Capítulo 3

Introdução à Programação Paralela com OpenMP: Além das Diretivas de Compilação

Rogério A. Gonçalves, João Martins de Queiroz Filho e Alfredo Goldman

Abstract

Compilation directives have been widely used for adapting legacy code applications to the available resources on platforms that evolve and become increasingly heterogeneous. `OpenMP` *has followed this evolution and has provided directives for these new application contexts. This text introduces* `OpenMP` *showing the main compilation directives that cover the parallel regions, loops, sections, tasks, and device accelerators. We want to show the structure of the code generated by expansion of the directives that are formed by its constructors and their clauses. The goal is to introduce* `OpenMP` *presenting a different view of commons tutorials, showing* `OpenMP` *from the generated code point of view, the code with the directives expansion and their relations with parallel programming concepts.*

Resumo

Diretivas de compilação tem sido amplamente utilizadas para a adaptação de aplicações de código legado aos recursos disponíveis em plataformas que evoluem e tornam-se cada vez mais heterogêneas. O `OpenMP` *tem acompanhado essa evolução fornecendo diretivas para esses novos contextos. Este material apresenta uma introdução ao* `OpenMP` *mostrando as principais diretivas de compilação para a cobertura de regiões paralelas, laços, seções, tasks e dispositivos aceleradores. Queremos mostrar a estrutura do código gerado pela expansão das diretivas, formadas por seus construtores e suas cláusulas. O objetivo é apresentar o* `OpenMP` *dando uma visão diferente dos tutoriais convencionais, mostrando o* `OpenMP` *do ponto de vista do código gerado com a expansão das diretivas e suas relações com conceitos de programação paralela.*

3.1. Introdução

O `OpenMP` [Dagum and Menon 1998] [OpenMP-ARB 2015] [OpenMP Site 2017] é um padrão bem conhecido e amplamente utilizado no desenvolvimento de aplicações parale-

las para plataformas *multicore* e *manycores*. E desde a versão 4.0 da sua especificação [OpenMP-ARB 2013] define o suporte a dispositivos aceleradores.

O uso de *diretivas de compilação* é uma das abordagens para paralelização de código que tem se destacado no contexto de Computação Paralela, pois anotar código é usualmente mais fácil do que reescrevê-lo.

As diretivas de compilação são como anotações que fornecem dicas sobre o código original e guiam o compilador no processo de paralelização das regiões anotadas. Estas diretivas são comumente implementadas usando-se as diretivas de pré-processamento `#pragma`, em `C/C++`, e sentinelas `!$`, no `Fortran`.

Durante o pré-processamento e a compilação, se o código anotado é analisado por um compilador com suporte ao `OpenMP` as diretivas tem seus construtores substituídos por um formato de código específico. O código gerado dessa expansão das diretivas é composto por chamadas às funções do *runtime* do `OpenMP` [Dagum and Menon 1998] [OpenMP-ARB 2015] [OpenMP Site 2017].

O `OpenMP` trabalha em sistemas com memória compartilhada e implementa o modelo `fork-join`, no qual múltiplas *threads* são criadas em regiões paralelas e executam tarefas definidas implicitamente ou explicitamente usando-se as diretivas do `OpenMP` [OpenMP-ARB 2011] [OpenMP-ARB 2013] [OpenMP-ARB 2015]. A Figura 3.1 apresenta a ideia do modelo `fork-join` implementado com regiões paralelas.

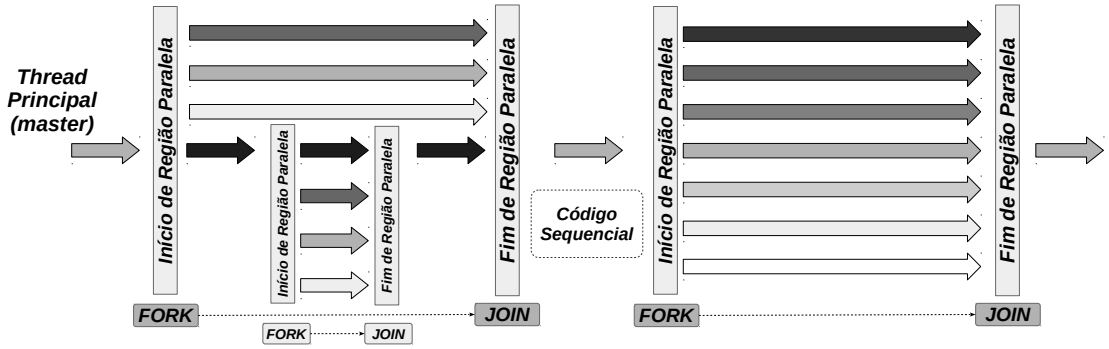

Figura 3.1. Modelo `Fork-join` baseado em regiões paralelas

A partir dos blocos anotados como regiões paralelas, o *runtime* cria implicitamente um time de *threads* que executarão o código da região paralela. A *thread* principal fará parte do time de *threads* e no final da região paralela, quando as outras *threads* forem destruídas, ela seguirá em execução.

Além da criação implícita de *threads*, é possível atribuir a alguma dessas *threads* a execução de tarefas explícitas definidas pelo programador usando a diretiva *task*.

O `OpenMP` traz aos usuários a possibilidade de aplicar conceitos de paralelismo em alto nível por meio das diretivas de compilação que permitem anotações no código. Além disso, em grande parte os tutoriais apresentam a ideia do uso de diretivas de compilação para gerar código para aplicações paralelas como algo extremamente trivial.

Utilizar `OpenMP` não é apenas colocar as diretivas no código e esperar que ma-

gicamente o *runtime* paralelize o código, é necessário que o código seja paralelizável e sabermos quais transformações serão aplicadas ao código para gerar a versão paralela. Antes de tudo é importante conhecer os conceitos sobre paralelismo, concorrência, sincronização e sobre como o código irá ser executado [Gonçalves et al. 2016].

Grande parte das linguagens de programação apresentam mecanismos para a criação de *threads* (`spawn`) e para a sincronização do trabalho entre as tarefas (`join, sync`) em comandos nativos da linguagem ou por extensões, como as diretivas de compilação fornecidas pelo `OpenMP`. Alguns desses conceitos comuns são apresentados ou implementados de maneiras diferentes, mas a ideia e o significado permanecem os mesmo.

As implementações de bibliotecas ou extensão para paralelismo podem ser de mais baixo nível ou de mais alto nível. A biblioteca `pthread` [Nichols et al. 1996] disponível em distribuições do GNU/Linux, fornece ao programador funções para a criação de *threads* (`pthread_create(...)`) e para a sincronização do trabalho compartilhado entre as *threads* (`pthread_join()`).

O `Cilk` [Blumofe et al. 1995] fornece também funções com as atribuições de criar *threads* (`cilk_spawn`) e de sincronização (`cilk_sync`), e também tem suporte à paralelização de laços com `cilk_for`.

A abordagem proposta pelo `OpenMP` é ser de mais alto nível, que o código seja anotado e não reescrito. Porém em muitos casos não se trata de somente uma forma simples de anotar o código, é necessário programar via diretivas de compilação, o que torna-se complexo.

A motivação inicial desse estudo foi a necessidade de interceptar código de aplicações `OpenMP` para fazer *offloading* de código para aceleradores. Nosso estudo foi baseado nas diretivas de compilação da biblioteca `libgomp` do `GCC` [GNU Libgomp 2016]. Outras aplicações como bibliotecas para a construção de *traces* e *logs* são possíveis via interceptação de chamadas de funções *(hooking)* das aplicações. O código que trata as chamadas de funções, mensagens ou intercepta eventos, fazendo uma espécie de *proxy* é chamado de *hook* (gancho).

O restante do texto está organizado da seguinte forma. Na Seção 3.2 apresentamos o padrão `OpenMP` e algumas de suas implementações. Algumas ds principais diretivas de compilação e o estudo sobre a expansão de código dessas diretivas está na Seção 3.3. A Seção 3.4 trata sobre possíveis aplicações e criação de uma biblioteca de interceptação de código. Por fim serão apresentadas as considerações finais na Seção 3.5.

3.2. Implementações do `OpenMP`

O `OpenMP` [OpenMP-ARB 2011] [OpenMP-ARB 2013] [OpenMP-ARB 2015] implementa o modelo `fork-join`, no qual a *master thread* executa sequencialmente até encontrar uma região paralela. Nesta região executa uma operação similar a um `fork` e cria um time de *threads* que irão executar na região paralela (regiões paralelas aninhadas são permitidas). Quando todas as *threads* alcançam a barreira no final da região paralela, o time de *threads* é destruído e a *master thread* continua sozinha a execução até encontrar uma nova região paralela.

O padrão OpenMP é suportado por praticamente todos os compiladores atuais. Compiladores como GCC [GCC 2015] [GNU Libgomp 2015a], Intel icc [Intel 2016b] e LLVM clang [Lattner and Adve 2004] [LLVM OpenMP 2015] tem implementações do OpenMP [OpenMP 2017]. Entre as implementações que são bem conhecidas atualmente estão a GNU GCC libgomp [GNU Libgomp 2016] e a Intel OpenMP* Runtime Library (libomp) [Intel 2016a] que trabalham com os compiladores GCC e clang.

A especificação do OpenMP foi expandida para dar suporte a *offloading* de código para dispositivos aceleradores, o que é um tópico de grande importância considerando que as plataformas tornam-se cada vez mais heterogêneas. A biblioteca libgomp [GNU Libgomp 2015a] [GNU Libgomp 2015b] [GNU Libgomp 2016] teve seu nome trocado recentemente de GNU OpenMP Runtime Library para GNU Offloading and Multi Processing Runtime Library sendo capaz de fazer offloading de código usando o padrão OpenACC [OpenACC 2015] [OpenACC 2017].

3.3. Expansão das Diretivas de Compilação

As *diretivas de compilação* são formadas por construtores e cláusulas que são substituídos por uma versão de código expandido durante a fase de pré-processamento. Nesta seção serão apresentados algumas das diretivas e os respectivos formatos de código pós expansão. O formato de código é estruturado e é composto de chamadas às funções do *runtime* do OpenMP.

Verificamos o formato de código gerado pelo GCC com a libgomp para os construtores de regiões paralelas *(parallel region)*, compartilhamento de trabalho em laços *(for)*, construtores para a declaração de tarefas explícitas *(task)* e algumas combinações com outros recursos, como *taskloop* para tarefas com compartilhamento de trabalho de laços e suporte a laços e tarefas com vetorização *(simd)*. Além disso apresentamos exemplos com o construtor target para *offloading* para dispositivos aceleradores.

3.3.1. Regiões paralelas: construtor `parallel`

Esta é uma das mais importantes diretivas, pois ela é responsável pela demarcação de regiões paralelas, indicando a região de código que será executada em paralelo. Se esse construtor não for especificado o programa será executado de forma sequencial. Regiões paralelas são criadas em OpenMP usando-se o construtor #pragma omp parallel.

Quando uma região paralela é encontrada pela *thread* principal, é criado um time de *threads* que irão executar o código da região paralela. A *thread* principal torna-se a *thread* mestre desse grupo. Porém, esse construtor não divide o trabalho entre as *threads*, apenas cria a região paralela e o grupo de *threads*. O formato de código para o construtor de região paralela é mostrado no Código 3.1.

Código 3.1. Formato do construtor *parallel*

```
1 #pragma omp parallel [clause[ [,] clause] ... ] new-line
2 {
3   /* Bloco estruturado. */
4 }
```

A diretiva *parallel* é implementada com a criação de uma nova função *(outlined function)* usando o código contido no bloco estruturado delimitado pelo construtor. A

libgomp [GNU Libgomp 2015a] [GNU Libgomp 2015b] usa funções para delimitar a região de código. As duas funções relacionadas com a construção do formato de regiões paralelas na ABI da libgomp estão listadas no Quadro 3.1.

Quadro 3.1: ABI libgomp – Funções relacionadas com a diretiva *parallel*

```
void GOMP_parallel_start(void (*fn)(void *), void *data, unsigned num_threads)
void GOMP_parallel_end(void)
```

De acordo com a documentação da libgomp [GNU Libgomp 2015a], o código gerado pós expansão assume o formato apresentado no Código 3.2. São inseridas chamadas às funções do *runtime* do OpenMP que demarcam o início e o fim da região paralela, entre essas chamadas a *thread* principal faz uma chamada à função que implementa o código extraído da região paralela.

Código 3.2. Formato de Código Expandido para o construtor *parallel*

```
1  /* Uma nova função é criada. */
2  void subfunction (void *data){
3    use data;
4    body;
5  }
6
7  /* A diretiva é substituída por chamadas ao runtime para criar a região paralela */
8  setup data;
9
10 GOMP_parallel_start(subfunction, &data, num threads);
11 subfunction(&data);
12 GOMP_parallel_end();
```

O Código 3.3 mostra o formato do código expandido gerado pelo GCC para a diretiva *parallel*. O código está escrito em GIMPLE, a representação intermediária utilizada pelo GCC.

Código 3.3. Código expandido gerado pelo GCC para *parallel*

```
1  /* Uma nova função é criada. */
2  main._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i) {
3    return;
4  }
5
6  main () {
7    int i;
8    int D.1804;
9    struct .omp_data_s.0 .omp_data_o.1;
10
11 <bb 2>:
12   .omp_data_o.1.i = i;
13   __builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 0);
14   main._omp_fn.0 (&.omp_data_o.1);
15   __builtin_GOMP_parallel_end ();
16   i = .omp_data_o.1.i;
17   D.1804 = 0;
18
19 <L0>:
20   return D.1804;
21 }
```

Uma estrutura omp_data é declarada para passar argumentos para a função que irá executar o código da região paralela. A *thread* principal fará uma chamada à função GOMP_parallel_start(...) passando como parâmetro o ponteiro da função ex-

traída (main._omp_fn.0) para a criação das *threads* pelo *runtime* do `OpenMP` e também fará uma chamada para a mesma função garantindo sua participação no time de *threads*. Todas as *threads* que terminarem a execução ficarão aguardando na barreira declarada implicitamente no fim da região paralela.

O construtor `parallel` apresenta algumas cláusulas para a definição do número de *threads* a serem criadas no time `(num_threads)`, para um teste condicional se a região paralela deve ou não ser criada `(if)`, e para definições de compartilhamento de dados `(shared e private)`. Pode ser utilizado em conjunto com outros contrutores como `single` e `master` para as situações nas quais seja necessário especificar qual das *threads* deve executar partes do código de uma região paralela. Outros construtores para sincronização entre as *threads* com uma barreira explícita como o `barrier` e para evitar condições de corrida em regiões críticas `(critical)`. O Código 3.4 apresenta um exemplo do uso de algumas cláusulas e desses construtores de compartilhamento de trabalho e sincronização.

