Open Multi-Processing - OpenMP
Papers and Abstracts

Nelsi Warken

[1]Programa de Pós-Graduação em Informática
Mestrado em Ciência da Computação
Universidade Católica de Pelotas, RS, Brasil

nelsi.warken@gmail.com

Resumo. *Este artigo mostra uma visão geral a respeito da linguagem de programação paralela OpenMP. OpenMP (*Open Multi-Processing*) é uma interface de programação de aplicação (API) que suporta uma multi-plataforma de multi-processamento de memória compartilhada. Os elementos essenciais do OpenMP são as construções para a criação de* threads*, distribuição de carga de trabalho, gestão dos dados do ambiente,* threads *de sincronização, rotinas em tempo de execução a nível de usuário e variáveis de ambiente. Esta linguagem é um modelo de execução paralela* fork-join *e consiste de um conjunto de diretivas de compilação, bibliotecas de funções e variáveis de ambiente, que influenciam o ambiente em tempo de execução. O OpenMp possui suportes em C/C++ e Fortran. O paralelismo ocorre em memória compartilhada e, sobretudo, paralelismo nos dados.*

1. Introdução

Nesta parte do artigo, foi elaborada uma introdução com a definição das características principais da linguagem e dos sistemas paralelos de memória compartilhada. Na seção dois é descrito o histórico do OpenMP, compiladores de fornecedores ou comunidades de *open source* que implementam a API OpenMP e projetos de pesquisa da OpenMP ARB. Na seção três é mostrada a motivação para a criação da linguagem OpenMP. Na seção quatro são descritos os objetivos principais e as aplicações do OpenMP. Na seção cinco são descritos a funcionalidade da linguagem, sua estrutura e o seu modelo de execução. Na seção seis é descrita a sintaxe da linguagem com a relação e definição dos comandos pricipais, referentes ao paralelismo. Na seção sete são relacionados vários programas, como exemplos da linguagem OpenMP. E na seção oito são relacionadas as conclusões, as vantagens e as desvantagens do uso da linguagem OpenMP.

OpenMP (*Open Multi-Processing*) é uma interface de programação de aplicação que suporta uma multi-plataforma de multi-processamento de memória compartilhada. Podendo ser programada em C/C++ e Fortran, para várias arquiteturas, incluindo plataformas UNIX e MicroSoft Windows. Consiste de um conjunto de diretivas de compilação, bibliotecas de funções e variáveis de ambiente, que influenciam o ambiente em tempo de execução.

A organização das arquiteturas paralelas pode dividir-se, basicamente, em dois tipos: arquitetura de memória distribuída e arquitetura de memória compartilhada.

OpenMP é uma API (*Application Programming Interface*) para programação paralela, explicitamente, onde o paralelismo aconteça em memória compartilhada e, sobretudo, paralelismo nos dados. Todos os processadores podem ler e gravar em todas as posições de memória, existe apenas uma unidade lógica de espaço de memória.

