Programação Paralela em Arquiteturas Multi-Core/Programação em OpenMP

Introdução e história

O OpenMP (Open Multi-Processing) é uma API multi-plataforma para processamento paralelo baseado em memória compartilhada para as linguagens C/C++ e Fortran. Ela consiste de um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente que influenciam na execução do programa.

O OpenMP foi especificado por um grupo dos grandes fabricantes de hardware/software visando algo que seja portável e escalável, com uma interface de utilização bem simples e que pudesse ser utilizado tanto para aplicações de grande porte, quanto para aplicações simples de desktop.

Ele foi desenvolvido inicialmente para Fortran em 1997 (pode ser considerado como algo relativamente novo). No ano seguinte, foi lançada a primeira versão para C/C++. Em 2000 foi lançada a versão 2.0 para Fortran e em 2002 foi lançada a versão para C/C++. A versão atual é a 2.5 e saiu em 2005. Nessa versão, finalmente foram combinados os padrões para Fortran e C/C++.

Modelo de programação

O OpenMP usa um modelo *fork*/*join* (que é como um mestre/escravo). Há um fluxo de execução principal (a master thread) e quando necessário (por exemplo, em uma seção paralela), novas threads são disparadas para dividir o trabalho. Por fim, ao fim de uma seção paralela, é feito um *join*.

Uma recurso interessante do OpenMP é que a sincronização entre os threads quase sempre ocorre de maneira implícita, de maneira automática. Isso faz com que sua utilização seja algo muito simples. Inclusive, é possível fazer com que os programas compilem com ou sem OpenMP. Caso a biblioteca seja utilizada eles serão paralelos, caso contrário, seriais.

É importante citar algumas características que diferenciam o OpenMP de outras alternativas. No OpenMP não é possível ver como cada *thread* é criada e inicializada. Também não é visível uma função separada contendo o código que cada *thread* executa. A divisão de trabalho realizado sobre um arranjo também não é visível explicitamente. Enfim, quase tudo acontece "por trás das cortinas" de modo que a utilização seja o mais transparente possível.

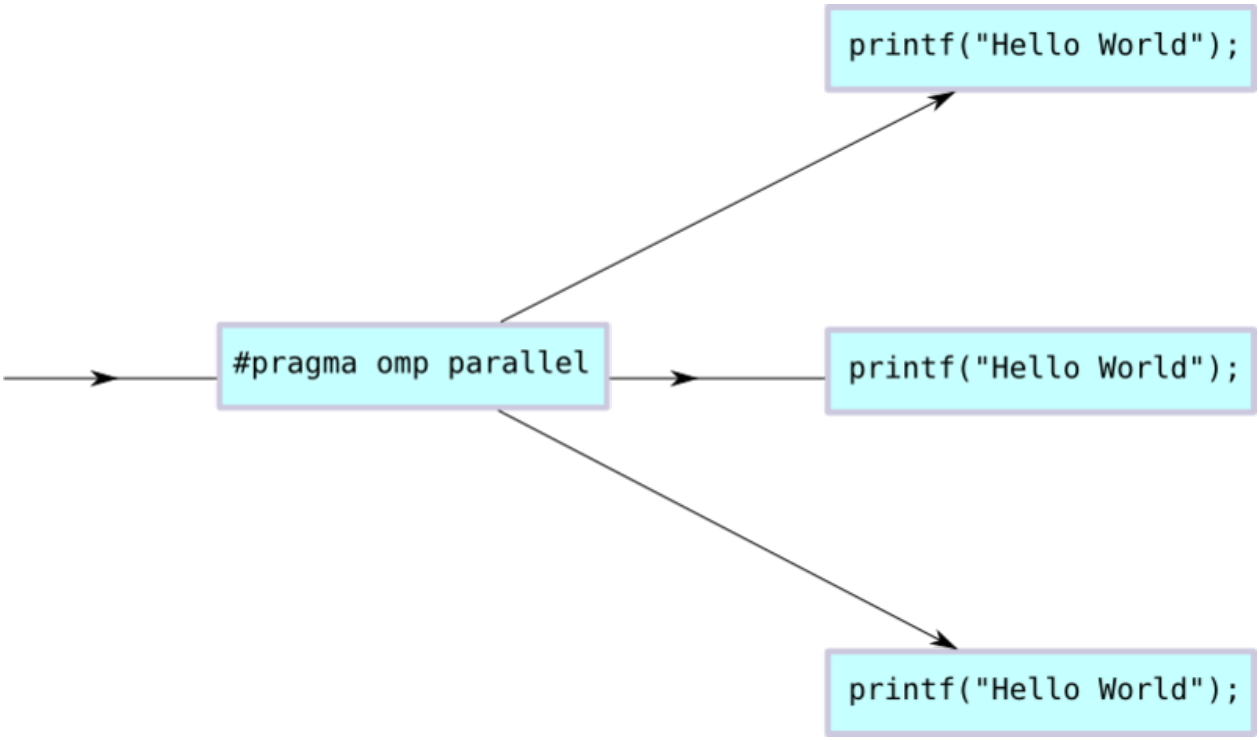
O modelo OpenMP é fortemente baseado em memória compartilhada, mas existem implementações para outras arquiteturas (tipo message-passing). Essas implementações não costumam ser tão eficientes quanto a implementação tradicional.

