
Conteúdo

[Aprenda](#) > Software livre

Aprendendo a usar a estrutura OpenMP com o GCC



Arpan Sen

Publicado em 08/Out/2012

A estrutura OpenMP é uma maneira eficiente de escrever programas paralelos. O GCC (GNU Compiler Collection) versão 4.2 tem suporte para o OpenMP 2.5, a versão mais recente. Outros compiladores, como o Microsoft Visual C++, também aprenderá a usar pragmas de compilador da estrutura OpenMP. Neste artigo, aprenderá a usar pragmas de compilador da estrutura OpenMP, a programação de aplicativos (APIs) que ela fornece e a testá-la com alguns algoritmos paralelos usando o compilador preferencial.



Page Saved!



O seu primeiro programa de OpenMP

Vamos começar com um simples programa para exibir a frase **Hello, World!**, que inclui um pragma adicional. [A Listagem 1](#) mostra o código.

Lista 1. Hello World com OpenMP

```
1  #include <iostream>
2  int main()
3  {
4      #pragma omp parallel
5      {
6          std::cout << "Hello World!\n";
7      }
8  }
```

Lista 2. Compilando e executando o código com o comando -fopenmp

Conteúdo

```
1 tintin$ g++ test1.cpp -fopenmp
2 tintin$ ./a.out
3 Hello World!
4 Hello World!
5 Hello World!
6 Hello World!
7 Hello World!
8 Hello World!
9 Hello World!
10 Hello World!
```

O que aconteceu? A mágica de `#pragma omp parallel` funciona apenas quando a opção `-fopenmp` é especificada. Internamente, durante a compilação, o GCC gera o código para criar o máximo possível no tempo de execução, com base no hardware e na configuração do sistema operacional. O encadeamento é o código no bloco após o pragma. Esse comportamento chama-se *parallelização*. OpenMP consiste em um conjunto de pragmas eficientes que livram o desenvolvedor da obra repetitivos. (Para comparação, confira como seria uma implementação do que você acaba de fazer com encadeamentos da Interface de Sistema Operacional). Seu computador possui um processador Intel® Core i7, com quatro núcleos físicos e oito encadeamentos lógicos adequados (8 encadeamentos = 8 núcleos lógicos).



Page Saved!



Add Tags

Agora vamos ver mais detalhes sobre pragmas paralelos.

Brincando com OpenMP paralelo

Para controlar o número de encadeamentos, basta usar o argumento `num_threads` do pragma da [Listagem 1](#) com o número de encadeamentos disponíveis especificados como 5 (como mostrado na [Listagem 2](#)).

Lista 3. Controlando o número de encadeamentos com `num_threads`

```
1 #include <iostream>
2 int main()
3 {
4     #pragma omp parallel num_threads(5)
5     {
6         std::cout << "Hello World!\n";
7     }
8 }
```

cabeçalho `omp.h`. Não é necessário vincular outras bibliotecas para que o código na [Listagem 4](#)

Conteúdo

Lista 4. Uso de `omp_set_num_threads` para ajuste fino da criação de encadeamentos

```
1 | #include <omp.h>
2 | #include <iostream>
3 | int main()
4 | {
5 |     omp_set_num_threads(5);
6 |     #pragma omp parallel
7 |     {
8 |         std::cout << "Hello World!\n";
9 |     }
10 | }
```

O OpenMP também usa variáveis de ambiente externas para controlar seu comportamento. [Listagem 2](#) para imprimir apenas **Hello World!** seis vezes, definindo a variável `OMP_NUM_THREADS` execução.

Lista 5. Usando variáveis de ambiente para ajustar o comportamento do OpenMP

```
1 | tintin$ export OMP_NUM_THREADS=6
2 | tintin$ ./a.out
3 | Hello World!
4 | Hello World!
5 | Hello World!
6 | Hello World!
7 | Hello World!
8 | Hello World!
```



Page Saved!



Add Tags

Você já descobriu todas as três facetas do OpenMP: pragmas do compilador, APIs do tempo ambiente. O que acontece se usarmos a variável de ambiente e a API do tempo de execução alta.

Um exemplo prático

O OpenMP usa técnicas de paralelização implícita, e é possível usar pragmas, funções explícitas para instruir o compilador. Vamos examinar um exemplo no qual OpenMP pode ser de grande ajuda. [Listagem 6](#).