Código 3.4. Exemplo de código com o construtor `parallel` e algumas cláusulas

```
1  int main(int argc, char *argv[]) {
2    int n = atoi(argv[1]);
3    int id, valor = 0;
4    printf("Thread[%d][%lu]: Antes da Região Paralela.\n", omp_get_thread_num(), (long int
         ) pthread_self());
5
6    #pragma omp parallel if(n>1024) num_threads(4) default(none) shared(valor) private(id)
7    {
8      id = omp_get_thread_num();
9      long int id_sys = (long int) pthread_self();
10     printf("Thread[%d][%lu]: Código Executado por todas as threads.\n", id, id_sys);
11
12     #pragma omp master
13     {
14       printf("Thread[%d][%lu]: Código Executado pela thread master.\n", id, (long int)
             pthread_self());
15     }
16
17     #pragma omp single
18     {
19       printf("Thread[%d][%lu]: Código Executado por uma das threads.\n", id, (long int)
             pthread_self());
20     }
21
22     if(omp_get_thread_num() == 3){
23       printf("Thread[%d][%lu]: Código Executado pela thread de id: 3.\n", id, (long int)
              pthread_self());
24     }
25
26     #pragma omp critical
27     {
28       printf("Thread[%d][%lu]: Executando a região crítica.\n", id, (long int)
             pthread_self());
29       printf("Thread[%d][%lu]: Antes... valor: %d\n", id, (long int) pthread_self(),
             valor);
30       valor = valor + id;
31       printf("Thread[%d][%lu]: Depois.. valor: %d\n", id, (long int) pthread_self(),
             valor);
32     }
33
34     printf("Thread[%d][%lu]: Barreira.\n", id, (long int) pthread_self());
35
36     #pragma omp barrier
37
38     printf("Thread[%d][%lu]: Depois da barreira.\n", id, (long int) pthread_self());
39   }
```

```
40
41    printf("Thread[%d][%lu]: Depois da Região Paralela.\n", omp_get_thread_num(), (long
          int) pthread_self());
42
43    return 0;
44 }
```

A saída da execução do Código 3.4 é apresentada no Terminal 3.1.

Terminal 3.1

```
rogerio@chamonix:/src/example-parallel-with-clauses$ ./example-parallel-with-clauses.
    exe 4096
Thread[0][18446744072495294336]: Antes da Região Paralela.
Thread[0][18446744072495294336]: Código Executado por todas as threads.
Thread[0][18446744072495294336]: Código Executado pela thread master.
Thread[0][18446744072495294336]: Código Executado por uma das threads.
Thread[1][18446744072482789120]: Código Executado por todas as threads.
Thread[3][18446744072466003712]: Código Executado por todas as threads.
Thread[2][18446744072474396416]: Código Executado por todas as threads.
Thread[3][18446744072466003712]: Código Executado pela thread de id: 3.
Thread[0][18446744072495294336]: Executando a região crítica.
Thread[0][18446744072495294336]: Antes... valor: 0
Thread[0][18446744072495294336]: Depois.. valor: 0
Thread[0][18446744072495294336]: Barreira.
Thread[2][18446744072474396416]: Executando a região crítica.
Thread[2][18446744072474396416]: Antes... valor: 0
Thread[2][18446744072474396416]: Depois.. valor: 2
Thread[2][18446744072474396416]: Barreira.
Thread[1][18446744072482789120]: Executando a região crítica.
Thread[1][18446744072482789120]: Antes... valor: 2
Thread[1][18446744072482789120]: Depois.. valor: 3
Thread[1][18446744072482789120]: Barreira.
Thread[3][18446744072466003712]: Executando a região crítica.
Thread[3][18446744072466003712]: Antes... valor: 3
Thread[3][18446744072466003712]: Depois.. valor: 6
Thread[3][18446744072466003712]: Barreira.
Thread[0][18446744072495294336]: Depois da barreira.
Thread[2][18446744072474396416]: Depois da barreira.
Thread[3][18446744072466003712]: Depois da barreira.
Thread[1][18446744072482789120]: Depois da barreira.
Thread[0][18446744072495294336]: Depois da Região Paralela.
rogerio@chamonix:/src/example-parallel-with-clauses$
```

3.3.2. Loops: construtor `for`

Um time de *threads* é criado quando uma região paralela é alcançada, porém com apenas o construtor de região paralela todas as *threads* irão executar o mesmo código que compõe o corpo da *outlined function*.

O construtor `for` é usado para distribuir o trabalho e coordenar a execução paralela entre as *threads* do time. Especifica que as iterações de um ou mais laços irão ser executadas em paralelo pelas *threads* no contexto de tarefas implícitas. As iterações são distribuídas entre as *threads* que estão em execução dentro da região paralela.

Os construtores para a especificação de região paralela e de laços podem ser utilizados separadamente ou combinados. O Código 3.5 mostra uma região paralela com um construtor *for*, o que é equivalente ao uso dos construtores em modo combinado apresentado no Código 3.6.

Código 3.5. Construtor *for* dentro de uma região paralela

```
1 #pragma omp parallel
2 {
3   #pragma omp for
4   for (i = lb; i <= ub; i++){
5     body;
6   }
7 }
```

Código 3.6. Construtores *parallel* e *for* combinados

```
1   #pragma omp parallel for
2   for (i = lb; i <= ub; i++){
3     body;
4   }
```

Semelhante ao que ocorre no processamento do construtor *parallel* individualmente, um construtor `parallel` com um construtor `for` ou o formato combinado deles `parallel for` também é implementado com a criação de uma nova função *(outlined function)*.

O código expandido que substitui a declaração do construtor `parallel` e do *for* associado ao laço paralelo é composto pelas chamadas para criar e finalizar a região paralela que são feitas ao *runtime* do `OpenMP` e pela *outlined function*. O que muda é que neste caso o corpo da nova função terá o código para controlar a distribuição das iterações do laço em *(chunks)* que são executados pelas *threads*.

O trabalho é dividido entre as *threads* de acordo com o algoritmo de escalonamento de iterações adotado. O escalonamento é definido usando-se a cláusula `schedule` e os tipos que estão disponíveis no `OpenMP` são: *static, auto, runtime, dynamic e guided*.

O Figura 3.2 apresenta como os *chunks* de iterações de um laço são executados pelas *threads*.

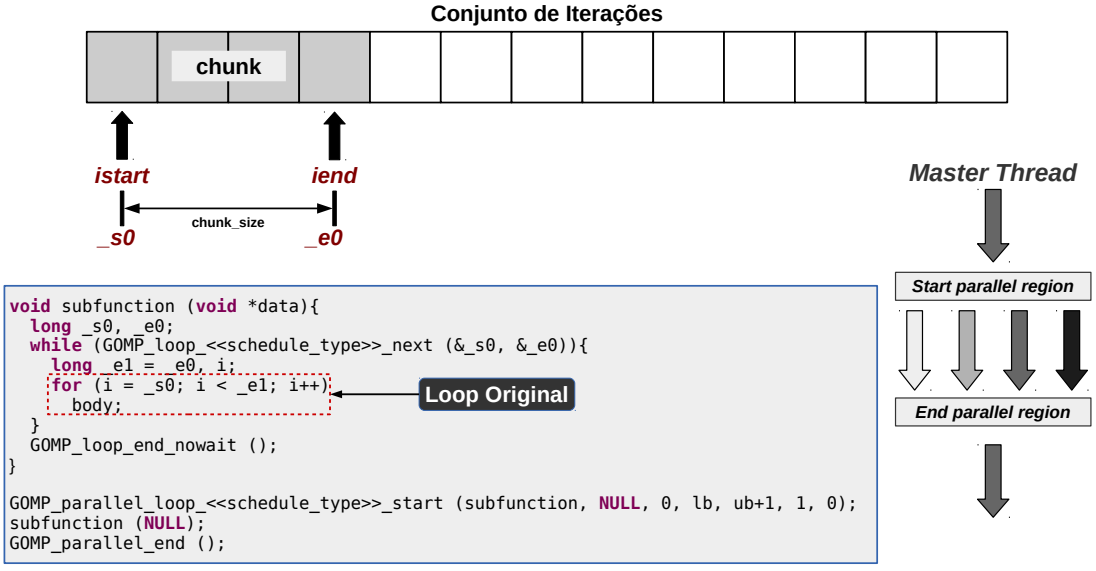

Figura 3.2. Esquema de execução das iterações do laço

O algoritmo de escalonamento define como as iterações do laço associado serão divididas em subconjuntos contíguos e não vazios, chamados de *chunks*, e como são dis-

tribuídos entre as *threads* pertencentes à região paralela [OpenMP-ARB 2015]. Os tipos de escalonamento de iterações de laços que podem ser utilizados no `OpenMP`:

1. Estático – `schedule(static, chunk_size)`: Baseia-se na ideia de que cada *thread* irá executar a mesma quantidade de iterações, se um *chunk_size* não for especificado irá dividir o número de iterações pelo número de *threads* formando *chunks* de tamanhos iguais e pelo menos um *chunk* é distribuído para cada *thread*, caso contrário seguirá no esquema *round-robin* pela ordem dos *ids* das *threads*, atribuindo *chunks* para cada uma delas até que todo o conjunto de iterações tenha sido executado.

2. Dinâmico - `schedule(dynamic, chunk_size)`: As iterações são distribuídas para as *threads* do time em *chunks*, conforme as *threads* requisitam mais trabalho. Cada *thread* executa um *chunk* de iterações e então requisita outro *chunk* até que não restem mais *chunks* para serem distribuídos. Cada *chunk* contém *chunk_size* iterações, exceto o último *chunk* a ser distribuído que pode ter um número menor de iterações. Quando a variável *chunk_size* não está definida, o valor padrão é 1.

3. Guiado – `schedule(guided, chunk_size)`: As iterações são atribuídas para as *threads* do time em *chunks* também conforme as *threads* requisitam mais trabalho. Cada *thread* executa um *chunk* de iterações e então requisita outro, até que não existam mais *chunks* a serem atribuídos. Para um *chunk_size* especificado como 1, o tamanho de cada *chunk* é proporcional ao número de iterações não atribuídas dividido pelo número de *threads* no time, decrescendo até 1. Para um *chunk_size* com um valor *k* (maior que 1), o tamanho de cada *chunk* é determinado da mesma forma, com a restrição de que os *chunks* não contenham menos que *k* iterações, exceto o último.

4. Auto – `schedule(auto)`: A decisão do escalonamento é delegada para o compilador ou para o *runtime*.

5. Runtime – `schedule(runtime)`: A decisão do escalonamento é adiada até o momento de execução, só é conhecida em tempo de execução. Tanto o *schedule* quando o *chunk_size* são obtidos do `run-sched-var ICV`. Se o `ICV` é definido para *auto*, o escalonamento é definido pela implementação. Quando o tipo especificado for *runtime* ou *auto* o valor de *chunk_size* não deve ser definido.

Quando não é especificado qual algoritmo de escalonamento a ser utilizado pelo *runtime* ou ele é do tipo *auto*, o `GCC` gera o código usando as funções da `libgomp` para o formato de escalonamento *static*, que por padrão faz uma divisão estática das iterações do laço pelo número de *threads*. O Código 3.7 apresenta um laço que terá suas iterações distribuídas entre as *threads* estaticamente.

Código 3.7. Laço sem escalonamento definido

```
1 int main() {
2   int id, i;
3
4   printf("Thread[%d][%lu]: Antes da Região Paralela.\n", omp_get_thread_num(), (long int
        ) pthread_self());
```

```
5
6   #pragma omp parallel num_threads(4) default(none) private(id)
7   {
8     // Todas as threads executam esse código.
9     id = omp_get_thread_num();
10
11    #pragma omp for
12    for(i=0; i<16; i++){
13      printf("Thread[%d][%lu]: Trabalhando na iteração %lu.\n", id, (long int)
            pthread_self(), i);
14    }
15  }
16  printf("Thread[%d][%lu]: Depois da Região Paralela.\n", omp_get_thread_num(), (long
        int) pthread_self());
17
18  return 0;
19 }
```

A saída produzida pela execução do Código 3.7 é apresentada no Terminal 3.2.

Terminal 3.2

```
rogerio@chamonix:/src/example-for$ ./example-for-constructor-static.exe
Thread[0][1476638592]: Antes da Região Paralela.
Thread[1][1464133376]: Trabalhando na iteração 4.
Thread[1][1464133376]: Trabalhando na iteração 5.
Thread[1][1464133376]: Trabalhando na iteração 6.
Thread[1][1464133376]: Trabalhando na iteração 7.
Thread[0][1476638592]: Trabalhando na iteração 0.
Thread[0][1476638592]: Trabalhando na iteração 1.
Thread[0][1476638592]: Trabalhando na iteração 2.
Thread[0][1476638592]: Trabalhando na iteração 3.
Thread[3][1447347968]: Trabalhando na iteração 12.
Thread[3][1447347968]: Trabalhando na iteração 13.
Thread[3][1447347968]: Trabalhando na iteração 14.
Thread[3][1447347968]: Trabalhando na iteração 15.
Thread[2][1455740672]: Trabalhando na iteração 8.
Thread[2][1455740672]: Trabalhando na iteração 9.
Thread[2][1455740672]: Trabalhando na iteração 10.
Thread[2][1455740672]: Trabalhando na iteração 11.
Thread[0][1476638592]: Depois da Região Paralela.
rogerio@chamonix:/src/example-for$
```

O Código 3.8 apresenta um laço anotado com o construtor `for` e com a cláusula `schedule(dynamic)`. Nesse tipo de escalonamento as *threads* ficam solicitando mais trabalho para o *runtime* até que todas as iterações tenham sido executadas. Desta maneira a execução depende de quais *threads* ficaram disponíveis, podendo uma *thread* receber mais *chunks* de iterações que outras.

Código 3.8. Laço com `schedule(dynamic)`

```
1  int main() {
2    int id, i;
3
4    printf("Thread[%d][%lu]: Antes da Região Paralela.\n", omp_get_thread_num(), (long int
         ) pthread_self());
5
6    #pragma omp parallel num_threads(4) default(none) private(id)
7    {
8      // All threads executes this code.
9      id = omp_get_thread_num();
10
11     #pragma omp for schedule(dynamic,2)
12     for(i=0; i<16; i++){
13       printf("Thread[%d][%lu]: Trabalhando na iteração %lu.\n", id, (long int)
             pthread_self(), i);
```

```
14      }
15    }
16    printf("Thread[%d][%lu]: Depois da Região Paralela.\n", omp_get_thread_num(), (long
          int) pthread_self());
17
18    return 0;
19  }
```

A saída produzida pela execução do Código 3.8 é apresentada no Terminal 3.3.

Terminal 3.3

```
rogerio@chamonix:/src/example-for$ ./example-for-constructor-dynamic.exe
Thread[0][18446744073366411136]: Antes da Região Paralela.
Thread[0][18446744073366411136]: Trabalhando na iteração 0.
Thread[0][18446744073366411136]: Trabalhando na iteração 1.
Thread[0][18446744073366411136]: Trabalhando na iteração 8.
Thread[0][18446744073366411136]: Trabalhando na iteração 9.
Thread[0][18446744073366411136]: Trabalhando na iteração 10.
Thread[0][18446744073366411136]: Trabalhando na iteração 11.
Thread[2][18446744073345513216]: Trabalhando na iteração 4.
Thread[2][18446744073345513216]: Trabalhando na iteração 5.
Thread[2][18446744073345513216]: Trabalhando na iteração 14.
Thread[2][18446744073345513216]: Trabalhando na iteração 15.
Thread[0][18446744073366411136]: Trabalhando na iteração 12.
Thread[0][18446744073366411136]: Trabalhando na iteração 13.
Thread[1][18446744073353905920]: Trabalhando na iteração 2.
Thread[1][18446744073353905920]: Trabalhando na iteração 3.
Thread[3][18446744073337120512]: Trabalhando na iteração 6.
Thread[3][18446744073337120512]: Trabalhando na iteração 7.
Thread[0][18446744073366411136]: Depois da Região Paralela.
rogerio@chamonix:/src/example-for$
```

Quando os tipos de escalonamento *runtime, dynamic* ou *guided* são usados, o formato do código gerado é o mesmo, mas ainda apresentam dois formatos distintos dependendo de como estão definidos o limite superior do laço e o *chunk_size*. Se o código utiliza valores numéricos para essas definições o código gerado é de um formato. Caso contrário, se as definições são feitas com base em variáveis ou expressões que precisam ser avaliadas, então o formato de código é outro.