Visão geral

Construções para divisão de trabalho

Uma das construções chave é o programa "omp parallel", que declara uma seção paralela. Quando uma seção paralela é encontrada, *threads* são disparadas conforme necessário, e todas elas começam a executar o código dentro daquela seção paralela. As construções para divisão de trabalho citadas a seguir só funcionam dentro de uma seção paralela. Exemplo:

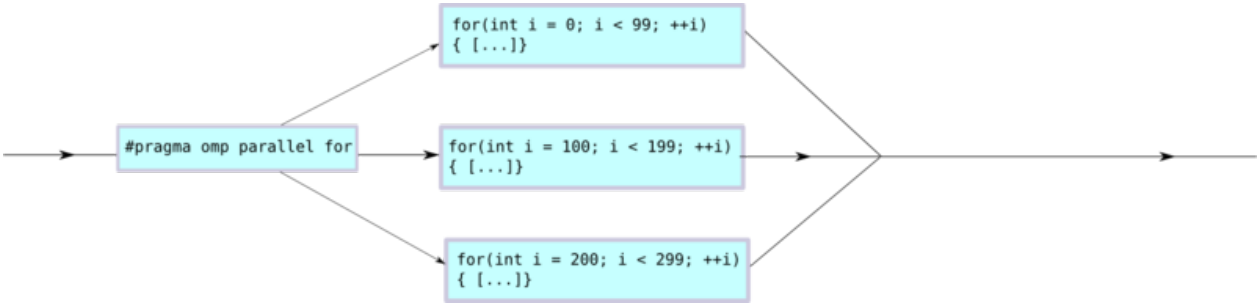
```
#pragma omp parallel
{
  printf("Hello World!");
}
```



Na definição da seção paralela deve ser informado ao compilador várias coisas como quais variáveis serão privadas de cada thread e quais serão compartilhadas - isso é feita através das palavras "shared" e "private". Pode-se também determinar explicitamente quantos *threads* são desejados, através da palavra "num_threads".

A primeira construção importante a ser citada é o "omp for". Nesta construção, o intervalo de iteração do *loop* é divido entre as threads e essa divisão acontece de forma automática. Por exemplo, dado o seguinte código:

```
#pragma omp parallel for
for(int i = 0; i < 400; ++i)
{
    z[i] = f(x[i], g(y[i]));
}
```



o corpo do loop seria executado uma vez para cada valor de i entre zero e 399, porém não de forma sequencial: os quatrocentos valores nesse intervalo seriam divididos entre as threads. Uma divisão possível seria que a thread 0 ficasse responsável pelos índices 0 a 99, a thread 1 pelos índices 100 a 199, e assim por diante. Essa divisão não é, porém, fixa. É possível alterar (até mesmo em tempo de execução) o número de threads, e a divisão de índices se adequará automaticamente.

