OpenMP

Ricardo de la Rocha Ladeira

Introdução

Na aula passada a aplicação prática de diversos conceitos abordados na disciplina foi vista na construção de uma aplicação de bate-papo. Novamente, aproveitando a base já obtida nas aulas anteriores, especialmente no que foi visto na Aula 4 – *Threads*, dar-se-á sequência no estudo sobre a biblioteca OpenMP, abordando-a com maior profundidade em relação à aula citada.

A proposta deste material não é ser um guia completo de OpenMP, mas trazer as principais diretivas e cláusulas fornecidas por essa API. Assim, para maiores informações e aprofundamento no conhecimento da API, recomenda-se a leitura da documentação oficial. Grande parte deste texto é baseada na especificação do OpenMP [1], que pode ser acessada diretamente em https://www.openmp.org/specifications/.

OpenMP

OpenMP é uma *Application Program Interface* – API composta por um conjunto de diretivas de compilador, rotinas de biblioteca e variáveis de ambiente para paralelismo em programas escritos nas linguagens C, C++ e Fortran, desenvolvida para ser portável entre diferentes arquiteturas [1].

De acordo com a OpenMP, instituições de áreas como engenharia, eletromagnetismo e astrofísica [2], entre outras, utilizam extensivamente a paralelização oferecida pela OpenMP para melhorar o desempenho e a eficiência de suas aplicações.

Diversos compiladores oferecem suporte à API OpenMP, entre eles

- GCC (*GNU Compiler Collection*)
- Clang
- Intel Compiler
- Microsoft Visual C++
- IBM XL C/C++ Compiler
- Oracle Solaris Studio (antigo *Sun Studio*)

- Crays Compiler (Cray C/C++)

Os compiladores que suportam a API OpenMP geralmente incluem opções de linha de comando para ativar ou desativar a interpretação de algumas ou de todas as diretivas OpenMP. Como já visto, para compilar um código C que utilize OpenMP com o GCC, é necessário utilizar o parâmetro `-fopenmp`:

```
> gcc arquivo.c -fopenmp
```

Caso deseje-se ignorar as diretivas OpenMP explicitamente, é necessário compilar o código com o parâmetro `-fno-openmp`:

```
> gcc -fno-openmp arquivo.c
```

É importante mencionar que a OpenMP utiliza *threads* POSIX (*Portable Operating System Interface*) para permitir a execução paralela de tarefas. Isso significa que utilizar o parâmetro `-fopenmp` implica uso implícito de `-pthread` [3]. Além disso, `-fopenmp` implica `-fopenmp-simd`, usada para ativar otimizações de vetorização SIMD (*Single Instruction, Multiple Data*).

SIMD é uma técnica de processamento de dados que envolve a execução de uma única instrução em múltiplos dados simultaneamente. Aproveitar as instruções SIMD disponíveis na arquitetura do processador é uma prática que pode levar a melhorias significativas de desempenho. Essa técnica é especialmente útil em situações em que há muitos dados que podem ser processados de maneira independente. Isso é comum em tarefas como processamento de imagens, áudio, simulações físicas e outras.

Por exemplo, supondo que seja necessário somar o i-ésimo valor de cada vetor, a saber, `A={1,2,3,4}` e `B={5,6,7,8}`. Com uma instrução de soma SIMD pode-se executar a adição de forma paralela, obtendo como resultado `{6, 8, 10, 12}`.

Na arquitetura x86 há um vasto conjunto de instruções SIMD, entre as quais estão `ADDPS`, `SUBPS`, `MULPS`, `SQRTPS` etc [4].

Um exemplo de código em linguagem C demonstrando o uso da instrução SIMD `ADDPS` por meio da instrução `asm` está disponível no **Apêndice I**.

Características

O modelo de programação do OpenMP é de paralelismo explícito, isto é, o programador define por meio de anotações os trechos que devem ser paralelizados. Além disso, a API utiliza o modelo de *memória compartilhada* (ver Aula 3 – Comunicação entre Processos) baseado em *threads*. Assim, todas as *threads* criadas com a OpenMP têm acesso a um local (memória) para armazenar e recuperar variáveis [1].

Outra característica da biblioteca é a flexibilidade. Embora defina internamente uma forma padrão de prover paralelismo, o programador pode, a seu critério, fazer novas definições. Uma delas é a quantidade de *threads*, que pode ser definida por cláusula, função ou variável de ambiente. Exemplos disso (baseados em [5]) estão disponíveis nos arquivos:

- `openMP-quant-threads-1.c`
- `openMP-quant-threads-2.c`
- `openMP-quant-threads-3.c`
- `openMP-quant-threads-4.c`

Os exemplos acima ajudam a ilustrar outra característica do OpenMP: a grande maioria das construções dessa API são aplicadas a blocos estruturados, que são blocos contendo uma ou mais instruções com um ponto de entrada no topo e um ponto de saída[1] na parte inferior [6].

