OpenMP

Mestrado em Ciência da Computação

Introdução ao Processamento Paralelo e Distribuído

Prof. Dr. Adenauer Corrêa Yamin

Mestranda: Nelsi Warken

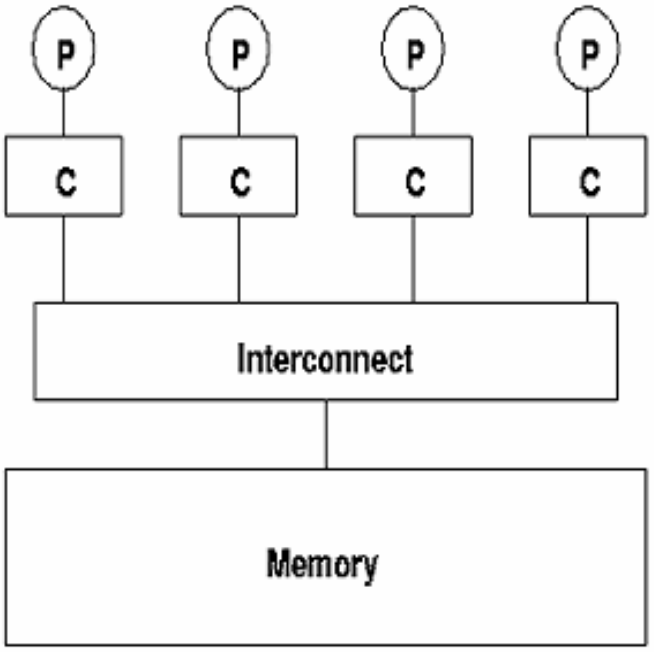
Sumário

1. Introdução

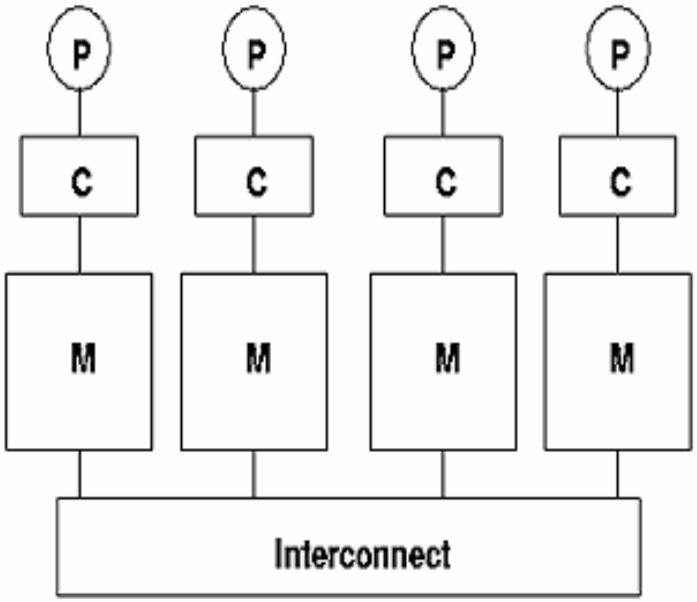
- OpenMP – Open Multi-Processing é uma interface de programação de aplicação (API), multi-plataforma, multi-processamento e de memória compartilhada.
- C/C++ e Fortran, plataformas UNIX e MicroSoft Windows.
- Conjunto de diretivas de compilação, bibliotecas de funções e variáveis de ambiente, que influenciam o ambiente em tempo de execução.
- Organização das arquiteturas paralelas: arquitetura de memória distribuída e arquitetura de memória compartilhada.
- API para programação paralela, onde o paralelismo utiliza 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.