Outra construção de divisão de trabalho é o "omp sections". Ela define blocos independentes, que podem ser distribuídos entre as *threads*. Cada *section* vai ser executada apenas por uma *thread*. Trata-se de uma forma simples de paralelizar tarefas distintas que não tem (ou tem pouca) dependência de dados entre si. Um exemplo simples de uso seria a inicialização de dois arranjos com conteúdos distintos e que podem ser inicializado de maneira independente um do outro. Exemplo:

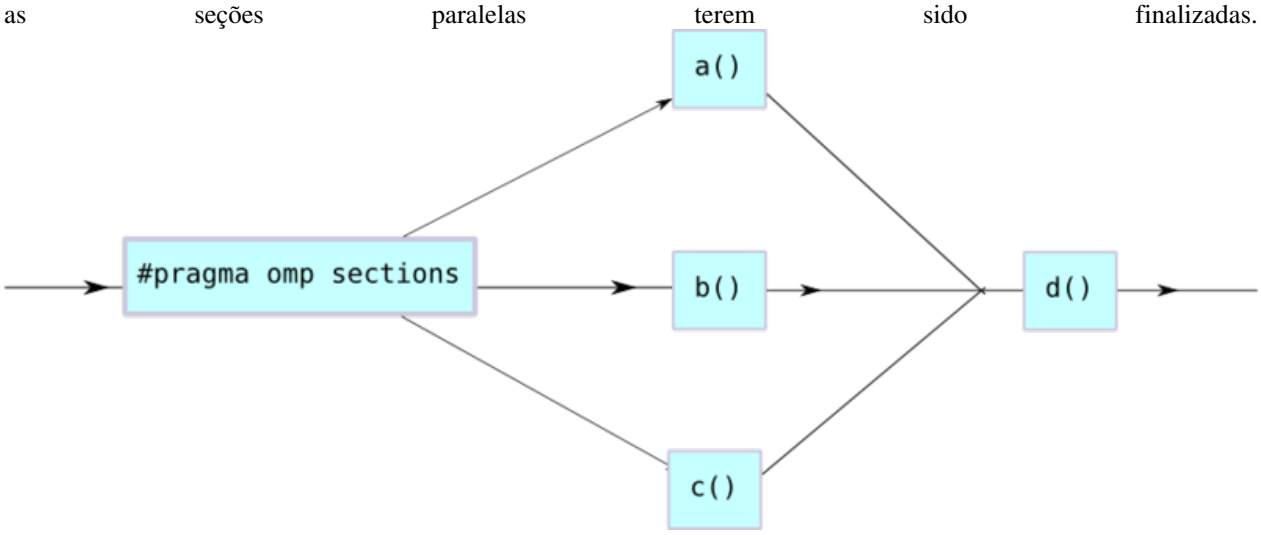
```
#pragma omp parallel sections
{
	#pragma omp section
	{
		a();
	}

	#pragma omp section
	{
		b();
	}

	#pragma omp section
	{
		c();
	}
}

d();
```

Nesse exemplo, as funções a, b e c seriam executadas em paralelo, enquanto a função d só seria chamada após todas as seções paralelas terem sido finalizadas.



Construções para sincronização

Apesar de o OpenMP tentar tratar o que pode de maneira transparente, em algumas situações são necessárias diretivas de sincronização para garantir consistência entre os estados dos diversos fluxos de execução.

As principais construções de sincronização são: barrier, critical, atomic, single, master, ordered.

