmetro tamanho define o número máximo de passos do laço que podem ser executadas concorrentemente sem quebrar uma dependência entre os passos;
- **linear (variável[:passo-linear],...)** : define que a variável é incrementada a cada passo do laço, e que o valor do incremento atribuído à variável a cada passo é igual ao valor passado no parâmetro passo-linear;
- **aligned (variável[:alinhamento])** : define que a variável foi previamente alinhada em memória, e o parâmetro alinhamento define o tamanho do bloco de alinhamento;
- **private(variável)**: define que a variável tem escopo válido apenas na região paralela;
- **lastprivate(variável)**: define que após o termino da região paralela, a variável assumirá o valor que possuia antes do início da região paralela;
- **reduction(variável)**: calcula a somatória da variável;

- **collapse (n)** : especifica o número de laços aninhados que serão agrupados em um único laço.

As cláusulas **linear**, **aligned**, **private**, **lastprivate** e **reduction** também aceitam uma lista de variáveis como parâmetro;

O Código 1.1 realiza o cálculo de multiplicação de duas matrizes bidimencionais **a** e **b** e armazena o resultado na matriz bidimencional **c** utilizando três laços aninhados. Diretivas OpenMP foram utilizadas para dividir os passos de laços em um time de *threads* e para indicar que o compilador deverá usar instruções vetoriais.

A estratégia utilizada para dividir a execução dos três laços aninhados foi a seguinte:

- Os laços mais externos das linhas 2 e 3 foram agrupados em um laço único passando o parâmetro 2 para a cláusula **collapse** da diretiva **#pragma omp for**. Os passos desse laço único serão divididas em múltiplas *threads*.

- O laço da linha 5 será executado de modo sequencial por cada uma das *threads* que estiverem executando os passos do laços único utilizando instruções vetoriais.

Código 1.1: Multiplicação de Matrizes.

```
1 #pragma omp parallel for collapse (2)
2 for(i=0; i<msize; i++) {
3   for(k=0; k<msize; k++) {
4     #pragma omp simd
5     for(j=0; j<msize; j++) {
6       c[i][j] = c[i][j] + a[i][k] * b[k][j];
7     }
8   }
9 }
```

1.4.2. Vetorização de Função

A diretiva **#pragma omp declare simd** pode ser aplicada à funções com o objetivo de criar uma versão vetorial da função, capaz de receber múltiplos elementos de cada argumento para serem processamdos simultaneamente pela função usando instruções SIMD.

Sintaxe:

```
1 #pragma omp declare simd [clausula[[,] clausula] ...]
2 definicao de uma funcao ou uma declaracao
```

A seguir são descritas as cláusulas que podem ser usadas em conjunto com a diretiva pragma omp declare simd:

- **uniform (variável)**: define que o valor da variável se mantém constante para todas as chamadas concorrentes da função;

- **simdlen (tamanho)**: O parâmetro tamanho define a quantidade máxima de elementos de cada argumento da função, que podem ser carregados por intruções SIMD;

- **inbranch**: define que a função pode ser chamada a partir de estruturas condicionais no laço;

- **notinbranch**: define que a função nunca é chamada a partir de estruturas condicionais no laço.

Além dessas, podem ser utilizadas também as cláusulas descritas na Seção 1.4.1. : **aligned**, **simdlen**, **linear**, **reduction** e **uniform**.

O Código 1.2 realiza uma operação de interpolação em todos os elementos de uma matriz [RAS 2015]. Na função main um laço percorre todos os elementos de uma matrix na linha 19. Para cada elemento a função **interpolate** realiza a interpolação de um elemento da matriz.

As funções **log** e **exp** chamadas pelas funções **FindPosition** e **Interpolate** serão substituídas por versões vetoriais. A matriz **vals** foi marcada como constante durante as execuções na linha 6 usando a cláusula **uniform**, e na função **FindPosition** serão carregados nos registradores vetoriais uma quantidade de elementos da variável **x** igual ao valor da constante **SIMD_LEN**, usando a cláusula **simdlen** na linha 1.

Código 1.2: Interpolação.

```
1  #pragma omp declare simd simdlen(SIMD_LEN)
2  int FindPosition(double x) {
3    return (int)(log(exp(x*steps)));
4  }
5
6  #pragma omp declare simd uniform(vals)
7  double Interpolate(double x, const point* vals){
8    int ind = FindPosition(x);
9    const point* pnt = &vals[ind];
10   double res = log(exp(pnt->c0*x+pnt->c1));
11   return res;
12 }
13 ...
14
15 int main(int argc, char* argv[])
16 {
17   ...
18   #pragma omp simd
19   for(i=0; i<ARRAY_SIZE;++i) {
20     dst[i] = Interpolate(src[i],vals);
21   }
22   ...
23 }
```

1.4.3. Paralelização e Vetorização em Conjunto

A diretiva **#pragma omp [parallel] for simd** com a variante **[parallel]** permite definir um laço em que os passos sejam executadas pelas *threads* do time e que cada passo seja executado utilizando instruções vetoriais conforme mostrado na Figura 1.4.

Sintaxe:

```
1 #pragma omp [parallel] for simd [clausula[[,] clausula] ...]
2 laco for
```

O Código 1.3 faz a leitura de todos os elementos de três matrizes, e realiza diferentes operações para os elementos de cada matriz. Nesse caso, o laço da linha 9 foi paralelizado para dividir os passos nas *threads* do time, porém cada passo é executado com instruções vetoriais.