1. Introdução

Sistemas Paralelos de Memória Compartilhada



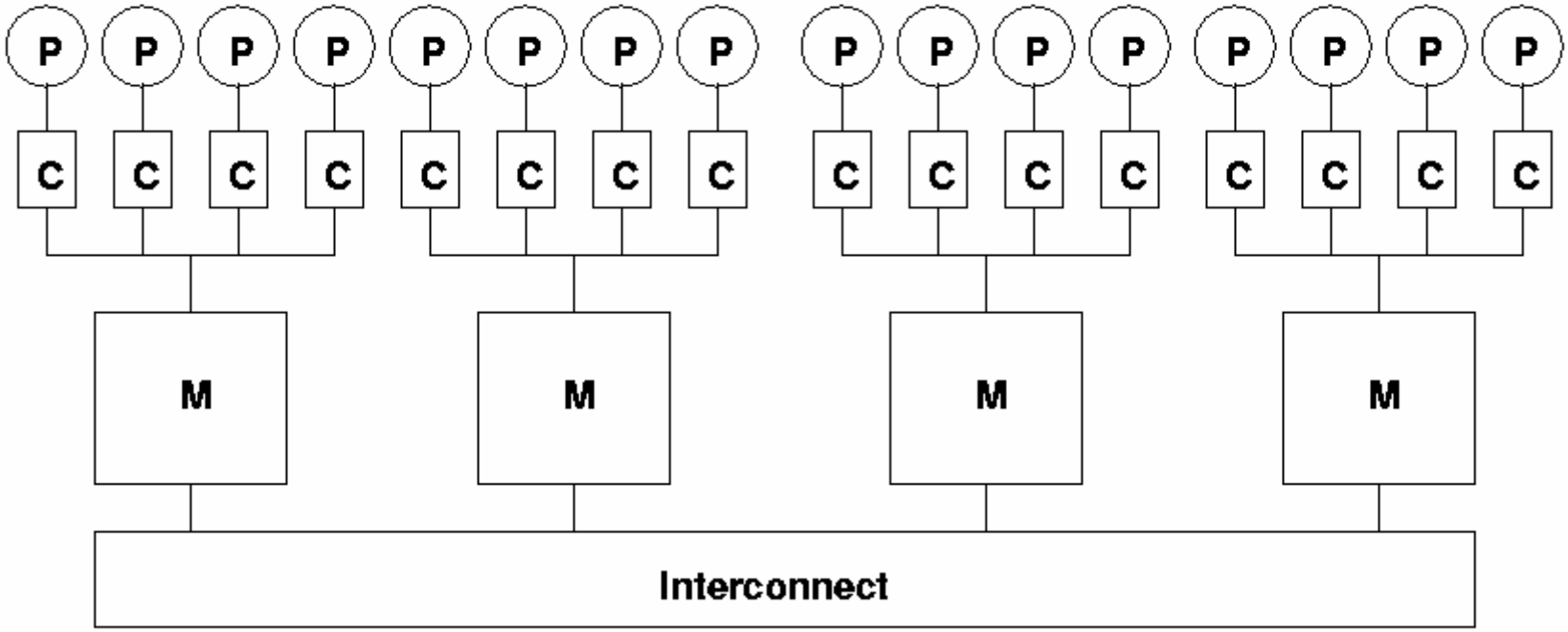
True shared memory



Distributed shared memory

1. Introdução

Sistemas Paralelos de Memória Compartilhada



Clustered distributed shared memory

2. Histórico

- OpenMP Architecture Review Board (ARB): corporação sem fins lucrativos, proprietária da marca OpenMP, supervisiona a especificação, produz e aprova novas versões do OpenMP. Também ajuda a organizar e financiar conferências, seminários, workshops e outros eventos relacionados, além de promover o OpenMP.

- Última versão oficial: 3.0 (Maio-2008): C/C++ e Fortran
- 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
- Traduções: Versão 3.0 - Japonês (*draft*)

2. Histórico

Compiladores de fornecedores ou comunidades de *open source* que implementam a API OpenMP.

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

2. Histórico

- 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 (University of Houston)
  - OpenMP Validation Suite (HPC Center, Stuttgart)
  - KOJAK: Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks (Jülich Supercomputing Center) - (gargalos).

