OpenMP

Hélio Crestana Guardia

Departamento de Computação
Universidade Federal de São Carlos

OpenMP

- **OpenMP** (***Open Multi-Processing*** - http://openmp.org) é uma interface de programação (API) que possibilita o desenvolvimento de programas em C/C++ e Fortran para ambientes multiprocessados.
- Definido por um grupo formado pelos principais fabricantes de *hardware* e *software*, **OpenMP** é um modelo **portável** e **escalável** que provê uma interface simples e flexível para o desenvolvimento de aplicações paralelas baseadas em memória compartilhada.
- Diferentes arquiteturas são suportadas, variando do *desktop* a supercomputadores, incluindo plataformas Unix e Windows.
- De maneira geral, **OpenMP** consiste de um conjunto de **diretivas** de compilação, **bibliotecas** e **variáveis de ambiente** que influenciam o comportamento da execução de programas.
- ***OpenMP*** *Architecture Review Board* (ARB) é uma corporação sem fins lucrativos que detém a marca OpenMP e supervisiona sua especificação, produzindo e aprovando novas versões

OpenMP

Latest Official OpenMP Specifications *Version 3.1* **Complete Specifications - (July 2011). (PDF)**

This document specifies a collection of ***compiler directives****,* ***library routines****, and* ***environment variables*** *that can be used to specify* ***shared-memory parallelism*** *in C, C++ and Fortran programs. This functionality collectively defines the specification of the OpenMP Application Program Interface (OpenMP API). This specification provides a* ***model for parallel programming*** *that is* ***portable across shared memory architectures*** *from different vendors. Compilers from numerous vendors support the OpenMP API.*

The directives, library routines, and environment variables defined in this document allow users to create and manage parallel programs while permitting portability. ***The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing and privatizing data****. The functionality to control the runtime environment is provided by library routines and environment variables. Compilers that support the OpenMP API often include a command line option to the compiler that activates and allows interpretation of all OpenMP directives.*

The OpenMP API covers only ***user-directed parallelization****, wherein the* ***user explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel****.* ***OpenMP-compliant implementations are not required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs****. In addition, compliant implementations are not required to check for code sequences that cause a program to be classified as non-conforming.*

The user is responsible for using OpenMP in his application to produce a conforming program*.* *OpenMP does not cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization.*

Referências

- http://openmp.org
- http://www.openmp.org/mp-documents/OpenMP3.1.pdf
- https://computing.llnl.gov/tutorials/openMP/
- http://www-users.cs.umn.edu/~karypis/parbook/
- http://en.wikipedia.org/wiki/OpenMP

- Chapman, B.; Jost, G. and van der Pas, R. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.
- Quinn, M.J. *Parallel Programming in C with MPI and OpenMP*. McGrawHill, 2004.

Compiladores

http://openmp.org/wp/openmp-compilers/

- Vendor/Source - Compiler Information - 11/2011

- GNU - gcc
- Free and open source - Linux, Solaris, AIX, MacOSX, Windows
- OpenMP 3.1 is supported since GCC 4.7
- Compile with -fopenmp

- Intel - C/C++ / Fortran (10.1)
- Tools: Thread Analyzer/Performance Analyzer
- Windows, Linux, and MacOSX
- Compile with **`-Qopenmp`** on Windows, or just **`-openmp`** on Linux or Mac OSX

- 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.

- Portland Group Compilers and Tools
- C/C++ / Fortran »More Information on PGI Compilers

- Oracle - C/C++ / Fortran
- Oracle Solaris Studio compilers and tools - free download for Solaris and Linux.

- **PathScale: C/C++ / Fortran**
- **HP: C/C++ / Fortran**
- **MS:** Visual Studio 2008-2010 C++
- Cray: Cray C/C++ and Fortran

OpenMP

Compilando com gcc:

// prog.c
#include <omp.h>
...

 # man gcc
/ openmp

-fopenmp
Enable handling of OpenMP directives "#pragma omp" in C/C++ and "!$omp" in Fortran.
When -fopenmp is specified, the compiler generates parallel code according to the OpenMP Application Program Interface v2.5 <http://www.openmp.org/>.
This option implies -pthread, and thus is only supported on targets that have support for -pthread.