Código 1.3: Operação com Matrizes.

```
1 int repetitions=1800000;
2 int contrep=0;
3 int auxrand=0;
4
5 for(contrep=0; contrep<repetitions; contrep++) {
6         auxrand=rand();
7         i=0;
```

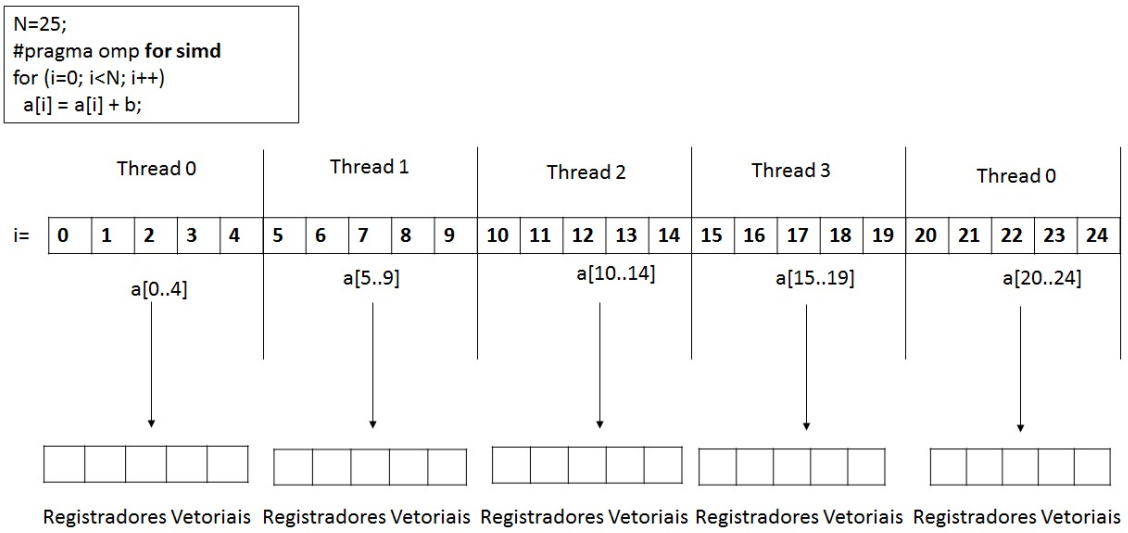

Figura 1.4: Divisão dos Passos de um Laço em *Threads* e Execução de Cada Passo Usando Unidades Vetoriais.

```
8           #pragma omp parallel for simd
9           for(j=0; j<NUM; j++) {
10                  a[i][j] = distsq(a[i][j], b[i][j])-auxrand;
11                  b[i][j] += min(a[i][j], b[i][j])+auxrand;
12                  c[i][j] = (min(distsq(a[i][j], b[i][j]), a[i][j]))/auxrand;
13          }
14  }
```

1.5. *Offloading*

As diretivas *offloading* têm como objetivo permitir a execução de regiões de código em um *device*. Nesse sentido as seguintes diretivas são disponibilizadas: **#pragma omp declare target**, **#pragma omp declare end target** e **#pragma omp target [data | update]**.

O processo de *offloading* é realizado seguindo os seguintes passos: transferência de dados e de código do *host* para um *device*, execução do código no *device*, e transferência de dados do *device* para o *host* conforme mostrado na Figura 1.5.

Outra forma de realizar o *offloading* é criar um bloco de fluxo de dados e dentro do bloco realizar transferências e *offloading* de regiões de código conforme mostrado na Figura 1.6.

A Seção 1.5.1. descreve como realizar *offloading* a Seção 1.5.2. descreve como criar um bloco de fluxo de dados entre *host* e *device* A Seção 1.5.3. descreve como atualizar dados do *host* e *device* e vice-versa, e a Seção 1.5.4. descreve a diretiva para criar um bloco de código que deve ser compilado tanto para ser usado no *host* quanto no *device*.

1.5.1. *Offload* de código

A diretiva **#pragma omp target** transfere um bloco de código para ser executado em um *device*. Essa diretiva pode ser utilizada com dois modificadores: **data** ou **update**.

Sintaxe:

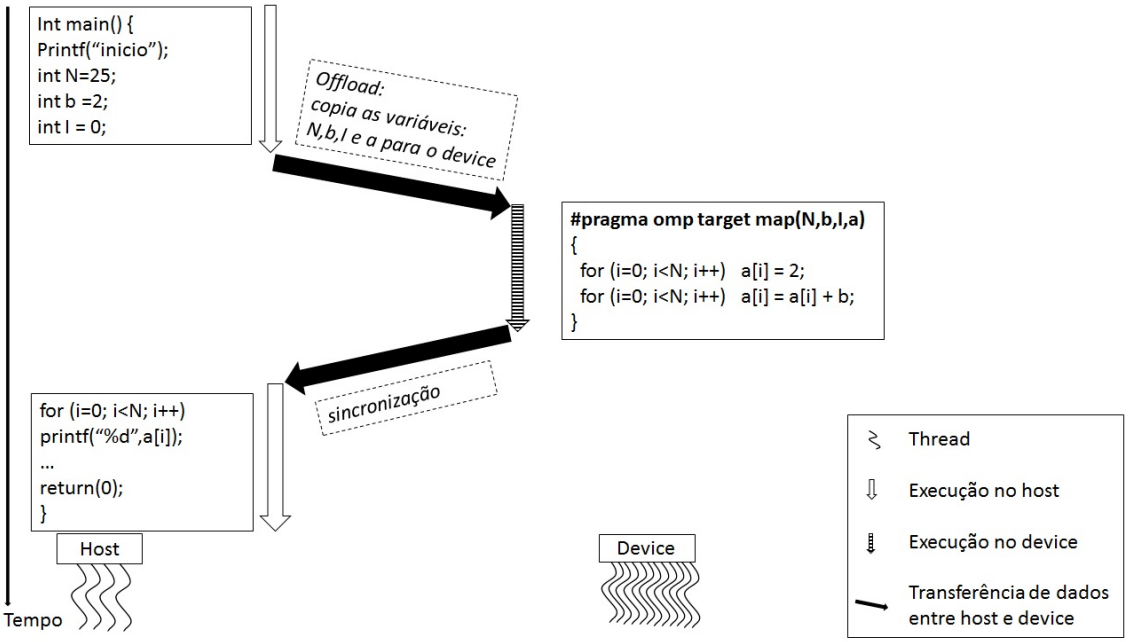

Figura 1.5: Sequência de Passos da Execução do *Offloading* de um Laço Para um *Device* Mapeando Três Variáveis.

