Alessandra Chan - *3308993*

Luis Felipe Martins Linhares - *3348507*

Murilo Carneiro Rodrigues - *3309121*
Rafael Soares Camargo - *3309375*

1. Resumo

Nesta monografia será apresentada a API OpenMP. Esta API é um importante auxílio àqueles que desejam criar aplicações para máquinas paralelas de memória compartilhada.

Também serão apresentados conceitos básicos para o entendimento da linguagem, como threads e memória compartilhada.

A motivação deste trabalho não é ensinar como implementar programas usando OpenMP e sim demonstrar conceitos sobre esta. Assim como mostrar os objetivos da linguagem, uma visão geral e comparação entre OpenMP e outras linguagens que se encaixam no âmbito de programação concorrente.

2. Introdução

2.1 **Alternativas para a programação paralela**

A programação paralela recorre à utilização de muitos conceitos como comunicação, sincronização e alocação de processadores, na implementação de algoritmos. Neste caso são necessárias ferramentas que suportem a manipulação desses conceitos. Algumas soluções são:

a. A utilização de uma nova linguagem;
b. A utilização de uma linguagem seqüencial modificada para lidar com o paralelismo;
c. O uso de compiladores que exploram a paralelização;
d. O uso de rotinas de biblioteca/diretivas de compilação em uma linguagem seqüencial existente.

No último item podem ser citados dois exemplos: para processadores com memória distribuída, a utilização de MPI e para processadores com memória compartilhada, a utilização de OpenMP, tema abordado nessa monografia.

2.2 **Alguns Conceitos Importantes**

2.2.1 **Memória Compartilhada**

OpenMP é um padrão usado em sistemas baseados em memória compartilhada. Nestes sistemas, a memória pode ser totalmente acessada por qualquer processador do sistema.

O tempo de acesso à memória pode não ser igual para todos os processadores. E o uso de memória compartilhada não garante a redução do tempo de uso da unidade de processamento.

A execução de programas paralelos em memória compartilhada, consiste na geração de múltiplos threads que executam de forma paralela. O processo de criação e sincronização de threads não é uma tarefa trivial ao programador. O número de threads usados no paralelismo não depende do número do de processadores.

2.2.2 **Threads**

Threads, também conhecidos como processos leves, consistem na unidade básica de utilização da Unidade Central de Processamento (UCP). Ao contrário de processos, threads contêm apenas o contador de programas (PC), um conjunto de valores de registradores e a pilha.

Todos os threads criados pelo mesmo processo pai compartilham o mesmo espaço de endereçamento. Além disso, threads possuem um baixo custo na criação e destruição, favorecendo a associação dos mesmos a múltiplos processadores de forma eficiente. Este processo consiste na base do paralelismo SMP (Shared-Memory Programming).

Abaixo está a Figura 1 que representa como são compartilhadas as informações entre os threads. Mostra que cada thread acessa seus próprios dados privados, assim como os dados compartilhados pelos mesmos. Será feita uma abordagem sobre dados compartilhados e privados mais completa adiante neste material.