- 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.

3. Motivação

- A evolução dos microprocessadores: muito maior desempenho, mas limitada pelas leis da física na tecnologia de desenvolvimento de semicondutores. As aplicações mais sofisticadas e complexas, e com exigências de computação de alta velocidade.
  Solução: arquiteturas inovadoras e o paralelismo.

- SMP, Multithread, Multicore, necessidade de uma linguagem de programação com ferramentas e mecanismos de controle de coerência, controle de concorrência, reengenharia e principalmente novos algoritmos.

- Ambiente motivador para o desenvolvimento de um padrão de programação para processamento paralelo.

- 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 compacto, mas poderoso, modelo de programação para aplicações paralelas de memória compartilhada;
- interface simples;
- suportar Fortran, C, C + +;
- programas portáveis para uma ampla gama de plataformas (Unix, Windows), desde desktop a supercomputadores;
- rodar em um cluster usando OpenMP e MPI (Interface de Passagem);
- ser escrito enquanto a versão seqüencial ainda está em construção.

4. Objetivos - Aplicações

- Previsão do tempo, modelos de ondas e modelos de oceanos - (LAMBO - Limited Area Model Bologna), um modelo de ondas (WA.M. - Wave Model, no Mar Mediterrâneo e no Mar Adriático) e um modelo de oceanos (M.O.M. -- Modular Ocean Model).
- Novel FDTD – Algorítmo 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, 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 várias distribuições como Fedora Core 5 gcc também suportam OpenMP.