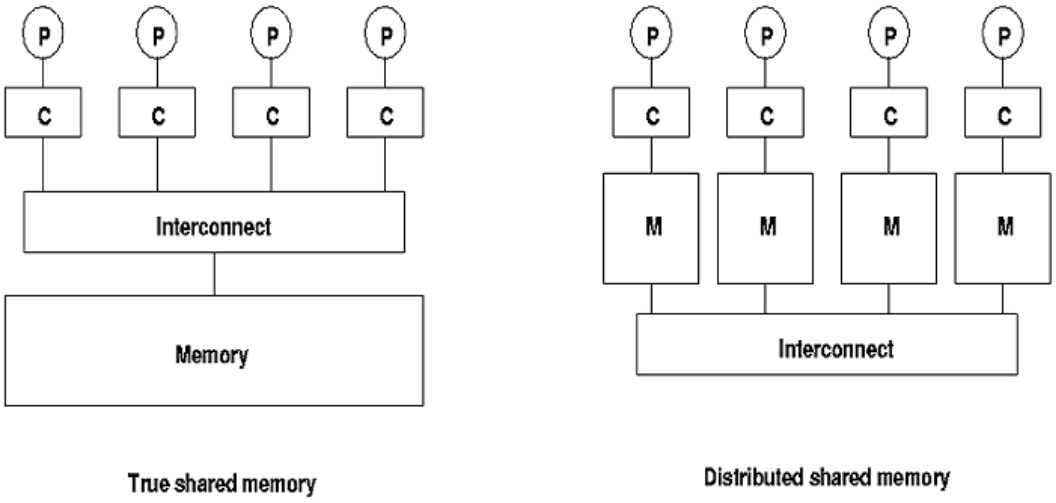


Figure 1. Memória Compartilhada

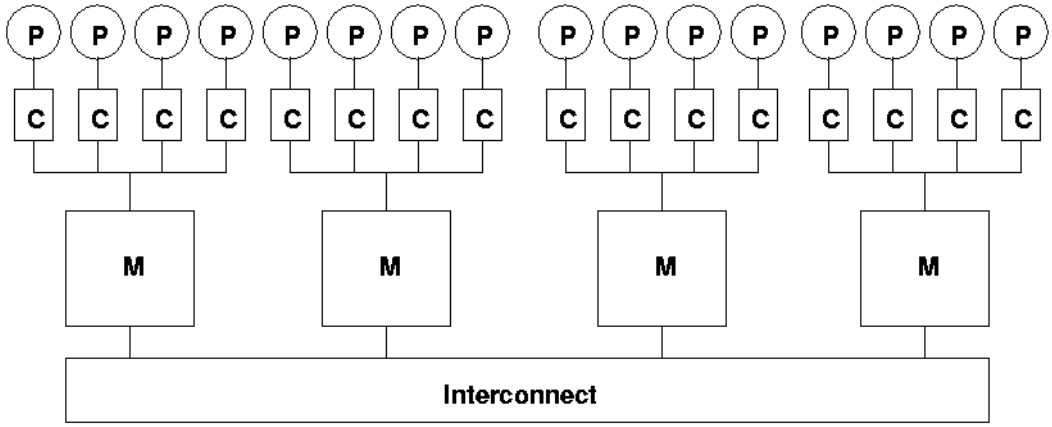


Figure 2. Memória Compartilhada em Clusters

As figuras 1 e 2 mostram as arquiteturas de sistemas paralelos com memória compartilhada.

OpenMP foi desenvolvido por um grupo de fornecedores de *hardware* e *software*. Constitui um modelo portável e escalável que fornece uma interface simples para desenvolver aplicações paralelas, desde *desktop* a supercomputadores. A aplicação pode ser construída com um modelo híbrido de programação paralela, podendo rodar em

um *cluster* usando OpenMP e MPI (Interface de Passagem de Mensagem). OpenMP *Architecture Review Board* (ARB) é uma corporação sem fins lucrativos, que é proprietária da marca OpenMP, supervisiona a especificação, produz e aprova novas versões do OpenMP. A ARB ajuda a organizar e financiar conferências, seminários, workshops e outros eventos relacionados, além de promover o OpenMP. A ARB é composta por membros permanentes e auxiliares. Membros permanentes são vendedores, que tem interesse a longo prazo, na criação de produtos para OpenMP. Membros auxiliares são normalmente organizações com interesse no padrão, mas que não criam ou vendem produtos OpenMP. A ARB, como a maior parte das sociedades sem fins lucrativos, possui membros corporativos oficiais (CEO, CFO, e secretário) e um Conselho de Administração. Os funcionários são responsáveis pelo dia-a-dia dos negócios da corporação. O Conselho de Administração é responsável pela definição de longo prazo das sociedades, dirigir e aprovar grandes gastos e atua como um órgão *OverSite* e de governança para a corporação.

2. Histórico

Dados do site oficial do OpenMP [OPENMP 2008]

Última versão oficial: 3.0 (Maio-2008): C/C++ e Fortran

Traduções: Versão 3.0 - Japonês (draft)

Versões anteriores:

- Versão 2.5 - (Maio 2005): C/C++ e Fortran)
- Versão 2.0 - (Março 2002): C/C++
- Versão 2.0 - (Novembro 2000): Fortran
- Versão 1.0 - (Outubro 1998): C/C++
- Versão 1.1 - (Novembro de 1999): Fortran
- Versão 1.0 - (Outubro 1997): Fortran

A figura 3 mostra os Compiladores de vários fornecedores ou de comunidades de *open source* que implementam a API OpenMP [OPENMP 2008].

Projetos de Pesquisa OpenMP ARB:

- *The INTONE project: C and Fortran compilers and analysis tools*
- *OdinMP: A Free, Portable OpenMP Implementation for C*
- *OpenUH: Open source UH compiler suite for OpenMP (Univ of Houston)*
- *OpenMP Validation Suite (HPC Center, Stuttgart)*
- *KOJAK: Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks (Jülich Supercomputing Center)* - (gargalos).

As comunidade de usuários OpenMP do meio acadêmico e da indústria podem discutir, tirar dúvidas e trocar informações sobre OpenMP e com especialistas de programação OpenMP. O registro nos fóruns de discussão é gratuito e aberto a todos, conforme sítio da comunidade de usuários OpenMP [COMUN 2008].

Fornecedor	Compilador	Disponibilidade
GNU	gcc (4.2)	Free and open source - Linux, Solaris, AIX, MacOSX, Windows
IBM	XL C/C++ / Fortran	Windows, AIX and Linux.
Sun Microsystems	C/C++ / Fortran	Sun Studio compilers and tools - free download for Solaris and Linux.
Intel	C/C++ / Fortran	Windows, Linux, and MacOSX.
Portland Group Compilers and Tools	C/C++ / Fortran	
Absoft Pro FortranMP	Fortran	
Lahey/Fujitsu Fortran 95	C/C++ / Fortran	
PathScale	C/C++ / Fortran	Linux 32/64 bit.
HP	C/C++ / Fortran	
MS	Visual Studio 2008 C++	Implements OpenMP 2.0

Ajustar linha da tabela

Figure 3. Compiladores OpenMP

3. Motivação

A incrível evolução dos microprocessadores proporcionam, atualmente, um desempenho maior que os supercomputadores do passado. Mas esta evolução hoje encontra-se limitada pelas leis da física na tecnologia de desenvolvimento de semicondutores. Além disso, as aplicações tornam-se cada vez mais sofisticadas e complexas e com exigências de computação de alta velocidade. Para atender estas demandas, cientistas procuram arquiteturas inovadoras e utilizam também o paralelismo como uma solução.