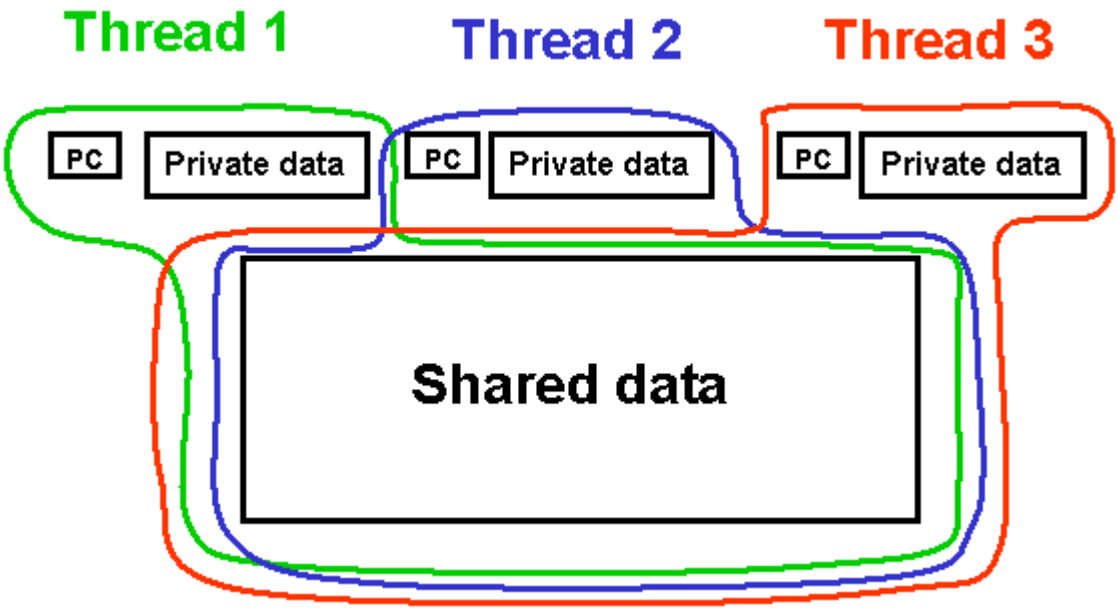

Figura 1: Representação de threads

3. OpenMP

OpenMP é uma API (application program interface) usada para paralelismo multi-thread em sistemas baseados em memória compartilhada. Esta API é formada por três componentes primários: Diretivas de Compilação, Biblioteca de Rotinas e Variáveis de Ambiente.

OpenMP é especificada para C/C++ e Fortran. Muitas plataformas possuem implementações desta API, incluindo a maior parte das plataformas UNIX e WindowsNT.

Com esta ferramenta, a programação de programas paralelos é facilitada, pois o programador não necessita fornecer detalhes sobre como será feita a paralelização (nível mais alto de abstração na utilização de threads), o que é uma característica mais atraente do que quando são usados threads de maneira explícita. Usa-se uma linguagem seqüencial modificada para se obter o paralelismo.

O uso de OpenMP também é mais vantajoso do que o uso de MPI em sistemas de memória compartilhada, pela facilidade de uso. Porém seu uso não garante a utilização mais eficiente da memória compartilhada.

4. Histórico

A primeira tentativa de se padronizar diretivas para o uso em máquinas com memória compartilhada foi o ANSI X3H5, em 1994. Este nunca foi adotado pelo fato de que nesta época havia um grande interesse em máquinas de memória distribuída, que se tornavam muito populares.

Portanto, em 1997, foi divulgada a especificação do padrão OpenMP, que é baseado no padrão X3H5. Porém, nesta época as novas arquiteturas de máquinas de memória compartilhadas estavam em evidência.

OpenMP foi desenvolvido e endossado por um grupo composto pelos maiores fabricantes de software e hardware. Almeja-se que este se torne um padrão ANSI no futuro. As especificações são construídas através de um processo colaborativo de trabalho entre as partes interessadas no mesmo, que são a indústria de hardware e software, governo e universidades.

Dentre os diversos parceiros, estão: Compaq/Digital, Hewlett-Packard, Intel, IBM, KAI, Silicon Graphics, Sun entre outros...

5. Objetivos

Padronização

> Prover uma padronização entre uma grande variedade de arquiteturas baseada em memória compartilhada.

Facilidade de Uso

Estabelecer um conjunto simples e limitado de diretivas para a programação paralela, fazendo com que um código paralelo significante necessite o uso de, no máximo, 3 ou 4 diretivas.

Prover a capacidade para o programador paralelizar uma parte do código de forma incremental.

Suportar paralelismo de granulosidade fina ou grossa.

Portabilidade

Suportar Fortran (77,90 e 95) e C/C++, além de várias arquiteturas como as *Unix-like* e Windows NT.

6. Visão Geral

6.1 **Diretivas e Sentinelas**

As diretivas são linhas especiais de código em linguagens de programação que dão inúmeros recursos ao programador. As diretivas são identificadas por sentinelas.

Na API OpenMP as diretivas são diferenciadas com a indicação de sentinelas no início da linha da diretiva. Na linguagem C/C++, as diretivas de OpenMP se iniciam com a seguinte sentinela: *#pragma omp*. Já na linguagem Fortran, a sentinela é a seguinte: *!$OMP*. Com isso o compilador saberá identificar a diretiva da maneira correta.

A seguir a Figura 2 indica as diretivas da API e suas respectivas cláusulas, que serão melhor abordadas adiante.