Código 3.9. Laço com limite superior usando valor

```
1 #pragma omp parallel for schedule(<<
      schedule_type>>)
2 for (i = 0; i < 1024; i++){
3   // body.
4 }
```

Código 3.10. Laço com limite superior usando variável

```
1 n = 1024;
2 #pragma omp parallel for schedule(<<
      schedule_type>>)
3 for (i = 0; i < n; i++){
4   // body.
5 }
```

Foram identificados dois formatos de código para a execução de laços, como em cada formato a estrutura é a mesma para os tipos de escalonamentos *dynamic, runtime* ou *guided*, estão representados nos códigos pela marcação «`schedule_type`». Desta forma, o `GCC` e a biblioteca `libgomp` usam as funções listadas no Quadro 3.2 para delimitar a região paralela e criar o *primeiro formato* de laço.

Quadro 3.2: `libgomp` ABI – Funções usadas para *parallel for* no primeiro formato

```
void GOMP_parallel_loop_<<schedule_type>>_start (void (*fn) (void *), void *data,
    unsigned num_threads, long start, long end, long incr);
void GOMP_parallel_end (void);
bool GOMP_loop_<<schedule_type>>_next (long *istart, long *iend);
void GOMP_loop_end_nowait (void);
```

Como no *segundo formato* é necessário avaliar a expressão ou variável que define o valor assumido pelo limite superior do laço ou do *chunk_size*, somente é criada a região paralela e a inicialização do laço é feita dentro da função criada para tratar o laço. As funções utilizadas no segundo formato são apresentadas no Quadro 3.3.

Quadro 3.3: `libgomp` ABI – Funções usadas para *parallel for* no segundo formato

```
void GOMP_parallel_start (void (*fn) (void *), void *data, unsigned num_threads);
void GOMP_parallel_end (void);
void GOMP_parallel_loop_<<schedule_type>>_start (void (*fn) (void *), void *data,
    unsigned num_threads, long start, long end, long incr);
bool GOMP_loop_<<schedule_type>>_next (long *istart, long *iend);
void GOMP_loop_end_nowait (void);
```

O código gerado para executar laços que se enquadram no *primeiro formato* é mostrado no Código 3.11.

Código 3.11. Código expandido para laços no primeiro formato

```
1  void subfunction (void *data){
2    long _s0, _e0;
3    while (GOMP_loop_<<schedule_type>>_next (&_s0, &_e0)){
4      long _e1 = _e0, i;
5      for (i = _s0; i < _e1; i++){
6        body;
7      }
8    }
9    GOMP_loop_end_nowait ();
10 }
11
12 /* O laço anotado é substituído. */
13 setup data;
14
15 GOMP_parallel_loop_<<schedule_type>>_start (subfunction, &data,
16 num_threads, start, end, incr, chunk_size,...);
17 subfunction (&data);
18 GOMP_parallel_end ();
```

O Código 3.12 apresenta a estrutura do código gerado para o *segundo formato*.

Código 3.12. Código expandido para laços no segundo formato

```
1  void subfunction (void *data){
2    long i, _s0, _e0;
3    if (GOMP_loop_<<schedule_type>>_start (0, n, 1, &_s0, &_e0)){
4      do {
5        long _e1 = _e0;
6        for (i = _s0; i < _e0; i++) {
7          body;
8        }
9      } while (GOMP_loop_<<schedule_type>>_next (&_s0, &_e0));
10   }
11   GOMP_loop_end ();
12 }
13
```

```
14 /* O laço anotado é substituído. */
15 setup data;
16
17 GOMP_parallel_start (subfunction, &data, num_threads);
18 subfunction (&data);
19 GOMP_parallel_end ();
```

O GCC utiliza o GIMPLE como formato de código intermediário, a visualização do código intermediário gerado para o *primeiro formato* é apresentada na Figura 3.3.

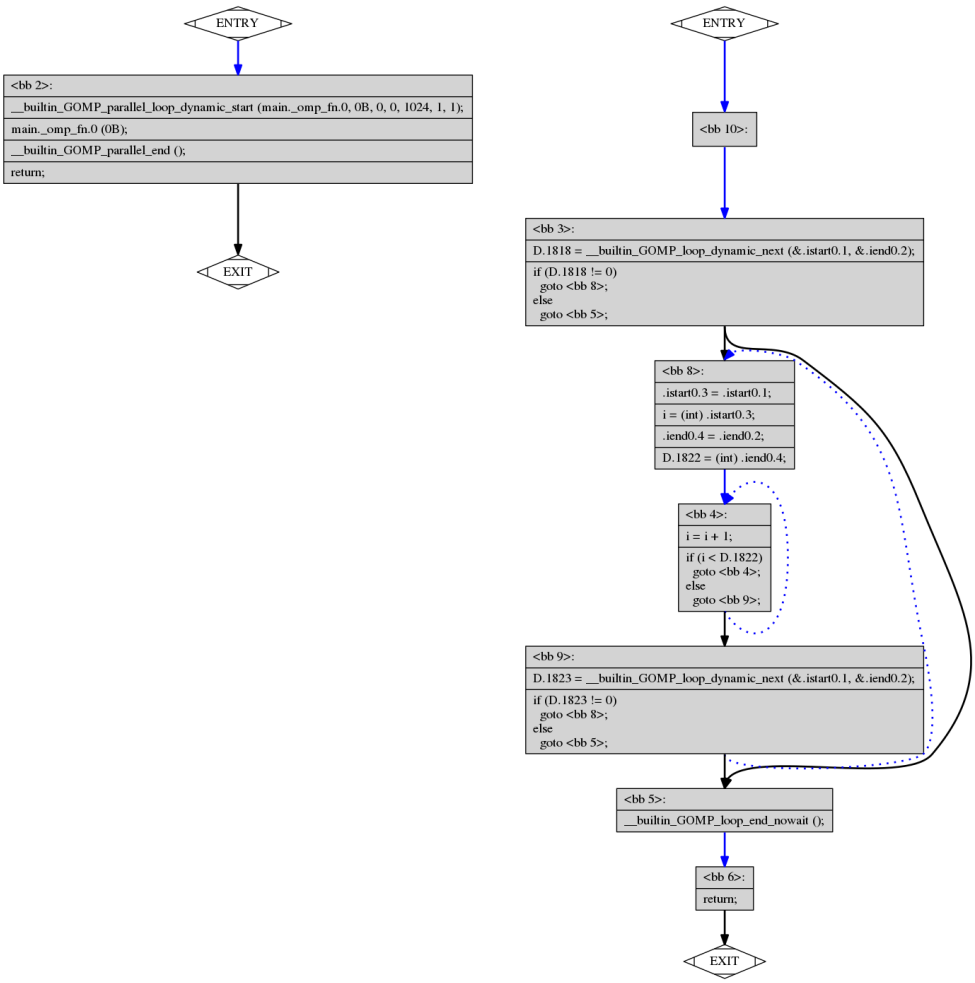

Figura 3.3. Visualização do primeiro formato laço utilizando `schedule(dynamic)`

No *primeiro formato* o início da região paralela é marcado com a chamada à função `GOMP_parallel_loop_«schedule_type»_start()`, que além de criar o time de *threads* também inicializa os controles da execução do laço. Dentro da *outlined function* a chamada à função `GOMP_loop_«schedule_type»_next(...)` é usada pela *thread* para recuperar o primeiro *chunk*. Cada *thread* executa este primeiro trabalho e depois entra em *loop* recuperando e executando novos *chunks* até que não tenha mais trabalho a ser feito. Quando as *threads* terminam a execução finalizam a execução do laço chamando `GOMP_loop_end_nowait()` e o compartilhamento de trabalho do laço é também finalizado. Então a região paralela é finalizada com a chamada

`GOMP_parallel_end()` que desaloca o time de *threads*.

A Figura 3.4 mostra a visualização do código para o *segundo formato*. A mesma semântica é aplicada ao *segundo formato*, mesmo que utilize diferentes funções. No *segundo formato* a chamada à função `GOMP_parallel_start(...)` inicia a região paralela e nesta chamada somente é criado o time de *threads*. A inicialização do compartilhamento de trabalho do laço é feito dentro da *outlined function* e a chamada `GOMP_loop_«schedule_type»_start(...)` é usada para recuperar o primeiro *chunk*. As *threads* que conseguem obter o seu primeiro *chunk* pode executá-lo e usam a função `GOMP_loop_«schedule_type»_next(...)` para recuperar os próximos *chunks* até terminarem as iterações do laço e então finalizarem com a chamada à `GOMP_loop_end_nowait()`. A região paralela também é finalizada usando a mesma chamada à função `GOMP_parallel_end()` que desaloca o time de *threads*.

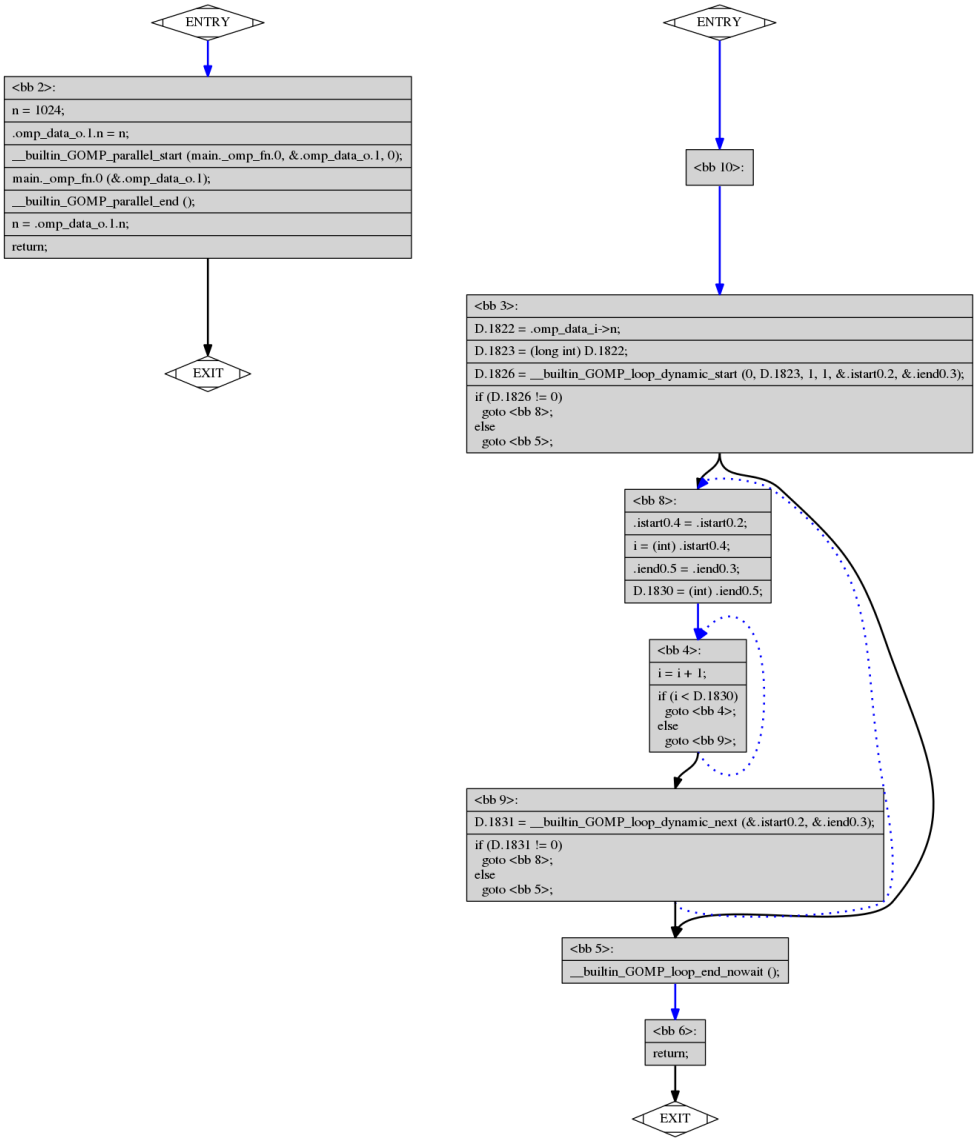

Figura 3.4. Visualização do segundo formato laço utilizando `schedule(dynamic)`

A Figura 3.5 resume os dois formatos de código que são gerados para laços.

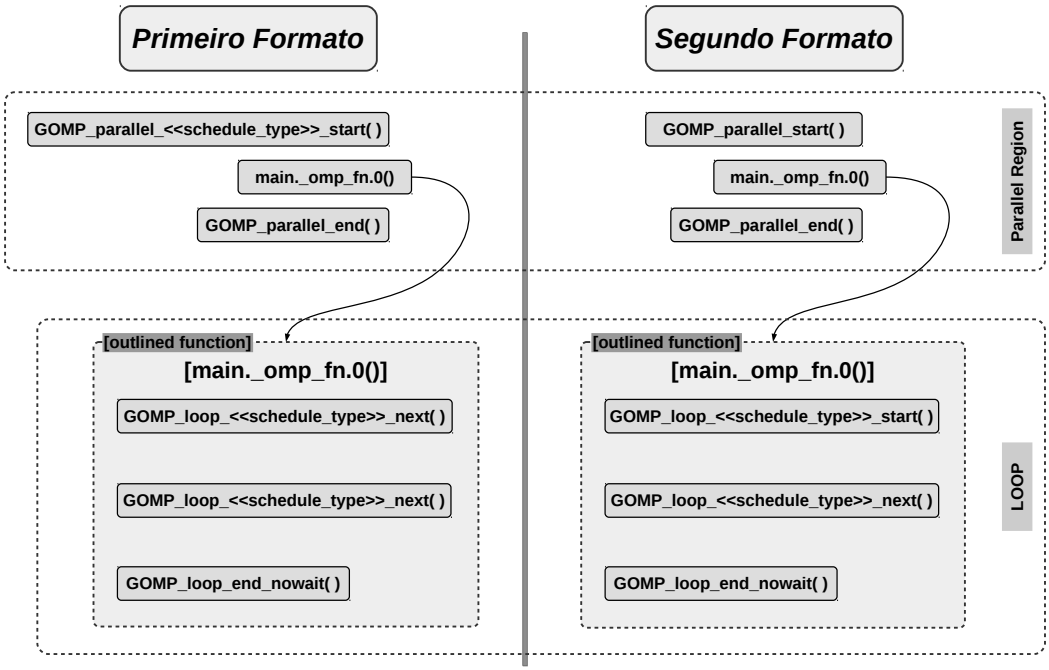

Figura 3.5. Comparativo dos dois formatos de laços

O Código 3.13 apresenta dois laços com diferentes escalonamentos e definições de limite superior e *chunk_size*.

Código 3.13. Código de uma região paralela com dois laços

```
1  num_t = 8;
2  #pragma omp parallel num_threads(num_t)
3  {
4    #pragma omp for schedule(runtime)
5    for (i = 0; i < 1024; i++){
6      body_1;
7    }
8    #pragma omp for schedule(dynamic, 32)
9    for (j = 0; j < n; j++){
10     body_2;
11   }
12 }
```

A Figura 3.6 mostra uma representação gráfica do código gerado para a região paralela com dois laços. No caso de códigos com múltiplos laços são geradas barreiras implíticas entre os laços no código final. Além as *threads* que terminarem seu trabalho antes das outras, aguardarão a conclusão na barreira implícita gerada pelo final da região paralela.