Com o surgimento dos conceitos de *SMP (Symmetric MultiProcessing)*, *Multithread*, *Multicore*, surgiu também a necessidade de uma linguagem de programação que utilize as ferramentas de mecanismos de controle de coerência, controle de concorrência, reengenharia e principalmente novos algoritmos.

Neste ambiente ocorreu a motivação para o desenvolvimento de um padrão de programação para processamento paralelo, apoiado pelos grandes fabricantes de software e hardware. O OpenMP foi desenvolvido em uma plataforma *open source*, para multi-processamento, através de um trabalho colaborativo entre indústrias de *software* e *hardware*, universidades e governo (EUA).

4. Objetivos

OpenMP tem como objetivos principais:

- fornecer um compacto, mas poderoso, modelo de programação de memória compartilhada;
- suportar Fortran, C, C + +;
- os programas são portáveis para uma ampla gama de plataformas;
- o programa pode ser escrito enquanto a versão sequencial ainda está em construção.

Aplicações do OpenMP:

- Previsão do tempo, modelos de ondas e modelos de oceanos - (LAMBO - *Limited Area Model Bologna*, modelo adaptado para o território italiano), um modelo de ondas (W.A.M. – *Wave Model*, no Mar Mediterrâneo e no Mar Adriático) e um modelo de oceanos (M.O.M. – *Modular Ocean Model*, usado com assimilação de dados). Estes três modelos foram escritos para máquinas vetoriais e foi convertido para código para arquitetura SMP.
- *Novel FDTD*: Algoritmo de diferenças temporais finitas para modelar características óticas de fontes de luz, criadas pela utilização de novas classes de materiais conhecidas como cristais de intervalos de banda fotônicas.
- OpenMP foi implementado em compiladores comerciais. Por exemplo, *Visual* C++ 2005 no *Professional and Team System editions* [MSDNOMP 2008], compiladores Intel para seus x86 e produtos da série IPF. A Sun Studio suporta versão OpenMP 2.5 para Solaris OS (UltraSPARC e x86/x64). Compiladores Fortran, C and C++ do Portland Group (*PGI CDK Cluster Development Kit*).
- Compiladores GCC e algumas distribuições, como Fedora Core 5 gcc tambem suportam OpenMP.

5. Funcionalidades

OpenMP é uma aplicação *multithreading*, um método de paralelização segundo o qual a *thread* mestre (uma série de instruções executadas consecutivamente) separa/divide (*forks*) um número determinado de *threads* escravas, onde uma tarefa é dividida entre elas. Estas *threads* são executadas simultaneamente, com o ambiente *runtime* alocando *threads* para diferentes processadores. A seção de código para rodar em paralelo estará marcada de acordo com uma diretiva pre-processador, que formará as *threads* antes da seção ser executada. *Cada thread* tem um *id* que a acompanha, que pode ser obtido através de uma função (chamada omp_get_thread_num () em C / C ++ e OMP_GET_THREAD_NUM () em FORTRAN). A *thread id* é um inteiro, e a *thread master* tem um *id* = 0. Após a execução do código paralelizado, as *threads* são unidas a *thread master*, que continuará a execução, sequencialmente, dali em diante até o fim do programa.

Estrutura do OpenMP:

Os elementos essenciais do OpenMP são as construções para a criação de *threads*, distribuição de carga de trabalho, gestão dos dados do ambiente, *threads* de sincronização, rotinas em tempo de execução a nível de usuário e variáveis de ambiente.

A figura 4 mostra estrutura da linguagem OpenMP, [WIKIOMP 2008].