OpenMP permite definir as políticas de compartilhamento de variáveis. Assim, é possível que variáveis sejam compartilhadas por todas as *threads*, por exemplo. Também é possível que elas sejam privadas. Neste caso, cópias das variáveis são feitas para cada *thread* [5]. É possível, inclusive, que em uma mesma rotina existam variáveis compartilhadas e variáveis privadas. A biblioteca OpenMP chama isso de cláusula de atributo de compartilhamento de dados e define, em sua especificação, dez valores diferentes para isso [1].

Sintaxe das Diretivas

As diretivas OpenMP são especificadas com uma *especificação de diretiva*. Uma *especificação de diretiva* consiste no especificador de diretiva acrescido de *cláusulas* que possam opcionalmente ser associadas à diretiva OpenMP:

```
especificador-de-diretiva [[,] cláusula[ [,] cláusula] ...]
```

[1] Porém, não há problema em haver, dentro do bloco, uma chamada à função `exit()` [6].

O especificador da diretiva pode ou não conter argumentos, e sua sintaxe geral é:
`nome-da-diretiva [(argumentos-da-diretiva)]`

Uma diretiva OpenMP sempre será especificada como uma *diretiva pragma*[2] (1) ou um *operador pragma* (2):

(1) **`#pragma omp especificador-de-diretiva`**

(2) **`_Pragma ("omp especificador-de-diretiva")`**

O uso de `omp` é reservado para diretivas OpenMP definidas nesta especificação [1].

Este material abordará algumas das principais diretivas da OpenMP.

Diretiva `parallel`

A diretiva `parallel` define uma região paralela, ou seja, o bloco de código que será executado em paralelo pelas *threads*. Essa diretiva pode ser usada com as diretivas `for` e `section`, bem como com as cláusulas `if`, `private`, `firstprivate`, `default`, `shared`, `copyin`, `reduction` e `num_threads`.

Para usar a diretiva, basta acrescentar o que segue:

```
#pragma omp parallel
{
      /* Bloco de instruções */
}
```

Exemplo simples (arquivo openMP-diretiva-parallel.c):

```
#include <stdio.h>
#include <omp.h>

int main () {
     int tid;

     #pragma omp parallel
     {
```

[2] Diretiva de propósito geral usada para (des)ativar recursos.

```
            tid = omp_get_thread_num();
            printf("Esta é a thread %d.\n", tid);
        }

        return 0;
}
```

Aqui, definiu-se um programa simples em C que utiliza a biblioteca OpenMP para criar uma região paralela. Nessa região, *threads* são criadas para executar o bloco de código dentro do escopo da diretiva `#pragma omp parallel`. Cada *thread* obtém seu próprio identificador de *thread* com a função `omp_get_thread_num()` e imprime esse identificador na saída padrão. Dessa forma, o programa demonstra como várias *threads* podem ser criadas e executadas em paralelo, cada uma realizando uma parte do trabalho.

Diretiva `for`

A diretiva `for` divide o código dentro de um laço de repetição `for` entre as *threads* utilizadas. Essa diretiva pode ser usada em conjunto com a diretiva `parallel`. Ela dá suporte às cláusulas `private`, `firstprivate`, `lastprivate`, `reduction`, `ordered`, `schedule`, `nowait`, sendo que esta última não pode ser usada se for estiver sendo usada em conjunto com `parallel`.

Exemplo simples (arquivo [openMP-diretiva-for.c]):

```
#include <stdio.h>

int main() {
    int i;
    int n = 2000000;
    int dados[n];

    #pragma omp parallel for
    for (i = 0; i < n; i++)
        dados[i] = i;

    #pragma omp parallel for
    for (i = 0; i < n; i++)
        dados[i] = dados[i] * 2;
```

```
    for (i = 0; i < n; i++)
        printf("%d ", dados[i]);

    printf("\n");

    return 0;
}
```

É possível perceber que a diretiva `for` foi utilizada em conjunto com a diretiva `parallel`. Neste exemplo, dois *loops* foram paralelizados, um para inicializar o *array* `dados` e outro para atualizar todos os seus valores com o dobro do valor inicial. Assim, as *threads* disponíveis farão essas atividades paralelamente.

Diretiva `sections`

A diretiva `sections` identifica seções de código a serem divididas entre todas as *threads* disponíveis [7]. Várias seções de código (diretiva `section`) podem ser criadas dentro de um bloco marcado com a diretiva `sections`. Essas seções serão executadas concorrentemente pelas *threads* disponíveis. Cada seção representa um bloco de código que pode ser executado em paralelo, e o número de *threads* disponíveis é distribuído entre as seções para realizar o trabalho. Assim, a diretiva `#pragma omp sections` é usada em conjunto com diretivas `#pragma omp section` para definir as seções individuais.

