5

Desenvolvendo Aplicações OpenMP

"*And there is more to come*"
Ruud van der Pas (Sun Microsystems)
An Overview of OpenMP 3.0- (IWOMP 2009)

Professores:

Nicolas Maillard[1]
(nicolas@inf.ufrgs.br)
Márcia Cristina Cera[2]
(marcia.cera@inf.ufrgs.br)

Resumo:

Este capítulo apresenta o OPENMP (Open Multi-Processing) [CHA 2008], o qual define uma API (Application Programming Interface) para programação paralela em memória compartilhada. Ele reune um conjunto de diretivas de compilação, rotinas e variáveis de ambiente que influenciam o comportamento da aplicação em tempo de execução. Em linhas gerais, o OPENMP possibilita extrair o paralelismo explícito de programas sequenciais e paralelos requerendo um baixo esforço de programação: a inclusão de algumas poucas diretivas de compilação possibilita a distribuição do trabalho entre fluxos de execução (threads). Em tempos em que máquinas com mais de um núcleo de processamento ou core estão cada vez mais difundidas, uma alternativa de programação com múltiplos fluxos de execução simples e portável como o OPENMP é de grande interesse. Nas próximas seções os leitores encontrarão as noções básicas para o desenvolvimento de programas OPENMP. A forma como acontece a distribuição dos cálculos e dados entre os fluxos de execução também será abordado, assim como a paralelização das iterações de laços e a sincronização dos fluxos de execução. Por fim, serão apresentadas características do OPENMP 3.0 e do paralelismo em nível de tarefas.

---

[1] Doutor em Sciences et Technologies de l'Information - Univ. Joseph Fourier (2001). Atualmente é professor adjunto no Instituto de Informática da UFRGS. Sua pesquisa envolve: programação e algoritmos paralelos, computação em Clusters e Grids, MPI, escalonamento de processos. Em seu currículo há um número significativo de publicações envolvendo MPI e aplicações em PAD.

[2] Doutoranda em Computação na UFRGS onde é bolsista CAPES, mestre na área de Tecnologia de Informação pela UFSM (2005) e graduada em Ciência da Computação na UFSM (2002). Fez seu doutorado sanduíche no Laboratoire d'Informatique de Grenoble, França (2008/2009). Atua em: PAD, escalonamento e balanceamento de carga, MPI e aplicações maleáveis. Seu doutorado envolve estudos sobre MPI e às aplicações maleáveis, sob co-orientação do Prof. Nicolas.

5.1. Noções Básicas

O OPENMP possibilita a distribuição das instruções de um programa entre vários fluxos de execução (ou *threads*), as quais compartilham parte da sua memória [CHA 2001]. Como qualquer interface de programação paralela para memória compartilhada, OPENMP define:

1 a forma como os cálculos serão distribuídos entre os fluxos de execução;

2 a forma como os dados estão replicados ou compartilhados entre os fluxos de execução;

3 as sincronizações entre os fluxos de execução;

O desenvolvimento de programas OPENMP dá-se através de `pragmas` acrescentados à programas C, C++ e Fortran e de rotinas definidas pela biblioteca OPENMP. As rotinas são declaradas no arquivo *header*, cujo nome padronizado é `omp.h` e que deve ser fornecido por qualquer compilador OPENMP.

Algumas das principais rotinas providas pela biblioteca OPENMP são: `omp_get_num_threads()` e `omp_set_num_threads(int n)` que servem respectivamente para identificar (durante a execução) quantos fluxos estão executando o programa e para atribuir o valor `n` a este número. Existe também a rotina `omp_get_thread_num()`, que retorna o identificador único dos fluxos de execução que é um número entre 0 e `omp_get_num_threads()`-1. Cuidado para não confundir `omp_get_num_threads()` e `omp_get_thread_num()`.

Quando se usa `pragmas` com um compilador não OPENMP, os mesmos serão automaticamente descartados. No entanto, não será o caso das chamadas à biblioteca OPENMP ou da inclusão do *header* `omp.h`. A norma OPENMP impõe a um compilador a definição da macro `_OPENMP`, a qual pode ser usada na precompilação para testar o uso efetivo de um compilador OPENMP. Tipicamente, poderá se encontrar a seqüência:

```
#ifdef _OPENMP
   #include <omp.h>
#endif
...
int nb_th=0;
#ifdef _OPENMP
   nb_th = omp_get_num_threads();
#endif
```

Assim, quando a macro `_OPENMP` não estiver definida, o compilador ignora as chamadas a biblioteca OPENMP e o programa mantém seu comportamento original. Nos exemplos apresentados neste capítulo, as chamadas à biblioteca OPENMP não serão protegidas para não sobrecarregar o código. No entanto, é uma boa ideia sistematizar o uso de `_OPENMP` em seus programas OPENMP.

Um primeiro programa OPENMP A Listagem 5.1 mostra como se programa o clássico "Ola Mundo!" com OPENMP.