#gcc prog.c –o prog –fopenmp

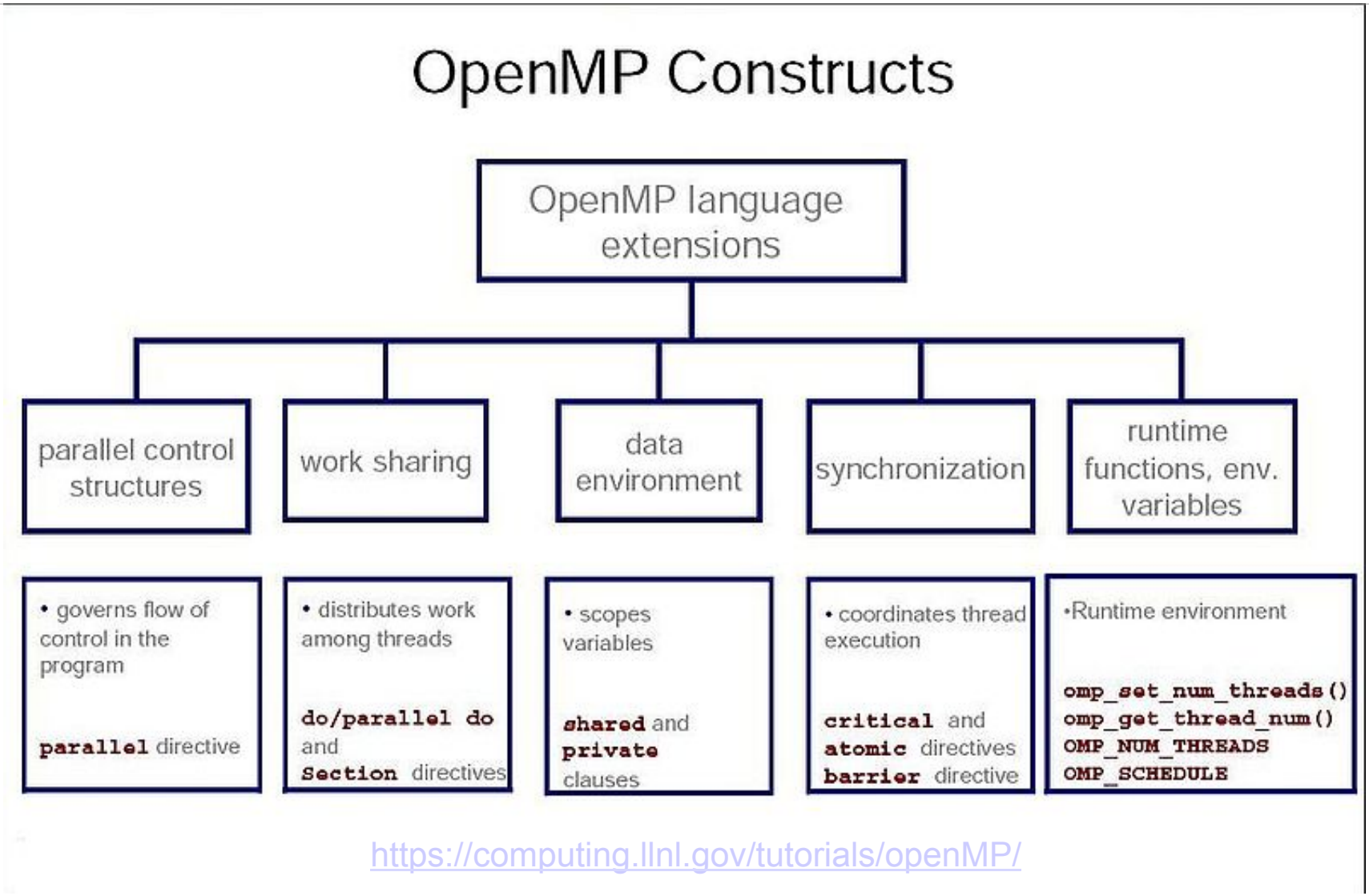
OpenMP

- Paralelismo baseado em ***threads*** para ambiente com **memória compartilhada** (*Shared Memory, Thread Based Parallelism*).
- OpenMP permite:
  - **Criar** times de *threads* para execução paralela de trechos de código
  - Especificar como **dividir** (*share*) as atividades entre os membros de um grupo
  - Declarar **variáveis** compartilhadas e privadas
  - **Sincronizar** *threads* e permitir que executem operações de maneira **exclusiva**
- OpenMP suporta ambos os modelos de paralelismo: de **código** (**funcional**) e de **dados** (de **domínio**).
- Paralelismo é definido de maneira **explícita**, não automática, com controle total do programador.
- Paralelismo especificado por diretivas de compilação (**pragmas em C/C++ (*pragmatic information*)**), que permitem passar informações ao compilador.
- Informações passadas pelos **pragmas** podem ser ignoradas pelo compilador sem alterar a correção do código gerado.
- Esforço de **paralelização de um programa** com OpenMP resume-se, em geral, à **identificação** do paralelismo e **não à reprogramação** do código para implementar o paralelismo desejado.

OpenMP

- Ao encontrar uma região paralela, um **time** de *threads* é criado, **replicando** ou **dividindo** o processamento indicado para execução paralela.
- Ambiente em tempo de execução é responsável por **alocar** as *threads* aos processadores disponíveis.
- Cada *thread* executa o trecho de código de uma seção paralela de maneira independente.
- Execução paralela é baseada no modelo ***Fork-Join***:
  - Programas OpenMP são iniciados com uma única *thread*, chamada ***master thread***.
  - *Master thread* executa sequencialmente até que região paralela seja encontrada.
  - FORK: *master thread* cria um **time** (*team*) de *threads* paralelas.
  - Comandos da região paralela são executados em paralelo pelas *threads* do time.
  - JOIN: quando time de *threads* termina de executar os comandos da região paralela, *threads* são sincronizadas e terminam.
  - Somente a *master thread* prossegue a execução depois da região paralela.
- Após a execução do código paralelo, a *master thread* **espera** automaticamente pelo fim da execução das demais *threads slaves* (*join*).
- Cada *thread* tem um **identificador** (id), do tipo inteiro, que pode ser consultado; *master thread* tem identificador 0.
- O número de *threads* para uma aplicação ou para cada trecho de código paralelo pode ser determinado de maneira automatizada ou selecionado pelo programador.

Construções OpenMP



https://computing.llnl.gov/tutorials/openMP/

Estrutura geral do código

```
#include <omp.h>

int main ()
{
  int var1, var2, var3;

  // Código serial
     ...
     ...
  // Início de seção paralela: geração das threads slaves
  // com definição do escopo de variáveis  usadas no trecho paralelo.
  #pragma omp parallel private(var1, var2) shared(var3)
  {
   // Seção paralela, executada por todas as threads
      ...
      ...
  } // Thread master espera pela conclusão (join) das demais threads

  // Retoma execução de trecho serial do código
     ...
     ...
}
```

OMP Directives, clauses, and functions

*OpenMP Compiler **directives**:*

- *parallel*
- *for / parallel for*
- *sections / parallel sections*
- *critical*
- *Single*
- *single*
- *master*
- *barrier*
- *taskwait*
- *atomic*
- *flush*

*OpenMP **Clauses**:*

- *if(scala-expression)*
- *num_threads(integer-expression)*
- *private(list)*
- *firstprivate(list)*
- *shared(list)*
- *default(none|shared)*
- *copyin(list)*
- *reduction(operator:list)*

*OpenMP **functions**:*

- *omp_get_num_procs(void);*
- *omp_set_num_threads(); omp_get_num_threads();*
- *omp_get_max_threads();*
- *omp_get_thread_num();*
- *omp_in_parallel();*
- *omp_set_dynamic(); omp_get_dynamic();*
- *omp_set_nested(); omp_get_nested();*
- *omp_get_thread_limit();*
- *omp_set_max_active_levels(); omp_get_max_active_levels()*
- *omp_get_level(); omp_get_active_level();*
- *omp_get_ancestor_thread_num();*
- *omp_get_team_size(int level);*
- *omp_in_final(void);*
- *omp_set_schedule(); omp_get_schedule();*
- *omp_init_lock(); omp_destroy_lock();*
- *omp_set_lock(); omp_unset_lock(); omp_test_lock();*
- *omp_init_nest_lock(); omp_destroy_nest_lock();*
- *omp_set_nest_lock(); omp_unset_nest_lock(); omp_test_nest_lock();*
- *double omp_get_wtime();*
- *double omp_get_wtick();*

OpenMP: Diretivas C/C++

#pragma omp	directive-name	[clause, ...]	newline
Obrigatório para todas as diretivas OpenMP C/C++	Diretiva OpenMP. Deve aparecer depois do *pragma* e antes de possíveis cláusulas (*clauses*).	Cláusulas opcionais. Podem ser definidas em qualquer ordem. Algumas não podem ser repetidas na mesma diretiva.	Comandos devem ser definidos na mesma linha (ou usando \). Precede um bloco estruturado que será tratado por esta diretiva.