Exemplo simples (arquivo [openMP-diretivas-sections-section.c]):

```
#include <stdio.h>

int main() {
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                printf("Seção 1 executada por uma thread.\n");
            }

            #pragma omp section
```

```
            {
                printf("Seção 2 executada por outra thread.\n");
            }

            #pragma omp section
            {
                printf("Seção 3 executada por outra thread.\n");
            }

            #pragma omp section
            {
                printf("Seção 4 executada por outra thread.\n");
            }
        }
    }

    return 0;
}
```

Ao executar este código, a ordem de execução de cada `section` não é garantida, mas é garantido que somente uma *thread* acessará cada seção.

Diretiva `single`

A diretiva `single` serve para criar uma região de código em que apenas uma das *threads* deve executar o bloco de código. Isso é particularmente útil quando há uma seção crítica em que apenas uma *thread* deva realizar uma tarefa, como inicializações, atualizações de valores, leituras de arquivos ou outras operações que não podem ser executadas concorrentemente.

Exemplo simples (arquivo [openMP-diretiva-single.c]):

```
#include <stdio.h>

int main() {
    int valor = 7;

    #pragma omp single
    {
```

```
        printf("Uma e somente uma thread entra aqui.\n");
        // Uma e somente uma thread modifica o valor da variável "valor".
        valor++;
    }

    printf("Valor compartilhado: %d\n", valor);

    return 0;
}
```

No exemplo, somente uma *thread* acessa o bloco com a diretiva `single` e atualiza o valor da variável `valor`. É comum esta diretiva estar dentro de um bloco com a diretiva `parallel`, em casos em que há códigos paralelizáveis mas há um código crítico que não pode ser acessado por mais de uma *thread*, como nos exemplos citados anteriormente.

Diretiva `master`

A diretiva `master` especifica um bloco de código que deve ser executado apenas pela *thread* principal (a *thread* com *ID = 0*) em um conjunto de *threads* disponíveis. É importante mencionar que o uso dessa diretiva não fornece garantias automáticas de sincronização entre a *thread* principal e as outras *threads*. Isso quer dizer que as *threads* filhas podem continuar a execução paralela após a diretiva `master` sem aguardar explicitamente pela *thread* principal.

Exemplo simples (arquivo openMP-diretiva-master.c):

```
#include <stdio.h>

int main() {
    int valor = 0;

    #pragma omp parallel
    {
        #pragma omp master
        {
                // Este bloco de código é executado apenas pela thread
principal.
            valor = 3;
        }
```

```
        // Todas as threads podem continuar daqui.
        printf("Valor compartilhado: %d\n", valor);
    }

    return 0;
}
```

Ao executar este código, uma resposta possível é:

```
Valor compartilhado: 3
Valor compartilhado: 3
Valor compartilhado: 0
Valor compartilhado: 0
```

Por que isso acontece? Como mencionado, não há garantias de que as *threads* aguardarão a conclusão dos trabalhos da *thread* ancestral.

Diretiva `critical`

A diretiva `critical` especifica um bloco de código que deve ser executado por uma única *thread* por vez, garantindo a exclusão mútua para proteger seções críticas de código.

Exemplo simples (arquivo [openMP-diretiva-critical.c]):

```
#include <stdio.h>
#include <omp.h>

#define TAM_MAX_FILA 100

int fila[TAM_MAX_FILA];
int inicio = -1;
int fim = -1;

void enfileira(int item) {
    #pragma omp critical
    {
        if (fim == TAM_MAX_FILA - 1)
            printf("Operação negada: fila cheia.\n");
        else {
```

```
            if (inicio == -1)
                inicio = 0;

            fim++;
            fila[fim] = item;
            printf("Enfileirado: %d\n", item);
        }
    }
}

int desenfileira() {
    int item = -1;
    #pragma omp critical
    {
        if (inicio == -1)
            printf("Operação negada: fila vazia.\n");
        else {
            item = fila[inicio];
            if (inicio == fim)
                inicio = fim = -1;
            else
                inicio++;

            printf("Desenfileirado: %d\n", item);
        }
    }
    return item;
}

int main() {

    #pragma omp parallel
    {
        int threadID = omp_get_thread_num();

        for (int i = 1; i <= 3; i++) {
             enfileira(i * (threadID + 1)); // Enfileira três elementos por
thread
            int item = desenfileira(); // Desenfileira um elemento
        }
    }
```

```
    return 0;
}
```

Neste exemplo, a diretiva `critical` garante que as operações de enfileiramento e desenfileiramento sejam realizadas de forma segura, evitando problemas de concorrência em uma fila compartilhada por várias *threads*. Cada *thread* enfileira três elementos e, em seguida, desenfileira um elemento. A diretiva `critical` garante que apenas uma *thread* execute cada operação crítica por vez, permitindo a operação segura em uma estrutura de dados compartilhada.

Diretiva `barrier`

A diretiva `barrier` cria um ponto de sincronização em que todas as *threads* esperam até que todas tenham alcançado o ponto de barreira. Uma barreira de sincronização, em computação paralela, é justamente um mecanismo que força um conjunto de *threads* ou processos a esperar até que todos tenham alcançado um determinado ponto de execução antes de continuar. Ela é uma ferramenta útil para coordenar a execução paralela (assim como as estratégias vistas na Aula 5 – Sincronização), garantindo que todas as *threads* ou processos atinjam um estado específico antes de prosseguir.