Listing 5.1: "Ola Mundo!" em paralelo com OPENMP.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  int main (int argc, char *argv[]) {
6  int nprocs = -1;
7  #pragma omp parallel
8    {
9       nprocs = omp_get_num_threads();
10      printf("Oi␣mundo!␣(Executando␣com␣%d␣fluxos)\n",
11                                                  nprocs);
12   }  /* Sincronizacao */
13   printf("Isso␣eh␣impresso␣por␣apenas␣uma␣thread.\n");
14 }
```

Na Listagem acima observa-se que a linha 7 especifica o bloco (definido pelas chaves '{' e '}') que deve ser executado em paralelo em cada *thread*. Neste exemplo, a linha 9 contendo uma chamada à biblioteca OPENMP não foi protegida pelo teste de `_OPENMP`. No fim do bloco (linha 12), haverá uma sincronização entre os fluxos. A partir da linha seguinte, volta-se a executar o programa com uma única *thread*. Por exemplo, ao executar este programa após ter atribuído 3 à variável de ambiente `OMP_NUM_THREADS`, obtém-se:

```
Oi mundo! (Executando com 3 fluxos)
Oi mundo! (Executando com 3 fluxos)
Oi mundo! (Executando com 3 fluxos)
Isso eh impresso por apenas uma thread.
```

5.2. Distribuição dos cálculos e dos dados

A forma mais simples de distribuir cálculos com OPENMP é usando a diretiva `parallel`: ela especifica o disparo de *threads* concorrentes (sendo o número de *threads* determinado na execução), e que o bloco entre `{...}` subsequente ao `pragma` deverá ser executado em paralelo. Esse paralelismo é do tipo *Fork/Join*, uma vez que se disparam *threads* (*Fork*) que se sincronizam ao executar a última instrução do bloco (*Join*). Entretanto, ela pode ser usada para paralelizar uma aplicação seguindo uma abordagem *"Single Program, Multiple Data"*: um único programa, executado n vezes por n processos distintos, processando dados distintos.

5.2.1. Especificação dos acessos concorrentes aos dados

Como qualquer interface para programação concorrente, OPENMP deve prover meios para acessar corretamente os dados na memória, e resolver os conflitos de eventuais escritas concorrentes. A cláusula `shared(x, y, z, ...)` especifica que as variáveis `x, y, z` (e as demais listadas) serão compartilhadas entre os fluxos: todos irão acessar concorrentemente um mesmo espaço na memória comum. Essa cláusula, assim como a próxima (`private`), atua em uma variável como um todo: por exemplo, não é possível declarar apenas um campo de uma `struct` como compartilhado. Isso não significa que haverá obrigatoriamente conflito no acesso a esses campos: se um fluxo acessa para escrita o campo `x` de uma *struct* e enquanto isso um outro fluxo acessa o campo `y` da mesma *struct*, mesmo se ela for *shared*, por definição não haverá problema de conflito. O mesmo vale se um vetor for compartilhado, porém com os fluxo acessando índices diferentes (ou seja: endereços distintos na memória) do vetor compartilhado.

A cláusula `private(x, y, z, ...)` especifica que as variáveis `x, y, z` (e as demais listadas) serão replicadas entre as *threads* de execução, de tal forma que cada um possa trabalhar em sua versão privada, em sua memória não compartilhada. Ao ser disparado, um fluxo declara novas instâncias de suas variáveis privadas, em seu espaço de endereçamento privado, e as usará durante sua execução. Ao encerrar uma *thread*, o Sistema Operacional irá reaproveitar seu espaço privado, o que significa que o conteúdo das variáveis privadas será perdido. Toda variável declarada localmente a um bloco `parallel` (ou seja, localmente ao escopo sintático definido pelo uso das chaves `{` e `}`) será alocada na pilha do fluxo que o executa, ou seja será automaticamente privada. Já toda variável global, ou mesmo toda variável declarada no escopo sintático mais abrangente do que o escopo que define os fluxos (por exemplo, o escopo do `main`), será por *default* compartilhada.

Voltando-se ao exemplo "Ola Mundo!" apresentado anteriormente, vamos considerar o caso da variável `nbprocs`: cada fluxo determina seu valor e o escreve no mesmo endereço na memória (o associado pelo compilador à variável). Ou seja, há um problema de escrita concorrente. Neste caso específico, como cada fluxo acaba escrevendo o mesmo valor, pode-se confiar que o resultado não será prejudicado pelo conflito se a atribuição de um valor `int` não puder ser interrompida por uma troca de contexto entre fluxos. De fato, o programa 5.1 executa corretamente, entretanto pode-se dizer que no mínimo ele não está bem escrito.

Indo mais além, o que aconteceria caso que se quisesse personalizar a mensagem em função do número que identifica cada fluxo? Considere a Listagem 5.2.

Listing 5.2: "Ola Mundo!" personalizado com OPENMP.