Exemplo:

#pragma omp parallel default(shared) private(ind,pi)

Regras gerais:

- *Parâmetros são sensíveis ao uso de letras maiúsculas e minúsculas*
- *Diretivas seguem as convenções do padrão C/C++ para diretivas de compilação*
- *Diretivas são aplicáveis a comando, ou bloco, apresentado a seguir.*
- *Linhas longas podem ser ajustadas com "\"*

OpenMP: Diretivas C/C++

parallel *Region Construct*

Uma região paralela é um bloco de código que será executado por múltiplas *threads*.
É a construção paralela fundamental.

#pragma omp parallel [clause[[,]clause]...] new-line
structured-block

Cláusulas (*clauses*) incluem:

- ***if****(scalar-expression)*
- ***private****(variable-list)*
- ***firstprivate****(variable-list)*
- ***default****(shared | none)*
- ***copying****(variable-list)*
- ***reduction****(operator: variable-list)*
- ***num_threads****(integer-expression)*

parallel Region

- Ao encontrar uma diretiva **parallel**, um **time** (*team*) de *threads* é criado pela *thread* corrente, que se torna a ***master*** desse time, com **identificador 0**.
- *Threads* são criadas quando:
  - Não há uma cláusula ***if***
  - O resultado da expressão analisada pelo ***if*** é **diferente de 0**
  - Se expressão do ***if*** resulta em 0, região é executada de forma **serial**
- Código dentro da região paralela é **replicado** e executado por **todas** as *threads* desta região.
- Uma **barreira** é inserida automaticamente ao final do código desta região e somente a *master thread* **prossegue** depois deste ponto.
- Caso alguma *thread* **termine** dentro de uma região paralela, todas as *threads* no time correspondente serão **encerradas**.

parallel Region

- Número de *threads* é determinado por diversos fatores:
  1. Avaliação da cláusula **IF**
  2. Ajuste da cláusula ***num_threads***
  3. Uso da função ***omp_set_num_threads***()
  4. Ajuste da variável de ambiente **OMP_NUM_THREADS**
  5. Uso do valor *default* da implementação – comumente igual ao **número de processadores** (ou *cores*) do sistema

Dynamic Threads:

- Número de *threads* que executam a região paralela também depende se o **ajuste dinâmico** do número de *threads* está habilitado.
- Se suportada, a crição dinâmica de *threads* pode ser ativada com o uso da função ***omp_set_dynamic***() ou com o ajuste da variável de ambiente **OMP_DYNAMIC**

Modelo de programação OMP

```
#include <omp.h>

int a, b;

int main()
{
   // trecho de código sequencial: ajustes, inicializações, ...
   ...

   #pragma omp parallel num_threads(8)
      // trecho paralelo: executado por várias (num_threads) threads
   ...

   return(0);
}
```

Modelo de programação OMP

Exemplo de programa **OpenMP** com possível tradução para uso da biblioteca *pthreads* realizada por compilador com suporte a OpenMP.

```
int a, b;
main() {
  // serial segment
  #pragma omp parallel num_threads (8) private (a) shared (b)
  {
    // parallel segment
  }
  // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
  // serial segment

  for (i = 0; i < 8; i++)
      pthread_create (......., internal_thread_fn_name, ...);

  for (i = 0; i < 8; i++)
      pthread_join (.......);

  // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;
  // parallel segment
}
```

Code inserted by the OpenMP compiler

Corresponding Pthreads translation

parallel Region Construct

Regiões paralelas aninhadas (*nested parallel regions*):

- Se uma *thread* de um time executando uma região paralela encontra outra região paralela, ela cria um novo time e se torna a *master* desse time.
- Regiões paralelas aninhadas (*nested parallel regions*) são serializadas por padrão. Assim, região paralela aninhada é executada por um time composto por uma única *thread*.
- Função ***omp_get_nested***() indica se é permitido criar regiões paralelas aninhadas.
- Se suportada, a criação de regiões paralelas aninhadas pode ser ativada com o uso da função ***omp_set_nested***(), ou pelo ajusta da variável de ambiente **OMP_NESTED**.
- Se não suportada, região paralela aninhadas resulta na criação de um novo time composto de uma única *thread*.

Restrições da construção ***parallel***:

- Região paralela deve ser bloco estruturado
- Não é permitido entrar ou sair (*branch*) do trecho de código de uma região paralela na forma de desvios (*jumps, gotos, ...*)
- Apenas uma cláusula ***if*** é permitida
- Apenas uma cláusula ***num_threads*** é permitida e, se usada, deve resultar em valor inteiro positivo.
- Um programa não deve depender da ordem em que as cláusulas especificadas são processadas.