Lista 6. Processamento sequencial em um loop `for`

```
1 | int main( )
2 | {
```

```

6   for (int i = 0; i < 10000000; ++i)
7       c[i] = a[i] * b[i] + a[i-1] * b[i+1];
8   // ... now do some processing with array c
9   }

```

Claramente, é possível dividir o loop `for` e executar em mais de um núcleo. O cálculo de outros elementos do array `c`. [A Listagem 7](#) mostra como o OpenMP ajuda a fazer isso.

Lista 7. Processamento paralelo em um loop `for` com o `pragma parallel for`

```

1   int main( )
2   {
3       int a[10000000], b[10000000];
4       // ... some initialization code for populating arrays a and b;
5       int c[10000000];
6       #pragma omp parallel for
7       for (int i = 0; i < 10000000; ++i)
8           c[i] = a[i] * b[i] + a[i-1] * b[i+1];
9       // ... now do some processing with array c
10  }

```

O `pragma parallel for` ajuda a dividir a carga de trabalho. Cada iteração pode ser executado em um núcleo diferente, o que acelera o processamento.



Page Saved!



Add Tags

Lista 8. Entendendo `omp_get_wtime`

developerWorks®

Aprenda

Desenvolva

Conecte-se

```

1   #include <omp.h>
2   #include <math.h>
3   #include <time.h>
4   #include <iostream>
5
6   int main(int argc, char *argv[]) {
7       int i, nthreads;
8       clock_t clock_timer;
9       double wall_timer;
10      double c[10000000];
11      for (nthreads = 1; nthreads <= 8; ++nthreads) {
12          clock_timer = clock();
13          wall_timer = omp_get_wtime();
14          #pragma omp parallel for private(i) num_threads(nthreads)
15          for (i = 0; i < 10000000; i++)
16              c[i] = sqrt(i * 4 + i * 2 + i);
17          std::cout << "threads: " << nthreads << " time on clock(): " <<
18              (double) (clock() - clock_timer) / CLOCKS_PER_SEC
19              << " time on wall: " << omp_get_wtime() - wall_timer << "\n";
20      }
21  }

```

Ordenação por intercalação (merge sort) com OpenMP

Na [Listagem 8](#), o código aumenta continuamente o número de encadeamentos para medir o tempo de execução. A API `omp_get_wtime` retorna o tempo em segundos a partir de algum ponto de partida.

Temas relacionados

processador dos encadeamentos individuais é somado antes de a chamada informar o número de threads. Como no caso do processador Intel Core i7, a [Listagem 9](#) mostra as informações exibidas.

Lista 9. Números para a execução do loop for interno

```
1 threads: 1 time on clock(): 0.015229 time on wall: 0.0152249
2 threads: 2 time on clock(): 0.014221 time on wall: 0.00618792
3 threads: 3 time on clock(): 0.014541 time on wall: 0.00444412
4 threads: 4 time on clock(): 0.014666 time on wall: 0.00440478
5 threads: 5 time on clock(): 0.01594 time on wall: 0.00359988
6 threads: 6 time on clock(): 0.015069 time on wall: 0.00303698
7 threads: 7 time on clock(): 0.016365 time on wall: 0.00258303
8 threads: 8 time on clock(): 0.01678 time on wall: 0.00237703
```

Embora o tempo de processador seja quase o mesmo em todas as execuções (como deveria ser, considerando o tempo adicional para criar os encadeamentos e o comutador de contexto), o que nos interessa é o tempo de execução total, que diminui progressivamente à medida que o número de encadeamentos aumenta, dando a entender que o trabalho é distribuído e calculado pelos núcleos em paralelo. Uma nota final sobre a sintaxe do pragma: `#pragma omp for` significa que a variável de loop `i` deve ser tratada como um armazenamento local de encadeamento, tendo uma cópia da variável. A variável local do



Page Saved!