```
1  int main (int argc, char *argv[]) {
2  int t_id = -1;
3  #pragma omp parallel private(t_id)
4    {
5  #ifdef _OPENMP
6       t_id = omp_get_thread_num();
7  #endif
8       printf("Oi␣mundo␣da␣thread␣%d\n", t_id);
9    }  /* Sincronizacao */
10   printf("Isso␣eh␣impresso␣por␣apenas␣uma␣thread.\n");
11 }
```

A variável `t_id` conterá o valor do identificador de cada fluxo. Cada fluxo, por definição, lhe atribuirá um valor distinto. Por isso, a variável deve ser `private` a cada fluxo, e isso é especificado pelo uso da diretiva na linha 7.

Cada variável pode ser privada ou compartilhada, mas não ambas. Um efeito colateral do uso da cláusula `private` é que uma variável declarada como privada não enxergará uma eventual inicialização feita antes do laço. Assim, no código abaixo:

```
1  int n = -1 ;
2  #pragma omp parallel private(n)
3  {
4     int p;
5     p = n;
6  }
```

a variável `p` é privada, já que é declarada localmente no bloco `parallel`. No entanto, sua inicialização na linha 5 será incorreta: o valor recuperado em `p` não será o valor -1, uma vez que a variável `n` foi declarada privada no bloco paralelo: ao ser executado por um fluxo, ele terá uma versão privada de `n`, que não herdará o valor da versão da memória compartilhada.

Existe uma subtileza ao acessar variáveis globais a partir de blocos paralelos no OPENMP: acessos feitos diretamente por instruções dentro do bloco são condicionados pelo uso de `private` e `shared`. Porém, nada impede que um bloco chame um procedimento, o qual pode fazer acesso a variáveis globais. Neste caso, OPENMP impõe que sejam acessados sistematicamente as versões dessas variáveis na memória compartilhada.

Assim, no código da Listagem 5.3, o resultado típico de uma execução com 2 fluxos será:

```
Fora do bloco, n = 10
Dentro do bloco, p = -1208075904
Dentro do bloco, n = -1208075904
n vale: 10
Dentro do bloco, p = -1208147540
```

Listing 5.3: Acesso a variáveis através de chamada a procedimento.

```
1  int n=10;
2
3  void imprime() {
4     printf("n␣vale:␣%d\n", n);
5  }
6
7  int main (int argc, char *argv[]) {
8    printf("Fora␣do␣bloco,␣&n␣=␣%d\n", &n);
9  #pragma omp parallel private(n)
10   {
11      int p = n;
12      printf("Dentro␣do␣bloco,␣p␣=␣%d\n", p);
13      printf("Dentro␣do␣bloco,␣&n␣=␣%d\n", &n);
14      imprime();
15   }
16 }
```

```
Dentro do bloco, n = -1208147540
n vale: 10
```

Conforme foi explicado, localmente no bloco, `p` e `n` não foram inicializados (apesar da versão global de n ter sido). A chamada a `imprime`, apesar de ser efetuada em cada fluxo de dentro do bloco paralelo, acessa a versão na memória global de `n`, por isso cada fluxo imprime "`n vale: 10`".

Para especificar que uma variável global deve ser local a cada fluxo, existe a diretiva `threadprivate`. A simples alteração no código anterior, que declara n como sendo `threadprivate` (ver o código 5.4), produz o seguinte resultado:

```
Fora do bloco, n = 10
Dentro do bloco, p = 10
Dentro do bloco, n = 10
n vale: 10
Dentro do bloco, p = 10
Dentro do bloco, n = 10
n vale: 10
```

Dessa vez, acessou-se a versão esperada de `n`, seja diretamente dentro do bloco paralelo, seja através da chamada a um procedimento.

5.2.2. Distribuição dos cálculos

O `pragma parallel`, já apresentado, possibilita delimitar um bloco a ser executado múltiplas vezes em paralelo. A diretiva `section` possibilita definir, caso a caso,

Listing 5.4: O código da Listagem 5.3 usando *threadprivate*.

```
#pragma omp threadprivate(n)
int n=10;

void imprime() {
   printf("n vale: %d\n", n);
}

int main (int argc, char *argv[]) {
  printf("Fora do bloco, n = %d\n", n);
#pragma omp parallel
  {
     int p = n;
     printf("Dentro do bloco, p = %d\n", p);
     printf("Dentro do bloco, n = %d\n", n);
     imprime();
  }
}
```

blocos de instruções a serem executados por fluxos de execução diferentes. Seu uso é exemplificado abaixo:

```
void f1();
void f2();
void f3();
#pragma omp sections
{
#pragma omp section
   { f1(); }
#pragma omp section
   { f2(); }
#pragma omp section
   { f3(); }
}
```

A diretiva `sections` inicia a especificação de uma série de blocos de instruções, cada qual sendo executado por um fluxo. Cada série é separada da seguinte pelo uso da diretiva `section` (sem 's' no final). No código acima, cada uma das funções `f1`, `f2` e `f3` serão executadas concorrentemente por fluxos diferentes. Obviamente, essas funções poderiam usar parâmetros (tomando-se os devidos cuidados a respeito das manipulações na memória: devendo-se usar, eventualmente, a cláusula `private`), ou ainda serem substituídas por uma seqüência de instruções separadas por ';'.

Caso sejam disparados um número menor de *threads* do que o de `section` s, cada *thread* executará mais de uma `section`. Mesmo com o número suficiente de *th-*

reads, mais de uma seção pode ser executada por *thread* (o que significa que algumas *threads* podem não executar nada) em função do escalonamento das mesmas. Dependendo do caso, pode-se levar mais tempo para criar uma *thread* e efetuar uma troca de contexto do que simplesmente executar as seções numa única *thread*.

Usar `sections` possibilita que cada fluxo execute sua parte de um procedimento maior. No entanto, a especificação das tarefas de cada fluxo deve estar explicitamente embutida no código: no programa acima, mesmo se houvesse mais de três fluxos, não existe um quarto procedimento `f4` a ser executado. A forma como se expressa o paralelismo da aplicação, com seções, não é escalável, nem muito genérico. O uso de paralelismo de laços permite melhorar isso.

5.3. Paralelismo de laços

A forma mais simples de se paralelizar um programa com OPENMP é identificar o paralelismo de laços: anota-se uma instrução de laço para indicar quando as iterações podem ser calculadas em paralelo. Paraleliza-se um laço com OPENMP através da diretiva:

```
#pragma omp for [cláusula [cláusula ...]]
for (i=0 ; i<n ; i++) { ...  }
```

A presença de um `for` logo após a diretiva é imperativa. As cláusulas, opcionais, serão discutidas na Seção 5.3.1. (página 155). Existe restrições quanto ao tipo de laços que podem ser paralelizados com essa diretiva. Quando o programa for executado, as iterações serão distribuídas entre os n fluxos que serão usados. OPENMP impõe que se possa determinar essa distribuição durante a compilação. Por isso, é necessário que a instrução `for` seja escrita da seguinte forma genérica:
`for (i = inicio; i op_rel fim; op(i))`, onde:

- `i`, `inicio` e `fim` são quaisquer variáveis inteiras cujo valor seja conhecido na compilação;
- `op_rel` é um dos operadores relacionais <, >, <=, ou >=;
- `op(i)` é um operador de iteração, sendo que o aumento ou decremento deve ser constante de uma iteração para a outra. Assim, `op(i)` pode ser igual a `i++`, `i--`, `i+=K` ou `i-=K`, `i=i+K` ou `i=i-K`, onde K é uma constante conhecida durante a compilação.

A constar, também não se pode forçar a saída do laço através de um `goto` ou de um `break`.

Pela mesma razão pode-se explicar porque o paralelismo de laços está limitado ao uso do `for`, e não a laços do tipo `while` ou `do...until`: laços condicionados por um

teste lógico não permitem a determinação, durante a compilação, de quantas iterações serão executadas (a não ser em casos triviais onde a condição de parada pode ser calculada na compilação). Nota-se que isso não impossibilita a paralelização, com o OPENMP, de tais laços; a seção 5.4. dará um exemplo de como isso pode ser alcançado; no entanto, não existe uma diretiva única e simples no OPENMP que poderia paralelizar essas construções.

A diretiva `#pragma omp for` pode ser combinada com a `#pragma omp parallel` para especificar numa única linha que deve-se criar fluxos concorrentes e distribuir entre eles as iterações do laço que começa logo a seguir.

A figura 5.1 ilustra o comportamento de um laço sendo paralelizado da forma seguinte:

```
1 printf(``Fim da inicializacao sequencial.\n'');
2 #pragma omp parallel for
3    for (i=0; i<N; i++) {
4            c[i] = a[i] + b[i];
5    }
6 printf(``Fim da parte paralela.\n'');
```

Ao iniciar o laço `for`, o fluxo mestre associado ao programa principal dispara novos fluxos, de acordo com o número indicado pelo usuário (por exemplo pela variável de ambiente `OMP_NUM_THREADS`). Cada um dos fluxos executa parte das iterações do laço. Ao chegar à chave `}` final, os fluxos se sincronizam, e o único fluxo mestre prossegue com sua execução com a instrução `printf("Fim da parte paralela.");`.

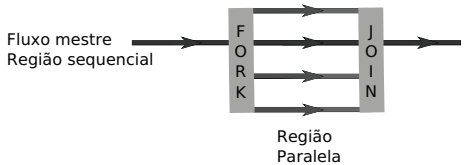

Figura 5.1: *Threads* executando um `#pragma omp parallel for`: o fluxo mestre que cria *threads* (*Fork*) para computar as interações e há uma sincronização (*Join*) ao final da região paralela.

A execução paralela é ilustrada simplesmente quando se manda um dos fluxos imprimir as iterações que está executando (ver o código na Listagem 5.5).

No código da Listagem 5.5, usa-se a rotina `omp_get_thread_num()` para recuperar o identificador de cada um dos fluxos disparados para executar as iterações. Apenas o fluxo cujo identificador seja zero irá imprimir na tela os valo-res de `i` nas iterações que ele está executando. A saída deste programa, quando executado com 4 fluxos, é:

```
Fim da inicializacao sequencial.
```

Listing 5.5: laço paralelo com OPENMP.

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define N       13
5
6  int main (int argc, char *argv[]) {
7    int td, i;
8    double a[N], b[N], c[N];
9
10   /* Inicializacoes */
11   for (i=0; i < N; i++)
12     a[i] = b[i] = i * 1.0;
13   printf("Fim da inicializacao sequencial.\n");
14   /* Laco paralelizado */
15   #pragma omp parallel for schedule(runtime) private(td)
16   for (i=0; i<N; i++) {
17     td = omp_get_thread_num();
18     c[i] = a[i] + b[i];
19     if (t_id == 0)
20       printf("Fluxo %d calculando i = %d.\n", td, i);
21   }
22   printf("Fim da parte paralela.\n");
23 }
```

```
Fluxo 0 calculando i = 0.
Fluxo 0 calculando i = 1.
Fluxo 0 calculando i = 2.
Fim da parte paralela.
```

Os três fluxos de execução cujos identificadores são $1, 2$ e 3 executaram as outras iterações. A Seção 5.3.2. apresentará como se controla quais iterações são distribuídas entre os fluxos.