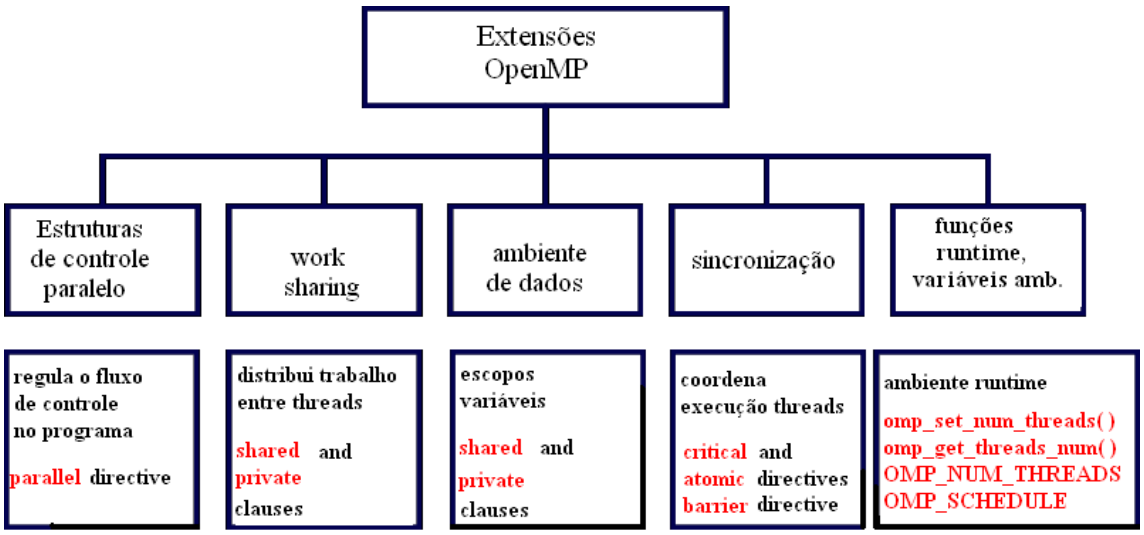


Figure 4. Estrutura da linguagem OPenMP

Modelo de Execução

OpenMp é um modelo de execução paralela *fork-join*. Projetado para programas que executam corretamente de forma sequencial ou paralela, facilitando a conversão de programas. Quando o programa é executado sequencialmente, os comandos de paralelização são considerados comentários. Pode também ser usado em programas que sejam estritamente paralelos em sua concepção.

Processo de execução do OpenMP:

- Inicia a execução com um processo *master thread*;
- no início do construtor paralelo, cria um grupo de *threads*;
- ao completar o grupo de *threads* sincroniza numa barreira implícita (*barrier*);
- apenas o *master* continua a execução.

A figura 5 mostra o modelo de execução *fork-join* [LLNLOMP 2008] .

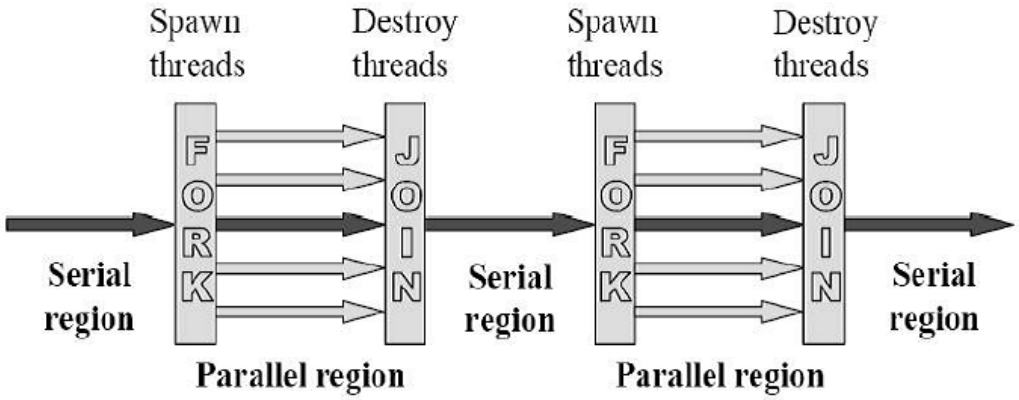


Figure 5. Modelo de execução *Fork-Join*

As variáveis no programa OpenMp devem ser definidas como variáveis compartilhadas ou como variáveis privadas.

Abaixo, algumas considerações a este respeito:

- A maioria das variáveis são compartilhadas;
- índices de *loop* são privadas;
- *loops* temporários são privadas;
- variáveis somente leitura são compartilhadas;
- *arrays* principais são compartilhadas;
- escalares escritas antes da leitura: normalmente são privadas;
- algumas vezes semanticamente está correto, mas há implicações na performance, em fazer uma escolha.

6. Sintaxe da Linguagem OPenMP

Os programas iniciam com apenas uma linha de execução (*thread*), chamada *thread inicial* ou *master thread*. O Programa segue a execução da *thread inicial*, como uma execução sequencial, até encontrar uma região de execução paralela. Ao término da região paralela, somente a *thread* principal segue a execução das instruções seguintes.

Uma diretiva do compilador em C/C++ é chamada *pragma* (*pragmatic information*). Ela é chamada de uma diretiva de pré-processador, e é declarada com um *hash* (♯)
.

Diretivas C/C++

♯pragma omp directive-name clause, newline

clause: opcional. Podem aparecer em qualquer ordem e se necessário repetidas.

Newline: Obrigatório. Seguido do bloco estruturado (será executado em paralelo).

♯pragma omp parallel default(shared) private(beta,pi)

Criação de Thread

omp parallel é a construção fundamental que inicia uma execução paralela. Usado para dividir threads adicionais para realizar o trabalho em paralelo. O processo original será marcado com master *thread* com *thread ID* 0.

Construções de compartilhamento de trabalho usado para especificar como marcar trabalho, independente, para uma ou todas as threads.

- *omp for or omp do*: usado para dividir iterações de *loop* entre as *threads*
- *sections*: marcação consecutiva, mas independente, blocos de código para diferentes *threads*
- *single*: especificar um bloco de código que é executado somente por uma *thread*, um limite é incluído no final
- *master*: similar ao single, mas o bloco de código será executado somente pelo *master thread* e nenhum limite é colocado no final.