"omp barrier" é uma barreira de memória, quando uma *thread* atinge uma barreira, ela só prossegue após todas as demais atingirem aquela mesma barreira. Esta é a construção mais básica de todas. Algumas construções openmp já incluem, implicitamente, uma barreira no final e se você não quiser essa barreira, tem que incluir uma cláusula "nowait". Por exemplo, considere o seguinte código:

```
#pragma omp parallel
{
```

```
  #pragma omp sections
  {
        #pragma omp section
        {
              a();
        }
        #pragma omp section
        {
              b();
        }
  } // Aqui há uma barreira implícita
  printf("Hello!");
} // Aqui há uma barreira implícita
```

Como se pode ver, há barreiras implícitas no fim das construções "omp parallel" e "omp section". Isso significa que caso, por exemplo, a thread que executa a primeira seção termine seu trabalho muito antes da thread que executa a segunda seção, ela irá parar e esperar a segunda seção terminar antes de imprimir "Hello!". Se incluirmos a cláusula "nowait":

```
#pragma omp parallel
{
  #pragma omp sections nowait
  {
        #pragma omp section
        {
              a();
        }
        #pragma omp section
        {
              b();
        }
  } // Não há mais barreira implícita aqui, devido a cláusula "nowait"
  printf("Hello!");
} // Aqui há uma barreira implícita
```

então assim que uma thread acabar de executar uma seção, ela pode sair do escopo de "omp sections" e imprimir "Hello", sem esperar que a outra seção esteja concluída. Além de barreiras implícitas, também é possível incluir explicitamente uma barreira:

```
#pragma omp parallel
{
  a();
  b();
  // Há uma barreira explícita aqui:
  #pragma omp barrier
  c();
  d();
} // Aqui há uma barreira implícita
```

As construções "critical" e "atomic" servem para dizer que o que estiver dentro deve ser executado somente por um *thread* ao mesmo tempo. A diferença entre as duas é que a "atomic" tenta utilizar recursos do hardware para ser mais eficiente. Se o código dentro de um atomic for complicado demais pro hardware, então ele é automaticamente convertido para um critical. Exemplo:

```
#pragma omp parallel for
for(int i = 0; i < 1048576; ++i)
{
  int partial_result = f(x[i], y[i], z[i]);
  #pragma omp atomic
  {
    sum += partial_result;
  }
}
```

Além dessas, existe também a diretiva "single". Ela especifica que o bloco de código deve ser executado por apenas um dos threads. A diretiva "master" é parecida: nela é garantido que a thread que vai executar aquele bloco é a *master thread*. Isso pode ser útil em alguns casos. Por exemplo, código pra atualizar interface gráfica em X11 normalmente só pode ser executado em uma thread específica, que é a principal. Se você quiser atualizar a tela, por qualquer motivo, de dentro de uma seção paralela, tem que garantir que essa atualização será feita pela master thread. Por exemplo, em

```
#pragma omp parallel
{
      a();
      b();
      c();
      #pragma omp barrier
      #pragma omp master
      {
            send_results_to_screen();
      }
      d();
      e();
      f();
}
```

a chamada à função send_results_to_screen() será feita apenas pela thread principal, mesmo ela estando dentro de uma seção paralela.

O OpenMP provê também a construção "ordered" que especifica que as interações do *loop* devem ser executadas na mesma ordem que seriam caso o programa fosse executado de maneira serial. O funcionamento dessa construção é mais facilmente explicado através de um exemplo:

```
#pragma omp parallel for
for(int i = 0; i < 1048576; ++i)
{
      std::string s = get_some_weird_string_in_a_weird_way(i);
      #pragma omp ordered
      {
            result = result + ", " + s;
```

```
        }
}
```

Nesse exemplo, o valor da variável "s" vai ser calculado em paralelo para diversos valores de "i", mas a redução desses valores calculados ao valor final (representado pela variável "result") se dará em ordem. Essa construção é extremamente interessante em situações do tipo map-reduce aonde a operação de mapeamento é relativamente cara e pode ser beneficiada por paralelismo, enquanto a operação de redução é relativamente barata mas necessita dos dados em ordem.