5.3.1. Cláusula `private` do `omp parallel for`

Assim como no caso da diretiva `parallel`, o `omp parallel for` pode incluir uma cláusula de restrição quanto ao acesso às variáveis manipuladas no laço. Pode-se usar `shared(x, y, z, ...)` ou `private(x, y, z, ...)` para que as variáveis `x`, `y` e `z` sejam compartilhadas ou privadas.

Um efeito colateral do uso da claúsula `private` é que uma variável declarada como privada não enxergará uma eventual inicialização feita antes do laço. Assim, no código abaixo:

```
1 int fator = 0.5;
2
3 #pragma omp parallel for private( fator )
4    for (i=0; i<N; i++) {
5            c[i] = fator * a[i] ;
6    }
```

o valor de `fator` usado na linha 4 será 0 e não 0.5, como provavelmente se quereria.

A especificação do modo de acesso às variáveis é crítica para a integridade e o desempenho do programa paralelo. Podem haver casos onde isso não é totalmente intuitivo. Por exemplo, a variável `i` do laço acima precisa ser compartilhada ou privada? A primeira vista, pode-se pensar que deva ser compartilhada, uma vez que se quer que todas as *threads* executem os laços de forma compartilhada. No entanto, as *threads* deverão escrever em `i`, pelo menos para poder incrementá-la. Além dessa escrita, todos deverão trabalhar com valores diferentes de `i`. Por isso, a variável do laço precisa ser privada. De modo geral, a necessidade de se acessar uma variável em escrita é uma indicação séria de que a mesma deverá ser declarada como privada. Se não é, o controle dos acessos em escrita deverá ser garantido pelo uso de mecanismos de sincronização, descritos na Seção 5.4.

O *default* para todas as variáveis cujo modo de acesso não foi explicitamente indicado pelo uso de `private` ou `shared` é que elas sejam compartilhadas. Pode-se alterar esse funcionamento pelo uso da diretiva `default(none)`, a qual fará com que o compilador retorne uma mensagem de erro se o programador não especificou o modo de acesso a uma das variáveis dentro do escopo de uma seção paralelizada com OpenMP.

Em um compilador C OpenMP, uma variável de laço (por exemplo o `i` dos programas acima) é sempre considerado como `private`. Faz sentido, uma vez que deve-se sempre incrementar a variável. No entanto, é importante entender que essa regra vale unicamente para o laço imediatamente após a diretiva `parallel for`, e não para os eventuais laços que possam ser aninhados. Seja o seguinte exemplo:

```
1 #pragma omp parallel for
2    for (i=0; i<N; i++) {
3       for (j=0; j<N; j++) {
4           c[i] = a[i]+b[j] ;
5       }
6    }
```

Por *default*, `i` é privada e todas as outras variáveis são compartilhadas. Isso inclui `j` (assim como `N`, `a`, `b` e `c`). Entretanto, o fato de se ter um laço iterando sobre `j` implica numa escrita em `j`, a qual é visível na instrução `j++`.

Este é um erro típico do principiante com OpenMP: neste programa, todas as *threads* vão tentar escrever no mesmo endereço na memória (onde o compilador terá decidido armazenar `j`) e tem-se uma condição de corrida. Consequentemente, o programa não está correto: seu resultado vai depender da ordem de execução das *threads*, do número

de *threads*, de decisões próprias ao compilador, etc... O que se quer é que cada fluxo de execução trabalhe com sua cópia própria de `j`:

```
1 #pragma omp parallel for private(j)
2    for (i=0; i<N; i++) {
3       for (j=0; j<N; j++) {
4           c[i] = a[i]+b[j] ;
5       }
6    }
```

Uma variável privada não enxerga uma atribuicão que lhe foi feita antes do bloco executado em paralelo. Caso seja necessário, pode-se usar a cláusula `firstprivate`: assim, em cada *thread*, a cópia privada herdará o valor inicial, tal como definido no ponto de entrada da seção paralela. Simetricamente, uma variável pode ser definida como sendo `lastprivate`, para que seu último valor (tal qual seria o valor na última iteração do laço, se o mesmo fosse executado seqüencialmente) seja atribuído ao sair da seção paralela. Ao contrário do que acontece com as cláusulas `private` e `shared`, uma variável pode ambas as cláusulas `firstprivate` e `lastprivate` associadas a ela.