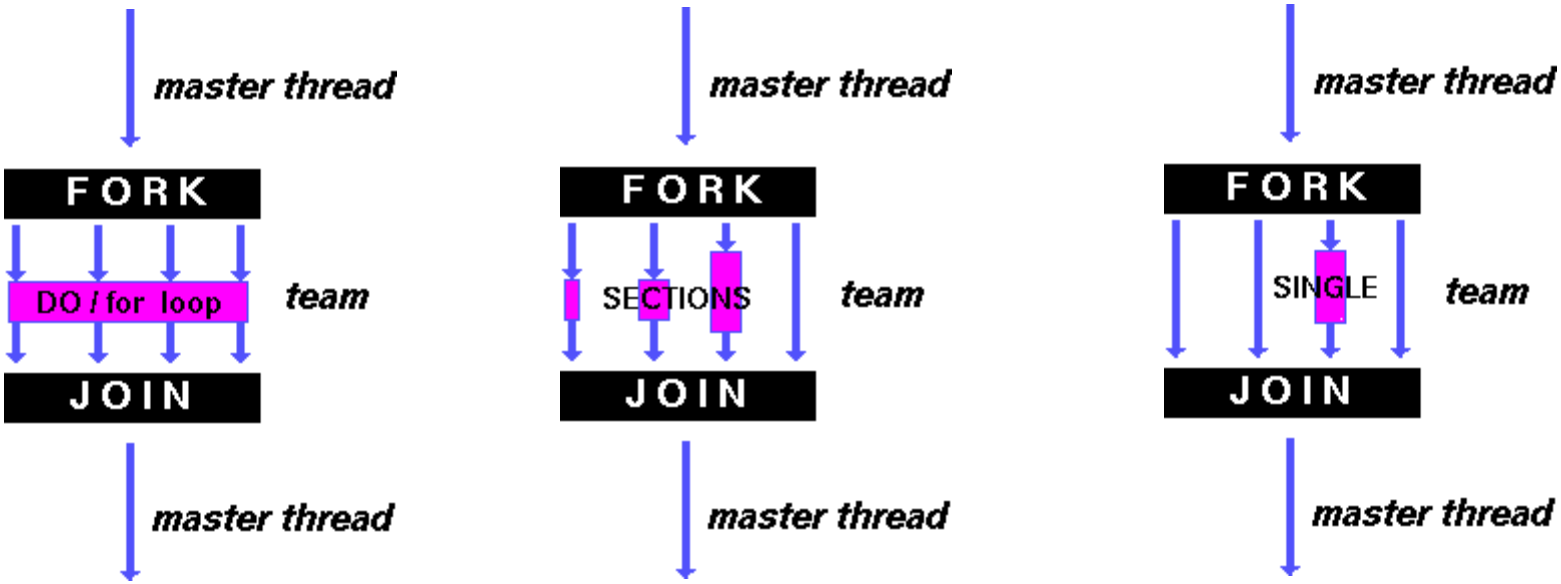
Work-Sharing Constructs

- Construções do tipo ***work-sharing*** possibilitam que o programador especifique como as operações de um bloco de código serão **distribuídas** (**divididas**) entre as *threads*.
- Construções do tipo ***work-sharing*** **não** geram novas *threads*.
- Não há uma barreira no início de uma construção desse tipo, mas uma é inserida automaticamente ao seu final.

Tipos de construções para divisão de carga (*work-sharing*):

- ***for / (DO*** *– fortran)*: dividem as iterações de um *loop* entre as *threads* do time. Representam o **paralelismo de dados**.
- ***sections***: dividem o trabalho em regiões explicitamente definidas. Cada seção é executada por uma *thread*. Pode ser usada para representar o **paralelismo funcional**.
- ***single***: serializa um trecho de código, que é executado por apenas 1 *thread* do time.
- ***workshare construct***: (*fortran* apenas)

Work-Sharing Constructs



https://computing.llnl.gov/tutorials/openMP/

Restrições:

- Uma construção do tipo ***work-sharing*** deve ser definida dentro de uma região paralela para que seja executada de forma paralela.
- Construções do tipo ***work-sharing*** devem estar no caminho de execução de **todas** as *threads* de um time ou de nenhuma.
- Construções do tipo ***work-sharing*** consecutivas devem ser atingidas no código na mesma ordem por todas as *threads* de um time.

Work-Sharing Constructs: for

for */ DO Directive*

- A diretiva ***for / DO*** especifica que as iterações do *loop* imediatamente abaixo devem ser executadas em paralelo pelas *threads* do time.
- Nesse caso, assume-se que esta diretiva está sendo emitida dentro de uma região paralela, ou a execução ocorrerá de maneira serial em um único processador.
- *for* deve ser especificado na forma canônica, sendo possível identificar o número de iterações, que não deve mudar durante a execução do *loop*.

```
#pragma omp for [clause ...]  newline
            schedule (type [,chunk])
            ordered
            private (list)
            firstprivate (list)
            lastprivate (list)
            shared (list)
            reduction (operator: list)
            collapse (n)
            nowait
  for_loop
```

Work-Sharing Constructs: for

- Para que o **compilador** possa **transformar** um *loop* sequencial em paralelo é preciso que seja capaz de verificar que o sistema em tempo de execução terá as informações necessárias para **determinar o número de iterações** ao avaliar a cláusula de **controle**.
- *For loops* devem possuir a seguinte forma canônica:

***for (indx** = start;*
***indx** {<,<=,>=,>} end;*
{indx++, ++indx, indx--, --indx, indx+=inc, indx -= inc, indx= indx+inc, indx=inc+indx, indx=indx-inc})

***for (init-expr**; **test-expr**; **incr-expr**) structured-block*

- ***Init-expr** is one of the following:var = lb, integer-type var = lb, random-access-iterator-type var = lb, pointer-type var = lb*
- ***Test-expr** is one of the following:var relational-op b, b relational-op var*
- ***Incr-expr** is one of the following:++var, var++, - -var, var- -, var+=incr, var-=incr, var=var+incr, var=incr+var, var=var– incr,*
- ***Var** is one of the following: a variable of a signed or unsigned integer type. For C++, a variable of a random access iterator type. For C, a variable of a pointer type. If this variable would otherwise be shared, it is implicitly made private in the loop construct. This variable must not be modified during the execution of the for-loop other than in incr-expr. Unless the variable is specified lastprivate on the loop construct, its value after the loop is unspecified.*
- ***Relational-op** is one of the following: <, <=, >, >=*
- ***lb and b**: loop invariant expressions of a type compatible with the type of var.*
- ***incr**: a loop invariant integer expression.*

Work-Sharing Constructs: for

for */ DO directive* - Cláusulas (*clauses*):

- ***schedule***: determina como as iterações do *loop* são divididas entre *threads* do time.
  - ***static***: iterações divididas em bloco de tamanho *chunk*.
  - ***dynamic***: iterações divididas em blocos de tamanho *chunk* e alocadas dinamicamente entre as *threads*, à medida que terminam as iterações atribuídas anteriormente.
  - ***guided***: número de iterações atribuído em cada rodada é calculado em função das iterações restantes divididas pelo número de *threads*, sendo o resultado decrescido de *chunk*.
  - ***runtime***: decisão de atribuição é realizada somente em tempo de execução, usando a variável de ambiente OMP_SCHEDULE.
  - ***auto***: decisão de atribuição é delegada ao compilador ou software de de tempo de execução.
- ***nowait***: se usada, indica que *threads* não devem ser sincronizadas no fim do *loop* paralelo.
- ***ordered***: indica que as iterações do loop devem ser executadas como se fossem trataras em um programa serial.
- ***collapse***: indica quantos *loops* em um aninhamento de *loops* (*nested loops*) devem ser agrupados (*collapsed*) em um bloco de iteração maior dividido de acordo com a cláusula *schedule*.