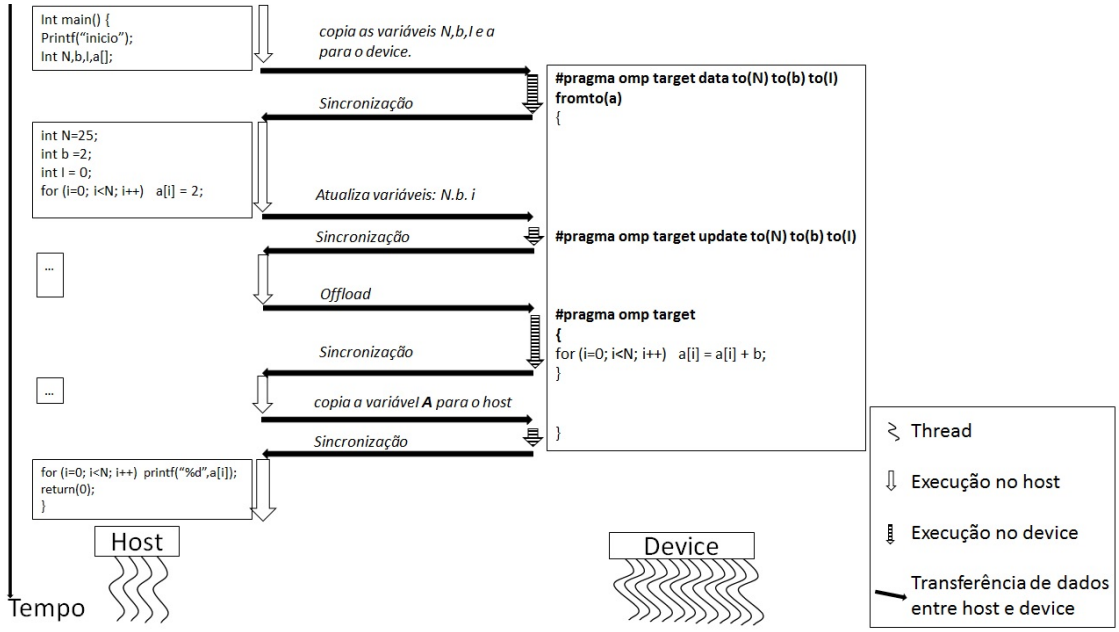

Figura 1.6: Sequência de Passos da Criação de um Bloco de Fluxo de Dados Entre o *Host* e o *Device*.

```
1 #pragma omp target [ update | data ] [clausula[[,] clausula] ...]
```

A seguir são descritas as cláusulas que podem ser usadas em conjunto com a diretiva **#pragma omp target [update | data]**:

- **device(identificação do *device*)**: o parâmetro identificação do *device* define o *device* para qual será realizado o *offloading*;

- **if (expressão condicional)**: o parâmetro expressão condicional define uma condição para realizar o *offloading*. Caso o teste da expressão seja verdadeiro a execução será realizada no *device*, caso contrário, a execução será realizada no *host*.

- **map([tipo de mapeamento :] variável, ...)**: essa cláusula define um mapeamento de uma variável entre o *host* e o *device*. Os seguintes tipos de mapeamento podem ser usados como parâmetro:

  - **to**: define que a variável será enviada do *host* para um *device*;
  - **from**: define que a variável será enviada de um *device* para o *host*;
  - **tofrom**: define que a variável vai ser transferida para o *device* na primeira linha do bloco e transferida do *device* para o *host* após o fim do bloco;
  - **alloc**: define que será alocada uma área de memória para a variável no *device*, que receberá valores a partir de um processamento realizado no próprio *device*.

O Código 1.4 mostra o uso da diretiva **#pragma omp target** para realizar *offloading* do Código 1.1 para um *device*. Na linha 1 a cláusula *device* foi utilizada para definir que será feito o *offloading* para o *device* 0 e as cláusulas *map* foram utilizadas para mapear o conteúdo das variáveis **a**,**b** e **c** para o *device* 0.

Código 1.4: *Offloading* do Código que Realiza Multiplicação de Matrizes.

```
1  #pragma omp target device(0) map(a[0:NUM][0:NUM]) \
2    map(b[0:NUM][0:NUM]) map(c[0:NUM][0:NUM])
3  {
4    int i,j, k;
5    #pragma omp parallel for collapse (2)
6    for(i=0; i<NUM; i++) {
7      for(k=0; k<NUM; k++) {
8        #pragma omp simd
9        for(j=0; j<NUM; j++) {
10         c[i][j] = c[i][j] + a[i][k] * b[k][j];
11       }
12     }
13   }
14 }
```

1.5.2. Definição de um Bloco de Fluxo de Dados entre *host* e *device*

A diretiva **#pragma omp target data** define um bloco de fluxo de dados. Os fluxos do *host* para o *devices* são realizados antes do início da execução do bloco, e os fluxos do *device* para o *host*, são executados imediatamente após a última linha do bloco. Dentro do bloco a execução de código nos *devices* deve ser realiza por meio da diretiva **#pragma omp target data**.