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Figura 2: Diretivas e suas classes

Vale lembrar que algumas diretivas não estão sendo abordadas na figura acima, pois não possuem clausulas ao serem chamadas. São estas: *MASTER, CRITICAL, BARRIER, ATOMIC, FLUSH, ORDERED* e *THREADPRIVATE*. Com estas, completa-se o conjunto de todas as diretivas reconhecidas pela API OpenMP.

As diretivas possuem o seguinte formato, para implementação em C++:

#pragma omp	**[nome_diretiva]**	**[clausula,...]**	**novalinha**
Obrigatório para diretivas em C++	Nome da diretiva OpenMP válida	Opcional. Podem ser em qualquer ordem	Necessária. Procede ao bloco referenciado por esta diretiva

Exemplos:

```
#pragma omp parallel default(shared) private(beta,pi)
#pragma omp critical
```

6.2 **Variáveis de Ambiente**

OpenMP provê quatro variáveis de ambiente que auxiliam na execução do código paralelo. O nome das variáveis de ambiente são todos escritos em letras maiúsculas. As variáveis são:

OMP_SCHEDULE

Esta variável aplica-se apenas para *do* e *for* – *PARALLEL DO* (Fortran) e *parallel for* (C/C++) – que possuem a clausula *schedule* definida como *RUNTIME*. O valor desta variável determina como as iterações do loop são escalonadas nos processadores.

Exemplos:

setenv OMP_SCHEDULE "guided, 4"

setenv OMP_SCHEDULE "dynamic"

OMP_NUM_THREADS

O valor desta variável define o número máximo de threads utilizados durante a execução do código paralelo.

Exemplo:

setenv OMP_NUM_THREADS 10

OMP_DYNAMIC

Esta variável serve para habilitar ou desabilitar o do número de threads disponíveis para a execução em regiões paralelas.

Exemplo:

setenv OMP_DYNAMIC TRUE

OMP_NESTED

Esta variável serve para habilitar ou desabilitar a possibilidade de paralelismo aninhado durante a execução.

Exemplo:

setenv OMP_NESTED FALSE

6.3 **Biblioteca de Rotinas**

A biblioteca de rotinas contém a implementação de diversas rotinas que auxiliam o programador na construção de um programa paralelo. Estas rotinas desempenham diversas funções como: consulta ao número de threads e de processadores sendo utilizados, definição do número de threads a serem utilizados, propósito geral para bloqueio (semáforos) e definição das variáveis de ambiente. As rotinas são

OMP_SET_NUM_THREADS

Define o número máximo de threads a ser utilizado durante a execução de um código paralelo

OMP_GET_NUM_THREADS

Esta função retorna o número de threads sendo usado no momento em que é chamada. Esta deve ser chamada dentro de um contexto paralelo. Se for chamada em uma parte seqüencial do programa, o retorno será 1.

OMP_GET_MAX_THREADS

Função que retorna o número máximo que a função *omp_get_num_threads* pode retornar. Isto reflete o valor da variável de ambiente

OMP_NUM_THREADS, ou seja, o número máximo de threads que podem ser criados durante uma execução paralela. Esta função pode ser chamada tanto na parte seqüencial como na parte paralela do programa.

OMP_GET_THREAD_NUM

Esta retorna o número do thread dentre o conjunto de threads criados para uma execução paralela. Este número pode ser um valor entre 0 e omp_get_num_threads – 1. O thread principal do conjunto é o thread 0. Esta função retorna 0 quando chamada em uma área seqüencial ou de paralelismo aninhado.

OMP_GET_NUM_PROCS

Função que retorna o número de processadores disponíveis para o programa.

OMP_IN_PARALLEL

Esta rotina retorna um valor verdadeiro se a seção do código sendo executada é paralela ou retorna um valor falso caso contrário.

OMP_SET_DYNAMIC

Esta sub-rotina habilita ou desabilita o ajuste dinâmico (para sistemas runtime) do número de threads disponíveis para a execução em regiões paralelas. Esta função tem precedência sobre a variável de ambiente *OMP_DYNAMIC*. Deve ser chamada em uma porção seqüencial do programa.

OMP_GET_DYNAMIC

Função é usada para determinar se o ajuste dinâmico para threads está habilitado ou não.

OMP_SET_NESTED

Esta sub-rotina é usada para habilitar ou não o paralelismo aninhado. Esta função tem precedência sobre a variável de ambiente *OMP_NESTED*.