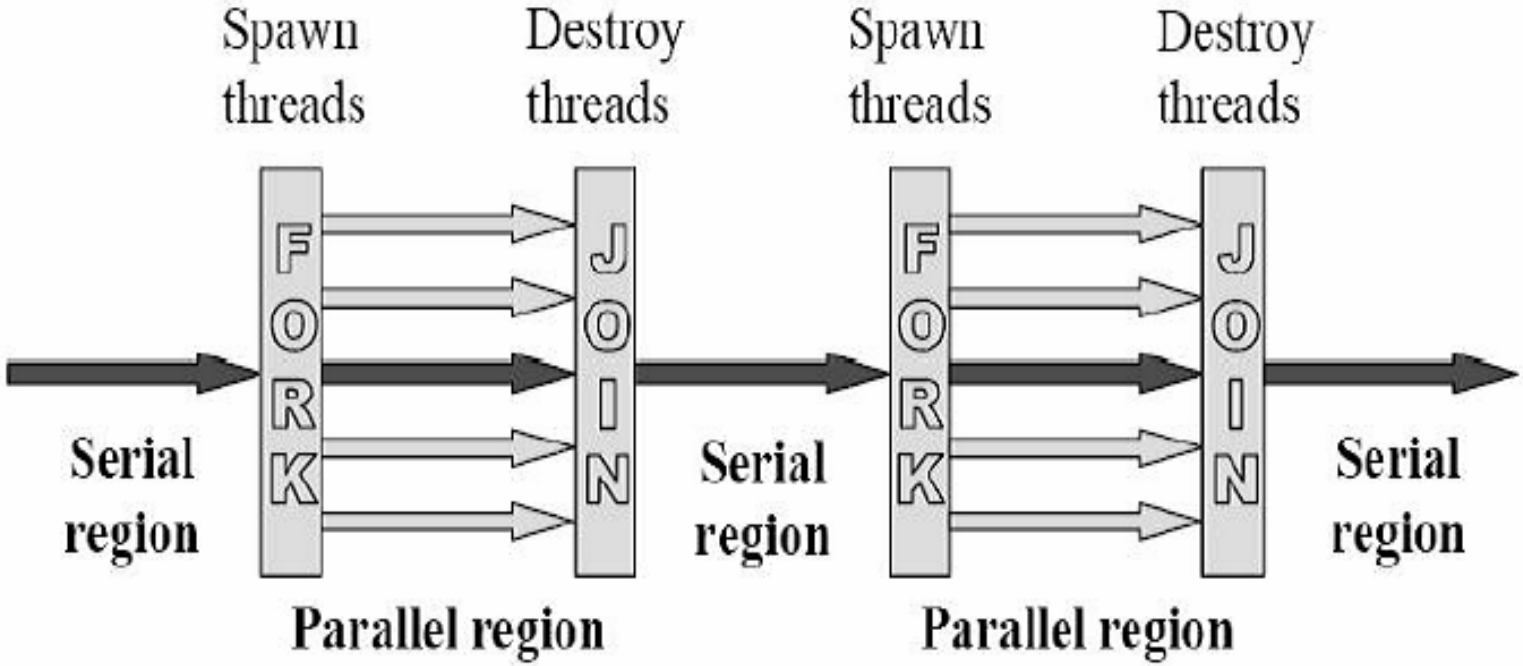
5. Funcionalidades

OpenMP é uma aplicação *multithreading*, um método de paralelização onde a *master thread* separa/divide (*forks)* um número determinado de *threads* escravas, onde uma tarefa é dividida entre elas. O ambiente *runtime* aloca *threads* para diferentes processadores, sendo executadas simultaneamente.

Após a execução do código paralelizado, as *threads* são unidas à *master thread* , que continuará a execução dali em diante até o fim do programa.

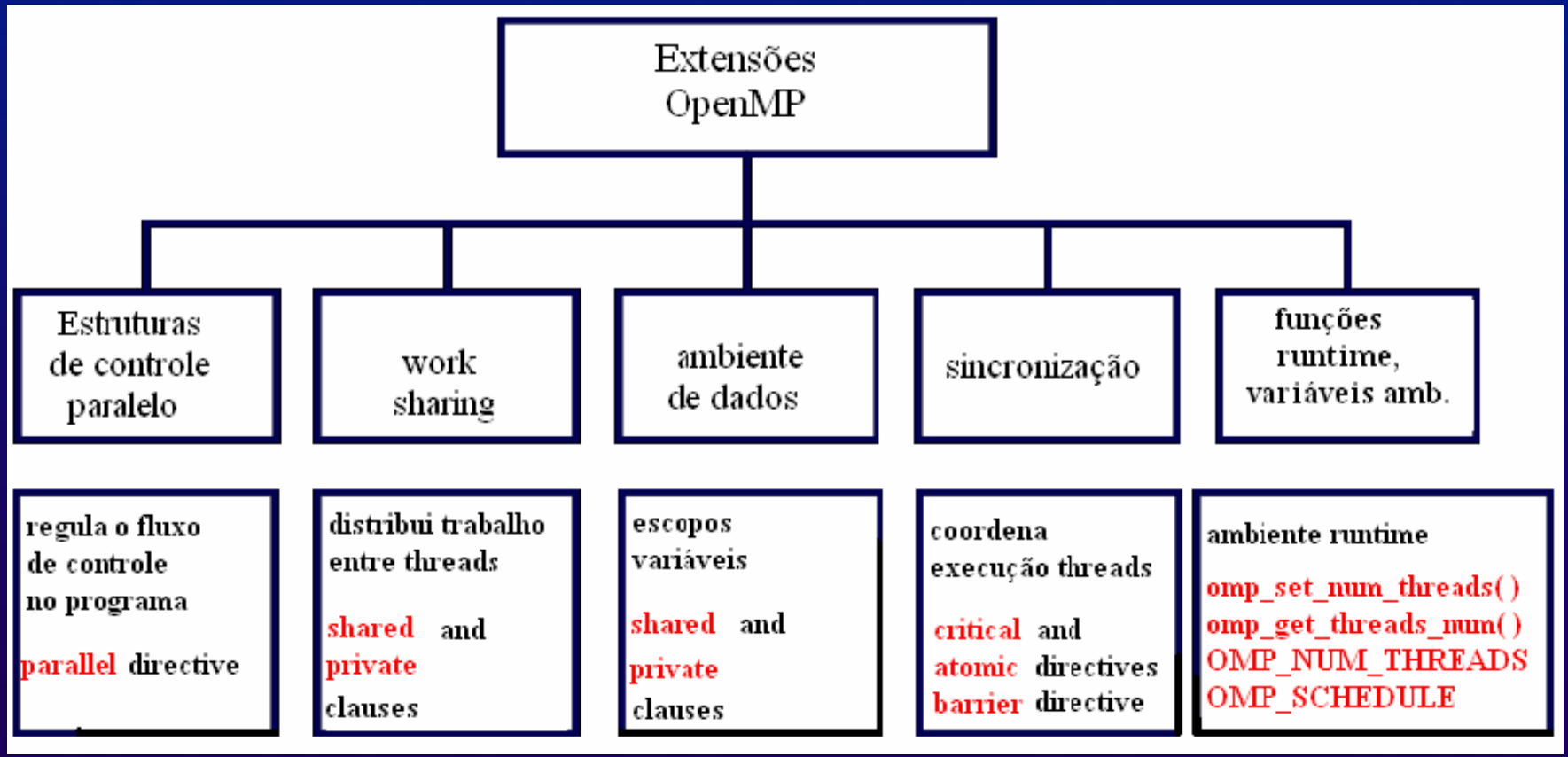
A seção de código para rodar em paralelo estará marcada de acordo com uma diretiva pré-processador que formará as *threads* antes da seção ser executada. Cada *thread* tem um "id". A *thread id* é um inteiro e a *thread master* id = 0.