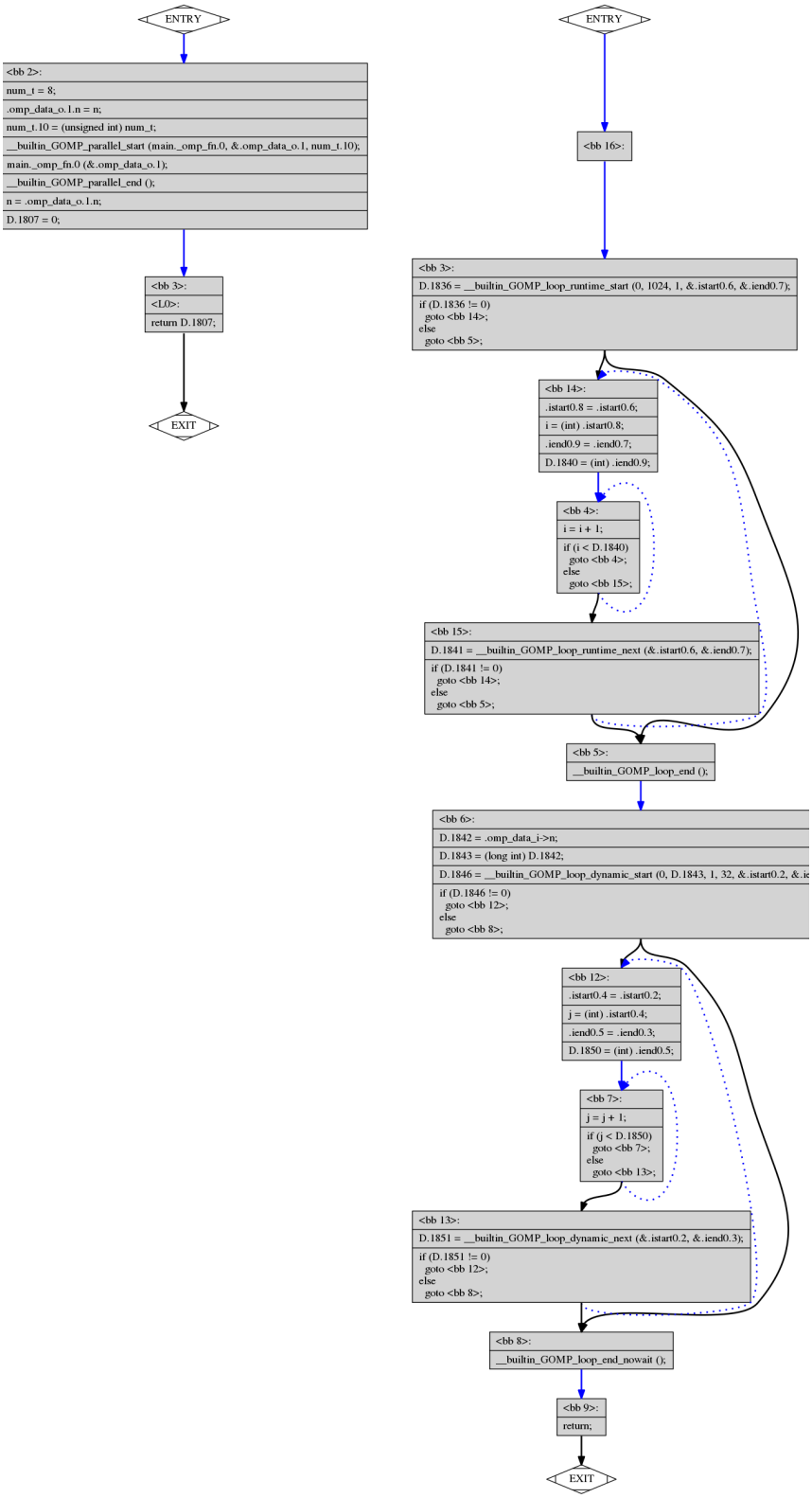

Figura 3.6. Representação gráfica de dois laços dentro de uma mesma região paralela

Podemos perceber que é seguido o mesmo processo, com o código da região paralela é criada uma nova função que agora terá o código dos dois laços, seguindo o formato

de laço. A Figura 3.7 mostra o código da função com dois laços separados por uma barreira que é gerada com a chamada à função `GOMP_loop_end()`.

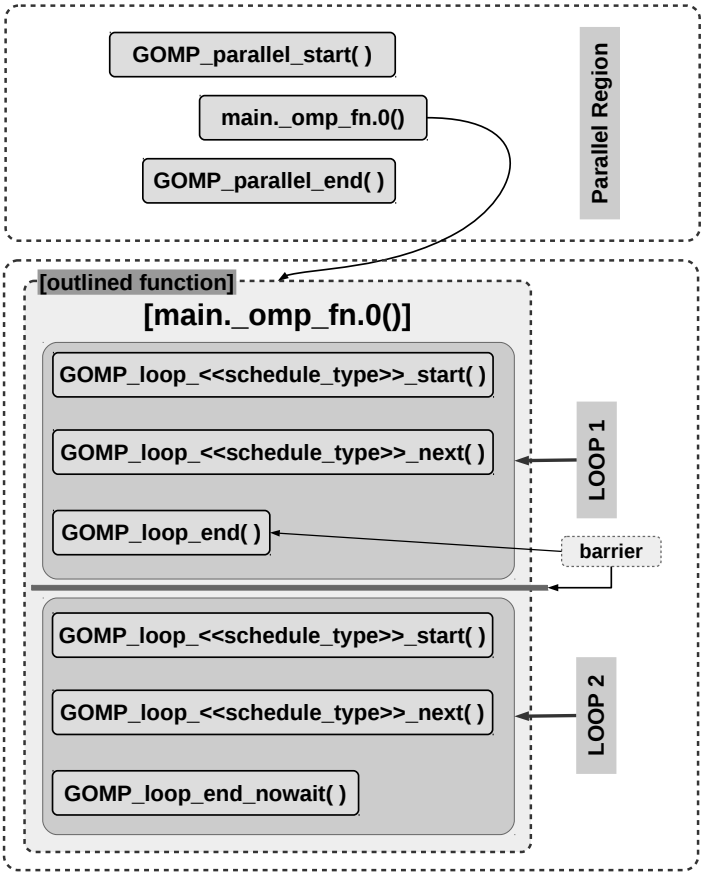

Figura 3.7. Formato para dois laços dentro de uma mesma região paralela

3.3.3. Seções: construtor `sections`

O *sections* é um construtor de compartilhamento de trabalho não iterativo que permite a definição de um conjunto de blocos estruturados utilizando a diretiva `#pragma omp sections` para indicar a criação de seções de código e a diretiva `#pragma omp section` para especificar cada bloco que será associado a uma seção. Os blocos são distribuídos para serem executados pelas *threads* do time criado pela região paralela, isto é, cada bloco é executado por uma das *threads* no contexto de uma tarefa implícita.

A sintaxe para uso dos construtores de seções é apresentado no Código 3.14. O conceito é que cada um dos blocos `(bloco_1, bloco_2 e bloco_3)` seja executado por alguma das *threads* do time criado pela região paralela.

Código 3.14. Uso dos construtores de seções

```
#pragma omp sections
{
  #pragma omp section
  bloco_1;
  #pragma omp section
  bloco_2;
  #pragma omp section
  bloco_3;
}
```

A estrutura de código que é gerada para a execução das seções é apresentada no Código 3.15. Este código estará dentro da função extraída para a execução da região paralela e todas as *threads* pertencentes ao time criado por essa região paralela irão executá-la. O bloco de seções é iniciado com a chamada `GOMP_sections_start(3)` (o argumento 3 indica o número de seções definidas no código) e as *threads* que atingirem o código do laço que itera sobre o conjunto de seções primeiro obterão uma das seções para executarem com a chamada à função `GOMP_sections_next()`, até que todas as seções definidas tenham sido executadas.

Código 3.15. Código dos construtores de seções expandido

```
for (i = GOMP_sections_start (3); i != 0; i = GOMP_sections_next ())
  switch (i) {
    case 1:
      bloco_1;
      break;
    case 2:
      bloco_2;
      break;
    case 3:
      bloco_3;
      break;
  }
  GOMP_barrier ();
```

O Código 3.16 apresenta um exemplo do uso de seções com a cláusula de redução `(reduction)`. Cada uma das *threads* irá trabalhar sobre o código de uma das seções produzindo um valor para sua cópia de *sum*. A cláusula indica que ao final da execução deve ser feita uma redução de soma `(reduction(+:sum))` nas cópias de *sum* que pertencem a cada uma das seções, gerando um único valor para *sum*.

Código 3.16. Exempo do uso dos construtores de seções

```
int main(int argc, char *argv[]) {
  int i, id;
  int sum = 0;

  fprintf(stdout, "Thread[%d][%lu]: Antes da Região Paralela.\n", omp_get_thread_num(),
      (long int)pthread_self());

  #pragma omp parallel num_threads(8) private(id)
  {
    id = omp_get_thread_num();
    #pragma omp sections reduction(+:sum)
    {
      #pragma omp section
      {
        fprintf(stdout, "   Thread[%lu,%lu]: Trabalhando na seção 1.\n", id, (long int)
            pthread_self());
        for(i=0; i<1024;i++){
          sum += i;
```

```
17          }
18        }
19
20        #pragma omp section
21        {
22          fprintf(stdout, "   Thread[%lu,%lu]: Trabalhando na seção 2.\n", id, (long int)
                pthread_self());
23          for(i=0; i<1024;i++){
24            sum += i;
25          }
26        }
27      }
28    }
29    fprintf(stdout, "Thread[%d][%lu]: Depois da Região Paralela.\n", omp_get_thread_num(),
          (long int)pthread_self());
30    fprintf(stdout, "Thread[%d][%lu]: sum: %d\n", omp_get_thread_num(), (long int)
          pthread_self(), sum);
31
32    return 0;
33  }
```

A execução das seções ocorre de maneira independente, cada uma das seções é atribuída a uma das *threads*. As *threads* que terminam a execução de sua parte do trabalho ficam aguardando em uma barreia implicíta adicionada no final do bloco de seções. Na saída da execução do Código 3.16 que é apresentada no Terminal 3.4 é possível visualizar que a seção 2 foi executada antes da seção 1.

Terminal 3.4

```
rogerio@chamonix:/src/example-sections-reduction$ ./example-sections-reduction.exe
Thread[0][18446744073314326400]: Antes da Região Paralela...
Thread[0,18446744073314326400]: Trabalhando na seção 2.
Thread[4,18446744073276643072]: Trabalhando na seção 1.
Thread[0][18446744073314326400]: Depois da Região Paralela.
Thread[0][18446744073314326400]: sum: 1047552
rogerio@chamonix:/src/example-sections-reduction$
```

O construtor `sections` foi a primeira forma de execução de blocos de código independentes e não iterativos, mesmo que a execução ainda ocorra dentro de uma região paralela com a criação de *threads* de maneira implícita.

A Figura 3.8 apresenta a visualização do código intermediário gerado para o Código 3.16.

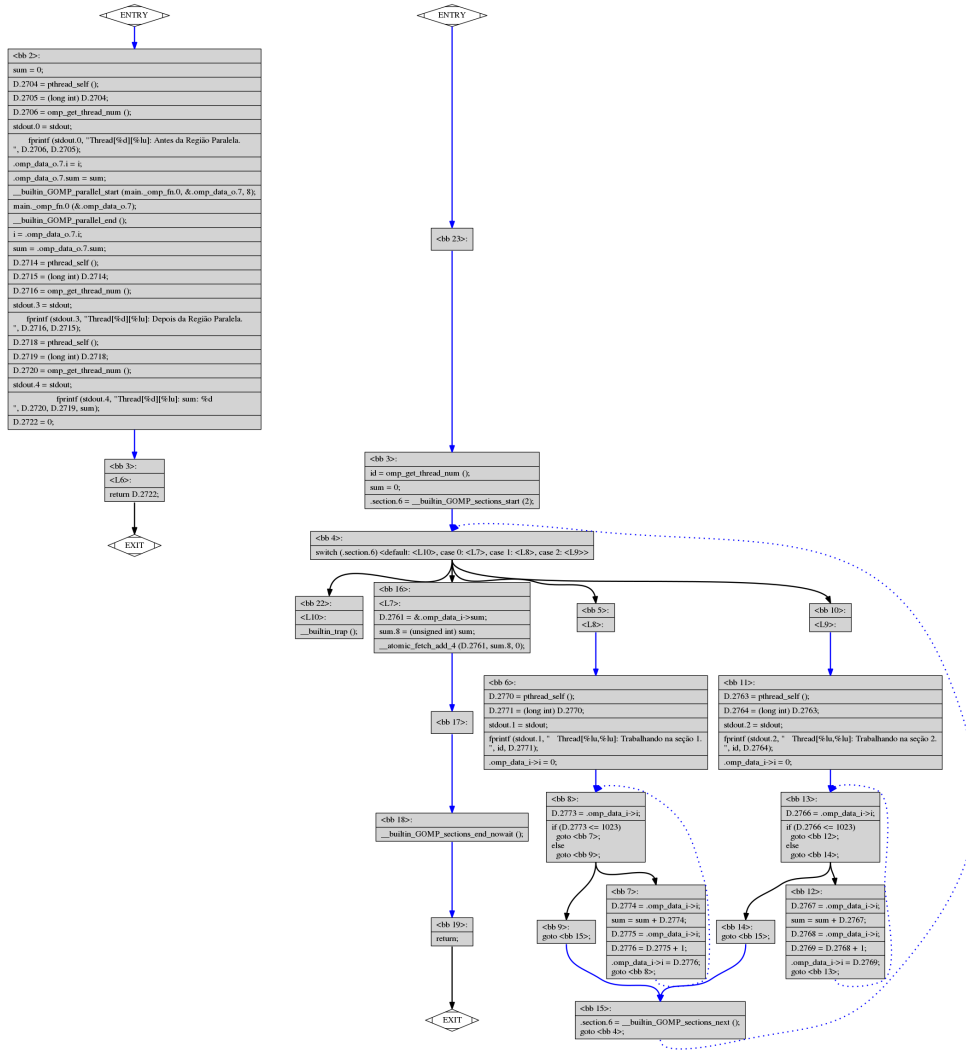

Figura 3.8. Código gerado para os construtores de seções

3.3.4. Tarefas: construtor `task`

O construtor *task* permite a criação de tarefas explícitas. O construtor `task` está disponível a partir das especificações 3.0 e 3.1 e é implementado pela `libgomp` do `GCC 4.4` e `GCC 4.7`, respectivamente. Quando uma *thread* encontra um construtor *task*, uma nova tarefa é gerada para executar o bloco associado ao construtor. A sintaxe do construtor `task` é apresentada no Código 3.17.

Código 3.17. Formato do construtor *task*

```
1 #pragma omp task [clause[ [,] clause] ... ] new-line
2 /* Bloco estruturado. */
```

O Código 3.18 apresenta como o construtor `task` é usado dentro de uma região paralela. Se for necessário criar apenas uma nova tarefa, o construtor `single` pode ser

utilizado para garantir esse comportamento, caso contrário todas a *threads* do time irão cria uma nova *task*.