Sintaxe:

```
1  #pragma omp target data [clausula[[,] clausula] ...]
```

Na Seção 1.5.1. foram apresentadas as cláusulas que podem ser utilizadas com essa diretiva.

O Código 1.5 realiza *offloading* do Código 1.1 que realiza multiplicação de matrizes para um *device*. O *offloading* é feito utilizando as diretivas: **#pragma omp target data** e **#pragma omp target**. A diretiva **#pragma omp target data** na linha 1 realiza a transferências de dados entre *host* e *device*, nesse caso, as variáveis **a**,**b** e **c** foram

transferida para o *device* antes do início do bloco. Dentro desse bloco na linha 4 foi utilizada a diretiva **#pragma omp target** para executar um bloco de código no *device* sem a necessidade de mapear dados entre o *host* e o *device*.

Código 1.5: *Offloading* do Código que Realiza Multiplicação de Matrizes Usando #pragma omp target data.

```
1  #pragma omp target data map(a[0:NUM][0:NUM]) map(b[0:NUM][0:NUM])\
2    map(c[0:NUM][0:NUM])
3  {
4    #pragma omp target {
5      omp_set_num_threads(60);
6      int i,j, k;
7      #pragma omp parallel for collapse (2)
8      for(i=0; i<NUM; i++) {
9        for(k=0; k<NUM; k++) {
10         #pragma omp simd
11         for(j=0; j<NUM; j++) {
12           c[i][j] = c[i][j] + a[i][k] * b[k][j];
13         }
14       }
15     }
16   }
17 }
```

1.5.3. Atualização de dados entre *host* e *devices*

A diretiva **#pragma omp target update** é utilizada para atualizar os dados internos a uma região delimitada por um **#pragma omp target data**.

Sintaxe:

```
1  #pragma omp target update [clausula[[,] clausula] ...]
```

Na Seção 1.5.1. foram apresentadas as cláusulas que podem ser utilizadas com essa diretiva.

O Código 1.6 mostra um exemplo de atualização do conteúdo de uma variável no *host* a partir do conteúdo gerado no *device*. Esse código foi escrito a partir do Código 1.5, incluindo após o fim da execução da região de código delimitada pela diretiva **#pragma omp target** na linha 18, uma transferência do conteúdo da variável **c** do *device* para o *host*, utilizando a diretiva **#pragma omp target update**, que foi chamada dentro do bloco **#pragma omp target data**.

Código 1.6: *Offloading* do Código que Realiza Multiplicação de Matrizes Usando #pragma omp target Update.

```
1  #pragma omp target data map(to:a[0:NUM][0:NUM]) map(to:b[0:NUM][0:NUM]) \
2    map(to:c[0:NUM][0:NUM])
3  {
4    printf("␣ex␣6␣c[11][12]␣%d", c[11][12] );
5    #pragma omp target{
6      omp\_set\_num\_threads(60);
7      int i,j, k;
8      #pragma omp parallel for collapse (2)
9      for(i=0; i<NUM; i++) {
10       for(k=0; k<NUM; k++) {
11         #pragma omp simd
12         for(j=0; j<NUM; j++) {
13           c[i][j] = c[i][j] + a[i][k] * b[k][j];
14         }
15       }
16     }
17   }
18   #pragma omp target update from(c[0:NUM][0:NUM])
19   printf("ex␣6␣c[11][12]␣%d", c[11][12] );
20 }
```

1.5.4. Bloco para definição de variáveis no *host* e nos *device*

As diretivas **#pragma omp declare target** e **#pragma omp end declare target** são utilizadas para delimitar uma região de código onde serão definidas variáveis e métodos, que podem ser usadas tanto no *host* quanto nos *devices*.

No início da execução do programa serão criadas versões das variáveis e funções definidas nesse bloco no *host* e nos *devices*. A execução desses métodos em um *device*, e a atualização do conteúdo dessas variáveis do *host* para o *device* e vice-versa é feito por meio da diretiva **#pragma omp target**.

Sintaxe:

```
1 #pragma omp declare target
2 ...
3 definicao de metodos e variaveis
4 ...
5 #pragma omp end declare target
```

O Código 1.7 mostra uma versão do Código 1.3 sendo executado no *device*, usando a diretiva **#pragma omp target** na linha 17. As funções **min** e **distsq** foram declarados dentro do bloco **#pragma omp declare simd**, entre as linhas 1 e 13, e por isso podem ser chamadas diretamente da região de código enviada para execução no *device*.

Código 1.7: Métodos Compartilhados Entre *Host* e *Device* Usando #pragma omp declare target.

```
1  #pragma omp declare target
2
3  #pragma omp declare simd
4  float min(float a, float b) {
5    return a < b ? a : b;
6  }
7
8  #pragma omp declare simd
9  float distsq(float x, float y) {
10   return (x - y) * (x - y);
11 }
12
13 #pragma omp end declare target
14
15 ...
16
```

```
#pragma omp target
{
   repetitions=180;//0000;
   contrep=0;
   auxrand=0;

   for(contrep=0; contrep<repetitions; contrep++) {
     auxrand=rand();
     i=0;

     #pragma omp parallel for simd
     for(j=0; j<NUM; j++) {
       a[i][j] = distsq(a[i][j], b[i][j])-auxrand;
       b[i][j] += min(a[i][j], b[i][j])+auxrand;
       c[i][j] = (min(distsq(a[i][j], b[i][j]), a[i][j]))/auxrand;
     }
   }
}
...
```

1.6. Liga de *Threads*

A versão 4.0 do OpenMP permite dividir a carga de trabalho de uma aplicação em uma liga de *threads*, usando as diretivas: **#pragma omp teams** e **#pragma omp distribute** conforme ilustrado na Figura 1.7.