5. Funcionalidades - *Multithreading*
Modelo de Execução: fork-join



5. Funcionalidades

Estrutura do OpenMP: elementos essenciais



6. Sintaxe da Linguagem

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

Uma diretiva do compilador em C/C++ é chamada "pragma" (pragmatic information). Ela é chamada de uma diretiva de pré-processador (#).

Criação de *Thread*

omp parallel *é a construção fundamental que inicia uma execução paralela. Divide threads adicionais para o trabalho em paralelo. O processo original será marcado com master thread* (id= 0).

#pragma omp directive-name [clause, …] **C/C++**

#pragma omp parallel [cláusula[[,]cláusula] ...]

!$omp parallel [cláusula[[,] cláusula]...] **Fortran**

bloco-estruturado

!$omp end parallel

6. Sintaxe da Linguagem

omp for or omp do: usado para dividir iterações de loop entre as threads

sections*: marcação consecutiva, independente, blocos de código para diferentes *threads.

***single*: especificar um bloco de código executado somente por uma *thread*, limite é incluído no final.**

***master*: similar ao single, mas o bloco de código é executado somente pelo *master thread*.**

***IF control*: a paralelização de tarefas somente ocorrerá se a condição for verdadeira.**

***If* (expressão-condicional)**

***private* (lista) : variáveis ficam privadas, cada *thread* terá uma cópia local e será usada como temporária. O valor não é atualizado para uso fora da região paralela. O contador de *loop* de iteração é privado.**