Exemplo simples (arquivo `openMP-diretiva-barrier.c`):

```
#include <stdio.h>

#define TAM 5

int main( ) {
    int a[TAM], i;

    #pragma omp parallel
    {
      #pragma omp for
        for (i = 0; i < TAM; i++)
            a[i] = i * i;

        #pragma omp master
        {
              printf("Antes:\n");
```

```
            for (i = 0; i < TAM; i++)
                printf("a[%d] = %d\n", i, a[i]);
        }

        #pragma omp barrier

        #pragma omp for
        for (i = 0; i < TAM; i++)
            a[i] += i;
    }

    printf("Depois:\n");
    for (i = 0; i < TAM; i++)
      printf("Valor[%d] = %d\n", i, a[i]);
}
```

O código utiliza algumas diretivas já vistas: `for`, `master`, `parallel` e agora `barrier`. A barreira (`barrier`) foi inserida logo após a impressão dos valores, que sucede a inicialização deles. Essa barreira é importante para garantir que o laço seguinte, que atualiza os valores do *array*, seja acessado somente após todas as *threads* terem concluído as atividades anteriores. Sugere-se executar novamente o código comentando o trecho que contém a diretiva da barreira (`#pragma omp barrier`), observando os resultados. É possível que após algumas execuções os valores sejam diferentes devido à falta de um mecanismo de sincronização.

Diretiva `atomic`

A diretiva `atomic` é usada para garantir que uma operação em uma variável compartilhada seja executada atomicamente, evitando condições de corrida.

Exemplo simples (arquivo openMP-diretiva-atomic.c, baseado em [7]):

```
#include <stdio.h>

#define MAX 10

int main() {
      int contador = 0;
```

```
	#pragma omp parallel num_threads(MAX)
	{
		#pragma omp atomic
		contador++;
	}

	printf("Quantidade de threads: %d\n", contador);

	return 0;
}
```

No exemplo, o contador é incrementado atomicamente por cada uma das dez *threads* definidas. As *threads* atuam uma de cada vez. Portanto, o incremento de contador é feito de forma segura e sem condições de corrida. Se a linha `#pragma omp atomic` for comentada (recomenda-se fazer esse teste!), várias *threads* podem tentar incrementar a variável `contador` simultaneamente. Isso pode criar condições de corrida, com mais de uma *thread* tentando ler, incrementar e gravar o valor do contador ao mesmo tempo. Isso pode resultar em uma variabilidade nos resultados, onde algumas *threads* podem sobrescrever o trabalho de outras, causando valores diferentes para contador.

Diretiva `flush`

A diretiva `flush` é usada para forçar a atualização das variáveis compartilhadas na memória, garantindo a consistência dos dados entre *threads*. Ela recebe como parâmetro uma lista de variáveis, separadas por vírgulas, que representam objetos a serem sincronizados. Se não houver especificação de variável(is), toda a memória é liberada.

Exemplo simples (arquivo openMP-diretiva-flush.c):

```
#include <stdio.h>
#include <omp.h>

void le(int *dado) {
	printf("lê o dado\n");
	*dado = 1;
}

void processa(int *dado) {
	printf("processa o dado\n");
```

```
	(*dado)++;
}

int main() {
	int dado;
	int flag = 0;

	#pragma omp parallel sections num_threads(2)
	{
		#pragma omp section
		{
			printf("Thread %d: ", omp_get_thread_num( ));
		le(&dado);
		#pragma omp flush(dado)
		flag = 1;
		#pragma omp flush(flag)
	}

	#pragma omp section
	{
		while (!flag) {
			#pragma omp flush(flag)
		}
		#pragma omp flush(dado)

		printf("Thread %d: ", omp_get_thread_num( ));
		processa(&dado);
		printf("dado = %d\n", dado);
	}
    }

    return 0;
}
```

O programa principal cria duas seções paralelas usando a diretiva `#pragma omp parallel sections`, onde cada seção é executada por uma *thread* diferente. Na primeira seção paralela, uma *thread* lê a variável `dado` chamando a função `le`. Em seguida, ela define a variável `flag` como `1`, indicando que a leitura está completa. A diretiva `flush` é usada após cada uma dessas operações para garantir que as alterações nas variáveis sejam visíveis para as outras *threads*. Essas diretrizes de `flush` são importantes para

evitar otimizações do compilador que poderiam reordenar as operações ou manter as variáveis em registradores, tornando as alterações não visíveis para outras *threads*. Na segunda seção paralela, outra *thread* aguarda até que a variável `flag` seja definida como `1`, indicando que a leitura da variável `dado` na primeira seção está completa. Isso é feito com um *loop* que verifica constantemente o valor de `flag` e usa a diretiva `flush(flag)` para garantir que a leitura de `flag` seja consistente entre as *threads*. Após a confirmação de que a leitura está completa (ou seja, `flag` é definida como `1`), a segunda *thread* lê a variável `dado` e, em seguida, chama a função `processa` para incrementar o valor de `dado`. Em seguida, ela imprime o valor atualizado de `dado`.