OMP_GET_NESTED

Função retorna se o paralelismo aninhado está habilitado ou não.

OMP_INIT_LOCK

Função que inicializa um semáforo com a variável *lock* (passada como parâmetro para a função). O estado inicial desta é desabilitado (*unlock*).

OMP_DESTROY_LOCK

Esta desassocia a variável *lock* (passada como parâmetro) de quaisquer *locks*.

OMP_SET_LOCK

Esta rotina faz com que o thread executando espere até que o *lock* indicado esteja disponível.

OMP_UNSET_LOCK

Rotina disponibiliza o *lock* indicado para a função executando.

OMP_TEST_LOCK

Função que tenta bloquear a execução com um *lock*, mas não bloqueia se o *lock* não estiver disponível.

6.4 **Regiões paralelas**

Regiões paralelas são as construções básicas do OpenMP e a principal motivação para esta dissertação. As regiões paralelas são seções do código de um programa que podem ser executadas paralela ou concorrentemente. Esta execução é feita com a criação de threads que realizam as operações contidas nestes blocos paralelos.

O programa se inicia sendo executado em apenas um thread principal e assim se mantém até que uma região paralela seja “encontrada”. Quando esta região é “encontrada” durante a execução do programa, o thread principal cria um conjunto (team) de threads para executar a tarefa paralelamente, seguindo o modelo fork/join. Em seguida, cada thread executa o código da região paralela, respeitando suas variáveis privadas e compartilhadas. Então ao final da execução do código paralelo, o thread principal espera pelo término da execução do conjunto de threads, reúne as informações (modelo fork/join) e continua executando o programa.

Este processo pode ser melhor visualizado na Figura 3 abaixo, onde *master thread* é o thread principal, *fork* é onde é criado o conjunto de threads, *parallel region* é o código paralelo que será executado pelos threads e *join* é onde é feita a sincronização e término do conjunto de threads:

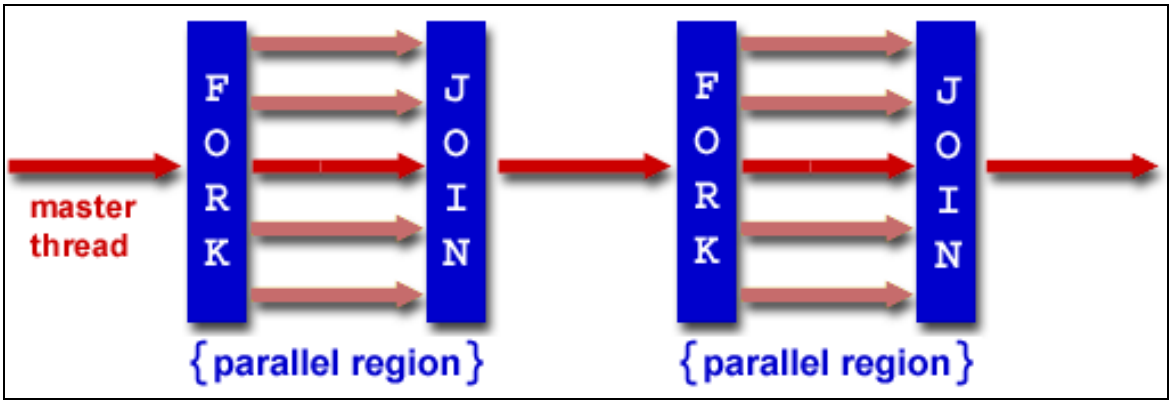

Figura 3 - Execução de um programa paralelo

Outro exemplo didático de como a região paralela é implementada em uma linguagem de programação é a Figura 4, onde as diretivas *!$OMP PARALLEL* são usadas para criar as regiões paralelas (fork) e *!$OMP END PARALLEL* são usadas para terminar uma região paralela (join). O código contido entre a primeira e a segunda diretiva é o código que será executado em paralelo pelo conjunto de threads.