***Firstprivate* (lista): permite que as variáveis privadas sejam inicializadas.**

***default* (shared | private | none): permite definir o *default* para as variáveis dentro de uma região paralela.**

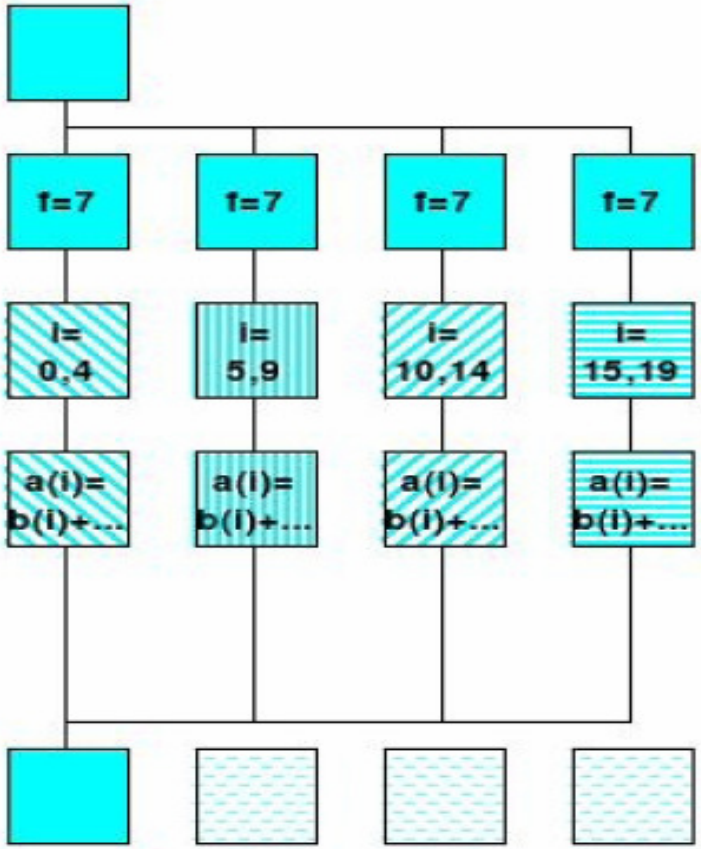
***shared* (lista): variáveis compartilhadas por todos os *threads* por *default*, exceto contador do *loop* de iteração.**

***copyin* (lista)** ***reduction* (operador: lista)**

6. Sintaxe da Linguagem - for

C / C++:

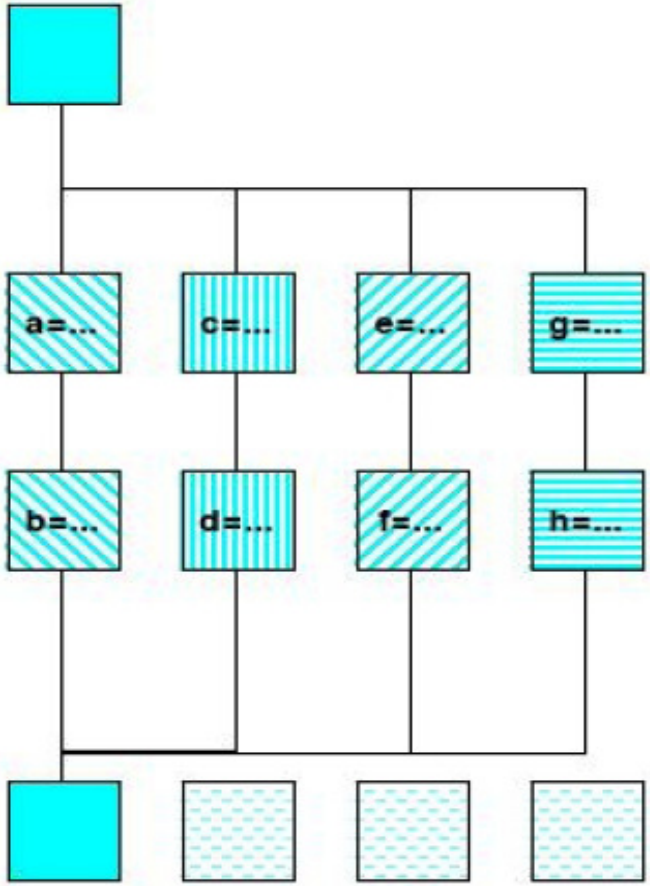
```
#pragma omp parallel private(f)
{
      f=7;

#pragma omp for
      for (i=0; i<20; i++)
          a[i] = b[i] + f * (i+1);



} /* omp end parallel */
```