Código 3.18. Formato do construtor *task*

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      /* Bloco estruturado. */
    }
  }
}
```

As funções da biblioteca `libgomp` que são utilizadas para gerar o código relacionado com o construtor *task* são listadas no Quadro 3.4.

Quadro 3.4: ABI `libgomp` – Funções usadas para a implementação do construtor *task*

```
void GOMP_parallel_start (void (*fn) (void *), void *data, unsigned num_threads);
void GOMP_parallel_end (void);
void GOMP_task (void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
  long arg_size, long arg_align, bool if_clause, unsigned flags,
void **depend);
void GOMP_taskwait (void);
```

O Código 3.19 apresenta um exemplo do uso da diretiva *task*. Neste exemplo três *tasks* são criadas dentro de uma região paralela. O construtor *single* é utilizado para garantir que o código seja executado apenas uma vez por uma das *threads* do time. Caso contário, as 8 *threads* criadas executariam o mesmo código criando cada uma delas três *tasks*.

Código 3.19. Exemplo de uso do construtor *task*

```
int main(int argc, char *argv[]) {
  int id = 0;
  int x = atoi(argv[1]);

  fprintf(stdout, "Thread[%lu,%lu]: Antes da região paralela.\n", omp_get_thread_num(),
      (long int) pthread_self());

  #pragma omp parallel num_threads(8) firstprivate(x) private(id)
  {
    id = omp_get_thread_num();
    fprintf(stdout, "  Thread[%lu,%lu]: Todas as threads executam.\n", id, (long int)
        pthread_self());

    #pragma omp single
    {
      fprintf(stdout, "  Thread[%lu,%lu]: Antes de criar tasks.\n", id, (long int)
          pthread_self());
      #pragma omp task if(x > 10)
      {
        fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na task 1.\n",
            omp_get_thread_num(), (long int) pthread_self());
      }

      #pragma omp task if(x > 20)
      {
        fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na task 2.\n",
            omp_get_thread_num(), (long int) pthread_self());
```

```
23        }
24
25        fprintf(stdout, "    Thread[%lu,%lu]: Antes do taskwait.\n", id, (long int)
              pthread_self());
26        #pragma omp taskwait
27        fprintf(stdout, "    Thread[%lu,%lu]: Depois do taskwait.\n", id, (long int)
              pthread_self());
28
29        #pragma omp task
30        {
31          fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na task 3.\n",
                omp_get_thread_num(), (long int) pthread_self());
32        }
33      }
34    }
35    fprintf(stdout, "Thread[%lu,%lu]: Depois da região paralela.\n", omp_get_thread_num(),
          (long int) pthread_self());
36
37    return 0;
38  }
```

Ainda no Código 3.19, pode ser visto o uso da cláusula `if` que também pode ser aplicada ao construtor `task` indicando uma condição para a criação da nova tarefa. No exemplo a `task 1` será criada somente se o valor da variável *x* recebido por parâmetro for maior que 10 e a `task 2` será criada se esse valor for maior que 20, já a `task 3` será criada sem nenhuma condição. A *thread* que entra no bloco do construtor `single` criará as duas primeiras *threads* e ficará aguardando o término da execução delas na diretiva `#pragma omp taskwait`. A saída da execução do Código 3.19 é apresentada no Terminal 3.5.

Terminal 3.5

```
rogerio@chamonix:/src/example-tasks$ ./example-tasks.exe 1024
Thread[0,140369357629312]: Antes da região paralela.
Thread[0,140369357629312]: Todas as threads executam.
Thread[0,140369357629312]: Antes de criar tasks.
Thread[7,140369294767872]: Todas as threads executam.
Thread[0,140369357629312]: Antes do taskwait.
Thread[3,140369328338688]: Todas as threads executam.
Thread[1,140369345124096]: Todas as threads executam.
Thread[6,140369303160576]: Todas as threads executam.
Thread[5,140369311553280]: Todas as threads executam.
Thread[4,140369319945984]: Todas as threads executam.
Thread[7,140369294767872]: Trabalhando na task 1.
Thread[0,140369357629312]: Trabalhando na task 2.
Thread[0,140369357629312]: Depois do taskwait.
Thread[5,140369311553280]: Trabalhando na task 3.
Thread[2,140369336731392]: Todas as threads executam.
Thread[0,140369357629312]: Depois da região paralela.
rogerio@chamonix:/src/example-tasks$
```

O formato de código gerado pelo `GCC` é apresentado no Código 3.20. Podemos perceber que são criadas novas funções para tratar a região paralela e o código das tarefas declaradas. Na função criada para tratar o código da região paralela as novas tarefas são criadas com as chamadas para a função `GOMP_task(...)`. Em cada uma das chamadas ao *runtime* do `OpenMP` é passado o ponteiro da função que deve ser executada pela nova *task*.

Código 3.20. Formato de código expandido para a diretiva *task*

```
main._omp_fn.3 (void * .omp_data_i)
{
  /* Declaração de variáveis suprimida. */

<bb 20>:

<bb 14>:
  D.3626 = pthread_self ();
  D.3627 = (long int) D.3626;
  D.3628 = omp_get_thread_num ();
  stdout.8 = stdout;
  fprintf (stdout.8, "    Thread[%lu,%lu]: Trabalhando na task 3.\n", D.3628, D.3627);
  return;
}

main._omp_fn.2 (void * .omp_data_i)
{
  /* Declaração de variáveis suprimida. */

<bb 22>:

<bb 11>:
  D.3630 = pthread_self ();
  D.3631 = (long int) D.3630;
  D.3632 = omp_get_thread_num ();
  stdout.5 = stdout;
  fprintf (stdout.5, "    Thread[%lu,%lu]: Trabalhando na task 2.\n", D.3632, D.3631);
  return;
}

main._omp_fn.1 (void * .omp_data_i)
{
  /* Declaração de variáveis suprimida. */

<bb 24>:

<bb 8>:
  D.3634 = pthread_self ();
  D.3635 = (long int) D.3634;
  D.3636 = omp_get_thread_num ();
  stdout.4 = stdout;
  fprintf (stdout.4, "    Thread[%lu,%lu]: Trabalhando na task 1.\n", D.3636, D.3635);
  return;
}

main._omp_fn.0 (struct .omp_data_s.10 & restrict .omp_data_i)
{
  /* Declaração de variáveis suprimida. */

<bb 26>:

<bb 5>:
  x = .omp_data_i->x;
  id = omp_get_thread_num ();
  D.3640 = pthread_self ();
  D.3641 = (long int) D.3640;
  stdout.2 = stdout;
  fprintf (stdout.2, "  Thread[%lu,%lu]: Todas as threads executam.\n", id, D.3641);

<bb 6>:
  D.3643 = __builtin_GOMP_single_start ();
  if (D.3643 == 1)
    goto <bb 7>;
  else
   goto <bb 16>;

<bb 16>:

```

```
<bb 17>:
  return;

<bb 7>:
  D.3644 = pthread_self ();
  D.3645 = (long int) D.3644;
  stdout.3 = stdout;
  fprintf (stdout.3, "  Thread[%lu,%lu]: Antes de criar tasks.\n", id, D.3645);
  D.3647 = x > 10;

<bb 25>:
  __builtin_GOMP_task (main._omp_fn.1, 0B, 0B, 0, 1, D.3647, 0, 0B, 0);

<bb 9>:

<bb 10>:
  D.3648 = x > 20;

<bb 23>:
  __builtin_GOMP_task (main._omp_fn.2, 0B, 0B, 0, 1, D.3648, 0, 0B, 0);

<bb 12>:

<bb 13>:
  D.3649 = pthread_self ();
  D.3650 = (long int) D.3649;
  stdout.6 = stdout;
  fprintf (stdout.6, "    Thread[%lu,%lu]: Antes do taskwait.\n", id, D.3650);
  __builtin_GOMP_taskwait ();
  D.3652 = pthread_self ();
  D.3653 = (long int) D.3652;
  stdout.7 = stdout;
  fprintf (stdout.7, "    Thread[%lu,%lu]: Depois do taskwait.\n", id, D.3653);

<bb 21>:
  __builtin_GOMP_task (main._omp_fn.3, 0B, 0B, 0, 1, 1, 0, 0B, 0);

<bb 15>:
  goto <bb 16>;
}

main (int argc, char * * argv)
{
  /* Declaração de variáveis suprimida. */

<bb 2>:
  if (argc <= 1)
    goto <bb 3>;
else
    goto <bb 4>;

<bb 3>:
  D.3562 = *argv;
  stderr.0 = stderr;
  fprintf (stderr.0, "Uso: %s <x>\n", D.3562);
  exit (0);

<bb 4>:
  id = 0;
  D.3564 = argv + 8;
  D.3565 = *D.3564;
  x = atoi (D.3565);
  D.3566 = pthread_self ();
  D.3567 = (long int) D.3566;
  D.3568 = omp_get_thread_num ();
  stdout.1 = stdout;
  fprintf (stdout.1, "Thread[%lu,%lu]: Antes da região paralela.\n", D.3568, D.3567);
  .omp_data_o.15.x = x;
  __builtin_GOMP_parallel (main._omp_fn.0, &.omp_data_o.15, 8, 0);
  .omp_data_o.15 = {CLOBBER};
```

```
139   D.3596 = pthread_self ();
140   D.3597 = (long int) D.3596;
141   D.3598 = omp_get_thread_num ();
142   stdout.9 = stdout;
143   fprintf (stdout.9, "Thread[%lu,%lu]: Depois da região paralela.\n", D.3598, D.3597);
144   D.3600 = 0;
145
146 <L4>:
147   return D.3600;
148 }
```

A Figura 3.9 mostra a representação gráfica do Código 3.20 com as quatro novas funções, uma para tratar a região paralela e as outras para as três *tasks*.

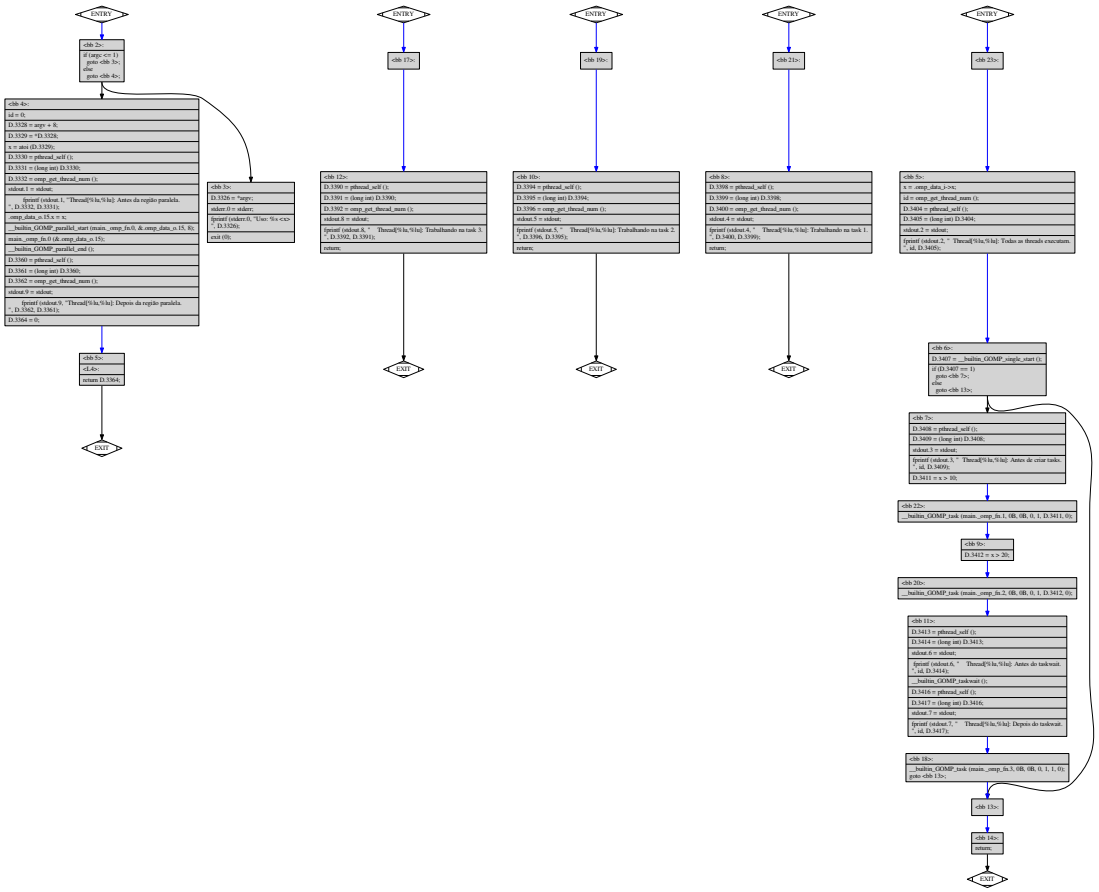

Figura 3.9. Visualização do código gerado para o construtor `task`

3.3.5. Tarefas com *loops*: construtor `taskloop`

O construtor *taskloop* é utilizado para distribuir as iterações de um ou mais laços aninhados para tarefas. Quando um construtor `taskloop` é encontrado uma nova função é criada e durante a execução subconjuntos de iterações do laço serão associados às tarefas criadas pelo construtor. O construtor `taskloop` está disponível a partir da especificação 4.5 e da `libgomp` do `GCC 6.x`. A sintaxe para uso do construtor `taskloop` é apresentada no Código 3.21.

Código 3.21. Formato do construtor *taskloop*

```
#pragma omp taskloop
{
  // for-loops.
}
```

O Código 3.22 apresenta um exemplo do uso do construtor `taskloop` aplicado a um laço. Com esse construtor é possível determinar o número de *threads* que serão criadas para a execução do laço com a cláusula `num_tasks()` e determinar o tamanho do subconjunto de iterações que cada *thread* irá executar através da cláusula `grainsize()`.

Código 3.22. Exemplo de uso do construtor *taskloop*

```
void func(){
  int i, j;
  fprintf(stdout, "Thread[%lu,%lu]: taskloop.\n", omp_get_thread_num(), (long int)
      pthread_self());
  #pragma omp taskloop num_tasks(8) private(j) grainsize(2)
  for (i = 0; i < 16; i++) {
    for (j = 0; j < i; j++) {
      fprintf(stdout, "Thread[%lu,%lu]: Trabalhando na iteração (%d,%d).\n",
          omp_get_thread_num(), (long int) pthread_self(),i,j);
    }
  }
}

int main(int argc, char *argv[]) {

  fprintf(stdout, "Thread[%lu,%lu]: Antes da Região Paralela.\n", (long int)
      omp_get_thread_num(), (long int) pthread_self());

  #pragma omp parallel num_threads(4)
  {
    #pragma omp single
    {
      fprintf(stdout, "  Thread[%lu,%lu]: Antes das tasks.\n", (long int)
          omp_get_thread_num(), (long int) pthread_self());
      #pragma omp taskgroup
      {
        #pragma omp task
        {
          fprintf(stdout, "Thread[%lu,%lu]: Trabalhando na task avulsa.\n",
              omp_get_thread_num(), (long int) pthread_self());
        }

        #pragma omp task
        {
          fprintf(stdout, "Thread[%lu,%lu]: Trabalhando na task func().\n",
              omp_get_thread_num(), (long int) pthread_self());
          func();
        }
      }
    }
  }

  fprintf(stdout, "Thread[%lu,%lu]: Depois da Região Paralela.\n", (long int)
      omp_get_thread_num(), (long int) pthread_self());

  return 0;
}
```

As funções da biblioteca `libgomp` que são utilizadas para gerar o código relacionado com o construtor *taskloop* são listadas no Quadro 3.5. As funções para a implementação de região paralela e tarefas que são utilizadas no exemplo já foram apresentadas.

Quadro 3.5: ABI `libgomp` – Funções usadas para a implementação do construtor *taskloop*