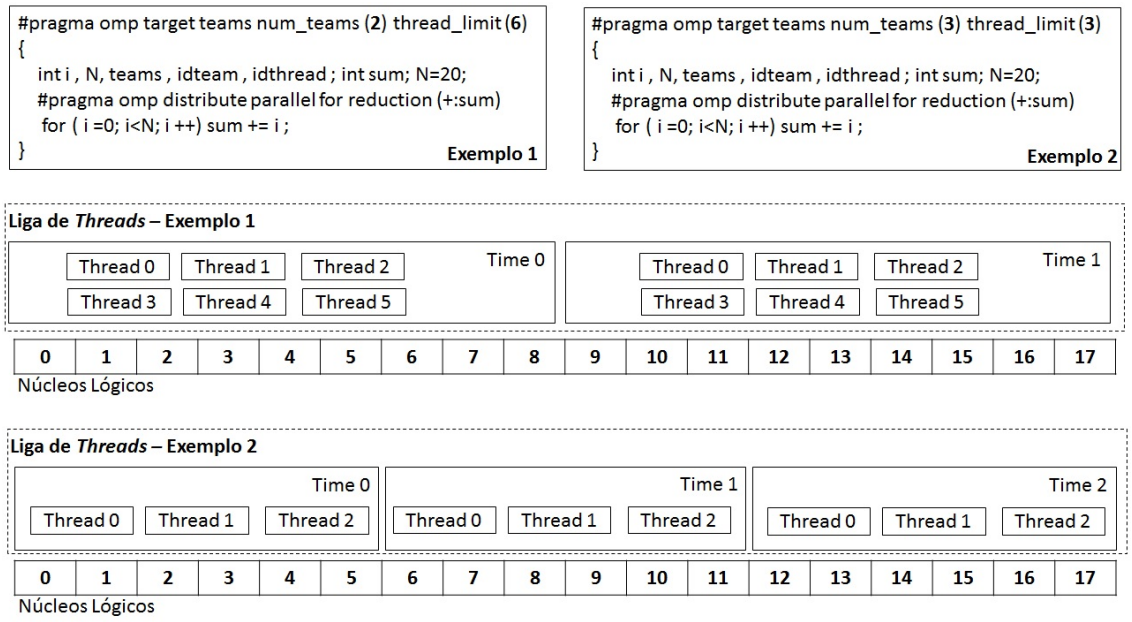

Figura 1.7: Dois Exemplos de Criação de Ligas de Times *Threads* e Alocação nos Recursos Computacionais.

1.6.1. Criação de Uma Liga de Times de *Threads*

A diretiva **#pragma omp teams** tem como objetivo criar uma liga de *threads* e executar um mesmo bloco de código em todas as *threads* principais de cada time.

Sintaxe:

```
#pragma omp teams [clausula[[,] clausula] ...]
```

As seguintes cláusulas podem ser utilizadas com essa diretiva:

- **num_teams(quantidade)**: define o número de times que serão criados;
- **thread_limit(limite)**: define o número máximo de *threads* em cada time.

A Seção 1.4.1. descreve as seguintes cláusulas: **private**, **firstprivate**, **shared** e **reduction**, que também podem ser usadas em conjunto com essa diretiva.

1.6.2. Divisão de Trabalho Entre os Times de *Threads*

A diretiva **#pragma omp distribute** define que passos de um laço devem ser executados nas *threads* principais de todos times de *threads* de uma liga.

Sintaxe:

```
#pragma omp distribute [clausula[[,] clausula] ...]
laço \textit{for}
```

A seguinte cláusula pode ser utilizada com essa diretiva:

- **dist_schedule (static[tamanho do bloco])**: essa cláusula define uma quantidade fixa de passos de um laço que deve ser executada em cada time por vez.

A Seção 1.4.1. descreve as seguintes cláusulas: **private**, **firstprivate** e **collapse**, que também podem ser usadas em conjunto com essa diretiva.

1.6.3. Exemplos de Uso das Diretivas de Controle de Liga de *Threads*

O Código 1.8 mostra um laço que calcula a somatória da variável sum. Os passos do laço são divididas em dois times que utilizam três *threads* cada um.

Código 1.8: Cálculo da Somatória da Variável sum Usando Uma Liga de Times de *Threads*.

```
#pragma omp target teams num_teams(2) thread_limit(3)
{
  int i, N, teams, idteam, idthread;
  int sum;
  N=20;

  #pragma omp distribute parallel for reduction(+:sum)
  for (i=0; i<N; i++) {
    sum += i;
    idthread = omp_get_thread_num();
    idteam = omp_get_team_num();
    teams = omp_get_num_teams();
    printf("␣i␣%d␣n␣%d␣idteam␣%d␣idthread␣%d␣teams␣%d\n", i,N, idteam, idthread, teams);
  }
}
```

O Código 1.9 mostra uma alteração realizada no *offloading* da região de código que realiza a multiplicação de matrizes do Código 1.5, para ser executada em dois times de *threads* com no máximo 30 *threads* em cada time na linha 4 e 5.

Código 1.9: Multiplicação de Matrizes Usando uma Liga de Times de *Threads*.

```
#pragma omp target data device(0) map(i,j,k) map(to:a[0:NUM][0:NUM])\
  map(to:b[0:NUM][0:NUM]) map(tofrom:c[0:NUM][0:NUM])
{
  #pragma omp target teams distribute parallel for collapse (2) \
        num_teams(2) thread_limit(30)
  for(i=0; i<NUM; i++) {
    for(k=0; k<NUM; k++) {
      #pragma omp simd
      for(j=0; j<NUM; j++) {
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
    }
  }
}
```

1.7. Paralelização da Simulação *N-Body* Usando OpenMP 4.0

Nessa Seção será descrito uma paralelização usando OpenMP 4.0 de uma aplicação do mundo real denominada N-Body [CIV 2015], com o objetivo de demonstrar o uso das novas diretivas presentes no OpenMP 4.0, para explorar o uma arquitetura computacional paralelas híbrida composta por processadores multi-core e many-core baseada nas arquiteturas *Intel Xeon* e *Intel Many Integrated Core*.