6. Sintaxe da Linguagem Sections

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*/
```



6. Sintaxe da Linguagem - schedule

Cláusula para construções de *loops*:

schedule - como dividir as iterações do ciclo.

Schedule **(tipo[, tamanho])**

Usado para construções *do-loop* or *for-loop*. As iterações são alocadas para *threads*. O escalonador de *threads* é controlado por esta cláusula.

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.

6. Sintaxe da Linguagem - Sincronização

Cláusulas de sincronização:

***critical section*: bloco de código é executado por todas as *threads*, mas apenas uma *thread* de cada vez, não simultaneamente com outra. Frequentemente usado para proteger dados compartilhados.**

***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:* execução ocorre na ordem em que as 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 pela *master thread;

***flush* – força o sincronismo da região de memória da thread com a memória.**

6. Sintaxe da Linguagem
Inicialização Variáveis

***firstprivate*: dados são privados para cada *thread*, mas inicializados usando o valor da variável de mesmo nome na *master thread*.**

***lastprivate*: dados são privados para cada *thread*. O valor será copiado para uma variável global de mesmo nome, se a iteração corrente for a última. Pode ser ambos: *firstprivate* e *lastprivate*.**

***threadprivate*: dados são 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, 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* (a *single* thread) para os objetos correspondentes em outras *threads* no grupo.**

***reduction (operator|intrinsic:list)*: as variáveis tem uma cópia local na *thread*, mas os valores das cópias locais podem ser reduzidos em uma variável global compartilhada. .**

6. Sintaxe da Linguagem

Variáveis Ambiente

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

6. Sintaxe da Linguagem
Funções - subrotinas de biblioteca

Rotinas em tempo de execução a nível de usuário: 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.

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..

void *omp_init_lock(omp_lock_t): i*nicializa um *lock* associado à variável de *lock.*

void *omp_destroy_lock (omp_lock_t) void omp_set_lock (omp_lock_t):* espera *lock.*

void *omp_unset_lock (omp_lock_t)* libera o *lock.*

double *omp_get_wtime (void):* retorna os segundos decorridos (*elapsed time*).

double *omp_get_wtick (void):* os segundos decorridos entre chamadas sucessivas.

7. Exemplos - “Hello”

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

7. Exemplos - for

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
  for (i=0;i<N;i++)
     a[i]= 2*i;
  return 0;
}
```

7. Exemplos - Sections

```
#include <omp.h>
#define N 1000
main() {
int i, chunk;
float a[N], b[N], c[N];
// Some initializations
for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;
#pragma omp parallel shared(a,b,c) private(i) {
#pragma omp sections nowait {
#pragma omp section
for (i=0; i < N/2; i++) c[i] = a[i] + b[i];
#pragma omp section
for (i=N/2; i < N; i++) c[i] = a[i] + b[i];
#include <stdio.h>
#include <omp.h>
int main () {
./ompforsections
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;   }
```

```
$ OMP_NUM_THREADS=4

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
```

7. Exemplos - Funções

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]){
int nthreads, tid, procs, maxt, inpar,
dynamic, nested;
/* Start parallel region */
#pragma omp parallel private(nthreads, tid) {
/* Obtain thread number */
tid = omp_get_thread_num();
/* Only master thread does this */
if (tid == 0) {
printf("Thread %d getting info...\n", tid);
/* Get environment information */
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();
/* Print environment information */
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);
}      } /* Done */        }
```

7. Exemplos

```
#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

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*, em uma plataforma com N processadores.

No entanto, isso raramente acontece, devido às seguintes razões:

- uma grande porção do programa não podem 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.