Por fim, é importante citar a diretiva "flush" que identifica um ponto de sincronização, aonde a implementação deve garantir que todas as threads tenham uma "visão" consistente da memória. Pode-se especificar explicitamente os pontos de flush, mas há também situações nas quais essa diretiva existe implicitamente:

- "omp barrier"
- entrada e saída de "omp critical"
- saída de "omp parallel"
- saída de "omp for"
- saída de "omp sections"
- saída de "omp single"

Funções da biblioteca

Além das diretivas implícitas da OpenMP existe também uma biblioteca com funções que podem ser chamadas de dentro dos programas. Com estas funções é possível ler e escrever em valores como um número de threads utilizados globalmente, etc. É possível também determinar, por exemplo, se a seção atual está sendo executada em paralelo ou não (através da função *omp_in_parallel*). É possível ligar ou desligar o ajustamento automático do número de *threads*, através das funções *omp_set_dynamic*. Entre várias outras que permitem a um programa ter informações úteis sobre o ambiente de execução.

Uma outra forma de especificar esses parâmetros é através de variáveis de ambiente que são lidas pelo runtime do OpenMP. As variáveis disponíveis são: OMP_SCHEDULE, OMP_NUM_THREADS, OMP_DYNAMIC, OMP_NESTED.

Também é importante comentar sobre as *Locking Functions* e *Timing Routines*. Existem dois tipos de *locks*: *simple* e *nestable*. As simples são *locks* simples, ordinárias. As *nestable* podem ser obtidas diversas vezes pelo mesmo threads e elas devem ser liberadas um número igual de vezes para que outro threads consiga obtê-la.

Estas funções são:

- Lock Functions:
  - omp_init_lock, omp_init_nest_lock
  - omp_destroy_lock, omp_destroy_nest_lock
  - omp_set_lock, omp_set_nest_lock
  - omp_unset_lock, omp_unset_nest_lock
  - omp_test_lock, omp_test_nest_lock
- Timing Routines:
  - omp_get_wtime Function, omp_get_wtick Function

Outros aspectos relevantes

Os compiladores que atualmente suportam OpenMP são o GCC a partir da versão 4.2 (4.1 no Fedora), o Visual Studio C++ 2005, os compiladores da Intel e o Sun Studio.

Exemplos

Exemplo 1: Hello World

```
int main()
{
  omp_set_num_threads(4); // Configura o número de treads
  #pragma omp parallel // Inicia uma seção paralela
  {
    // Código aqui dentro será executado por todas as threads...
    #pragma omp critical // ...porém uma de cada vez
    {
      printf("Hello, World! I'm thread %d.\n", omp_get_thread_num());
    }
  } // Fim da seção paralela
}
```

Esse exemplo não é exatamente útil, mas ilustra bem vários aspectos da API OpenMP.

Exemplo 2: "omp for": Mesmo código, dados diferentes

```
int sample(int **input_array, int size)
{
  int i, j, k, result;
  result = 0;
  #pragma omp parallel shared(i, result) private(j,k) // As variáveis
"result" e "i" são comuns a todas as threads, as variáveis j e k são
diferentes para cada thread
  {
    #pragma omp for
    for(i = 0; i < size; ++i) // Note que o limite de iterações é
dado por uma variável, ele não precisa ser estático
    {
      k = 0;
      for(j = 0; j < 42000; ++j)
      {
        k += input_array[i][j] - 1;
      }
      #pragma omp atomic
      {
        result += k;
      }
    }
  }
  return k;
}
```

Esse exemplo mostra o funcionamento da construção "omp for", que divide as iterações de um dado loop "for" entre as threads, bem como a especificação de escopo para variáveis.