Diretiva `ordered`

A diretiva `ordered` garante a execução ordenada de regiões paralelas, onde as *threads* executam o código em ordem específica. Essa diretiva precisa estar dentro da extensão dinâmica de um constructo `for` ou `parallel for` com uma cláusula `ordered` [7].

Exemplo simples (arquivo openMP-diretiva-ordered.c):

```
#include <stdio.h>
#include <omp.h>

int main() {
      int i;

    #pragma omp parallel num_threads(4)
    {
      #pragma omp for ordered
        for (i = 1; i <= 10; i++) {
            #pragma omp ordered
            printf("Thread %d, Iteração %d\n", omp_get_thread_num(), i);
        }
    }

    return 0;
}
```

Nesse exemplo, o uso da diretiva `ordered` garante que a *thread* 0 executará suas iterações antes da *thread* 1, que, por sua vez, executará suas iterações antes da *thread* 2, e

assim por diante. Ao remover essa diretiva, percebe-se que a ordem de execução passa a ser indefinida.

Diretiva `threadprivate`

Especifica que uma variável é privada para um *thread* [7]. Essa diretiva é utilizada da seguinte maneira:

```
#pragma omp threadprivate(var)
```

Onde var é o nome da variável (ou uma lista de variáveis separadas por vírgula) que se deseja tornar privada para um *thread*. Na linguagem C, a variável **precisa ser global** ou **estática**.

Um exemplo simples (arquivo [openMP-diretiva-threadprivate.c]):

```
#include <stdio.h>
#include <omp.h>

int global;

#pragma omp threadprivate(global)

int main() {
	global = 0;

	#pragma omp parallel num_threads(4)
	{
	// Cada thread terá sua própria cópia da variável "global"
	global += omp_get_thread_num();
	printf("Thread %d - Valor da variável global: %d\n",
omp_get_thread_num(), global);
	}

	// O valor original não é alterado
	printf("Valor da variável global: %d\n", global);

	return 0;
}
```

No exemplo, a variável `global` é de escopo global. A diretiva `threadprivate` faz com que a variável seja privada à *thread*. A cláusula `num_threads` foi usada para especificar o número de *threads* a serem usadas na região paralela. Cada *thread* adiciona seu número identificador à variável `global` e o valor da variável é exibido para cada *thread*. Finalmente, o valor original da variável é exibido, e não estará alterado.

Cláusulas de Controle de Variáveis

Com OpenMP é possível controlar o escopo das variáveis na região paralela (na diretiva `parallel`) por meio de cláusulas. Algumas delas são:

- `default`: assume valor predeterminado na especificação do OpenMP [1]. Por exemplo, objetos com duração de armazenamento dinâmica são compartilhados (`shared`).
- `private`: as variáveis citadas ficam privadas a cada *thread*.
- `shared`: as variáveis citadas são compartilhadas por todas as *threads*.

Exemplo:

```
#pragma omp parallel for private(i, j, k) shared(A, B, C)
```

Nessa diretiva, tem-se a paralelização de um laço de repetição `for` com variáveis `i`, `j` e `k` sendo privadas e `A`, `B` e `C` sendo compartilhadas.

Outras Cláusulas

OpenMP oferece cláusulas e diretivas além das abordadas neste material. Algumas das outras cláusulas disponíveis são:

- `reduction`: Realiza operações de redução em variáveis compartilhadas. Uma operação de redução é uma operação que combina os valores de várias *threads* em uma única *thread*, geralmente realizando uma operação associativa e comutativa, como soma, produto, mínimo, máximo, entre outras.
- `nowait`: Indica que as *threads* não devem esperar umas pelas outras ao sair de uma região paralela. Normalmente, quando uma região paralela é executada, as *threads* aguardam umas pelas outras ao final dessa região, garantindo que todas tenham concluído antes de continuar a execução fora da região. No entanto, em certos casos, pode ser desejável que as *threads* não esperem, principalmente

quando cada *thread* está trabalhando em tarefas independentes e não precisa compartilhar resultados ou coordenar entre si no final da região paralela. A cláusula `nowait` é usada para sinalizar essa intenção.