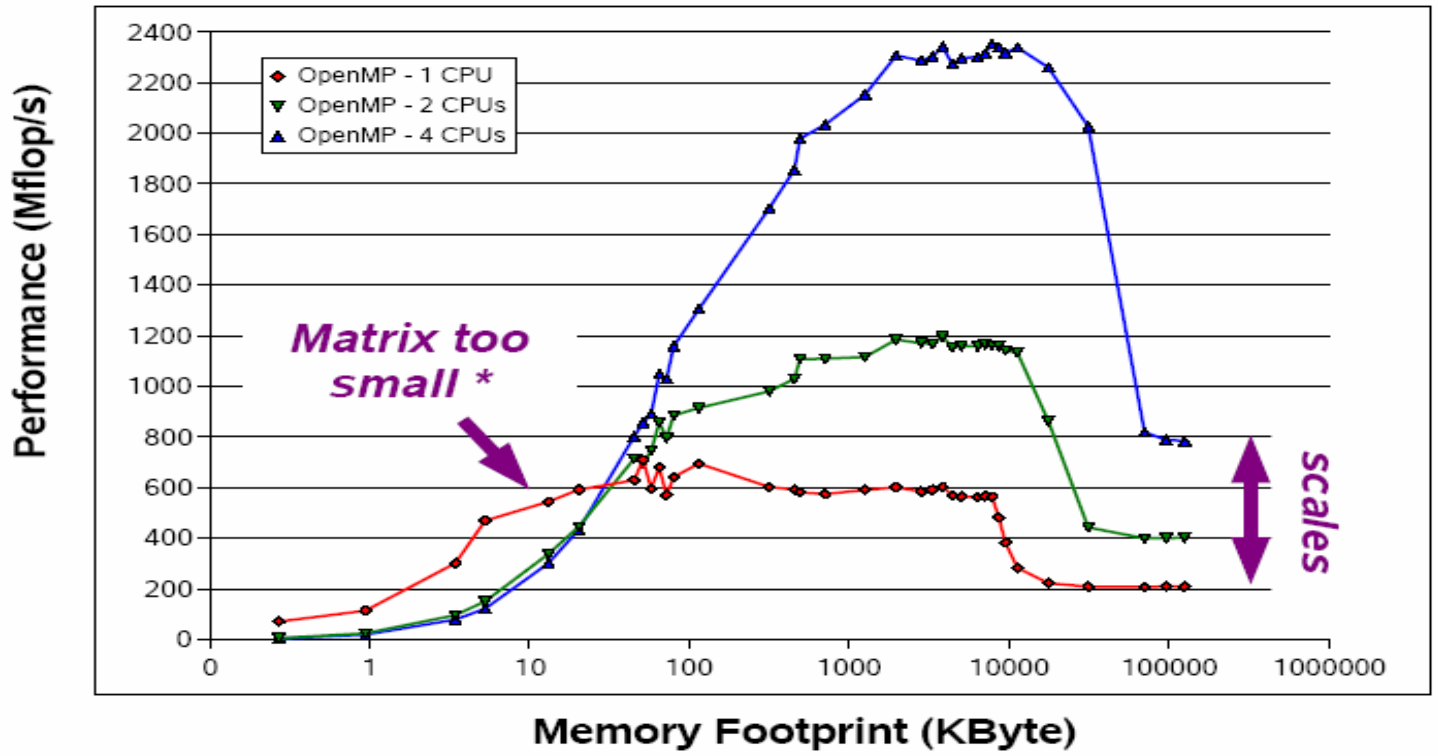
8. Conclusões - Vantagens

- Simples: não necessita negociar com *message passing* como MPI.
- Dados: layout e decomposição é manuseada automaticamente pelas diretivas.
- Paralelismo incremental: pode trabalhar em uma parte do programa em um momento, sem 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 introduzir bugs.
- Granulosidade: paralelismos coarse-grained e fine-grained são possíveis.

8. Conclusões - 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.

8. Conclusões - Sun



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

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

9. Referências

- http://en.wikipedia.org/wiki/OpenMP
- http://openmp.org/wp/
- https://computing.llnl.gov/tutorials/openMP/
- http://www.cOMPunity.org Community of OpenMP Users, Researchers, Tool Developers and Providers
- http://msdn.microsoft.com/pt-br/magazine/default.aspx MSDN Magazine article on OpenMP
- http://developers.sun.com/sunstudio/index.jsp Sun Studio multithreading tools for Solaris and Unix