Uma simulação *N-body* consiste em fazer uma aproximação do movimento de partículas, que interagem entre si de acordo com alguma força física. Tal simulação é muita usada para estudar a movimentação de corpos (*bodies*), tais como, satélites, planetas, estrelas, galaxias, etc, que interagem entre si de acordo com a força da gravidade [AAR 2003].

Para definir a movimentação de corpos em uma simulação *N-body* no contexto apresentado, pode ser utilizada a segunda lei de movimentação Newton, que define a movimentação de corpos da seguintes forma: a somatória das forças aplicada a cada corpo é igual ao produto de sua massa pela aceleração. Além disso, cada par de corpos se atraem segundo uma força proporcional a suas massas, e inversamente proporcional ao quadrado das distâncias entre elas.

A versão paralela da implementação da simulação *N-Body* utiliza diretivas OpenMP para paralelizar o laço que possui o maior custo computacional. Nos laços internos é utilizado diretivas SIMD do OpenMP para compilar versões vetoriais dos laços. Tal implementação permite executar a simulação *N-Body* utilizando todos recursos disponíveis no *host* ou em um *device*. Porém, não é possível utilizar o *host* e os *devices* em conjunto.

A otimização aplicada ao código da simulação *N-Body* para utilizar o *host* e os *devices* em conjunto consiste no uso das diretivas de *offload* do OpenMP, para executar passos do laço que possui o maior custo computacional, no *host* e nos *devices* disponíveis no nó computacional.

A Seção 1.7.1. descreve a versão paralela da simulação *N-Body* e a Seção 1.7.2. descreve as otimizações feitas na versão paralela para balancear a carga de execução entre o *host* e os *devices*.

1.7.1. Implementação Paralela da Simulação *N-Body*

Os corpos da simulação são representados por sete matrizes unidimensionais: a posição nos eixos x,y e z é representando por três matrizes, a velocidade também é representada nos eixos x,y e z por três matrizes e a massa é representa por uma matriz conforme mostrado no algoritmo 1.10.

Código 1.10: Matrizes para Representar os Corpos.

```
1 real *x, *y, *z, *vx, *vy, *vz, *m;
```

O Código 1.11 mostra o laço que executa os passos de tempo da simulação. Para cada passo de tempo a lei de Newton é aplicada à todos os corpos. Como resultado novos valores para posição e velocidade de cada corpo são definidos, a partir da posição e velocidade do passo de tempo anterior.

O Código 1.12 apresenta a implementação do algoritmo para aplicar a lei de Newton aos corpos em uma simulação *N-Body*.

Código 1.11: Laço que Aplica a Função Newton a Cada Passo Temporal.

```
1 for ( int it = 0; it < 100; ++it ) {
2         Newton( n, 0.01 );
3 }
```

Um laço realiza a leitura de todos os corpos para calcular a velocidade no passo de tempo atual Na linha 6. Em cada passo desse laço é calculada a força exercida no corpo atual em relação aos outros corpos da simulação. Esse cálculo é feito em dois laços:

- Na linha 9 um laço realiza o cálculo da força exercida desde o primeiro até o corpo anterior ao corpo atual. Esse cálculo é feito entre as linhas 10 a 15;
- Na linha 18 um laço realiza o cálculo da força exercida desde o próximo corpo após o corpo atual até o último corpo. Esse cálculo é feito entre as linhas 19 a 24, de maneira idêntica ao código da linha de 10 a 15;

Em seguida, a nova velocidade do corpo é calcula entre as linhas 26 a 28.

Após o fim do laço da linha 6 um laço calcula a posição de todos os corpos no passo de tempo atual entre as linhas 31 a 35.

A paralelização do cálculo do Newton é feita dividindo os passos do laço da linha 6 entre todas as *threads* do time. Os laços das linhas 9, 18 e 31 estão anotados para serem compilados usando instruções vetoriais.

Código 1.12: Função Newton.

```
1  void Newton( size_t n, real dt ) {
2    const real dtG = dt * G;
3   #pragma omp parallel
4    {
5      #pragma omp for schedule( auto )
6      for ( size_t i = 0; i < n; ++i ) {
7        real dvx = 0, dvy = 0, dvz = 0;
8        #pragma omp simd
9        for ( size_t j = 0; j < i; ++j ) {
10         real dx = x[j] - x[i], dy = y[j] - y[i], dz = z[j] - z[i];
11         real dist2 = dx*dx + dy*dy + dz*dz;
12         real mOverDist3 = m[j] / (dist2 * Sqrt( dist2 ));
13         dvx += mOverDist3 * dx;
14         dvy += mOverDist3 * dy;
15         dvz += mOverDist3 * dz;
16       }
17       #pragma omp simd
```

```
18        for ( size_t j = i+1; j < n; ++j ) {
19          real dx = x[j] - x[i], dy = y[j] - y[i], dz = z[j] - z[i];
20          real dist2 = dx*dx + dy*dy + dz*dz;
21          real mOverDist3 = m[j] / (dist2 * Sqrt( dist2 ));
22          dvx += mOverDist3 * dx;
23          dvy += mOverDist3 * dy;
24          dvz += mOverDist3 * dz;
25        }
26        vx[i] += dvx * dtG;
27        vy[i] += dvy * dtG;
28        vz[i] += dvz * dtG;
29      }
30      #pragma omp for simd schedule( auto )
31      for ( size_t i = 0; i < n; ++i ) {
32        x[i] += vx[i] * dt;
33        y[i] += vy[i] * dt;
34        z[i] += vz[i] * dt;
35      }
36    }
37  }
```