5.3.2. Cláusula `schedule` do `omp parallel for`

A cláusula `schedule` especifica a forma como as iterações de um laço serão distribuídas entre as *threads*. A representação genérica desta cláusula é: `schedule(tipo[ ,`*chunk*`])`. O *chunk* é um conjunto de t iterações contíguas que serão atribuídas a uma *thread*. O tipo do escalonamento pode ser:

- `static`. Se o programador especificou um valor para *chunk*, este valor será usado para calcular o tamanho dos *chunks* e eles serão atribuídos as *threads* seguindo um *Round-Robin*. Se não houver um valor especificado, o *default* é 1. Portanto, é durante a compilação que se determina quais *chunks* serão executados por quais *threads*.

- `dynamic`. Se *chunk* foi especificado, será usado para determinar o tamanho dos *chunks*, caso contrário será usado o tamanho 1. Neste modo, cada *chunk* é atribuído a uma *thread* durante a execução, sob-demanda quando uma *thread* terminar o cálculo de um *chunk* anterior.

  A vantagem é que este modo atende aos casos em que há iterações com tempos de duração diferentes e consegue manter cada *thread* sempre ocupada, até não ter mais nenhum *chunk* a ser calculado. O inconveniente é que este mecanismo impõe um sobrecusto, durante a execução, para a manutenção dos *chunks* a serem calculados.

- `guided`. É uma variação do escalonamento dinâmico, porém onde o tamanho do *chunk* evolui a medida que as iterações vão sendo calculadas. A partir de um valor inicial (que depende da implementação do compilador OPENMP), cada *chunk* sucessivo tem seu tamanho diminuído de um fator proporcional ao tamanho do *chunk* anterior.

iteração	0	1	2	3	4	5	6	7	8	9	10	11	12
static	0	1	2	0	1	2	0	1	2	0	1	2	0
static,2	0	0	1	1	2	2	0	0	1	1	2	2	0
dynamic	0	0	0	0	0	0	0	0	0	0	0	0	0
dynamic,2	2	2	2	2	2	2	2	2	2	2	2	2	2

Tabela 5.1: Os identificadores das *threads* para cada iteração executada, conforme o escalonamento empregado.

O escalonamento `guided` das iterações é um meio-termo entre o estático e o dinâmico: ele tenta, assim como no caso estático, alocar *chunks* grandes as *threads*, para diminuir o sobrecusto de gerenciamento; porém, ele também os atribui de forma dinâmica durante a execução. O fato de deixar *chunks* menores para o fim da computação ajuda a balancear a carga.

Assim, o uso típico de `schedule` será algo como: `schedule(static, 10)`, `schedule(dynamic)`, etc... Nota-se que existe uma outra opção para a clausula `schedule`: `schedule(runtime)`, a qual deixa a decisão de escalona-mento das iterações para o momento da execução do programa. Neste caso, dever-se-á atribuir à variável de ambiente `OMP_SCHEDULE` um valor que seja compatível com as opções acima. Por exemplo:

```
setenv OMP_SCHEDULE ``static,10''
```

junto com `schedule(runtime)`, produzirá exatamente o mesmo resultado que `schedule(static, 10)`.

A tabela 5.1 mostra, a título de exemplo, quais *threads* irão executar quais iterações para cada tipo de escalonamento, com $N = 13$ iterações e $p = 3$ *threads*.

Nota-se que, no caso `dynamic`, por definição, determinar qual *thread* irá executar qual iteração dependerá das condições durante a execução. Por acaso, nas execuções que levaram aos dados dessa tabela, apenas uma *thread*, em ambos os casos dinâmicos, acabou executando todas as iterações; os demais não tiveram o tempo suficiente para serem escalonados e calcular parte das iterações.

5.3.3. Cláusula `reduce` do `omp parallel for`

As cláusulas `firstprivate` e `lastprivate` possibilitam a transmissão inicial e final de valores entre variáveis compartilhadas e privadas as *threads*. Um esquema de computação frequentemente encontrado é o cálculo de um valor através da acumulação de valores parciais; um caso típico é quando um laço efetua `r += f(...)`, onde `f(...)` se encarrega de calcular um valor parcial, e a operação de acumulação é uma soma aritmética.

Para paralelizar tais laços, pode-se pensar em usar `firstprivate(r)`, `lastprivate(r)` e inicializar `r` com 0:

```
1 double r = 0;
2
3 #pragma omp parallel for firstprivate(r) lastprivate(r)
4    for (i=0; i<N; i++) {
5       r += f() ;
6    }
7    printf("Resultado␣final␣=␣%lf\n", r);
```

No entanto, o valor final será apenas o último valor calculado para `r`, de acordo com a ordem seqüencial (ou seja, o valor calculado pela *thread* que, de acordo com o escalonamento das iterações, executará a iteração $i = N - 1$). Serão perdidos todos os valores locais de `r` calculados pelos outros fluxos de execução, que nunca foram somados a este `r` especial. Logo, não se calcula o que se espera.

A solução é usar uma operação de *redução* que, além de cuidar da transmissão de valores iniciais e finais, aplicará a operação necessária para acumular os resultados parciais. O OpenMP prevê a cláusula `reduction(op:var)` para efetuar tal operação. Nela, `op` é um dos operadores `+`, `*`, `&` ou `|`, e `var` é a (ou a lista de) variável a ser acumulada. Assim, pode-se escrever o programa anterior da seguinte forma:

```
1 double r = 0;
2 #pragma omp parallel for reduction(+:r)
3    for (i=0; i<N; i++) {
4       r += f() ;
5    }
6    printf("Resultado␣final␣=␣%lf\n", res);
```

Nota-se que a operação de redução deve ser associativa, pois a ordem dos cálculos não deve importar nessa operação.