Work-Sharing Constructs: for

Restrições:

- ***for loop*** não pode ser um ***while*** ou um *loop* sem controle.
- Variável de controle das iterações do *loop* deve ser do tipo inteiro e os parâmetros de controle do *loop* devem ser o mesmo para todas as *threads*.
- Corretude do programa não pode depender da ordem em que as iterações são executadas.
- Não é permitido executar um desvio de saída de um *loop* associado à diretiva ***for***.
- Tamanho do *chunk* para escalonamento deve ser especificado como uma expressão de valor inteiro que não varia durante o *loop*; não há sincronização na sua avaliação por diferentes *threads*.
- Cláusulas ***ordered***, ***collapse*** e ***schedule*** podem ser usadas apenas uma vez no *loop*.

Work-Sharing Constructs: sections

sections *directive*

- Diretiva ***sections*** permite especificar **seções** dentro de uma região paralela que devem ser divididas entre as *threads* de um time.
- Cada *section* é executada uma vez por uma *thread* no *time*.
- Diferentes seções podem ser executadas por *threads* diferentes.
- Uma *thread* pode executar mais de uma seções, se houver mais seções do que *threads* no time

#pragma omp sections [clause ...] newline

private (list)
firstprivate (list)
lastprivate (list)
reduction (operator: list)
nowait

```
{
[#pragma omp section
/* structured block */
]
[#pragma omp section
/* structured block */
]
...
}
```

- **Barreira** é inseria ao final da diretiva ***section***, a menos que a cláusula ***nowait*** seja especificada.
- Número de seções pode diferir do número de *threads* disponíveis. Escalonamento depende da implementação
- **Restrição**:Não é permitido realizar desvios para fora ou para dentro dos blocos de seções.

Work-Sharing Constructs: single

single *directive*

- Diretiva *single* especifica que o código associado deve ser executado por apenas uma *thread* no time, não necessariamente a *master*.
- Normalmente, é útil no tratamento de seções de código que não são *thread safe* (como E/S)

#pragma omp single [clause ...] newline
private (list)
firstprivate (list)
nowait

structured_block

Cláusulas (*clauses*):
- *Theads* no time que não executam a diretiva *single* esperam no final do código associado, exceto se a cláusula *nowait* for especificada.

Restrição:
- Não é permitido realizar desvios para fora ou para dentro de um bloco *single*.

Combined parallel work-sharing Constructs

- Diretivas para particionamento do código podem ser expressas de maneira **condensada**
- Diretivas condensadas têm o mesmo comportamento de uma diretiva *parallel* para um segmento de código, seguida apenas de uma diretiva de divisão de carga (*work-sharing*).
- Maioria das regras, cláusulas e restrições são as mesmas das diretivas não condensadas.
- *parallel for*
- *parallel sections*

Versão completa	Versão combinada
#pragma omp parallel *{* * #pragma omp for* * for-loop* *}*	***#pragma omp parallel for*** *[clause[[,] clause] ...] new-line* * for-loop*
#pragma omp parallel *{* * #pragma omp setcions* * {* * [#pragma omp section]* * Structured block* * #pragma omp section* * Structured block* * ...* * }* *}*	***#pragma omp parallel sections*** *[clause[[,]clause] ...] new-line* *{* *[**#pragma omp section** new-line]* *structured-block* *[**#pragma omp section** new-line* *structured-block]* *...* *}*

OpenMP Directives

task *construct*

- ***Task*** (introduzida com OpenMP 3.0) define uma tarefa específica que pode ser executada pela *thread* que a encontrar ou deferida para execução por qualquer outra *thread* num time.
- O ambiente de dados de uma *task* é determinado pelas cláusulas de atributos de compartilhamento de dados e valores *default*.
- Execução de uma *task* está sujeita ao escalonamento de tarefas.

#pragma omp task [clause ...] newline
structured_block

if (scalar expression)
untied
default (shared | none)
private (list)
firstprivate (list)
shared (list)

Synchronization Directives

Construções OpenMP para **sincronização**:

- *#pragma omp **barrier***
- *#pragma omp **single** [clause list]*
  structured block
- *#pragma omp **master***
  structured block
- *#pragma omp **critical** [(name)]*
  structured block
- *#pragma omp **ordered***
  structured block
- *#pragma omp **flush** [(list)] new-line*
- *#pragma omp **taskwait** newline*
- *#pragma omp **atomic** new-line*
  Expression-stmt
- *#pragma omp **for ordered** [clauses...]*
  (loop region)
- *#pragma omp **ordered** newline*
  structured_block
  (endo of loop region)
- *#pragma omp **threadprivate**(list) new-line*

Synchronization Directives

master *directive*

- A diretiva ***master*** especifica um bloco de código que deve ser executado pela *thread* ***master*** **do time**.
- Outras threads no time ignoram esse trecho de código.
- Não há barreira nem no início e nem no final da execução desta construção.

#pragma omp ***master*** *new-line*
*structured-block*MASTER Directive

Synchronization Directives

***critical** directive*

- Especifica região de código que deve ser executada por apenas uma *thread* de cada vez.
- Uma *thread* espera no início de uma região crítica até que nenhuma outra *thread* esteja executando a mesma região crítica.
- Restrição de execução é para **todas as *threads* do programa**, e não apenas entre as *threads* do time atual.
- Um nome pode ser usado opcionalmente para identificar a região crítica.
- Todas as especificações de regiões críticas sem nome são consideradas tratar-se da mesma região.

#pragma omp critical [name] newline
* structured_block*

Exemplo: todas as threads em um time executam em paralelo mas uso da diretiva critical faz com que os acessos à variável x ocorram um de cada vez.

```
int x = 0;
#pragma omp parallel shared(x)
{
  #pragma omp critical
  x = x + 1;
}
```

Synchronization Directives

barrier *Directive*

- Diretiva ***barrier*** sincroniza **todas** as *threads* em um **time**.
- Ao encontrar uma diretiva *barrier*, uma *thread* será bloqueada até que todas as outras *threads* do time cheguem na barreira.
- Todas as threads prosseguem a execução depois da barreira.