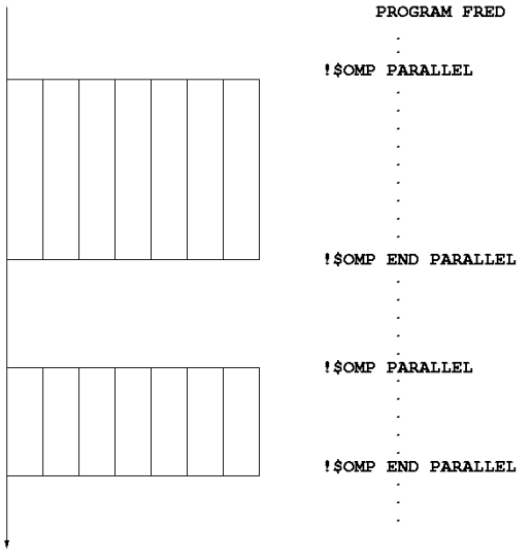

Figura 4 - Exemplo de região paralela

6.5 **Dados Compartilhados e Privados**

Na execução de uma região paralela, é criado um conjunto de threads que compartilham informações entre si. Sendo assim, cada thread compartilha dados de natureza comum a todo o conjunto de threads designados a resolver um problema paralelo. Cada thread também possui dados de natureza privada.

Dados compartilhados podem ser lidos e modificados por qualquer um dos threads pertencentes ao conjunto de threads. Todos enxergam a mesma cópia das variáveis compartilhadas.

Dados privados apenas podem ser lidos e modificados pelo próprio thread. Estes dados são invisíveis aos outros threads do conjunto.

Exemplificando, analisaremos o código abaixo:

```
void exemplo()
```

```
{
      int i, n;
      float z[n], a, x[n], y;
      #pragma omp parallel shared(z,a,x,y,n) private(i)
      {
         #pragma omp parallel for
         for (i=1; i<n; ++i)
           z[i] = a * x[i] + y;
         return;
     }
}
```

No código acima, notamos que uma região paralela foi criada e foram indicadas quais variáveis devem ser tratadas como compartilhadas (cláusula *share*) e quais devem ser tratadas como privadas para cada thread (cláusula *private*). Se nada for especificado, por padrão, todas as variáveis de uma região paralela serão compartilhadas. Dentro da região paralela, foi indicado o paralelismo de um comando *for*. Sendo assim, a Figura 5 ilustra como as variáveis são armazenadas durante uma execução seqüencial e paralela e como é a visibilidade das mesmas perante aos diferentes threads na execução paralela.

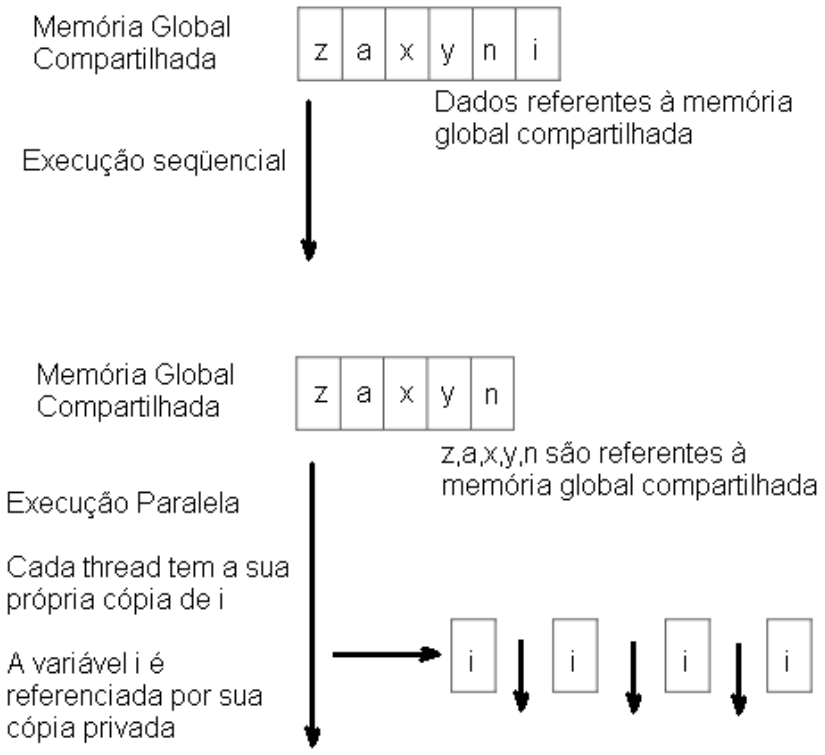

Figura 5 - Dados compartilhados e privados

Ainda considerando o exemplo acima, a Figura 6 ilustra como é o armazenamento das variáveis durante a execução do laço *for*, para 40 iterações e 4 threads.