5.4. Sincronização das *Threads*

As diretivas `parallel`, `sections` e `for` servem para especificar atividades a serem executadas em paralelo. No entanto, é importante poder especificar que um trecho de instruções só deve ser executado por uma única *thread*, ou talvez por uma *thread* de cada vez, ou seja, em exclusão mútua. Este será o caso quando um programa precisa manipular dados que só podem ser acessado sequencialmente; pense, por exemplo, na escrita em um arquivo.

A diretiva `single` delimita um bloco a ser executado por uma, e apenas uma, *thread* independentemente de qual das *threads* o executará. Serve, tipicamente, para fazer entradas/saídas. Pode-se usar também a diretiva `master`, que fará com que apenas a

thread mestre execute o bloco especificado. Nos exemplos acima, usou-se testes relativos ao identificador da *thread*: `if(t_id == 0)...;`. Em lugar disso, pode-se usar simplesmente `#pragma omp single` (ou então `#pragma omp master`) e, em seguida definir o bloco de instruções a serem executadas entre `{` e `}`.

Outra situação comum é existir um trecho de código que deve ser executado por todas as *threads*, porém esta execução não pode ser concorrente: tal trecho é chamado de seção crítica. Neste caso, o OPENMP provê a diretiva `omp critical`, com sua ação restrita ao seu bloco de instruções definido entre `{` e `}`. Veja a Listagem 5.6.

Listing 5.6: Paralelização de um laço while com OPENMP (1).

```
1  int main (int argc, char *argv[]) {
2  int tid, i;
3  #pragma omp parallel private(i, tid)
4    {
5       tid = omp_get_thread_num();
6       i = next_indice();
7       while (i != -1) {
8          calculo(tid,i);
9          i = next_indice();
10      }
11   }  /* Fim do bloco paralelo */
12 }
```

Segundo a Listagem 5.6, disparam-se *threads* (na linha 29), cada qual executando um laço `while`, até que `i` seja igual a `-1`. Para cada valor de `i`, cada *thread* executa um `calculo` em função de seu identificador `tid` e de `i`. Este tipo de programa paralelo é bastante comum em algoritmos "*task-farms*", ou "*master/worker*": mantem-se uma lista de tarefas (aqui, "cálculo"s) identificadas por um índice (`i`), que podem ser executadas em paralelo.

De uma iteração para a outra, `i` é atualizado através de `next_indice()`. Esse procedimento deve ser tal que ele retorne valores de `i` distintos para cada *thread*, caso contrário eles irão executar as mesmas tarefas de cálculo. Para isso, basta manter uma variável global `indice` que será incrementada a cada chamada de `next_indice()`, até que um valor máximo seja ultrapassado, para então atribuir o valor `-1`. Ou seja:

```
1  int indice;
2  int next_indice() {
3    int res;
4    if (indice == N)
5       res = -1;
6    else
7       res = ++indice;
8    return(res);
9  }
```

No entanto, haverá uma condição de corrida caso todas as *threads* possam atualizar `indice` concorrentemente: um pode começar a executar `next_indice()` com um valor de `indice`, perder a CPU para um outro, que executará também `next_indice()` e deixará o primeiro com um valor incoerente de `indice`. A solução é executar o trecho de atualização de `indice` numa seção crítica, conforme mostra o código da Listagem 5.7.

Listing 5.7: Paralelização de um laço while com OPENMP (2).

```
1  int next_indice() {
2    int res;
3  #pragma omp critical
4  {
5     if (indice == N)
6             res = -1;
7     else
8       res = ++indice;
9  }
10    return(res);
11 }
```

Este código funciona corretamente e cada *thread* executará o laço `while` com valores de `i` diferentes.

Quando uma seção crítica limita-se a fazer acessos em escrita na memória, pode-se usar o `omp atomic`, que se aplica apenas à linha seguinte a diretiva. Espera-se que essa linha efetue apenas um acesso na memória.

Enfim, quando fala-se em sincronização de *threads*, deve-se mencionar a cláusula `omp nowait`, que, quando acrescentada a qualquer diretiva de distribuição de trabalho entre *threads*, tira a sincronização implícita na chave `}` que encerra o trecho de código a ser executado em cada *thread*.

5.5. OPENMP 3.0 e o Paralelismo de Tarefas

A norma OPENMP 3.0 foi lançada em 2008 e começa a estar disponível nos compiladores recentes (na data da escrita deste texto, o compilador `icc` da Intel já incluia o OPENMP 3.0).