Seções críticas com Op

Você não achou que o OpenMP ia cuidar das seções críticas sozinho, achou? Claro, não é nem assim, mas ainda é preciso especificar a seção crítica. Aqui está a sintaxe:

```
1 #pragma omp critical (optional section name)
2 {
3 // no 2 threads can execute this code block concurrently
4 }
```

O código que vem depois de `pragma omp critical` pode apenas ser executado por um único thread a cada momento. Além disso, `optional section name` é um identificador global, e dois encadeamentos críticos com o mesmo identificador global ao mesmo tempo. Considere o código da [Listagem 10](#).

Lista 10. Mais de uma seção crítica com o mesmo nome

```
1 #pragma omp critical (section1)
2 {
3 myhashtable.insert("key1", "value1");
4 }
5 // ... other code follows
6 #pragma omp critical (section1)
```

Conteúdo

Com base nesse código, podemos supor com segurança que as duas inserções de hashtable simultaneamente, pois os nomes da seção crítica são os mesmos. Isso é um pouco diferente do que é acostumado a lidar com seções críticas usando pthreads, que são, em grande parte, caracterizados por bloqueios.

Bloqueios e mutexes com OpenMP

Curiosamente, OpenMP tem suas próprias versões de mutexes (então não se trata apenas de `omp_lock_t`, definido como parte do arquivo de cabeçalho `omp.h`). As operações de mutex nos nomes das APIs são semelhantes. Há cinco APIs que o desenvolvedor deve conhecer:

- `omp_init_lock`: Essa deve ser a primeira API a acessar `omp_lock_t`. É usada para inicializar o bloqueio, considera-se que o bloqueio não foi definido.
- `omp_destroy_lock`: Essa API destrói o bloqueio. O bloqueio deve estar no estado não bloqueado, o que significa que não é possível chamá-lo.
- `omp_set_lock`: Essa API define o bloqueio. O bloqueio deve estar no estado não bloqueado, o que significa que não é possível chamá-lo.
- `omp_test_lock`: Essa API tenta bloquear se o bloqueio estiver disponível, e retorna 1 em caso de sucesso. Essa é uma *API sem bloqueio* — ou seja, essa função não faz o encadeamento de chamadas.
- `omp_unset_lock`: Essa API libera o bloqueio.

A [Listagem 11](#) mostra uma implementação trivial de uma fila legada de um encadeamento e multiencadeamento usando bloqueios do OpenMP. Observe que isso pode não ser o ideal, pois é apenas uma ilustração rápida.

Lista 11. Usando OpenMP para estender uma fila de um encadeamento

```
1  #include <openmp.h>
2  #include "myqueue.h"
3
4  class omp_q : public myqueue<int> {
5  public:
6      typedef myqueue<int> base;
7      omp_q( ) {
8          omp_init_lock(&lock);
9      }
10     ~omp_q() {
11         omp_destroy_lock(&lock);
12     }
13     bool push(const int& value) {
```

```
17     return result;
18 }
19 bool trypush(const int& value)
20 {
21     bool result = omp_test_lock(&lock);
22     if (result) {
23         result = result && this->base::push(value);
24         omp_unset_lock(&lock);
25     }
26     return result;
27 }
28 // likewise for pop
29 private:
30     omp_lock_t lock;
31 };
```

Bloqueios aninhados

Outros tipos de bloqueios fornecidos pelo OpenMP são variações do `omp_nest_lock_t`. São vantagem de que podem ser bloqueados várias vezes pelo encadeamento que já está realizando bloqueio aninhado é readquirido pelo encadeamento aumentado. O bloqueio é liberado pelo encadeamento reconfiguram o contador do bloqueio interno para

**Page Saved!**

- `omp_init_nest_lock(omp_nest_lock_t*)`: Essa API inicializa o contador de aninhamento.
- `omp_destroy_nest_lock(omp_nest_lock_t*)`: Essa API destrói o bloqueio. Um chamado com contagem de aninhamento interno diferente de zero resulta em comportamento indefinido.
- `omp_set_nest_lock(omp_nest_lock_t*)`: Essa API é semelhante a `omp_set_lock`, mas funciona mais de uma vez enquanto mantém o bloqueio.
- `omp_test_nest_lock(omp_nest_lock_t*)`: Essa API é uma versão sem bloqueio de `omp_test_lock`.
- `omp_unset_nest_lock(omp_nest_lock_t*)`: Essa API libera o bloqueio quando o contador é diminuído com cada chamada para esse método.