```
void GOMP_taskloop (void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
    long arg_size, long arg_align, unsigned flags, unsigned long num_tasks, int
    priority, TYPE start, TYPE end, TYPE step)
```

O Código 3.23 apresenta o código intermediário gerado pelo `GCC`. Foram criadas quatro funções, uma para tratar a região paralela (`main._omp_fn.1`) que cria um grupo de tarefas com o construtor `taskgroup` e duas *tasks* são criadas. Uma das tarefas executará a função (`main._omp_fn.2`) e outra que executará a função `main._omp_fn.2` que tem o construtor `taskloop`. A chamada à `GOMP_taskloop (func._omp_fn.0, ...)` irá executar a função `func._omp_fn.0` que tem o código do laço.

Código 3.23. Formato de código expandido para o construtor *taskloop*

```
1  func._omp_fn.0 (struct  & restrict .omp_data_i)
2  {
3    /* Declaração de variáveis suprimida. */
4
5  <bb 15>:
6
7  <bb 4>:
8    D.3591 = .omp_data_i->D.3579;
9    D.3592 = .omp_data_i->D.3581;
10   D.3593 = (int) D.3591;
11   D.3594 = (int) D.3592;
12   i = D.3593;
13
14 <bb 5>:
15   j = 0;
16
17 <bb 7>:
18   if (j < i)
19     goto <bb 6>;
20   else
21     goto <bb 8>;
22
23 <bb 8>:
24   i = i + 1;
25   if (i < D.3594)
26     goto <bb 5>;
27   else
28     goto <bb 9>;
29
30 <bb 9>:
31
32 <bb 10>:
33   return;
34
35 <bb 6>:
36   D.3597 = pthread_self ();
37   D.3598 = (long int) D.3597;
38   D.3599 = omp_get_thread_num ();
39   stdout.1 = stdout;
40   fprintf (stdout.1, "Thread[%lu,%lu]: Trabalhando na iteração (%d,%d).\n", D.3599, D
         .3598, i, j);
41   j = j + 1;
42   goto <bb 7>;
43 }
44
45 func ()
46 {
47   /* Declaração de variáveis suprimida. */
```

```
48
49 <bb 2>:
50   D.3565 = pthread_self ();
51   D.3566 = (long int) D.3565;
52   D.3567 = omp_get_thread_num ();
53   stdout.0 = stdout;
54   fprintf (stdout.0, "Thread[%lu,%lu]: taskloop.\n", D.3567, D.3566);
55   D.3574 = 0;
56   D.3573 = 16;
57   __builtin_GOMP_taskloop (func._omp_fn.0, &.omp_data_o.3, 0B, 16, 8, 1280, 8, 0, D
       .3574, D.3573, 1);
58   .omp_data_o.3 = {CLOBBER};
59   return;
60 }
61
62 main._omp_fn.3 (void * .omp_data_i)
63 {
64   /* Declaração de variáveis suprimida. */
65
66 <bb 17>:
67
68 <bb 10>:
69   D.3642 = pthread_self ();
70   D.3643 = (long int) D.3642;
71   D.3644 = omp_get_thread_num ();
72   stdout.7 = stdout;
73   fprintf (stdout.7, "Thread[%lu,%lu]: Trabalhando na task func().\n", D.3644, D.3643);
74   func ();
75   return;
76 }
77
78 main._omp_fn.2 (void * .omp_data_i)
79 {
80   /* Declaração de variáveis suprimida. */
81
82 <bb 19>:
83
84 <bb 7>:
85   D.3646 = pthread_self ();
86   D.3647 = (long int) D.3646;
87   D.3648 = omp_get_thread_num ();
88   stdout.6 = stdout;
89   fprintf (stdout.6, "Thread[%lu,%lu]: Trabalhando na task avulsa.\n", D.3648, D.3647);
90   return;
91 }
92
93 main._omp_fn.1 (void * .omp_data_i)
94 {
95   /* Declaração de variáveis suprimida. */
96
97 <bb 21>:
98
99 <bb 3>:
100
101 <bb 4>:
102   D.3650 = __builtin_GOMP_single_start ();
103   if (D.3650 == 1)
104     goto <bb 5>;
105   else
106     goto <bb 13>;
107
108 <bb 13>:
109
110 <bb 14>:
111   return;
112
113 <bb 5>:
114   D.3651 = pthread_self ();
115   D.3652 = (long int) D.3651;
116   D.3653 = omp_get_thread_num ();
```

```
117   D.3654 = (long int) D.3653;
118   stdout.5 = stdout;
119   fprintf (stdout.5, "  Thread[%lu,%lu]: Antes das tasks.\n", D.3654, D.3652);
120
121 <bb 6>:
122   __builtin_GOMP_taskgroup_start ();
123
124 <bb 20>:
125   __builtin_GOMP_task (main._omp_fn.2, 0B, 0B, 0, 1, 1, 0, 0B, 0);
126
127 <bb 8>:
128
129 <bb 9>:
130
131 <bb 18>:
132   __builtin_GOMP_task (main._omp_fn.3, 0B, 0B, 0, 1, 1, 0, 0B, 0);
133
134 <bb 11>:
135
136 <bb 12>:
137   __builtin_GOMP_taskgroup_end ();
138   goto <bb 13>;
139 }
140
141 main (int argc, char * * argv)
142 {
143   /* Declaração de variáveis suprimida. */
144
145 <bb 2>:
146   D.3601 = pthread_self ();
147   D.3602 = (long int) D.3601;
148   D.3603 = omp_get_thread_num ();
149   D.3604 = (long int) D.3603;
150   stdout.4 = stdout;
151   fprintf (stdout.4, "Thread[%lu,%lu]: Antes da Região Paralela.\n", D.3604, D.3602);
152   __builtin_GOMP_parallel (main._omp_fn.1, 0B, 4, 0);
153   D.3619 = pthread_self ();
154   D.3620 = (long int) D.3619;
155   D.3621 = omp_get_thread_num ();
156   D.3622 = (long int) D.3621;
157   stdout.8 = stdout;
158   fprintf (stdout.8, "Thread[%lu,%lu]: Depois da Região Paralela.\n", D.3622, D.3620);
159   D.3624 = 0;
160
161 <L2>:
162   return D.3624;
163 }
```

A Figura 3.10 apresenta a visualização gráfica do código gerado para o exemplo com o construtor `taskloop`.

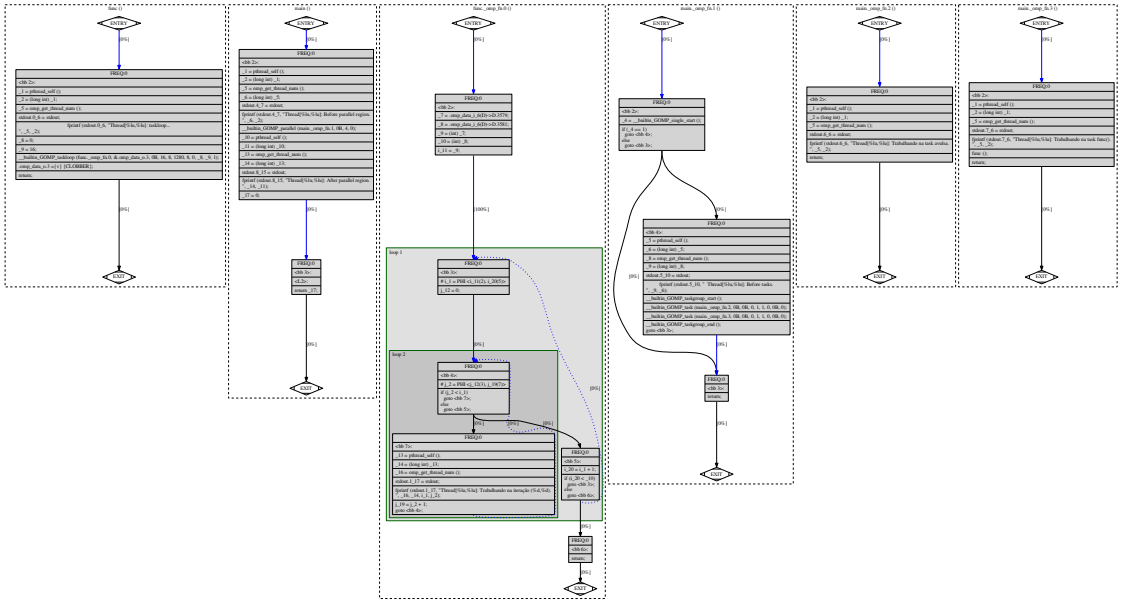

Figura 3.10. Visualização do exemplo com o construtor `taskloop`

3.3.6. Suporte à Vetorização: `construtor simd`

O construtor `simd` pode ser aplicado a um laço diretamente indicando que múltiplas iterações do laço podem ser executadas concorrentemente usando instruções `SIMD`. Também pode ser combinado com construtores como o `for` e `taskloop` para que o conjunto de iterações seja dividido entre as *threads* e essas iterações possam ser executadas usando instruções `SIMD`. A sintaxe para uso do construtor `simd` é apresentada no Código 3.24.

Código 3.24. Formato do construtor *simd*

```
1 #pragma omp simd [clause[[,] clause] ...]new-line
2 for-loops
```

O Código 3.25 apresenta um exemplo de código que utiliza o construtor `simd` em um laço que faz a multiplicação de dois *arrays*.

Código 3.25. Exemplo de Código usando o construtor *simd*

```
1  int main(int argc, char **argv) {
2    int i;
3    double res;
4    init_array();
5
6    #pragma omp simd
7    for (i = 0; i < N; i++) {
8      h_c[i] += h_a[i] * h_b[i];
9    }
10
11   return 0;
12 }
```

O corpo do laço com instruções `SIMD` é apresentado no Código 3.26.

Código 3.26. Código do corpo do laço com instruções `SIMD`

```
.L6:
  movl  -20(%rbp), %eax
  cltq
  movsd h_c(,%rax,8), %xmm1
  movl  -20(%rbp), %eax
  cltq
  movsd h_a(,%rax,8), %xmm2
  movl  -20(%rbp), %eax
  cltq
  movsd h_b(,%rax,8), %xmm0
  mulsd %xmm2, %xmm0
  addsd %xmm0, %xmm1
  movq  %xmm1, %rax
  movl  -20(%rbp), %edx
  movslq  %edx, %rdx
  movq  %rax, h_c(,%rdx,8)
  addl  $1, -20(%rbp)
```

O construtor `simd` pode ser combinado com o `for`. O Código 3.27 apresenta um exemplo que utiliza os construtores `for` e `simd` combinados.

Código 3.27. Exemplo de Código usando os construtores `for` e `simd`

```
int main(int argc, char *argv[]) {
  int i;
  /* Inicialização dos vetores. */
  init_array();

  #pragma omp parallel for simd schedule(dynamic, 32) num_threads(4)
  for (i = 0; i < N; i++) {
    h_c[i] = h_a[i] * h_b[i];
  }

  /* Resultados. */
  print_array();
  check_result();

  return 0;
}
```

Os Códigos 3.28 e 3.29 apresentam o código *assembly* gerado para a função *main* e a função para execução do laço com instruções `SIMD` em seu corpo.

Código 3.28. Código da função main

```
1  main:
2    pushq %rbp
3    movq  %rsp, %rbp
4    subq  $48, %rsp
5    movl  %edi, -36(%rbp)
6    movq  %rsi, -48(%rbp)
7    movl  $0, %eax
8    call  init_array
9    leaq  -32(%rbp), %rax
10   pushq $0
11   pushq $32
12   movl  $1, %r9d
13   movl  $1048576, %r8d
14   movl  $0, %ecx
15   movl  $4, %edx
16   movq  %rax, %rsi
17   movl  $main._omp_fn.0, %edi
18   call  GOMP_parallel_loop_dynamic
19   addq  $16, %rsp
20   movl  -32(%rbp), %eax
21   movl  %eax, -4(%rbp)
22   movl  $0, %eax
23   call  check_result
24   movl  $0, %eax
25   leave
26   ret
27   .size main, .-main
```

Código 3.29. Código da função extraída para tratar o laço com instruções `SIMD`

```
1  .type main._omp_fn.0, @function
2  main._omp_fn.0:
3    /* Código Suprimido. */
4    call  GOMP_loop_dynamic_next
5    testb %al, %al
6    je  .L13
7  .L17:
8    /* Código Suprimido. */
9  .L15:
10   cmpl  %edx, -20(%rbp)
11   jge .L14
12   movl  -20(%rbp), %eax
13   cltq
14   movsd h_a(,%rax,8), %xmm1
15   movl  -20(%rbp), %eax
16   cltq
17   movsd h_b(,%rax,8), %xmm0
18   mulsd %xmm1, %xmm0
19   movl  -20(%rbp), %eax
20   cltq
21   movsd %xmm0, h_c(,%rax,8)
22   addl  $1, -20(%rbp)
23   jmp .L15
24 .L14:
25   cmpl  $1048576, -20(%rbp)
26   je  .L16
27 .L18:
28   /* Código Suprimido. */
29   call  GOMP_loop_dynamic_next
30   testb %al, %al
31   jne .L17
32   jmp .L13
33 .L16:
34   /* Código Suprimido. */
35   jmp .L18
36 .L13:
37   cmpl  $1048576, %ebx
38   je  .L19
39 .L20:
40   call  GOMP_loop_end_nowait
41   jmp .L21
42 .L19:
43   /* Código Suprimido. */
```

O construtor `simd` pode também ser combinado com o `taskloop`. O Código 3.30 apresenta um exemplo que utiliza os construtores `taskloop` e `simd` combinados. As iterações do laço serão executadas em paralelo por *tasks* e as iterações que cada *thread* executa podem ser transformadas em instruções `SIMD`.

Código 3.30. Exemplo de Código usando os construtores `taskloop` e `simd`