- `copyin`: Copia o valor de uma variável de um *thread* mestre para todas as outras *threads*. Isso é útil quando deseja-se que todas as *threads* tenham acesso ao mesmo valor inicial de uma variável, mesmo que ela não seja compartilhada.
- `if`: Especifica uma condição para a execução paralela de uma região. Ela permite controlar dinamicamente a execução paralela com base em uma condição em tempo de execução. Se a condição especificada na cláusula `if` for avaliada como verdadeira, a região paralela será executada; caso contrário, ela será ignorada. Um exemplo está disponível no arquivo `openMP-clausula-if.c`.
- `num_threads`: Define o número de *threads* a serem usadas em uma região paralela. Já foi utilizada em alguns códigos apresentados neste material (exemplo: `openMP-diretiva-atomic.c`).
- `schedule`: Controla a forma como as iterações de um *loop* são distribuídas (ou *agendadas*) entre as *threads* em uma região paralela. Ela determina como as iterações são agendadas e divididas entre as *threads*, afetando o equilíbrio de carga e o desempenho da paralelização. Um exemplo está disponível no arquivo `openMP-clausula-schedule.c`. Neste exemplo, utilizou-se o agendamento dinâmico (`dynamic`), que distribui as iterações de forma dinâmica entre as *threads*. Isso significa que as iterações do *loop* na região paralela são alocadas para as *threads* conforme necessário. Algumas iterações podem ser muito mais demoradas do que outras, o que favorece o agendamento dinâmico. Neste caso, as *threads* mais rápidas podem pegar novas iterações para processar, enquanto as mais lentas trabalham nas iterações anteriores. Como dito, isso ajuda a equilibrar a carga de trabalho entre as *threads* e a melhorar o desempenho geral da execução paralela.

Multiplicação de Matrizes

A multiplicação de matrizes é uma operação de álgebra linear onde duas matrizes são combinadas para produzir uma terceira matriz. Ela é definida para matrizes retangulares para as quais a quantidade de linhas de pelo menos uma delas deve ser igual à quantidade de colunas da outra matriz. Em outras palavras, dadas duas matrizes *A* e *B*, *A* precisa ter dimensões *m* x *n* e *B* precisa ter dimensões *n* x *p*. Assim, a matriz resultante terá dimensões *m* x *p*.

Para construir uma matriz resultante (matriz *C*) dessa multiplicação, é necessário calcular os elementos C_{ij} como a multiplicação de cada elemento da linha *i* da matriz *A* pelo elemento correspondente da coluna *j* da matriz *B*, e então esses produtos são somados. Em outras palavras:

$$C_{ij} = A_{i1} \times B_{1j} + A_{i2} \times B_{2j} + ... + A_{in} \times B_{nj}$$

A multiplicação de matrizes é uma operação fundamental em várias áreas, incluindo Álgebra Linear, Estatística, Física, Engenharia e, claro, Ciência da Computação. Na Ciência da Computação, diversas aplicações dependem dessa operação, entre elas as que fazem uso de compressão de imagens e aprendizagem de máquina.

Dada a importância e a aplicabilidade dessa operação na Ciência da Computação, a implementação de um código para multiplicação de matrizes é um exemplo importante para demonstrar o uso do paralelismo com a API OpenMP. No código implementado e apresentado a seguir, duas matrizes de tamanho 1000 x 1000 são declaradas e preenchidas com valores para depois serem multiplicadas.

O código de multiplicação de matrizes utilizando OpenMP, exposto na sequência, também está disponível no arquivo openMP-multiplicação-de-matrizes.c.

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

#define TAM 1000  // Tamanho das matrizes

int main() {
    int i, j, k;
    // Alocação de memória dinamicamente para a matriz a fim de evitar
estouro de pilha
    float **A = (float **)malloc(TAM * sizeof(float *));
    float **B = (float **)malloc(TAM * sizeof(float *));
    float **C = (float **)malloc(TAM * sizeof(float *));
    for (i = 0; i < TAM; i++) {
        A[i] = (float *)malloc(TAM * sizeof(float));
        B[i] = (float *)malloc(TAM * sizeof(float));
        C[i] = (float *)malloc(TAM * sizeof(float));
    }
```

```
    // Inicialização das matrizes
    for (i = 0; i < TAM; i++)
        for (j = 0; j < TAM; j++) {
            // Inicializa A com a soma de i e j, e B com a diferença de i e
j
            A[i][j] = i + j;
            B[i][j] = i - j;
        }

    #pragma omp parallel for private(i, j, k) shared(A, B, C)
    // Loop paralelo para a multiplicação de matrizes usando OpenMP
    /*
    - #pragma omp: A diretiva de pré-processamento que indica que o código
contém instruções do OpenMP.
    - parallel: Indica que a região de código a seguir será executada por
várias threads simultaneamente.
    - for: Especifica que um loop será paralelizado.
    - private(i, j, k): Define as variáveis i, j e k como variáveis
privadas para cada thread. Isso significa que cada thread
        terá sua própria cópia dessas variáveis e elas não interferirão nas
variáveis com o mesmo nome em outras threads.
    - shared(A, B, C): Indica que as variáveis A, B e C são compartilhadas
entre todas as threads. Portanto, todas as threads
        podem acessar e modificar essas variáveis.
    */
    for (i = 0; i < TAM; i++)
        for (j = 0; j < TAM; j++) {
            C[i][j] = 0.0;
            for (k = 0; k < TAM; k++)
                C[i][j] += A[i][k] * B[k][j];
        }

    // Exibindo a matriz resultante
    printf("Resultado da multiplicacao:\n");
    for (i = 0; i < TAM; i++) {
        for (j = 0; j < TAM; j++)
            printf("%.2f ", C[i][j]);
        printf("\n");
    }