Controle de baixa granularidade sobre a execução

Você já viu que todos os encadeamentos executam o bloco de códigos após `pragma omp parallel`. Você pode categorizar ainda mais o código dentro desse bloco para ser executado por encadeamentos da [Listagem 12](#).

Lista 12. Aprendendo a usar o `pragma` de seções paralelas

```

3  #pragma omp parallel
4  {
5      cout << "All threads run this\n";
6      #pragma omp sections
7      {
8          #pragma omp section
9          {
10             cout << "This executes in parallel\n";
11         }
12         #pragma omp section
13         {
14             cout << "Sequential statement 1\n";
15             cout << "This always executes after statement 1\n";
16         }
17         #pragma omp section
18         {
19             cout << "This also executes in parallel\n";
20         }
21     }
22 }
23 }

```

O código que vem antes de `pragma omp sections`, mas logo após `pragma omp parallel`, é o encadeamento em paralelo. O bloco que vem depois de `pragma omp sections` é classificado em encadeamentos individuais usando `pragma omp section`. Cada bloco de código dentro de um `section` é executado individualmente. No entanto, as instruções dentro de um `section` são executadas na sequência. [A Listagem 13](#) mostra a saída do código.



Page Saved!



Add Tags

Lista 13. Saída do código da Listagem 12

```

1  tintin$ ./a.out
2  All threads run this
3  All threads run this
4  All threads run this
5  All threads run this
6  All threads run this
7  All threads run this
8  All threads run this
9  All threads run this
10 This executes in parallel
11 Sequential statement 1
12 This also executes in parallel
13 This always executes after statement 1

```

Na Listagem 13, temos novamente oito encadeamento sendo criados inicialmente. Desses criamos apenas três deles no bloco `pragma omp sections`. Na segunda seção, especificamente as instruções de impressão são executadas. É esse o motivo para usar o `pragma sections`. Se não especificar a ordem dos blocos de códigos.

loops paralelos

Conteúdo

Anteriormente, você viu o uso de `private` para declarar o armazenamento local de encadeamento de variáveis locais de encadeamento? Talvez sincronizá-las com a variável no encadeamento por operações? É nessa situação que a diretiva `firstprivate` é útil.

A diretiva `firstprivate`

Ao usar `firstprivate(variable)`, é possível inicializar a variável em um encadeamento paralelo. Considere o código da [Listagem 14](#).

Lista 14. Usando a variável local do encadeamento que não está sincronizada com o encadeamento

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main()
5  {
6      int idx = 100;
7      #pragma omp parallel private(idx)
8      {
9          printf("In thread %d idx = %d\n",
10      }
11 }
```



Page Saved!



Add Tags

Aqui está a saída que eu recebi. Seus resultados podem ser diferentes.

```
1  In thread 1 idx = 1
2  In thread 5 idx = 1
3  In thread 6 idx = 1
4  In thread 0 idx = 0
5  In thread 4 idx = 1
6  In thread 7 idx = 1
7  In thread 2 idx = 1
8  In thread 3 idx = 1
```

A [Listagem 15](#) mostra o código com a diretiva `firstprivate`. A saída, como era esperado, in em todos os encadeamentos.

Lista 15. Usando a diretiva `firstprivate` para inicializar as variáveis locais do encadeamento

```
1  #include <stdio.h>
2  #include <omp.h>
3
```

```

7 | #pragma omp parallel firstprivate(idx)
8 | {
9 |     printf("In thread %d idx = %d\n", omp_get_thread_num(), idx);
10 | }
11 | }

```

Observe também que usamos o método `omp_get_thread_num()` para acessar o ID de um encadeamento e o ID de encadeamento que o comando `top` do Linux® mostra. Esse esquema é apenas uma maneira de visualizar as contagens de encadeamento. Outra nota sobre a diretiva `firstprivate`, caso você queira usar uma variável que a diretiva `firstprivate` usa é um construtor de cópias para inicializar-se a partir do valor principal. Portanto, ter um construtor de cópias privado em uma classe invariavelmente resulta em passar para a diretiva `lastprivate`, que é, de certa forma, o reverso da moeda.

A diretiva `lastprivate`

Em vez de inicializar uma variável local de encadeamento com os dados do encadeamento principal, a diretiva `lastprivate` sincroniza os dados do encadeamento principal com o valor da variável local quando o encadeamento executa um loop `for` paralelo.

Lista 16. Usando um loop `for` paralelo sem sincronização