```
1  void func(){
2    int i;
3
4    #pragma omp taskloop simd num_tasks(4)
5    for (i = 0; i < N; i++) {
6      h_c[i] = h_a[i] * h_b[i];
7    }
8  }
```

A Figura 3.11 apresenta a visualização do código para o construtor `taskloop`. Pode ser visto que toda a estrutura de execução do construtor `taskloop` e dos construtores utilizados na região paralela é criada. É em `func()` que o construtor `taskloop` foi declarado e então quando `GOMP_taskloop (func._omp_fn.0,...)` é chamada, como parâmetro é passado a função que executa o laço. O efeito que o construtor `simd` causa é perceptível somente na geração do código final.

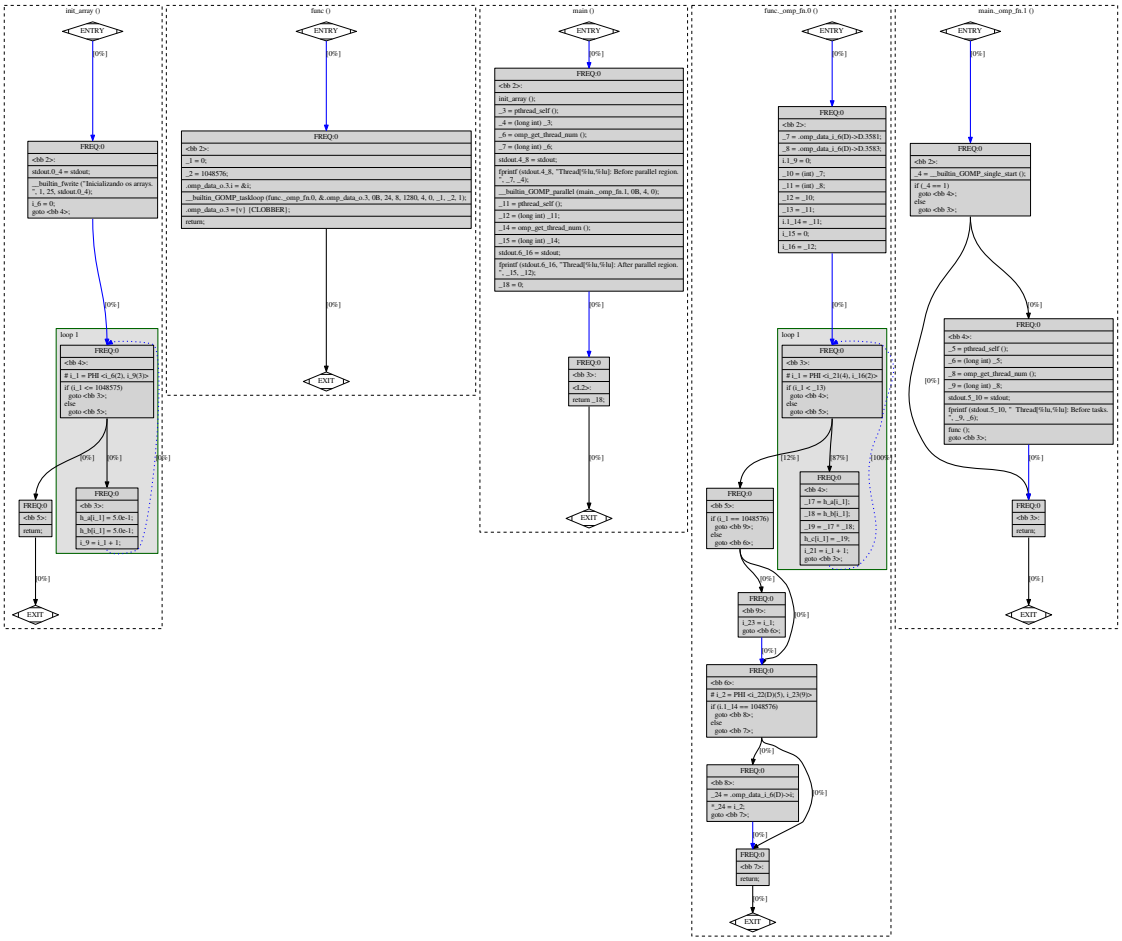

Figura 3.11. Visualização do código gerado para o construtor `taskloop` combinado com `simd`

3.3.7. *Offloading* para Aceleradores: construtor `target`

Para falarmos sobre *diretivas de compilação* para aceleradores temos que introduzir o modelo de programação para aceleradores como as `GPUs`. Para esse tipo de dispositivo acelerador é necessário definir uma função *kernel* que terá sua execução lançada no dispositivo. Um *kernel* para a soma de vetores escrito em `CUDA` [NVIDIA 2017] pode ser visto no Código 3.31.

Código 3.31. Função *kernel* em CUDA para soma de vetores

```
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  if (id < n)
    c[id] = a[id] + b[id];
}
```

O Código 3.32 mostra como os dados são declarados. Como o exemplo é de soma de vetores, temos a declaração de três *arrays* (`h_a, h_b e h_c`) que são alocados e representam os dados na memória principal do *host* e mais três ponteiros que são alocados na memória da GPU que irão representar os três vetores do lado do dispositivo *(device)* (`d_a, d_b e d_c`. Para a alocação de memória do lado *device* existe uma função `cudaMalloc(...)` equivalente à função `malloc(...)`.

Código 3.32. Declaração e alocação de dados do lado *host* e do lado *device*

```
int main( int argc, char* argv[] ){
float *h_a;
float *h_b;
float *h_c;

// Declaracao dos vetores de entrada na memoria da GPU.
float *d_a;
float *d_b;
// Declaracao do vetor de saida do dispositivo.
float *d_c;

// Tamanho em bytes de cada vetor.
size_t bytes = n * sizeof(float);

// Alocacao de memoria para os vetores do host.
h_a = (float*) malloc(bytes);
h_b = (float*) malloc(bytes);
h_c = (float*) malloc(bytes);

// Alocacao de memoria para cada vetor na GPU.
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);

// Inicializacao dos arrays.
```

CUDA fornece função para realizar transferências de dados entre a memória principal e a memória do dispositivo, o Código 3.33 apresenta a cópia dos dados dos *arrays*.

Código 3.33. Transferência dos dados para a memória do dispositivo

```
// Copia dos vetores do host para o dispositivo.
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
```

A chamada à função *kernel* pode ser vista no Código 3.34. Na ativação do *kernel* a configuração da estrutura do arranjo de *threads* (`grid e bloco`) precisa ser definida explicitamente pelo programador. Essa configuração determina quantas *threads* serão criadas e como estarão organizadas em blocos dentro do *grid* mapeado para o dispositivo.

Código 3.34. Ativação da função *kernel*

```
1  int blockSize, gridSize;
2
3  // Numero de threads em cada bloco de threads.
4  blockSize = 1024;
5
6  // Numero de blocos de threads no grid.
7  gridSize = (int)ceil((float)n/blockSize);
8
9  // Chamada a funcao kernel.
10 vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
```

O Código 3.35 apresenta a cópia do resultado (`d_c`) da soma de vetores realizada no dispositivo para (`h_c`) na memória do *host*.

Código 3.35. Cópia do resultado e liberação da memória alocada

```
1  // Copia do vetor resultado da GPU para o host.
2  cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost );
3
4  // Liberacao da memoria da GPU.
5  cudaFree(d_a);
6  cudaFree(d_b);
7  cudaFree(d_c);
8
9  // Liberacao da memoria do host.
10 free(h_a);
11 free(h_b);
12 free(h_c);
13
14 return 0;
15 }
```

Com o exemplo de soma de vetores escrito em `CUDA`, no modelo clássico de execução, no qual as transferências são declaradas explicitamente, é possível ter uma ideia das operações envolvidas na execução de código em dispositivo aceleradores.

No contexto de *diretivas de compilação* o padrão `OpenACC` [OpenACC 2015] [OpenACC 2017] fornece um conjunto de diretivas para que da mesma maneira que podemos anotar código em `OpenMP`, possamos anotar código de laços e regiões paralelizáveis que podem ser transformados em *kernels* e ter sua execução acelerada por uma `GPU`.

Como no `OpenMP` as diretivas em `C/C++` são especificadas usando `#pragma` e se o compilador não tiver suporte as anotações são ignoradas na compilação. Cada diretiva em `C/C++` inicia com `#pragma acc` e existem construtores e cláusulas para a criação de *kernels* com base em laços, por exemplo.

O modelo de execução do `OpenACC` tem três níveis: *gang*, *worker* e *vector*. É um mapeamento dos elementos presentes no contexto de GPUs que utilizam `CUDA`, sendo `gang=bloco`, `worker=warp`, `vector=threads`, sendo um *warp* é um conjunto de *threads* escalonáveis num multiprocessador (SM) [Denise Stringhini 2012]. O Código 3.36 apresenta o formato das diretivas do `OpenACC` .

Código 3.36. Formato das diretivas do `OpenACC`

```
1  #pragma acc directive-name [clause [[,] clause]...] new-line
```

O Código 3.37 apresenta o exemplo soma de vetores escrito com as diretivas do `OpenACC`. Na função `main` podemos ver o construtor `data` que especifica através da

cláusulas `copyin` e `copyout` os dados que devem ser copiados da memória do *host* para a memória do dispositivo e vice-versa. Nesse exemplo, os *arrays* `a` e `b` serão copiados como entrada para a execução do *kernel* e o `c` será copiado após a execução do *kernel* como resultado.

Código 3.37. Exemplo de Soma de Vetores anotado com diretivas `OpenACC`

```
1  void vecaddgpu(float *restrict c, float *a, float *b, int n){
2    #pragma acc kernels for present(c,a,b)
3    for( int i = 0; i < n; ++i )
4      c[i] = a[i] + b[i];
5  }
6
7  int main( int argc, char* argv[] ){
8
9    #pragma acc data copyin(a[0:n],b[0:n]) copyout(c[0:n])
10   {
11     vecaddgpu(c, a, b, n);
12   }
13
14   return 0;
15 }
```

A saída gerada pelo compilador `pgcc` [PGROUP 2015] é apresentada no Teminal 3.6. As mensagens indicam que o laço anotado com o construtor `kernels for` foi detectado como paralelizável e no lançamento da execução do *kernel* cada bloco será criado com 256 *threads*. Também foram geradas as operações de transferências de dados.

Terminal 3.6

```
rogerio@chamonix:/src/example-openacc$ pgcc -acc -ta=nvidia,time -Minfo=accel -fast
    vectoradd.c -o vectoradd-acc-gpu
  vecaddgpu:
    12, Generating present(b[0:])
        Generating present(a[0:])
        Generating present(c[0:])
        Generating compute capability 1.0 binary
        Generating compute capability 2.0 binary
    13, Loop is parallelizable
        Accelerator kernel generated
        13, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
        CC 1.0 : 5 registers; 36 shared, 4 constant, 0 local memory bytes; 100%
            occupancy
        CC 2.0 : 5 registers; 4 shared, 48 constant, 0 local memory bytes; 100%
            occupancy
  main:
    44, Generating copyout(c[0:n])
        Generating copyin(b[0:n])
        Generating copyin(a[0:n])
rogerio@chamonix:/src/example-openacc$
```

Para *offloading* de código para dispositivos aceleradores, no `OpenMP` temos o construtor `target`. A sintaxe de uso desse construtor é apresentada no Código 3.38.

Código 3.38. Sintax do Construtor `target` do OpenMP

```
1 #pragma omp target [clause[ [ , ] clause] ... ] new-line
2 bloco-estruturado
```

O Código 3.39 apresenta os construtores `target` e `parallel for` combinados. O construtor `target` faz o mapeamento de variáveis para a memória do dispositivo e lança a execução do código no dispositivo. Uma função com o código associado ao construtor `target` é criada para ser executada no dispositivo alvo. O dispositivo alvo *(device target)* pode ser definido chamando a função `omp_set_default_device(int device_num)` com o número do dispositivo sendo passado como argumento ou definindo-se a variável de ambiente `OMP_DEFAULT_DEVICE` ou ainda usando a cláusula `device (device_num)`.

Código 3.39. Exemplo com construtor `target` combinado com laço paralelo

```
1 void vecaddgpu(float *restrict c, float *a, float *b, int n){
2   #pragma omp target device(0)
3   #pragma omp parallel for private(i)
4   for( int i = 0; i < n; ++i ){
5     c[i] = a[i] + b[i];
6   }
7 }
```

O mapeamento de dados para o dispositivo pode ser feito usando-se a cláusula `map` admitida pelo construtor `target`.

As variáveis a, b e c são mapeadas explicitamente para o dispositivo alvo. A variável n é mapeada implicitamente, pois é referenciada no código. Os tipos de mapeamento aceitos pela cláusula `map` indicam o sentido da transferência de dados a ser realizada: `map(to:vars)` *(host→device)* e `map(from:vars)` *(device→host)*. O Código 3.40 apresenta o uso da cláusula `map`. A declaração `map(to:a[0:n],b[:n])` indica que os arranjos a, b devem ser copiados para a memória do dispositivo e `map(from: c[0:n])` que o arranjo c será copiado de volta para a memória do *host*, como um resultado da execução do *kernel*.

Código 3.40. Mapeando dados para o disposito com a cláusula `map`

```
1 void vecaddgpu(float *restrict c, float *a, float *b, int n){
2   #pragma omp target map(to: a[0:n], b[:n]) map(from: c[0:n])
3   #pragma omp parallel for private(i)
4   for( int i = 0; i < n; ++i ) {
5     c[i] = a[i] + b[i];
6   }
7 }
```

O construtor `target` também permite a escolha de fazer o *offloading* do código para o dispositivo ou não, com base no tamanho dos dados, por exemplo. Isso pode ser feito utilizando a cláusula `if`, que possui o comportamento semelhante ao que vimos para região paralela. Um exemplo com a cláusula `if` para que a execução do *kernel* será lançada no dispositivo somente para tamanho de n que ultrapasse um limiar de valores é apresentado no Código 3.41.

Código 3.41. Decidindo sobre *offloading* utilizando a cláusula `if`

```
1  #define THRESHOLD 1024
2
3  void vecaddgpu(float *restrict c, float *a, float *b, int n){
4    #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:n]) if(n>THRESHOLD)
5    {
6      #pragma omp target if(n>THRESHOLD)
7      #pragma omp parallel for if(n>THRESHOLD)
8      for( int i = 0; i < n; ++i )
9        c[i] = a[i] + b[i];
10   }
11 }
```

Ainda no Código 3.41 é possível percebermos que as transferências de dados também podem ser declaradas com o construtor `target data` que cria um novo ambiente de dados que será utilizado pelo *kernel*. A cópia dos dados também pode ser condicionada a um tamanho dos dados utilizando a cláusula `if`, e as transferências somente devem ser feitas para a memória do dispositivo se o objetivo for lançar a execução do *kernel*.

Especificar uma região de dados pode ser útil quando múltiplos *kernels* irão executar sobre os mesmos dados. O Código 3.42 apresenta uma região de dados definida com o construtor `target data` que especifica somente a cópia de volta do *array c*, pois entre as execuções das *target regions* há uma atualização dos elementos de *a* e *b* que são copiados da memória do *host* para a memória do dispositivo antes da execução de cada *kernel*.

Código 3.42. Declarando dois *kernels* para mesma região de dados

```
1  #define THRESHOLD 1048576
2
3  void vecaddgpu(float *restrict c, float *a, float *b, int n){
4    #pragma omp target data map(from: c[0:n])
5    {
6      #pragma omp target if(n>THRESHOLD) map(to: a[0:n], b[:n])
7      #pragma omp parallel for
8      for( int i = 0; i < n; ++i )
9        c[i] = a[i] + b[i];
10
11     // Reinicialização dos dados.
12     init(a,b);
13
14     #pragma omp target if(n>THRESHOLD) map(to: a[0:n], b[:n])
15     #pragma omp parallel for
16     for( int i = 0; i < n; ++i )
17       c[i] = c[i] + (a[i] * b[i]);
18     }
19   }
20 }
```

Uma outra maneira de se fazer a atualização dos dados entre as execuções dos *kernels* é utilizando o construtor `target update` que atualiza os dados de uma seção de mapeamento para o ambiente de dados do dispositivo. O Código 3.43 apresenta o código do exemplo anterior modificado para usar o construtor `target update`, que também admite a cláusula `if` que pode ser utilizada para atualizar os dados entre as execuções dos *kernels* se esses foram modificados.

Código 3.43. Atualizando os dados entre as execuções dos *kernels*

```
void vecaddgpu(float *restrict c, float *a, float *b, int n){
  int changed = 0;
  #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:n])
  {
    #pragma omp target
    #pragma omp parallel for
    for( int i = 0; i < n; ++i )
      c[i] = a[i] + b[i];

    changed = init(a,b);

    #pragma omp target update if (changed) to(a[0:n], b[:n])

    #pragma omp target
    #pragma omp parallel for
    for( int i = 0; i < n; ++i )
      c[i] = c[i] + (a[i] * b[i]);
  }
}
```

O Código 3.44 apresenta o mesmo exemplo de soma de vetores feito em `CUDA` e em `OpenACC` no `OpenMP` utilizando o construtor `target` e suas combinações vistas nos exemplos anteriores.

Código 3.44. Atualizando os dados entre as execuções dos *kernels*

```
#define THRESHOLD 1024

float *h_a;
float *h_b;
float *h_c;
int n = 0;