#pragma omp barrier newline

Restrição:

- Todas as *threads* (ou nenhuma) em um time devem atingir o código da barreira.

Synchronization Directives

***taskwait** Directive*

- ***taskwait*** determina uma espera pelo término das tarefas geradas desde o início da tarefa corrente.

C/C++ format:

#pragma omp taskwait newline

- Restrições: *because the taskwait construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The taskwait directive may be placed only at a point where a base language statement is allowed. The taskwait directive may not be used in place of the statement following an if, while, do, switch, or label.*

Exemplos:

```
void a23_wrong()
{
  int a = 1;
  #pragma omp parallel
  {
  if (a != 0)
    #pragma omp taskwait
    /* incorrect as taskwait cannot be
       immediate substatement of if statement */
  }
}

void a23()
{
  int a = 1;
  #pragma omp parallel
  {
  if (a != 0) {
    #pragma omp taskwait
  }
  }
}
```

Synchronization Directives

atomic *Directive*

- ***atomic*** determina que uma posição de memória específica deve ser atualizada de maneira **atômica**, impedindo múltiplas *threads* de tentar escrever nessa localização simultaneamente.

#pragma omp atomic new-line
expression-stmt

expression-stmt é uma expressão com um entre os formatos seguintes:

- *x binop= expr*
- *x++*
- *++x*
- *x--*
- *--x*

Nessas expressões:

- *x is an lvalue expression with scalar type.*
- *expr is an expression with scalar type, and it does not reference the variable designated by x.*
- *binop is one of +, *, -, /, &, ^, |, <<, or >>.*
- *binop, binop=, ++, and -- are not overloaded operators.*

Restrições:

- *The directive applies only to a single, immediately following statement*
- *An atomic statement must follow a specific syntax.*
- *All atomic references to the storage location of each variable that appears on the left-hand side of an atomic assignment statement throughout the program are required to have a compatible type.*

Synchronization Directives

flush *Directive*

- ***flush*** torna a memória visível pela *thread* consistente com a memória e garante uma ordem nas operações de memória das variávies especificadas ou relacionadas (*implied*).
- Variáveis visíveis pela *thread* são gravadas na memória neste ponto.

#pragma omp ***flush*** *[(list)] new-line*

- *Note that because the flush construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program.*
- *The binding thread set for a flush region is the encountering thread. Execution of a flush region affects the memory and the temporary view of memory of only the thread that executes the region. It does not affect the temporary view of other threads. Other threads must themselves execute a flush operation in order to be guaranteed to observe the effects of the encountering thread's flush operation.*
- *A flush construct with a list applies the flush operation to the items in the list, and does not return until the operation is complete for all specified list items.*
- *A flush construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed.*
- *If a pointer is present in the list, the pointer itself is flushed, not the memory block to which the pointer refers.*

Synchronization Directives

flush *Directive*

A flush region without a list ***is implied*** *at the following locations:*

- *During a barrier region.*
- *At entry to and exit from parallel, critical, and ordered regions.*
- *At exit from worksharing regions unless a nowait is present.*
- *At entry to and exit from combined parallel worksharing regions.*
- *During omp_set_lock and omp_unset_lock regions.*
- *During omp_test_lock, omp_set_nest_lock, omp_unset_nest_lock and omp_test_nest_lock regions, if the region causes the lock to be set or unset.*
- *Immediately before and immediately after every task scheduling point.*

A flush region with a list ***is implied*** *at the following locations:*

- *At entry to and exit from atomic regions, where the list contains only the variable updated in the atomic construct.*

A flush region ***is not implied*** *at the following locations:*

- *At entry to worksharing regions.*
- *At entry to or exit from a master region.*

Synchronization Directives

ordered *Directive*

- ***ordered*** especifica que um bloco estruturado em um *loop* será executado na ordem das iterações do *loop*. Isto serializa e ordena o código dentre de uma região, permitindo que o código fora desta região execute em paralelo.

#pragma omp ***for ordered*** *[clauses...]*
(loop region)
#pragma ***omp ordered*** *newline*
structured_block
(endo of loop region)

Binding

- *The binding thread set for an ordered region is the current team. An ordered region binds to the innermost enclosing loop region. ordered regions that bind to different loop regions execute independently of each other.*

Description

- *The threads in the team executing the loop region execute ordered regions sequentially in the order of the loop iterations. When the thread executing the first iteration of the loop encounters an ordered construct, it can enter the ordered region without waiting. When a thread executing any subsequent iteration encounters an ordered region, it waits at the beginning of that ordered region until execution of all the ordered regions belonging to all previous iterations have completed.*

Restrictions to the ordered construct are as follows:

- *The loop region to which an ordered region binds must have an ordered clause specified on the corresponding loop (or parallel loop) construct.*
- *During execution of an iteration of a loop or a loop nest within a loop region, a thread must not execute more than one ordered region that binds to the same loop region.*

Synchronization Directives

threadprivate *directive*

- ***threadprivate*** *especifica que variávies sejam repilicadas, com cada thread tendo sua própria cópia.*
- ***threadprivate*** é usada para tornar variáveis globais locais e persistentes às *threads* ao longo da execução de múltiplas regiões paralelas.

#pragma omp ***threadprivate****(list) new-line*

- *where list is a comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.*

Restrições:

- *Cada cópia de uma variável threadprivate é inicializada apenas uma vez, da forma especificada pelo programa, mas em um ponto qualquer no programa antes da primeira referência àquela cópia.*
- Uma *thread* não deve acessar a cópia de uma variável *threadprivate* de outra *thread*.
- *The directive must appear after the declaration of listed variables/common blocks. Each thread then gets its own copy of the variable/common block, so data written by one thread is not visible to other threads.*
- *On first entry to a parallel region, data in THREADPRIVATE variables and common blocks should be assumed undefined, unless a COPYIN clause is specified in the PARALLEL directive*
- *THREADPRIVATE variables differ from PRIVATE variables because they are able to persist between different parallel sections of a code.*
- *Data in THREADPRIVATE objects is guaranteed to persist only if the dynamic threads mechanism is "turned off" and the number of threads in different parallel regions remains constant. The default setting of dynamic threads is undefined.*
- *The THREADPRIVATE directive must appear after every declaration of a thread private variable/common block*