Cláusulas OpenMP

OpenMP é um modelo de programação de memória compartilhada, onde muitas variáveis, no código OpenMP, são visíveis para todas as *threads* por default. Mas, algumas vezes, variáveis privadas são necessárias para evitar dependência de dados. Além de poder existir a necessidade de passar valores entre a parte sequencial e o bloco de código paralelo. Então, o gerenciamento de dados do ambiente é introduzido como *data clauses* adicionando-as nas diretivas OpenMP.

IF control - A paralelização de tarefas somente ocorrerá se a condição for verdadeira. Caso contrário, o bloco de código será executado sequencialmente.

if(expressão-condicional)

Caracterizando os diferentes tipos de cláusulas de dados:

- *shared* (lista): as varáveis da lista são compartilhadas por todos as *threads por default*, exceto o contador do *loop* de iteração.

- *private* (lista): as variáveis da lista ficam privadas a cada *thread*, isto é, cada *thread* terá uma cópia local e será usada como uma variável temporária. A variável privada não é inicializada e o valor não é atualizado para uso fora da região paralela. Por *default*, o contador de *loop* de iteração é privado.
- *Firstprivate* (lista): permite que as variáveis privadas sejam inicializadas.
- *default (shared, private, or none)*: permite definir o *default* para as variáveis dentro de uma região paralela. A opção *none* força a declaração de cada variável na região paralela (*shared ou private*).
- *copyin*(lista)
- *reduction*(operador: lista)
- *num_threads*(inteiro)

Construções de compartilhamento de trabalho: especificam as regras de divisão de trabalho entre as *threads*, não as cria. Tipos de divisão:

Cláusula *for*: a construção *loop* - divide as iterações de um ciclo pelas *threads* da *team* (paralelismo de dados).

Cláusula *sections*: divide o trabalho em seções discretas, distintas, que são executadas pelas *threads*. Pode ser usado para paralelismo funcional.

Cláusula *single*: serializa o código

A figura 6 mostra a aplicação da cláusula *FOR* [SUN 2008].

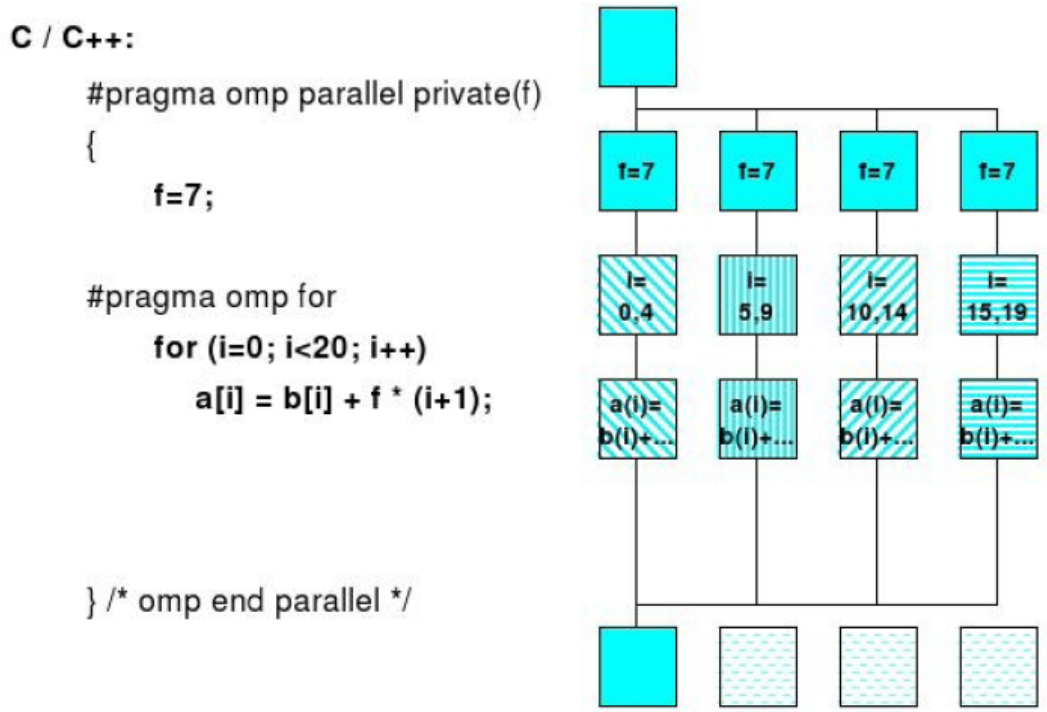


Figure 6. Cláusula *FOR*

A figura 7 mostra a aplicação da cláusula *SECTIONS* [SUN 2008].

Claúsula *schedule* - é uma cláusula importante na construções de *loops*. Define como dividir as iterações de ciclo.
Usada para construçoes *do-loop* or *for-loop*. As iterações são alocadas para *threads*. O escalonador de *threads* é controlado por esta cláusula.

schedule(tipo[, tamanho])

O tipo define como o trabalho é dividido (tipos de escalonamento):

- *static*: as iterações são agrupadas em conjuntos (*chunks*), estaticamente atribuídas às *threads*.
- *dynamic*: as iterações são agrupadas em grupos (*chunks*), e são dinamicamente distribuídas pelas *threads*. Quando um termina, recebe dinamicamente outro *chunk*.
- *guided*: indica o número mínimo de iterações a agrupar numa tarefa.
- *runtime*: a decisão é tomada em tempo de execução.