Uma das principais novidades é o suporte a um outro modelo de programação paralela além do paralelismo de laços: o paralelismo de tarefas. Ele é bem apropriado a algoritmos recursivos, onde uma (ou várias) tarefa(s), ao executar, dispara recursivamente outras tarefas. Observa-se que esse paradigma de programação paralela é ao paralelismo de laços o que programação recursiva (em particular usada na programação funcional) é à programação iterativa. Muitos algoritmos eficientes usam recursividade, e é importante que o OpenMP possa dar suporte aos mesmos. Claramente, isso não significa que um

programa que execute um laço naturalmente paralelo deva ser reprogramado de forma recursiva. Mas nos casos em que a recursividade fornece um algoritmo simples, é bom que a versão paralela seja semelhante.

Como exemplo, voltaremos a tratar do cálculo dos números de Fibonacci, conforme foi apresentado na seção 4.4.2. Ao olhar o código sequencial, identifica-se diretamente que as duas chamadas recursivas podem ser executadas em paralelo — exatamente como foi feito ao se programar com MPI-2. Após as duas chamadas recursivas, há a necessidade de uma sincronização para garantir que os valores de x e y foram calculados, antes de retornar a soma deles.

Isso se programa da seguinte forma com OPENMP 3.0:

```
int fib(int n) {
   if (n < 2) return n;
   else {
      int x, y;
#pragma omp task shared(x)
{     x = fib(n-1); }
#pragma omp task shared(y)
{     y = fib(n-2); }
#pragma omp taskwait
      return(x + y);
   }
}
```

Graças à diretiva `task`, foram especificados quais os blocos de instruções C que podem ser executados de forma assíncrona por uma das *threads* do programa. (No caso, cada tarefas é constituída de apenas uma linha de código e as chaves são opcionais.) A diretiva `taskwait` possibilita sincronizar uma tarefa com suas tarefas filhas, antes de prosseguir.

Nota-se que essas diretivas são utilizadas de forma integrada com o resto das diretivas OPENMP: espera-se uma diretiva `parallel` para disparar as *threads*, especificar quais variáveis são compartilhadas ou não, etc. Por exemplo, veja a seguir a primeira chamada ao procedimento que calcula o número de Fibonacci para um determinado valor de n:

```
#pragma omp parallel
{
  #pragma omp single nowait
  { result = fib(n); }
}
```

Apenas uma *thread* deve chamar $fib(n)$ desencadeando a execução das tarefas conforme mostrado anteriormente.

5.6. Conclusão

Este capítulo apresentou o OPENMP, o qual é capaz de extrair o paralelismo de programas sequenciais e paralelos de forma explícita (porém não automática, já que o programador tem total controle sobre a paralelização). Uma das grandes vantagens do OPENMP é que através da inclusão de algumas poucas diretivas de compilação em um código fonte preexistente, este torna-se um programa com múltiplos fluxos de execução. Porém, como foi visto neste minicurso, existem pequenos detalhes e cuidados a serem tomados para garantir a eficiência na execução de tais programas, bem como sua corretude.

Inicialmente foram apresentadas as noções básicas para o desenvolvimento de programas OPENMP, tais como rotinas, macros e a definição básica de um bloco paralelo. Em seguida, falou-se sobre a especificação dos acessos concorrentes as variáveis globais e locais (ou privadas a cada fluxo de execução), assim como a distribuição de cálculos entre *threads* específicas através da definição de seções paralelas.

A forma mais simples de se extrair o paralelismo usando o OPENMP é através da identificação das instruções de laços que podem ser executadas concorrentemente por *threads* diferentes. Seguindo nesta linha, foram apresentados o comportamento das cláusulas `private`, `schedule` e `reduce` com relação as iterações de um laço paralelizado com `#pragma omp parallel for`.

O OPENMP baseia-se num modelo de execução paralela *Fork/Join*, onde um conjunto de *threads* será criado para computar paralelamente um trabalho (*Fork*) e após a computação, acontecerá a sincronização e finalização das *threads* (*Join*). Adicionalmente, foram introduzidas algumas diretivas para delimitar regiões a serem executadas por uma única *thread* e regiões críticas. Em seguida, foram apresentadas características do OPENMP 3.0, o qual proporciona outro modelo de programação: o paralelismo de tarefas. Essa forma de paralelismo tem recebido atenção dado que é diretamente aplicável e apropriado a algoritmos recursivos.

Ao final deste minicurso gostaríamos de ressaltar que o OPENMP é uma ferramenta que, se bem empregada, é capaz de oferecer alto desempenho através da exploração do paralelismo explícito, tanto em programas sequenciais quanto paralelos. Para estes últimos, cabe ressaltar que OPENMP pode (e em alguns casos, deve) ser associado a outras ferramentas. Um bom exemplo disso é a grande quantidade de aplicações que associam OPENMP a programas MPI. Dessa forma, têm-se um uso intensivo do paralelismo local graças ao OPENMP (em especial quando há máquinas multicore envolvidas) e também a possibilidade de associar em paralelo arquiteturas com memória distribuída, através do modelo com troca de mensagens do MPI.

5.7. Bibliografia

[CHA 2001] CHANDRA, R. et al. **Parallel programming in openmp**. [S.l.]: Morgan Kaufmann, 2001.

[CHA 2008] CHAPMAN, B.; JOST, G.; PAS, R. van der. **Using openMP**: portable shared memory parallel programming. Cambridge, MA: MIT Press, 2008. (Scientific and Engineering Computation Series).