OpenMP functions

void ***omp_set_num_threads*** *(int);* *int* ***omp_get_num_threads*** *(void);* *int* ***omp_get_max_threads*** *(void);* *int* ***omp_get_thread_num*** *(void);* *int* ***omp_get_num_procs*** *(void);* *int* ***omp_get_thread_limit****(void);*	*void* ***omp_init_lock*** *(omp_lock_t *);* *void* ***omp_destroy_lock*** *(omp_lock_t *);* *void* ***omp_set_lock*** *(omp_lock_t *);* *void* ***omp_unset_lock*** *(omp_lock_t *);* *int* ***omp_test_lock*** *(omp_lock_t *);*
void ***omp_set_dynamic****(int dynamic_threads);* *int* ***omp_get_dynamic****(void);* *void* ***omp_set_nested****(int nested);* *int* ***omp_get_nested****(void);*	*void* ***omp_init_nest_lock*** *(omp_nest_lock_t *);* *void* ***omp_destroy_nest_lock*** *(omp_nest_lock_t *);* *void* ***omp_set_nest_lock*** *(omp_nest_lock_t *);* *void* ***omp_unset_nest_lock*** *(omp_nest_lock_t *);* *int* ***omp_test_nest_lock*** *(omp_nest_lock_t *);*
int ***omp_in_parallel*** *(void);*	*double* ***omp_get_wtime*** *(void);* *double* ***omp_get_wtick*** *(void);*
void ***omp_set_max_active_levels****(int max_active_levels);* *int* ***omp_get_max_active_levels****(void);*	*void* ***omp_set_schedule****(omp_sched_t kind, int modifier);* *void* ***omp_get_schedule****(omp_sched_t *kind, int *modifier);*
int ***omp_get_level****(void);* *int* ***omp_get_active_level****(void);*	*int* ***omp_get_ancestor_thread_num****(int level);* *int* ***omp_get_team_size****(int level);*

OpenMP functions

- ***omp_get_num_procs( )***: retorna o número de processadores físicos disponíveis para o programa paralelo.
- ***omp_set_num_threads( )***: determina o número de *threads* que estarão ativas nas seções paralelas de código. Permite ajustar o nível de paralelismo de acordo com a granularidade ou outras características de um bloco de código.
- ***omp_get_num_threads( )***: retorna o número de *threads* ativas num time.
- ***omp_get_thread_num( )***: retorna o identificador na *thread* no time atual.

```
int t;
...
t=omp_get_num_procs();
omp_set_num_threads(t);
```

- *void omp_set_num_threads (int);*
- *int omp_get_num_threads (void);*
- *int omp_get_max_threads (void);*
- *int omp_get_thread_num (void);*
- *int omp_get_num_procs (void);*

OpenMP Clauses

- Cláusulas (*clauses*) usadas nas diretivas permitem especificar a **paralelização condicional**, definir o **número de *threads*** e tratar o **compartilhamento** dos dados.
  - **Paralelização condicional**: *if (scalar expression)* determina se a construção paralela será executada por múltiplas *threads*.
  - **Grau de concorrência**: *num_threads(integer expression)* determina o número de threads que serão criadas.
  - **Compartilhamento de dados**: *private (variable list)*, *firstprivate (variable list)*, *shared (variable list)* e *lastprivate(variable list)* tratam do compartilhamento de variáveis entre as threads.
- No modelo de programação baseado em memória compartilhada provido por ***OpenMP***, a maioria da variáveis é naturalmente visível por todas as *threads*.
- Cláusulas de compartilhamento de dados (***data sharing clauses***) podem ser inseridas nas diretivas e permitem especificar variáveis que serão privadas para as *threads*.
- Cláusulas também servem para evitar condições de disputa (*race conditions*) e possibilitam passar valores entre a parte sequencial e regiões paralelas.

Três tipos de cláusulas (*clauses*) são definidos:

- ***Data-sharing attribute clauses / Data Copying clauses***
- ***Synchronization clauses***
- ***Scheduling clauses***

OpenMP Clauses

Data-shaging clauses são aplicadas a variáveis visíveis na construção em que a cláusula é usada.

- ***default****(shared|none)*: controla o modo de compartilhamento padrão para variáveis referenciadas nas construções *parallel* ou *task*. Opção *none* força o programador a declarar explicitamente cada item como compartilhado ou privado.
- ***shared****(list)*: declara um ou mais itens que serão compartilhados pelas *threads* geradas por *parallel* ou *task*. Por *default*, todas as variáveis em uma região paralela são compartilhadas, exceto pelo contador das iterações.
- ***private****(list)*: declara um ou mais itens privados para uma tarefa. Cada *thread* terá uma cópia desses itens, cujos valores iniciais não são controlados. Variáveis privadas não são preservadas ao final da execução das *threads*.
- ***firstprivate****(list)*: declara um ou mais itens privados para uma *thread* e inicia seu valor de acordo com o valor original quando a diretiva é emitida.
- ***lastprivate****(list)*: declara um ou mais itens privados para as *threads* e faz com que o valor final seja atualizado na variável original ao final da execução da região paralela.
- ***reduction****(operator**:**list)*: declara o acúmulo de valores de uma lista de itens usando o operador indicado. Acúmulo é parcialmente feito em cópias locais de cada item da lista, posteriormente combinadas no item original.

OpenMP Clauses

Synchronization clauses

- ***critical section***: indica que o bloco de código será executado por apenas uma *thread* de cada vez. É comumente usada para proteger o acesso a dados compartilhados e evitar condições de disputa (*race conditions*).
- ***atomic***: é similar à seção crítica, mas indica ao compilador para usar instruções do *hardware* visando ao melhor desempenho. Sugestão pode ser ignorada pelo compilador, que usaria seção crítica.
- ***ordered***: indica que o bloco de código será executado na ordem em que as iterações seriam executadas em um *loop* sequencial.
- ***barrier***: faz com que cada *thread* espere até que todas as outras *threads* do time tenham atingido esse ponto. Construções do tipo *work-sharing* têm uma barreira para sincronização implícita no final.
- ***nowait***: indica que as *threads* que completarem suas atribuições podem prosseguir sem ter que esperar pelas demais concluírem suas respectivas partes. Na ausência dessa cláusula, *threads* serão sincronizadas numa barreira ao final de uma construção *work sharing*.

OpenMP Clauses

Scheduling *clauses*

schedule*(type, chunk)*: determina como as iterações do *loop* são divididas entre *threads* do time.