Cláusulas de sincronização:

- *critical section*: o bloco de código será executado por todas as *threads*, mas apenas uma *thread* de cada vez, não é executada simultaneamente com outra. Frequentemente usado para proteger dados compartilhados (dependência de dados).
- *atomic*: similar ao critical, mas avisa o compilador para usar instruções especiais de *hardware* para melhor *performance*. Compiladores podem ignorar a sugestão dos usuários e utilizar a *critical section*. Evita que uma região de memória seja atualizada simultaneamente por mais de uma *thread*.
- *ordered*: a execução ocorre na ordem em que a iterações seriam executadas em um loop sequencial.
- *barrier*: threads aguardam outras chegarem a este ponto antes de continuarem a execução. A construção de compartilhamento tem uma sincronização *barrier* implícita no final.
- *nowait*: especifica quais *threads* poderão prosseguir.
- *single*: executada em uma *thread*.
- *master*: executada apenas pelo *master thread*, os outros ignoram.
- *flush*: força o sincronismo da região de memória da *thread* com a memória principal.

Inicialização

- *firstprivate*: os dados são privados para cada *thread*, mas inicializados usando o valor da variável com o mesmo nome na *master thread*.
- *lastprivate*: os dados são privados para cada *thread*. O valor desta variável será copiado para uma variável global usando o mesmo nome, se a iteração corrente for a última no *loop* paralelizado. Uma variável pode ser ambos, *firstprivate* e *lastprivate*.
- *threadprivate*: Os dados são dados globais, mas ela é privada em cada região paralela em tempo de execução. A diferença entre threadprivate e private é o escopo global associado com *threadprivate* e o valor preservado nas regiões paralelas.

- *copyin*: similar ao *firstprivate* para variáveis privadas, *threadprivate* variáveis não são inicializadas, a não ser usando *copyin* para passar o valor das correspondentes variáveis globais. Não é necessário o *copyout* porque o valor da variável *threadprivate* é mantido durante toda execução do programa.
- *copyprivate*: usado com *single* para suportar a cópia dos valores de dados dos objetos privados em uma *thread* (*single thread*) para os objetos correspondentes em outras *threads* no grupo.

Redução

reduction (*operator / intrinsic: list*): as variáveis tem uma cópia local em cada *thread*, mas os valores das cópias locais podem ser reduzidos em uma variável global compartilhada. Podem ser operadas as variáveis da lista.

Variáveis de Ambiente

Um método para alterar os recursos de execução das aplicações OpenMP. Usado para controlar escalonamento de iterações de *loop*, número default de *threads*, etc. Por exemplo, OMP_NUM_THREADS para especificar o número de *threads* para uma aplicação.

OMP_SCHEDULE

OMP_NUM_THREADS

OMP_DYNAMIC

OMP_NESTED

Funções OpenMP (subrotinas da biblioteca)

Rotinas em tempo de execução a nível de usuário: usadas para modificar/checar o número de *threads*. Detecta se o contexto de execução está em uma região paralela quantos processadores tem no sistema corrente, setar/desetar *locks*, funções de *timing*, etc.

Algumas funções:

- void omp_set_num_threads (int) invocada na parte sequencial;
- int omp_get_num_threads (void) retorna o número de *threads* ativos;
- int omp_get_max_threads (void) retorna o número máximo de *threads* permitidos;
- int omp_get_thread_num (void) retorna o *ID* do *thread* (entre 0 e t-1);
- int omp_get_num_procs(void) retorna o número de processadores disponíveis para o programa;
- void omp_init_lock(omp_lock_t*) Inicializa um *lock* associado à variável de *lock*;
- void omp_destroy_lock(omp_lock_t*) void omp_set_lock(omp_lock_t*) espera até conseguir o *lock* ;
- void omp_unset_lock(omp_lock_t*) libera o *lock*;
- double omp_get_wtime(void) retorna o o número de segundos decorridos (*elapsed time*);
- double omp_get_wtick(void) retorna os segundos decorridos entre chamadas sucessivas.

7. Exemplos

Inicializar o valor de um grande *array* em paralelo, usando cada *thread* para fazer uma parte do trabalho.

```
{}
#define N 100000
int main(int argc, char *argv[])
{
  int i, a[N];
  #pragma omp parallel for   // início da paralelização
  for (i=0;i<N;i++)          // iteração
     a[i]= 2*i;
  return 0;
}
```

Exemplos sections

```
{}

#include <omp.h>
#define N 1000
main() {
int i, chunk;
float a[N], b[N], c[N];
// Inicializações  a[0]=0.0... a[999]=999.0
for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;
/*
// paralelização com variáveis a,b e c compartilhadas pelas threads
// e variável i privada nas threads
*/
#pragma omp parallel shared(a,b,c) private(i) {
#pragma omp sections nowait {
#pragma omp section        // região paralela 1
for (i=0; i < N/2; i++) c[i] = a[i] + b[i];
#pragma omp section        // região paralela 2
for (i=N/2; i < N; i++) c[i] = a[i] + b[i];
```

```
#include <stdio.h>
#include <omp.h>
int main () {
int i;
#pragma omp parallel sections private(i)
{
#pragma omp section
for (i=1; i<5; i++) printf("loop=1 i=%d thread=%d\n", i,
```

```
omp_get_thread_num());
#pragma omp section
for (i=1; i<5; i++) printf("loop=2 i=%d thread=%d\n", i,
omp_get_thread_num());
}
return 0;
}
```