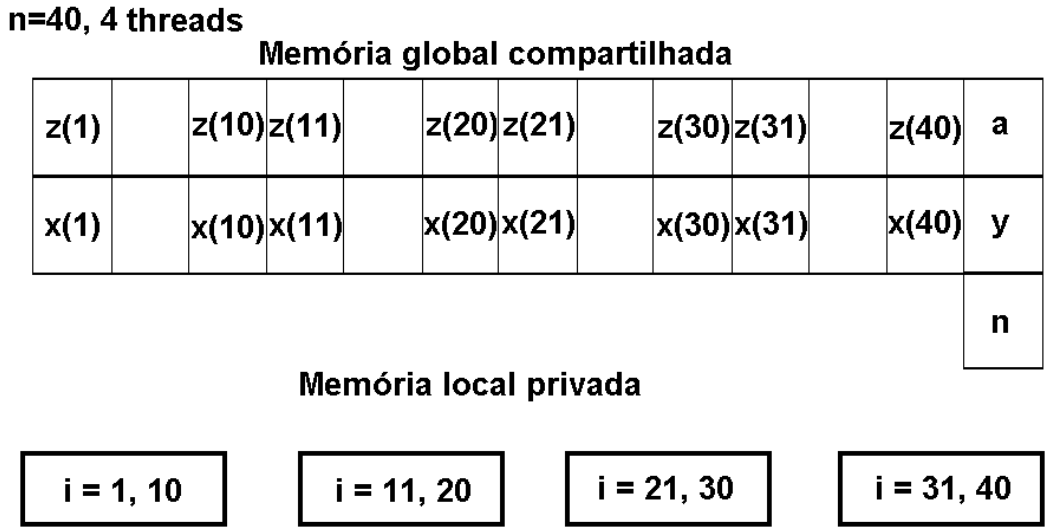

Figura 6 - Exemplo com 4 threads e 40 iterações

Nota-se que cada um dos quatro threads executou uma parte da iteração. A tarefa de dividir os trabalhos entre os threads fica a cargo da API OpenMP.

6.6 **Loops Paralelos**

Loops são iterações executadas por um programa e são a principal fonte de códigos paralelos. *Loops* são facilmente paralelizados.

Se as iterações de um *loop* forem independentes entre si, então se pode executá-las em qualquer ordem e as iterações podem ser compartilhadas entre os threads.

Por exemplo, temos um *loop for* que possui iterações independentes:

```
for (i=0; i<=100; ++i)
      a[i] += b[i];
```

Se tivermos dois threads para executar este pequeno trecho, o primeiro se encarregaria das iterações de 1 a 50 e o segundo, de 51 a 100.

6.7 **Sincronização**

Durante a execução de um código paralelo, há a necessidade de garantir uma certa sincronia entre os threads criados.

No que diz respeito a dados compartilhados, deve ser estabelecida uma ordem a ser obedecida para que ações aplicadas nestes dados garantam a integridade dos mesmos. Por exemplo: o thread 1 deve escrever a variável A antes que o thread 2 a leia ou o thread 2 deve ler a variável B antes que o thread 1 a escreva.

Vale lembrar que operações de atualização de variáveis compartilhadas não são atômicas, portanto devem ser usados alguns artifícios para contornar este fato e garantir que os dados não serão corrompidos. Por exemplo, pode-se estabelecer uma região crítica usando a diretiva de OpenMP (C/C++): *#pragma omp critical*

6.8 **Reduções**

Redução é uma grande “aliada” na paralelização de um laço *for*. As reduções produzem um único valor derivado de operações associativas (soma, multiplicação, max, min, and e or).

Em OpenMP, a redução é indicada como uma cláusula da diretiva de paralelização de um laço *for*, informando qual operação associativa será reduzida e em que variável ela será aplicada. Assim sendo cada thread fará a sua própria associação da variável a ser reduzida e, em seguida, todos os threads são associados. Por exemplo, temos o seguinte código de um laço *for* com redução:

```
#pragma omp parallel for default(shared) private(i) reduction(+:result)
for (i=0; i < n; i++)
    result += (a[i] * b[i]);
```

No código acima, a diretiva de paralelismo indica que a variável *i* é privada para cada thread e o restante das variáveis são compartilhadas. Também indica uma redução na somatória da variável *result*. Isso significa dizer que serão criados *t* threads e cada um deles calculará a sua própria acumulação da variável *result* e ao final da execução do laço, todos os *t* threads compartilharão seus valores de *result* para a redução de acumulação final, obtendo assim seu valor final. Se essa redução não fosse feita, a variável result só poderia ser atualizada por apenas um thread por vez, o que removeria totalmente o paralelismo do laço.