```

1 | #include <stdio.h>
2 | #include <omp.h>
3 |
4 | int main()
5 | {
6 |     int idx = 100;
7 |     int main_var = 2120;
8 |
9 |     #pragma omp parallel for private(idx)
10 |    for (idx = 0; idx < 12; ++idx)
11 |    {
12 |        main_var = idx * idx;
13 |        printf("In thread %d idx = %d main_var = %d\n",
14 |            omp_get_thread_num(), idx, main_var);
15 |    }
16 |    printf("Back in main thread with main_var = %d\n", main_var);
17 | }

```

No meu computador de desenvolvimento, com oito núcleos, o OpenMP cria seis encadeamentos. Cada encadeamento, por sua vez, é responsável por duas iterações do loop. O valor final de `main_var` é o valor de `idx` no encadeamento executado e, portanto, o valor de `idx` nesse encadeamento. Em outras palavras, o valor final de `main_var` depende do último valor de `idx`, mas não do valor de `idx` no encadeamento executado por último.

```

1  In thread 4 idx = 8 main_var = 64
2  In thread 2 idx = 4 main_var = 16
3  In thread 5 idx = 10 main_var = 100
4  In thread 3 idx = 6 main_var = 36
5  In thread 0 idx = 0 main_var = 0
6  In thread 1 idx = 2 main_var = 4
7  In thread 4 idx = 9 main_var = 81
8  In thread 2 idx = 5 main_var = 25
9  In thread 5 idx = 11 main_var = 121
10 In thread 3 idx = 7 main_var = 49
11 In thread 0 idx = 1 main_var = 1
12 In thread 1 idx = 3 main_var = 9
13 Back in main thread with main_var = 9

```

Execute o código na Listagem 17 algumas vezes para convencer-se de que o valor de `main_var` sempre depende do valor de `idx` no último encadeamento executado. E se quisermos sincronizar o principal com o valor final de `idx` no loop? É nessa parte que entra a diretiva `lastprivate` 18. Assim como no código da Listagem 17, execute o código da Listagem 18 algumas vezes para ver o valor final de `main_var` no encadeamento principal é 121 (`idx` é o valor do final do contador de loop).

Lista 18. Usando a diretiva `lastprivate` para sincronizar o principal com o valor final de `idx` no loop.

```

1  #include <stdio.h>
2  #include <omp.h>
3
4  int main()
5  {
6      int idx = 100;
7      int main_var = 2120;
8
9      #pragma omp parallel for private(idx) lastprivate(main_var)
10     for (idx = 0; idx < 12; ++idx)
11     {
12         main_var = idx * idx;
13         printf("In thread %d idx = %d main_var = %d\n",
14             omp_get_thread_num(), idx, main_var);
15     }
16     printf("Back in main thread with main_var = %d\n", main_var);
17 }

```



Page Saved!



Add Tags

A Listagem 19 mostra a saída da Listagem 18.

Lista 19. Saída do código na Listagem 18 (observe que o valor de `main_var` always é 121 no encadeamento principal).

```

1  In thread 3 idx = 6 main_var = 36
2  In thread 2 idx = 4 main_var = 16
3  In thread 1 idx = 2 main_var = 4
4  In thread 4 idx = 8 main_var = 64
5  In thread 5 idx = 10 main_var = 100
6  In thread 3 idx = 7 main_var = 49
7  In thread 0 idx = 0 main_var = 0

```

```
11 In thread 5 idx = 11 main_var = 121
12 In thread 0 idx = 1 main_var = 1
13 Back in main thread with main_var = 121
```

Uma observação final: para que um objeto C++ tenha suporte para o operador `lastprivate`, `operator=` esteja disponível publicamente na classe correspondente.

Ordenação por intercalação (merge sort) con

Vamos observar um exemplo prático no qual o conhecimento sobre OpenMP ajuda a economizar. Não é uma versão muito otimizada do algoritmo merge sort, mas é o suficiente para mostrar o código. [A Listagem 20](#) mostra o código de exemplo.

Lista 20. Ordenação por intercalação usando OpenMP