Execução do programa acima, informando que o numero de threads a ser utilizado = 4

```
$ OMP_NUM_THREADS=4 ./ompforsections

// resultado da execução

loop=1 i=1 thread=0
loop=1 i=2 thread=0
loop=1 i=3 thread=0
loop=1 i=4 thread=0
loop=2 i=1 thread=2
loop=2 i=2 thread=2
loop=2 i=3 thread=2
loop=2 i=4 thread=2
```

Exemplo de funções OpenMP

```
{}
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]){
int nthreads, tid, procs, maxt, inpar,
dynamic, nested;

/* Início da região paralela */

#pragma omp parallel private(nthreads, tid) {

/* Obtem o número de  threads */

tid = omp_get_thread_num();

/* Executado somente pela master thread  */
```

```
if (tid == 0) {
printf("Thread %d getting info...\n", tid);

/* Obtem informações do ambiente (retorno de funções) */

procs = omp_get_num_procs();
nthreads = omp_get_num_threads();
maxt = omp_get_max_threads();
inpar = omp_in_parallel();
dynamic = omp_get_dynamic();
nested = omp_get_nested();

/* Imprime o retorno das funções */

printf("Number of processors = %d\n", procs);
printf("Number of threads = %d\n", nthreads);
printf("Max threads = %d\n", maxt);
printf("In parallel? = %d\n", inpar);
printf("Dynamic threads? = %d\n", dynamic);
printf("Nested parallelism? = %d\n", nested);
}
} / Fim /
}
```

Exemplo Hello em C:

```
{}
#include <stdio.h>
#include <omp.h>
int main() {
int tid;
if(omp_in_parallel()) printf("oops... parallel\n");
#pragma omp parallel private(tid)
{

/* obtem número de threads a ser utilizado pela paralelização,
informado pelo usuário */

tid = omp_get_thread_num();
if(tid == 0) {
int tnum = omp_get_num_threads();
printf("%d threads.\n", tnum);
}
printf("Hello world! Thread %d.\n", tid);
if(omp_in_parallel()) printf("parallel\n");
```

```
}
if(omp_in_parallel()) printf("oops again... parallel\n");
return 0;
}
```

```
Exemplo em C (saída em tela):
$ gcc-4.2 -fopenmp -lgomp hello2.c -o hello2
$ ./hello2
threads 2
Hello world! Thread 0.
parallel
Hello world! Thread 1.
parallel
$
```

Programa Hello World em Fortran

```
{}

PROGRAM HELLO
      INTEGER ID, NTHRDS
      INTEGER OMP_GET_THREAD_NUM, OMP_GET_NUM_THREADS
C$OMP PARALLEL PRIVATE(ID)
      ID = OMP_GET_THREAD_NUM()
      PRINT *, 'HELLO WORLD FROM THREAD', ID
C$OMP BARRIER
      IF ( ID .EQ. 0 ) THEN
        NTHRDS = OMP_GET_NUM_THREADS()
        PRINT *, 'THERE ARE', NTHRDS, 'THREADS'
      END IF
C$OMP END PARALLEL
      END
```

A figura 8 mostra um exemplo de paralelização OpenMP.

8. Conclusões

Como expectativa de desempenho do OpenMP, poderíamos esperar obter um aumento de performance N vezes menor tempo de execução ou N vezes o *speedup*, quando es-

C / C++:

```
#pragma omp parallel
{
#pragma omp sections
 {{  a=...;
     b=...; }
#pragma omp section
   {  c=...;
      d=...; }
#pragma omp section
   {  e=...;
      f=...; }
#pragma omp section
   {  g=...;
      h=...; }
 } /*omp end sections*/
} /*omp end parallel*/
```