7. OpenMP em C/C++

OpenMP possui diversas implementações, sendo que estas variam de linguagem para linguagem, principalmente quando se trata de escopo de variáveis, alocação de memória e a própria definição das diretivas (sintaxe utilizada). Será citado alguns pontos importantes da implementação de OpenMP em C/C++, principalmente por esta ser uma linguagem presente no nosso cotidiano.

Quanto ao escopo de variáveis a API OpenMP trata, por padrão, todas as variáveis como compartilhadas, se nada for especificado na diretiva. Todas as diretivas devem ser seguidas de um comando válido que pode ser um bloco estruturado de código.

Variáveis estáticas declaradas e memória alocada utilizando-se de *malloc* dentro de uma região paralela são compartilhadas entre os threads. Porém um ponteiro para esta memória pode ser privado.

O escopo de ação das diretivas funciona semelhantemente ao de palavras reservadas em C/C++ (o escopo de ***if*** , por exemplo). As diretivas agem dentro de um bloco de código em C/C++({,}), não sendo obrigatória a informação de um comando OpenMP (*#pragma omp end parallel*, por exemplo).

Diretivas de grande tamanho podem ser continuadas na linha seguinte adicionando uma contra-barra (“\”) na linha anterior. Isso é um padrão da linguagem C/C++.

8. Overhead

OpenMP é uma API fácil de ser utilizada por programadores, pois permite um tratamento de alto nível sobre o paralelismo em sistemas de memória compartilhada. Esta facilidade, porém, tem um preço.

Este preço é um *overhead* criado pela linguagem. Sendo assim, o uso da linguagem OpenMP deve ser compensado por um código paralelo significantemente grande. Um estudo mostra que são necessários 10 kFLOPS para amortizar o *overhead* gerado por esta.

Por exemplo: para executar-se uma diretiva de paralelismo de um laço *for* em uma arquitetura com 16 processadores SGI Origin 2000, são consumidos 8000 ciclos de *clock* de *overhead* da linguagem. Ao multiplicar-se isso por um programa extenso, há uma perda de eficiência grande, porém compensada pelo tempo ganho no tempo de programação do mesmo. Pois grande parte do tempo, que seria gasto com criação, manipulação, sincronização e destruição de threads, é poupado pela facilidade de se programar tudo isso em alto nível.

9. OpenMP vs. MPI

A Tabela 1 ilustra diferenças entre a API OpenMP e MPI. É conveniente salientar que esta comparação refere-se apenas ao uso de OpenMP e MPI em memória compartilhada.

OpenMP	MPI
Apenas para computadores de memória compartilhada	Portável para todas plataformas
Pequena API baseada em diretivas e rotinas limitadas	Vasta coleção de rotinas
O mesmo programa pode ser usado para execução seqüencial e paralela (bastando apenas retirar as diretivas do código)	Difícil de usar o mesmo código para execução seqüencial e paralela
Variáveis compartilhadas e privadas podem causar confusão	Variáveis são locais para cada processador

Tabela 1 - OpenMP vs. MPI

10. Conclusão

OpenMP é uma API composta por três componentes (diretivas, variáveis de ambientes e biblioteca de rotinas) e é utilizada para paralelização de programas com memória compartilhada. Esta faz uso de threads para executar códigos paralelos permitindo uma configuração de como estes serão criados, executados e manipulados.

Sua fácil implementação e abstração em alto nível de problemas paralelos fazem com que esta API seja uma importante e poderosa ferramenta para desenvolvedores de sistemas paralelos. Porém, o sistema deve ser grande o bastante para que o custo de sobrecarga da criação e manipulação de threads pelo OpenMP seja vantajoso à árdua tarefa de implementar estas manipulações sem o uso da ferramenta.

11. Referências

- www.openmp.org
- www.llnl.gov/computing/tutorials/openMP
- www.mcc.ac.uk/HPC/OpenMP/
- www.inf.ufrgs.br/~elgio/trabs-html/Nucleo/openMP.html