```
1  #include <omp.h>
2  #include <vector>
3  #include <iostream>
4  using namespace std;
5
6  vector<long> merge(const vector<long> &left, const vector<long> &right)
7  {
8      vector<long> result;
9      unsigned left_it = 0, right_it = 0;
10
11     while(left_it < left.size() && right_it < right.size())
12     {
13         if(left[left_it] < right[right_it])
14         {
15             result.push_back(left[left_it]);
16             left_it++;
17         }
18         else
19         {
20             result.push_back(right[right_it]);
21             right_it++;
22         }
23     }
24
25     // Push the remaining data from both vectors onto the resultant
26     while(left_it < left.size())
27     {
28         result.push_back(left[left_it]);
29         left_it++;
30     }
31
32     while(right_it < right.size())
33     {
34         result.push_back(right[right_it]);
35         right_it++;
36     }
37 }
```



Page Saved!



Add Tags

```
41 vector<long> mergesort(vector<long>& vec, int threads)
42 {
43     // Termination condition: List is completely sorted if it
44     // only contains a single element.
45     if(vec.size() == 1)
46     {
47         return vec;
48     }
49
50     // Determine the location of the middle element in the vector
51     std::vector<long>::iterator middle = vec.begin() + (vec.size() / 2);
52
53     vector<long> left(vec.begin(), middle);
54     vector<long> right(middle, vec.end());
55
56     // Perform a merge sort on the two smaller vectors
57
58     if (threads > 1)
59     {
60         #pragma omp parallel sections
61         {
62             #pragma omp section
63             {
64                 left = mergesort(left, threads/2);
65             }
66             #pragma omp section
67             {
68                 right = mergesort(right, threads/2);
69             }
70         }
71     }
72     else
73     {
74         left = mergesort(left, 1);
75         right = mergesort(right, 1);
76     }
77
78     return merge(left, right);
79 }
80
81 int main()
82 {
83     vector<long> v(1000000);
84     for (long i=0; i<1000000; ++i)
85         v[i] = (i * i) % 1000000;
86     v = mergesort(v, 1);
87     for (long i=0; i<1000000; ++i)
88         cout << v[i] << "\n";
89 }
```

**Page Saved!**

Usando oito encadeamentos para executar esse merge sort, a duração do tempo de execução ficou em 3,7 segundos, enquanto um único encadeamento deu 3,7 segundos. A única coisa que se deve tomar cuidado com o número de encadeamentos. Eu comecei com oito encadeamentos. Sua experiência pode variar de acordo com a configuração do seu sistema. No entanto, sem a contagem explícita de encadeamentos, senão milhares, deles, com altas chances de que o desempenho do sistema decaia. O uso de `sections`, discutido anteriormente, foi bem utilizado com o código merge sort.

Conclusão

Conteúdo

Assim acaba o artigo. Nós avançamos bastante aqui: você conheceu os pragmas paralelos e maneiras de criar encadeamentos; ficou convencido das melhorias em desempenho de tem baixa granularidade que o OpenMP oferece; e terminou com uma aplicação prática do OpenMP. Para estudar, é o melhor lugar para isso é o site do projeto OpenMP. Não deixe de conferir os detalhes.

Recursos para download



[PDF desse conteúdo](#)

Temas relacionados

- Não deixe de conferir o [site do projeto OpenMP](#)
- Para mais informações sobre como melhorar a performance de algoritmos de ordenação, confira o artigo [Passing, and Hybrid Merge Sorts for Standalone](#)



Page Saved!



Add Tags

Comentários

[Acesse](#) ou [registre-se](#) para adicionar e acompanhar os comentários.

☐ Receba notificações dos comentários

developerWorks

Sobre

Ajuda

Relatar abuso

Aviso de termos legais de terceiros/parceiros

Nos siga!

Conheça

Programa Acadêmico da IBM (em inglês)

Programa IBM de apoio a startups (em inglês)

Jornadas de aprendizado (em inglês)

Selecione um idioma

English

中文

日本語

Русский

Português (Brasil)

Español

한글

Downloads e trials

Feeds RSS

Newsletters (Inglês)

Tutoriais & treinamentos

Contato

Privacidade

Termos de uso

Acessibilidade

Feedback

Preferências de cookie