- ***static***: iterações são divididas em bloco de tamanho *chunk*.
- ***dynamic***: iterações são divididas em blocos de tamanho *chunk* e alocadas dinamicamente entre as *threads*, à medida que terminam as iterações a elas atribuídas anteriormente.
- ***guided***: número de iterações atribuído em cada rodada é calculado em função das iterações restantes divididas pelo número de *threads*, sendo o resultado decrescido de *chunk*.
- ***runtime***: decisão de atribuição é realizada somente em tempo de execução, usando a variável de ambiente OMP_SCHEDULE.
- ***auto***: decisão de atribuição é delegada ao compilador ou *software* em tempo de execução.

OpenMP Clauses

if control

- ***if*** faz com que *threads* sejam utilizadas para paralelizar um bloco de código apenas **se uma condição for satisfeita**. Caso contrário, código é executado **serialmente**.

Initialization

- ***firstprivate***: dados são privados para cada *thread* e valores iniciais são atribuídos em função das variáveis de mesmo nome na *master thread*.
- ***lastprivate***: dados são privados para cada *thread*. Valores são copiados da *thread* que realizar a última iteração de um *loop*. Variáveis podem ser simultaneamente *firstprivate* e *lastprivate*.
- ***threadprivate***: dados são globais mas privados em cada região paralela em tempo de execução. Diferença de *threadprivate* e *private* é o escopo global associado a *threadprivate* e a preservação dos valores entre regiões paralelas.

Data copying

- ***copyin***: do mesmo modo que ocorre com *firstprivate* para variáveis privadas, variáveis definidas com *threadprivate* não são inicializadas a menos que se use *copyin* para atribuir o valor correspondente das variáveis globais. Não é necessário existir ***copyout*** pois variáveis *threadprivate* preservam o valor durante toda a execução do programa.
- ***copyprivate***: usado com diretiva *single* para possibilitar a cópia de variáveis privadas em uma *thread* para os objetos correspondentes nas demais *threads* no time.

OpenMP Clauses

Reduction

- ***reduction(operator | intrinsic : list)***: especifica como múltiplas cópias de uma variável nas diferentes *threads* são combinadas (*reduced*) em uma única cópia na *thread master* quando as *threads* terminarem.
- Variáveis na lista são tratadas como **privadas** (*private*) para as *threads*.
- **Operadores**: +, *, -, &, |, ^,&&, ||

```
#pragma omp parallel reduction(+: sum)
{
  /* calcula somas parciais (sum) nas threads */
}
/* aqui, sum contém a soma de todas as instâncias de sum */
```

Operadores	Valores iniciais
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

OpenMP: variáveis de ambiente

Variáveis de ambiente permitem alterar aspectos da execução de aplicações OpenMPI.

São usadas para controlar o escalonamento das iterações de *loops*, o número *default* de *threads*, etc.

- **OMP_SCHEDULE**: usada na diretiva parallel for, determina como as iterações do loop são escalonadas aos processadores. Ex: *setenv OMP_SCHEDULE "dynamic"*
- **OMP_NUM_THREADS**: determina o número máximo de threads para uso durante a execução. Ex: *setenv OMP_NUM_THREADS 8*
- **OMP_DYNAMIC**: habilita (TRUE ou FALSE) o ajuste dinâmico do número de threads disponíveis para a execução de regiões paralelas. Ex: *setenv OMP_DYNAMIC TRUE*
- **OMP_NESTED**: habilita o uso de paralelismo aninhado.
- **OMP_STACKSIZE**: controla o tamanho da pilha para *threads* criadas (non-Master). Ex: *setenv OMP_STACKSIZE 2000500B*, *setenv OMP_STACKSIZE "3000 k "*, *setenv OMP_STACKSIZE 10M*
- **OMP_WAIT_POLICY**: fornece dicas à implementação OpenMP sobre o comportamento de threads em estado de espera (*waiting threads*). *ACTIVE specifies that waiting threads should mostly be active, i.e., consume processor cycles, while waiting. PASSIVE specifies that waiting threads should mostly be passive, i.e., not consume processor cycles, while waiting. The details of the ACTIVE and PASSIVE behaviors are implementation defined. Ex: setenv OMP_WAIT_POLICY ACTIVE*; *setenv OMP_WAIT_POLICY active*; *setenv OMP_WAIT_POLICY PASSIVE*
- **OMP_MAX_ACTIVE_LEVELS**: controla o número máximo de regiões paralelas ativas aninhadas. Ex: *setenv OMP_MAX_ACTIVE_LEVELS 2*
- **OMP_THREAD_LIMIT**: ajusta o número de *threads* a serem usadas ao todo no programa OpenMP. Ex: *setenv OMP_THREAD_LIMIT 8*

Questões de desempenho

Desempenho esperado com o uso de **N** *threads* pode **não** ser **N** vezes **superior**:

- *Lei de Amdahl*: ganho de desempenho com o paralelismo está teoricamente limitado pela porção sequencial do programa.
- Equipamento com N processadores pode ter poder computacional N vezes superior, mas outros elementos do sistema podem limitar a escalabilidade, como a concorrência no acesso à memória compartilhada.
- Problemas de balanceamento de carga e sincronização podem limitar ganhos de desempenho.

OpenMP x Explicit Thread programming

Prós:

- Uso de diretivas **facilita** o geranciamento das *threads*.
- Programação não precisa preocupar-se com atributos e argumentos para *threads* e com o particionamento do código.
- **Decomposição** dos dados é feita de maneira automática.
- Diferentes **granularidades** de código podem ser ajustadas.
- **Paralelismo** pode ser implementado de maneira **incremental**, sem mudança drástica no código.
- **Mesmo código** pode ser usado para versão sequencial e paralela da aplicação.

Contras:

- Requer **compilador** com suporte a **OpenMP** (maioria atualmente).
- Limitado a multiprocessadores com memória compartilhada (ver *Intel's Cluster OpenMP*).
- **Escalabilidade** pode ser limitada pela arquitetura de acesso à memória.
- Tratamento de **erros** mais refinado pode ser necessário.
- Não possui mecanismo para controle do mapeamento das *threads* entre processadores.
- Trocas de dados explícitas na programação com *threadas* oferece controle mais refinado nos **compartilhamentos**.
- Tratamento explícito das *threads* oferece API com mais recursos para controlar **sincronizações** e **bloqueios**.