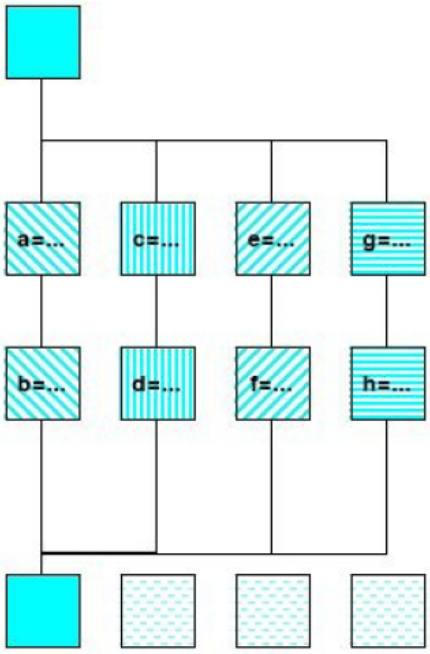


Figure 7. Cláusula *SECTIONS*

```
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
       z[i] = x[i] + y[i];

#pragma omp for nowait

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

#pragma omp barrier

        ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
        ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

parallel loop
(work will be distributed)

parallel loop
(work will be distributed)

synchronization

Statement is executed by all threads

parallel region

Figure 8. Exemplo de paralelização OpenMP

tivéssemos executando um programa paralelizado, usando OpenMP em uma plataforma com N processadores. No entanto, isso raramente acontece, devido às seguintes razões:

- uma grande porção do programa não pode ser paralelizada pelo OpenMP, o que significa que o limite máximo teórico de aceleração está de acordo com a Lei de Amdahl;
- N processadores em um SMP pode ter N vezes o poder de computação, mas a largura de banda de memória geralmente não pode escalar até N vezes. Muitas vezes, a memória original é compartilhada por vários processadores e a degradação de desempenho pode ser observada quando eles concorrem pela largura de banda da memória compartilhada;
- muitos outros problemas comuns que afetam o *speedup* final na computação paralela também aplicam-se ao OpenMP, como o balanceamento de carga e *overhead* de sincronização.

Vantagens

- Simples: não necessita negociar com *message passing* como o MPI.
- Dados: *layout* e decomposição é manuseada automaticamente pelas diretivas.
- Paralelismo incremental: pode trabalhar em uma parte do programa em um momento, não são necessárias mudanças dramáticas no código.
- Código unificado para ambas as aplicações: serial e paralela. Construções OpenMP são tratadas como comentários quando compiladores sequenciais são utilizados.
- Código original (serial), em geral, não necessita ser modificado quando for paralelizado com OpenMP. Isto reduz a chance de inadvertidamente introduzir bugs.
- Granulosidade: paralelismos *coarse-grained* e *fine-grained* são possíveis.

Desvantagens

- Atualmente só é executado, de forma eficiente, em plataformas de multiprocessadores de memória compartilhada.
- Necessita de um compilador que suporta OpenMP.
- Escalabilidade é limitada pela arquitetura da memória.
- Não tem o tratamento do erro de confiabilidade.
- Granulosidade: faltam mecanismos para controlar o mapeamento *thread-processor* (*fine-grained*).
- Sincronização entre um subconjunto de threads não é permitido.

A figura 9 mostra a performance do OpenMP, segundo a SUN [SUN 2008]. A performance em *MFlops (Mega-floating point)* - milhões de intruções de ponto flutuante por segundo e a *memory footprint*, a memória realmente utilizada pelo programa. Mostra comparações utilizando uma, duas ou quatro CPUs.

References

COMUN (2008). Comunidade de usuários openmp. <http://www.cOMPunity.org>.

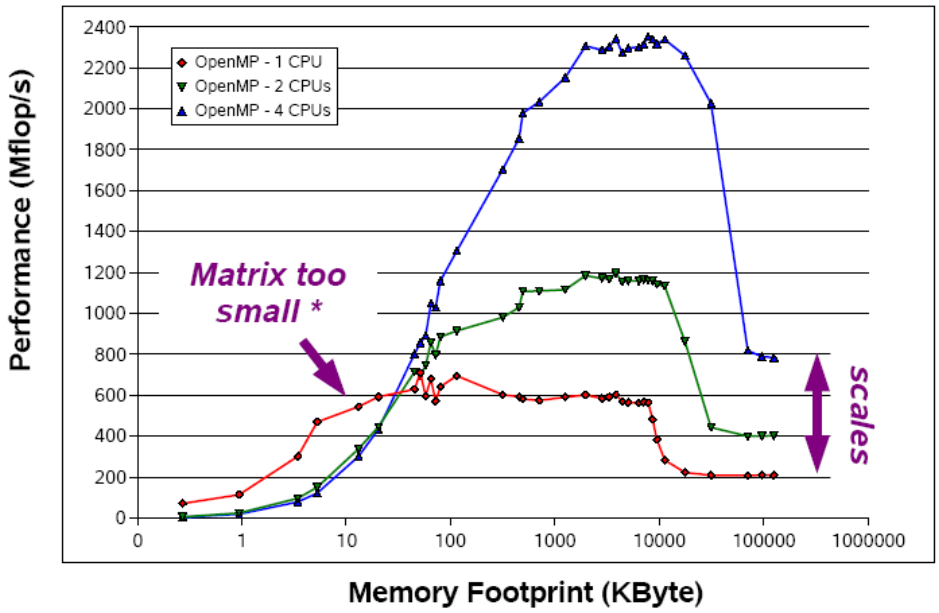
LLNLOMP (2008). Llnl tutorial openmp. <https://computing.llnl.gov/tutorials/openMP/>.

MSDNOMP (2008). Msdn microsoft openmp. <http://msdn.microsoft.com/en-us/library/tt15eb9t(VS.80).aspx>.

OPENMP (2008). Site oficial da openmp. <http://openmp.org/wp/>.

SUN (2008). Sun microsystems. <http://developers.sun.com/sunstudio/index.jsp>.

WIKIOMP (2008). Wikipedia openmp. <http://en.wikipedia.org/wiki/OpenMP>.



SunFire 6800
UltraSPARC III Cu @ 900 MHz
8 MB L2-cache

*) With the IF-clause in OpenMP this performance degradation can be avoided

Figure 9. Performance OPenMP - Sun Microsystems