1.7.2. Implementação Paralela com Balanceamento de Carga da Simulação *N-Body*

A simulação *N-Body* com balanceamento de carga apresenta uma otimização em relação a versão paralela que divide os passos do laço principal da função Newton apresentado no Código 1.12 em *threads* no *host* e em todos os *devices*

O balanceamento é feito dividindo a quantidade de corpos que deve ser calculada em cada passo de tempo entre o *host* e os *devices* da seguinte forma:

- Da linha 1 até a linha 6 os corpos são divididos entre a quantidade de recursos disponíveis *host* e *devices*, definindo um intervalo de corpos no qual o *host* e cada *device* irá aplicação a função Newton.
- O laço que chama a função Newton para cada passo de tempo dentro da função main 1.13 inicia uma região paralela para criar uma *thread* para o *host* e uma *thread* para cada *device* na linha 9.
- As *threads* que vão executar a função Newton em um *device* criam um bloco para definir um fluxo de dados entre *host* e o *device* na linha 12, para copiar as matrizes de corpos para os *devices*.
- Cada *thread* faz uma chamada à função Newton 1.14 passando como parâmetro o intervalo de corpos que irá processar na linha 19.
- Na linha 22 uma região crítica é aberta para que as *threads* executem de modo exclusivo o bloco que transfere as matrizes de corpos do *device* para o *host*.
- Na linha 30 e 35 a diretiva target update device transfere os corpos calculados pelos outros *devices* e pelo *host* para o *device*.

Código 1.13: Divisão da Aplicação da Função Newton Distribuída Entre *host* e *devices*.

```
1   size_t displ[dev+2];
2   displ[0] = 0;
3   for ( int i = 1; i < dev+1; ++i ) {
```

```
4       displ[i] = (i * n) / (dev + 1);
5     }
6     displ[dev+1] = n;
7
8     int dev = omp_get_num_devices();
9     #pragma omp parallel num_threads( dev + 1 )
10    {
11      const int tid = omp_get_thread_num();
12      #pragma omp target data device( tid ) if( tid < dev ) \
13      map( to: x[0:n], y[0:n], z[0:n], vx[0:n], vy[0:n], vz[0:n], m[0:n] )
14      {
15        for ( int it = 0; it < 100; ++it ) {
16          size_t s = displ[tid], l =  displ[tid+1] - displ[tid];
17
18          double tt = omp_get_wtime();
19          Newton( n, s, l, 0.01, tid, dev );
20          tth[tid] = omp_get_wtime() - tt;
21
22          #pragma omp critical
23          {
24            #pragma omp target update device( tid ) if( tid < dev ) \
25            from( x[s:l], y[s:l], z[s:l], vx[s:l], vy[s:l], vz[s:l] )
26          }
27
28          #pragma omp barrier
29
30          #pragma omp target update device( tid ) if( tid < dev ) \
31          to( x[0:s], y[0:s], z[0:s], vx[0:s], vy[0:s], vz[0:s] )
32
33          size_t s1 = s+l, l1 = n-s-l;
34
35          #pragma omp target update device( tid ) if( tid < dev ) \
36          to( x[s1:l1], y[s1:l1], z[s1:l1], vx[s1:l1], vy[s1:l1], vz[s1:l1] )
37
38          #pragma omp single
39          computeDisplacements( displ, tth, dev );
40        }
41      }
42    }
```

Dois aspectos da função Newton 1.14 foram alterados:

- Foi incluída uma diretiva para realizar o *offloading* do bloco de código que calcula a movimentação dos corpos para o *device* na linha 11. A cláusula if foi utilizada para definir uma expressão condicional antes de realizar o *offloading*. Caso o número da *thread* que está executando o *offloading*, seja a última *thread* do time o *offloading* não é realizado e o código é executado no *host*;
- Os laços que calculam as forças exercidas no corpo foram alterados para executarem em um intervalo de corpos definido por duas variáveis chamadas **s** e **l** que são passada como parâmetro à função Newton, nas linhas 7 e 11 respectivamente.

Código 1.14: Função Newton com Balanceamento de Carga.

```
1   void Newton( size_t n, size_t s, size_t l, real dt, int tid, int dev ) {
2     const real dtG = dt * G;
3     #pragma omp target device( tid ) if( tid < dev )
4     #pragma omp parallel
5     {
6       #pragma omp for schedule( auto )
7       for ( size_t i = s; i < s+l; ++i ) {
8         ...
9       }
10      #pragma omp for simd schedule( auto )
11      for ( size_t i = s; i < s+l; ++i ) {
12        ...
13      }
```

```
14    }
15  }
```

1.8. Conclusões

Temos presenciado um rápido aumento na capacidade de processamento paralelo dos aceleradores [LIU 2013], que é um reflexo do aumento no número de núcleos disponíveis nessas arquiteturas.

Arquiteturas de computadores heterogêneas que utilizam CPUs multinúcleos de propósito geral e placas aceleradoras têm se mostrado como uma boa alternativa para a construção de supercomputadores para computação de alto desempenho [LIA 2013].

De fato, na lista top500[6], que apresenta o ranking dos 500 supercomputadores mais poderosos em termos de poder de computação, podemos ver em suas primeiras posições arquiteturas híbridas que utilizam aceleradores.

O modelo de programação OpenMP baseado em diretivas é uma das formas mais largamente adotadas para a implementação do paralelismo em nível de threads [DIE 2014]. Em sua versão 4.0 foram incluídas diretivas que possibilitam a realização do *offload* para placas aceleradoras, provendo mecanismos para a utilização máxima do paralelismo disponível em arquiteturas heterogêneas.