    // Libera a memória alocada
```

```
    for (i = 0; i < TAM; i++) {
        free(A[i]);
        free(B[i]);
        free(C[i]);
    }
    free(A);
    free(B);
    free(C);

    return 0;
}
```

Para testar o código, a proposta é compilá-lo com e sem o parâmetro `-fopenmp`, executar o código nas duas situações e analisar o tempo de execução do programa com a ferramenta `time`[3].

Primeiro teste sugerido:

```
> gcc openMP-multiplicação-de-matrizes.c
> for i in {0..20}; do time -p ./a.out > saida_sem_openMP_$i; done
```

Exemplo de resposta (parcial) obtida na execução do código do primeiro teste:

```
real 18,66
user 18,50
sys 0,14
real 17,90
user 17,76
sys 0,13
real 17,84
```

[3]O comando time fornece três medidas: `real`, `user` e `sys`.

– `real` (tempo real): tempo "real" (de relógio) que o programa leva para ser executado, desde o início até o término. Inclui os tempos de CPU e de espera (por exemplo de operações de E/S).

– `user` (tempo do usuário): tempo que a CPU gasta executando o código do programa. Inclui apenas o tempo que a CPU passa executando instruções no espaço de usuário, desconsiderando o tempo de operações do sistema operacional.

– `sys` (tempo do sistema): tempo que a CPU gasta executando chamadas de sistema em nome do seu programa. Inclui o tempo que a CPU passa executando operações do sistema operacional em resposta às chamadas do programa. Isso pode incluir operações de E/S, gerenciamento de memória e outros serviços do sistema operacional.

```
user 17,65
sys 0,18
(...)
```

Segundo teste sugerido:

```
> gcc openMP-multiplicação-de-matrizes.c -fopenmp
> for i in {0..20}; do time -p ./a.out > saida_com_openMP_$i; done
```

Exemplo de resposta (parcial) obtida na execução do código do segundo teste:

```
real 7,79
user 26,95
sys 0,15
real 7,76
user 26,85
sys 0,20
real 8,03
user 26,87
sys 0,19
(...)
```

Cabe citar que os exemplos foram realizados de uma máquina com processador Intel® Core™ i3-2310M CPU @ 2.10GHz × 4 e executados em modo gráfico, sem qualquer controle sobre outros processos em execução e que eventualmente poderiam interferir diretamente no resultado. No entanto, cabem algumas observações sobre os resultados.

É possível notar que o tempo real do primeiro exemplo é superior ao tempo real do segundo exemplo. Isso significa que, ao cronometrar o tempo de relógio de ambos os códigos, obteve-se um tempo inferior no código paralelizado.

Outro ponto interessante diz respeito ao tempo de usuário do segundo exemplo. Ele é maior que o tempo real. Como isso pode acontecer?

Um programa paralelo pode utilizar mais tempo de CPU (tempo do usuário) e ainda ser mais rápido em termos de tempo de relógio (tempo real). Isso ocorre porque o tempo de CPU é uma métrica que se refere à quantidade de tempo que as CPUs realmente gastam processando as instruções do programa, enquanto o tempo de relógio é a duração total

desde o início até a conclusão do programa, incluindo qualquer espera ou atividade em segundo plano.

Resumo

Esta aula aprofundou os conhecimentos na API OpenMP, vista de forma resumida em aulas anteriores. As principais diretivas dessa API foram definidas e exploradas com exemplos práticos, permitindo melhor compreensão sobre suas funções. Por fim, um exemplo de multiplicação de matrizes foi fornecido e discutido, buscando trazer um caso real de um algoritmo fundamental para diversas aplicações reais que, se paralelizado (com OpenMP, por exemplo), pode trazer ganhos significativos em termos de tempo real.

Exercícios

1) No exemplo de multiplicação de matrizes, como paralelizar com OpenMP o trecho abaixo? Essa mudança traz benefícios significativos no desempenho do programa? Justifique.

```
// Inicialização das matrizes
for (i = 0; i < TAM; i++)
    for (j = 0; j < TAM; j++) {
        // Inicializa A com a soma de i e j, e B com a diferença de i e j
        A[i][j] = i + j;
        B[i][j] = i - j;
    }
```

2) Considerando o código de multiplicação de matrizes, refaça os testes com e sem o uso de OpenMP em cada uma das situações abaixo e interprete os resultados.
    a) Com um valor pequeno para `TAM` (ex: 50)
    b) Com um valor grande para `TAM` (ex: 3000)
3) (**EXTRA**) Conjunto de Mandelbrot é um fractal conhecido na área da matemática. Ele foi descoberto e nomeado em homenagem ao matemático Benoit Mandelbrot, que o estudou extensivamente. O Conjunto de Mandelbrot é uma estrutura geométrica complexa que se forma a partir de iterações matemáticas. Esse fractal é criado através de uma equação iterativa envolvendo números complexos. A definição matemática do Conjunto de Mandelbrot é a seguinte:

$$Z_{n+1} = Z^2_n + C$$

Z é um número complexo, *C* é uma constante complexa, e *n* é um número inteiro não negativo que representa o número de iterações. O Conjunto de Mandelbrot é formado por todos os valores de *C* para os quais a sequência gerada por essa equação não diverge para o infinito, ou seja, os valores de *Z* permanecem limitados em magnitude durante iterações infinitas. Implemente o Conjunto de Mandelbrot utilizando OpenMP.

Gabarito

1) É possível criar duas rotinas com laços aninhados, uma para inicializar os valores da matriz *A* e outra para a inicialização da matriz *B*. Assim, o preenchimento de cada matriz poderia ser paralelizado. O código ficaria da seguinte maneira:

```
#pragma omp parallel for private(i, j) shared(A)
 for (i = 0; i < TAM; i++)
     for (j = 0; j < TAM; j++) {
         A[i][j] = i + j;
     }

 #pragma omp parallel for private(i, j) shared(B)
 for (i = 0; i < TAM; i++)
     for (j = 0; j < TAM; j++) {
         B[i][j] = i - j;
     }
```

   Essa solução contém o dobro de laços, mas também utiliza as *threads* disponíveis pelo processador. Ao executar o código, não houve mudança significativa no desempenho. Uma das justificativas para isso é o *overhead* que naturalmente ocorre pelo paralelismo, pois introduzir *threads* para tarefas simples exigirá, de qualquer forma, que existam mecanismos de gerenciamento e sincronização. Esse custo de criação e coordenação de *threads* pode superar os benefícios da paralelização em tarefas pequenas ou em laços aninhados simples.
2) Com tamanho pequeno (ex: *TAM = 50*) os tempos obtidos foram muito parecidos, indicando que o paralelismo não influenciou positivamente o desempenho. Dessa forma, se somente valores pequenos forem utilizados, não é vantajoso paralelizar, pois o custo dessa ação não compensa o desempenho obtido. Com tamanho grande (ex: *TAM = 3000*) os resultados obtidos com paralelismo foram aproximadamente a metade dos valores obtidos com a solução sequencial. Dessa forma, tamanhos

maiores de matrizes permitem que o paralelismo atue de forma mais significativa no desempenho dos códigos.

3) Implementações do Conjunto de Mandelbrot com OpenMP e sem OpenMP.

Referências

[1] OPENMP. OpenMP 5.2 API Syntax Reference Guide. Disponível em: https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf. Acesso em: 18 ago. 2023.

[2] OPENMP. Who's Using OpenMP? Disponível em: https://www.openmp.org/about/whos-using-openmp/. Acesso em: 18 out. 2023.

[3] GCC TEAM. Options Controlling C Dialect. GCC Online Documentation. 2011. Disponível em: https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html. Acesso em: 14 out. 2023.

[4] INTEL. Intel® Processor Architecture: SIMD Instructions. Disponível em: https://www.intel.sg/content/dam/www/public/apac/xa/en/pdfs/ssg/Intel_Processor_Architecture_SIMD_Instructions.pdf. Acesso em: 15 out. 2023.

[5] SILVA, Fernando. Introdução ao OpenMP. Disponível em: http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/intro_openmp-Fernando-Silva.pdf. Acesso em: 21 out. 2023.

[6] MATTSON, Tim. A "Hands-on" Introduction to OpenMP. Disponível em: https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf. Acesso em: 22 out. 2023.

[7] MICROSOFT. Diretivas (OpenMP). Disponível em: https://learn.microsoft.com/pt-br/cpp/parallel/openmp/reference/openmp-directives. Acesso em: 27 out. 2023.

Apêndice I – Exemplo de uso de `ADDPS`

```
#include <stdio.h>

int main() {

    float a [] = {1.1f, 1.2f, 1.3f, 1.4f};
    float b [] = {2.1f, 2.2f, 2.3f, 2.4f};
    float c [] = {0.0f, 0.0f, 0.0f, 0.0f};

    printf("%f %f %f %f\n", a[0], a[1], a[2], a[3]);
    printf("%f %f %f %f\n", b[0], b[1], b[2], b[3]);
    printf("%f %f %f %f\n", c[0], c[1], c[2], c[3]);

    asm (   "movaps %1, %%xmm0\n\t" // mover a para xmm0
            "movaps %2, %%xmm1\n\t" // mover b para xmm1
            "addps %%xmm0, %%xmm1\n\t" // somar jogando em xmm0
            "movaps %%xmm1, %0\n\t" // mover xmm1 para c
            : "=m" (c)
            : "m" (a), "m" (b)
    );

    printf("%f %f %f %f\n", a[0], a[1], a[2], a[3]);
    printf("%f %f %f %f\n", b[0], b[1], b[2], b[3]);
    printf("%f %f %f %f\n", c[0], c[1], c[2], c[3]);

    return 0;
}
```

Autor do código: Eder Augusto Penharbel