Exemplo 3: Execução paralela

O exemplo abaixo mostra duas threads executando ao mesmo tempo, realizando cada uma seu trabalho.

```
void do_two_things()
{
      #pragma omp sections nowait
      {
            #pragma omp section
            {
                  printf("Hi, I'm thread %d, I'm doing something.\n",
omp_get_thread_num());
                  while(not_done_yet("me"))
                  {
                        do_something();
                  }

            }

            #pragma omp section
            {
                  printf("Hi, I'm thread %d, I'm doing some other
thing.\n", omp_get_thread_num());
                  while(not_done_yet("myself"))
                  {
                        do_some_other_thing();
                  }
            }
      }
}
```

Vantagens e desvantagens

As principais vantagens são:

- Simplicidade, pois o OpenMP cuida da maioria das coisas para o usuário e a distribuição de tarefas é feita automaticamente pela implementação;
- Paralelizar código já existente é simples, requer poucas modificações, e o paralelismo pode ser implementado de forma incremental, já que não requer grandes modificações estruturais no código. Por exemplo, no caso de um "for", o OpenMP cuida de quais threads irão fazer o que automaticamente e, em geral, o programa continua sendo válido como um programa serial, caso o OpenMP não seja utilizado;
- Compacto, poderoso e simples.

As desvantagens são:

- Atualmente, só executa de maneira eficiente em arquiteturas de memórias compartilhada;
- Requer suporte do compilador;
- Escalabilidade é limitada pela arquitetura da memória;
- Tratamento de erros ainda é um problema;

- Falta alguns tipos de controle mais "fino" sobre os threads, alteração de alguns parâmetros, etc.

Comparação com pthreads

Como já citado anteriormente, o que fica mais claro quando se compara OpenMP com pthreads é que o OpenMP é bem mais simples. Contudo, também é possível perceber que com pthreads você pode conseguir alguma performance melhor caso otimize bastante seu código pois ele permite um controle mais fino.

Referências

Introduções ao assunto

- http://equipe.nce.ufrj.br/gabriel/progpar/OpenMP.pdf
- http://developers.sun.com/sunstudio/articles/omp-intro.html?feed=DSC
- http://www.cesup.ufrgs.br/Downloads/Outros/OpenMP/OPENMP_O.PDF
- http://www.inf.ufrgs.br/~elgios/trabs-html/Nucleo/openMP.html

Tutoriais

- http://www.llnl.gov/computing/tutorials/openMP/
- http://scv.bu.edu/tutorials/OpenMP/
- http://bisqwit.iki.fi/story/howto/openmp/
- http://kallipolis.com/openmp/
- http://www.plutospin.com/files/OpenMP_reference.pdf

Artigos

- http://www.thinkingparallel.com/2006/08/12/why-openmp-is-the-way-to-go-for-parallel-programming/
- http://msdn.microsoft.com/msdnmag/issues/05/10/OpenMP/#S7
- http://www-usr.inf.ufsm.br/~reis/publicacoes/wsl2007_gomp.pdf
- http://cache-www.intel.com/cd/00/00/29/78/297875_297875.pdf
- http://www.openmp.org/specs/mp-documents/paper/paper.ps

Fontes e Editores da Página

Programação Paralela em Arquiteturas Multi-Core/Programação em OpenMP *Fonte*: http://pt.wikibooks.org/w/index.php?oldid=127094 *Contribuidores*: Jorge Morais, Nilsonsfj, Tawhaki, 4 edições anónimas

Fontes, licenças e editores da imagem

Imagem:Omp-parallel-for.png *Fonte*: http://pt.wikibooks.org/w/index.php?title=Ficheiro:Omp-parallel-for.png *Licença*: desconhecido *Contribuidores*: Tawhaki

Imagem:omp-parallel-for-exemplo.png *Fonte*: http://pt.wikibooks.org/w/index.php?title=Ficheiro:Omp-parallel-for-exemplo.png *Licença*: desconhecido *Contribuidores*: Tawhaki

Imagem:omp-sections.png *Fonte*: http://pt.wikibooks.org/w/index.php?title=Ficheiro:Omp-sections.png *Licença*: desconhecido *Contribuidores*: Tawhaki

Licença