Uma novidade também disponível no OpenMP 4.0 é a capacidade de vetorizar laços, utilizando a diretiva *simd*, de forma que o programa desenvolvido pode utilizar a unidade de vetorização disponível nos mdernos processadores e aceleradores. Assim, o programa desenvolvido pode executar um maior número de instruções pelo mesmo número de ciclos de processador.

Esse minicurso apresentou, de forma introdutória - com exemplos em código fonte de aplicações reais - novas diretivas introduzidas pelo OpenMP 4.0 para realização de *offloading* e melhor uso da vetorização.

1.9. Bibliografia

[AAR 2003] AARSETH, S. J. **Gravitational n-body simulations**. [S.l.]: Cambridge University Press, 2003. Cambridge Books Online.

[BER 2014] BERTOLLI, C. et al. Coordinating gpu threads for openmp 4.0 in llvm. In: LLVM COMPILER INFRASTRUCTURE IN HPC (LLVM-HPC), 2014, 2014. **Anais...** [S.l.: s.n.], 2014. p.12–21.

[CAB 2009] CABRERA, D. et al. Openmp extensions for fpga accelerators. In: SYSTEMS, ARCHITECTURES, MODELING, AND SIMULATION, 2009. SAMOS '09. INTERNATIONAL SYMPOSIUM ON, 2009. **Anais...** [S.l.: s.n.], 2009. p.17–24.

[CIV 2015] CIVARIO, G.; LYSAGHT, M. Chapter 11 - dynamic load balancing using openmp 4.0. In: REINDERS, J.; JEFFERS, J. (Eds.). **High performance**

[6] `http://www.top500.org/`

parallelism pearls: multicore and many-core programming approaches. Boston, MA, USA: Morgan Kaufmann, 2015. v.1, p.185 – 200.

[DIE 2014] LOPES, L. et al. (Eds.). **Euro-par 2014**: parallel processing workshops: euro-par 2014 international workshops, porto, portugal, august 25-26, 2014, revised selected papers, part ii. Cham: Springer International Publishing, 2014. p.291–301.

[FLY 72] FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Transactions on Computers**, v.C-21, n.9, p.948–960, Sept 1972.

[FOR 94] FORUM, M. P. **Mpi**: a message-passing interface standard. Knoxville, TN, USA: [s.n.], 1994.

[HAR 2014] HARODE, A. et al. Optimization of molecular dynamics application for intel xeon phi coprocessor. In: HIGH PERFORMANCE COMPUTING AND APPLICATIONS (ICHPCA), 2014 INTERNATIONAL CONFERENCE ON, 2014. **Anais...** [S.l.: s.n.], 2014. p.1–6.

[HEI 2012] HEINECKE, A.; KLEMM, M.; BUNGARTZ, H. J. From gpgpu to many-core: nvidia fermi and intel many integrated core architecture. **Computing in Science Engineering**, v.14, n.2, p.78–83, March 2012.

[LIA 2013] RENDELL, A. P.; CHAPMAN, B. M.; MÜLLER, M. S. (Eds.). **Openmp in the era of low power devices and accelerators**: 9th international workshop on openmp, iwomp 2013, canberra, act, australia, september 16-18, 2013. proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p.84–98.

[LIU 2013] LIU, X.; MELLOR-CRUMMEY, J.; FAGAN, M. A new approach for performance analysis of openmp programs. In: INTERNATIONAL ACM CONFERENCE ON INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 27., 2013, New York, NY, USA. **Proceedings...** ACM, 2013. p.69–80. (ICS '13).

[MIT 2013] MITRA, G. et al. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and intel platforms. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING, WORKSHOPS AND PHD FORUM, CAMBRIDGE, MA, USA, MAY 20-24, 2013, 2013., 2013. **Anais...** [S.l.: s.n.], 2013. p.1107–1116.

[MIT 2014] MITRA, G. et al. Implementation and optimization of the openmp accelerator model for the TI keystone II architecture. In: USING AND IMPROVING OPENMP FOR DEVICES, TASKS, AND MORE - 10TH INTERNATIONAL WORKSHOP ON OPENMP, IWOMP 2014, SALVADOR, BRAZIL, SEPTEMBER 28-30, 2014. PROCEEDINGS, 2014. **Anais...** [S.l.: s.n.], 2014. p.202–214.

[Ope 2013] OpenMP Architecture Review Board. **OpenMP application program interface version 4.0**.

[RAH 2013] RAHMAN, R. **Intel xeon phi coprocessor architecture and tools**: the guide for application developers. 1st.ed. Berkely, CA, USA: Apress, 2013.

[RAS 2015] RASKULINEC, G. M.; FIKSMAN, E. Chapter 22 - simd functions via openmp. In: REINDERS, J.; JEFFERS, J. (Eds.). **High performance parallelism pearls volume two**: multicore and many-core programming approaches. Boston, MA, USA: Morgan Kaufmann, 2015. v.2, p.421 – 440.

[SIL 2008] SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating system concepts**. 8th.ed. [S.l.]: Wiley Publishing, 2008.

[STO 2013] STOTZER, E. et al. Openmp on the low-power TI keystone II ARM/DSP system-on-chip. In: OPENMP IN THE ERA OF LOW POWER DEVICES AND ACCELERATORS - 9TH INTERNATIONAL WORKSHOP ON OPENMP, IWOMP 2013, CANBERRA, ACT, AUSTRALIA, SEPTEMBER 16-18, 2013. PROCEEDINGS, 2013. **Anais...** [S.l.: s.n.], 2013. p.114–127.

[VAJ 2011] VAJDA, A. **Programming many-core chips**. 1st.ed. [S.l.]: Springer Publishing Company, Incorporated, 2011.