/* Código Suprimido. */

void vecaddgpu(float *restrict c, float *a, float *b){
  #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:n]) if(n>THRESHOLD)
  {
    #pragma omp target if(n>THRESHOLD)
    #pragma omp parallel for if(n>THRESHOLD)
    for( int i = 0; i < n; ++i ){
      c[i] = a[i] + b[i];
    }
  }
}

int main(int argc, char *argv[]) {
  int i;
  n = atoi(argv[1]);

  h_a = (float*) malloc(n*sizeof(float));
  h_b = (float*) malloc(n*sizeof(float));
  h_c = (float*) malloc(n*sizeof(float));

  init_array();

  vecaddgpu(h_c, h_a, h_b);

  return 0;
}
```

A Figura 3.12 apresenta a estrutura do código gerado para o exemplo do Código 3.44.

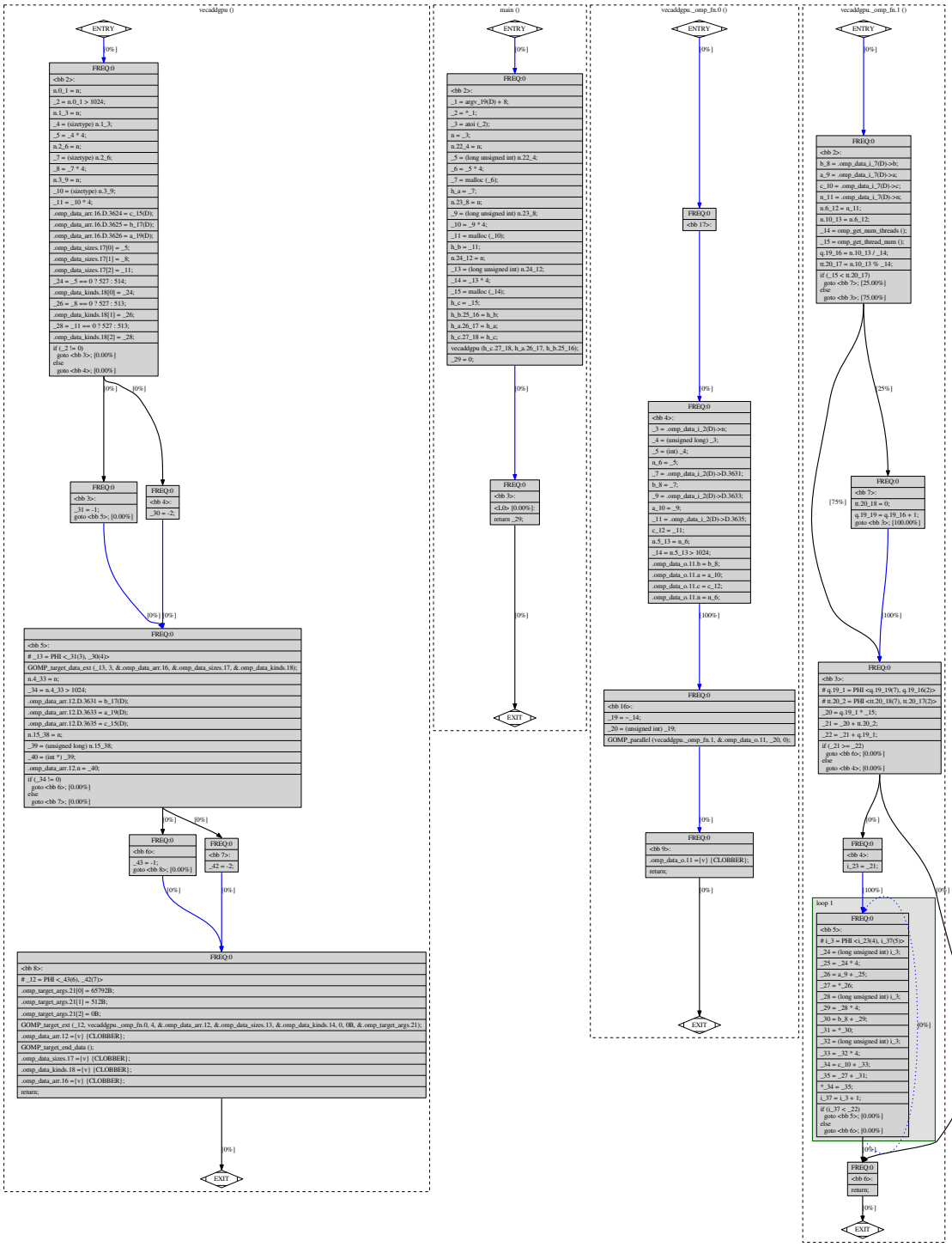

Figura 3.12. Visualização do Código intermediário gerado para o construtor `target`

As funções relacionadas com a geração de código para o construtor `target` que identificamos na `ABI` da `libgomp` estão listadas no Quadro 3.6.

Quadro 3.6: ABI `libgomp` – Funções relacionadas com o construtor *target*

```
void GOMP_parallel (void (*fn) (void *), void *data, unsigned num_threads, unsigned
    int flags)
void GOMP_target_data_ext (int device, size_t mapnum, void **hostaddrs, size_t *sizes,
     unsigned short *kinds)
void GOMP_target_end_data (void)
void GOMP_target_update (int device, const void *unused, size_t mapnum, void **
    hostaddrs, size_t *sizes, unsigned char *kinds)
void GOMP_target_ext (int device, void (*fn) (void *), size_t mapnum, void **hostaddrs
    , size_t *sizes, unsigned short *kinds, unsigned int flags, void **depend, void **
    args)
```

Como o Código 3.44 utiliza a cláusula `if` para decidir se deve ou não fazer o *offloading* para o dispositivo com base no tamanho dos dados. Executamos o exemplo soma de vetores com tamanho de dados $n = 16384$ e utilizamos a ferramenta de perfilamento, o `nvprof` para nos certificarmos que as transferências de dados e o *offloading* de código para o dispositivo seria feito. A saída da execução é apresentada no Terminal 3.7.

Terminal 3.7

```
rogerio@ragserver:~/example-target$ nvprof ./example-target.exe 16384
Inicializando os arrays.
==2381== NVPROF is profiling process 2381, command: ./example-target.exe 16384
Verificando o resultado.
Resultado Final: (16384.000000, 1.000000)
==2381== Profiling application: ./example-target.exe 16384
==2381== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
97.56%  2.6697ms         1  2.6697ms  2.6697ms  2.6697ms  vecaddgpu$_omp_fn$0
1.61%  44.160us         6  7.3600us  1.0560us  21.408us  [CUDA memcpy HtoD]
0.82%  22.496us         1  22.496us  22.496us  22.496us  [CUDA memcpy DtoH]

==2381== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
60.98%  131.59ms         1  131.59ms  131.59ms  131.59ms  cuCtxCreate
34.12%  73.631ms         1  73.631ms  73.631ms  73.631ms  cuCtxDestroy
1.24%  2.6735ms         1  2.6735ms  2.6735ms  2.6735ms  cuCtxSynchronize
1.17%  2.5168ms        22  114.40us  32.989us  999.11us  cuLinkAddData
1.07%  2.3177ms         1  2.3177ms  2.3177ms  2.3177ms  cuModuleLoadData
0.45%  961.91us         1  961.91us  961.91us  961.91us  cuLinkComplete
0.25%  544.36us         1  544.36us  544.36us  544.36us  cuLaunchKernel
0.18%  388.84us         3  129.61us  125.77us  135.86us  cuMemAlloc
0.18%  387.49us         1  387.49us  387.49us  387.49us  cuMemAllocHost
0.11%  239.10us         3  79.698us  74.062us  85.600us  cuMemFree
0.09%  186.99us         1  186.99us  186.99us  186.99us  cuMemFreeHost
0.05%  116.62us        11  10.601us     114ns  114.64us  cuDeviceGetAttribute
0.05%  105.62us         6  17.604us  7.6740us  41.930us  cuMemcpyHtoD
0.03%  69.121us         1  69.121us  69.121us  69.121us  cuMemcpyDtoH
0.02%  38.732us         1  38.732us  38.732us  38.732us  cuLinkCreate
0.00%  3.9020us         9     433ns     315ns     605ns  cuMemGetAddressRange
0.00%  2.9470us        14     210ns     144ns     398ns  cuCtxGetDevice
0.00%  1.6580us         1  1.6580us  1.6580us  1.6580us  cuModuleGetFunction
0.00%  1.4500us         3     483ns     127ns     906ns  cuDeviceGetCount
0.00%     756ns         2     378ns     310ns     446ns  cuFuncGetAttribute
0.00%     751ns         1     751ns     751ns     751ns  cuMemHostGetDevicePointer
0.00%     698ns         1     698ns     698ns     698ns  cuLinkDestroy
0.00%     639ns         1     639ns     639ns     639ns  cuModuleGetGlobal
0.00%     592ns         1     592ns     592ns     592ns  cuInit
0.00%     553ns         2     276ns     223ns     330ns  cuDeviceGet
0.00%     155ns         1     155ns     155ns     155ns  cuCtxGetCurrent
rogerio@ragserver:~/example-target$
```

Da mesma forma o exemplo foi executado com $n = 512$ e podemos verificar com o `nvprof` que nenhuma operação relacionada ao dispositivo (transferências de dados e

lançamento da execução de *kernels*) que caracterizaria o *offloading* de código foi realizada. A saída da execução é apresentada no Terminal 3.8.

Terminal 3.8

```
rogerio@ragserver:~/example-target$ nvprof ./example-target.exe 512
Inicializando os arrays.
Verificando o resultado.
Resultado Final: (512.000000, 1.000000)
======== Warning: No CUDA application was profiled, exiting
```

3.4. Aplicações

Conhecer como é o formato de código gerado e as funções da `ABI` do *runtime* do `OpenMP` pode ser útil para a construção de bibliotecas de interceptação de código via *hooking*.

Essas bibliotecas podem ser pré-carregadas para alterarem o comportamento da execução de aplicações `OpenMP`. Essa técnica pode ser utilizada para a execução de código pré ou pós chamada ao *runtime* do `OpenMP`. O que pode cobrir desde *logging*, criação de *traces* [Trahay et al. 2011], monitoramento [Mohr et al. 2002] e avaliação de desempenho ou *offloading* de código para dispositivos aceleradores.

Para criar *hooks* para funções da `libgomp` é necessário criar uma biblioteca que tenha funções com o mesmo nome das funções disponibilizadas em sua `ABI`. Uma vez que a biblioteca de *hooking* seja carregada antes da biblioteca `libgomp`, os símbolos como as chamadas para as funções do *runtime* do `OpenMP` serão ligados aos símbolos da biblioteca de interceptação. A ideia é recuperar do *linker* via *dlsym* um ponteiro para a função original para que a chamada original possa ser feita de dentro da função *proxy*. Um *hook* para a função `GOMP_parallel_start()` é apresentado no Código 3.45.

Código 3.45. Exemplo de criação de uma hook para a função *GOMP_parallel_start*

```
1 void GOMP_parallel_start (void (*fn) (void *), void *data, unsigned num_threads){
2   PRINT_FUNC_NAME;
3
4   /* Retrieve the OpenMP runtime function. */
5   typedef void (*func_t) (void (*fn) (void *), void *, unsigned);
6   func_t lib_GOMP_parallel_start = (func_t) dlsym(RTLD_NEXT, "GOMP_parallel_start");
7
8   lib_GOMP_parallel_start(fn, data, num_threads);
9 }
```

No Código 3.46 é definida uma macro para a recuperação do ponteiro para a função original, no caso ponteiros para funções do *runtime* `OpenMP`.

Código 3.46. Definição de macro para recuperar o ponteiro para a função original

```
1 #define GET_RUNTIME_FUNCTION(hook_func_pointer,func_name)    \
2   do {                                                       \
3     if (hook_func_pointer) break;                            \
4     void *__handle = RTLD_NEXT;                              \
5     hook_func_pointer = (typeof(hook_func_pointer)) (uintptr_t) dlsym(__handle,
          func_name);                                  \
6     PRINT_ERROR();                                           \
7   } while(0)
```

O Código 3.47 apresenta a mesma função proxy usando a macro para recuperar o ponteiro para a função original. Além disso, apresenta a ideia de chamadas de funções

para executar algum código antes (`PRE_`) ou algum código depois (`POST_`).

Código 3.47. Definição de macro para recuperar o ponteiro para a função original

```
1  void GOMP_parallel_start (void fn)(void), void *data, unsigned num_threads){
2    PRINT_FUNC_NAME;
3
4    /* Retrieve the OpenMP runtime function. */
5    GET_RUNTIME_FUNCTION(lib_GOMP_parallel_start, "GOMP_parallel_start");
6
7    /* Código a ser executado antes. */
8    PRE_GOMP_parallel_start();
9
10   /* Chamada à função original. */
11   lib_GOMP_parallel_start(fn, data, num_threads);
12
13   /* Código a ser executado depois. */
14   POST_GOMP_parallel_start();
15 }
```

O Código 3.48 apresenta a função *proxy* para a função de inicialização de laço com escalonamento do tipo *dynamic*.

Código 3.48. Definição de macro para recuperar o ponteiro para a função original

```
1  void GOMP_parallel_loop_dynamic_start (void fn)(void), void *data,
2  unsigned num_threads, long start, long end,
3  long incr, long chunk_size){
4    PRINT_FUNC_NAME;
5
6    /* Retrieve the OpenMP runtime function. */
7    GET_RUNTIME_FUNCTION(lib_GOMP_parallel_loop_dynamic_start, "
         GOMP_parallel_loop_dynamic_start");
8
9    /* Código a ser executado antes. */
10   PRE_GOMP_parallel_loop_dynamic_start();
11
12   /* Chamada à função original. */
13   lib_GOMP_parallel_loop_dynamic_start(fn, data, num_threads, start, end, incr,
         chunk_size);
14
15   /* Código a ser executado depois. */
16   POST_GOMP_parallel_loop_dynamic_start();
17 }
```

A função *proxy* para a função de término de laços de repetição é apresentada no Código 3.49.

Código 3.49. Definição de macro para recuperar o ponteiro para a função original

```
1  void GOMP_loop_end (void){
2    PRINT_FUNC_NAME;
3
4    /* Retrieve the OpenMP runtime function. */
5    GET_RUNTIME_FUNCTION(lib_GOMP_loop_end, "GOMP_loop_end");
6
7    /* Código a ser executado antes. */
8    PRE_GOMP_loop_end();
9
10   /* Chamada à função original. */
11   lib_GOMP_loop_end();
12
13   /* Código a ser executado depois. */
14   POST_GOMP_loop_end();
15 }
```

O Código 3.50 apresenta a função *proxy* capaz de interceptar a função de criação de *tasks*. Podendo da mesma forma que outras funções de interceptação, executar um código antes e outro depois da chamada à função original.

Código 3.50. Definição de macro para recuperar o ponteiro para a função original

```
1  void GOMP_task (void fn)(void), void *data, void cpyfn)(void *, void),
2  long arg_size, long arg_align, bool if_clause, unsigned flags,
3  void **depend){
4    PRINT_FUNC_NAME;
5
6    /* Retrieve the OpenMP runtime function. */
7    GET_RUNTIME_FUNCTION(lib_GOMP_task, "GOMP_task");
8
9    /* Código a ser executado antes. */
10   PRE_GOMP_task();
11
12   /* Chamada à função original. */
13   lib_GOMP_task(fn, data, cpyfn, arg_size, arg_align, if_clause, flags, depend);
14
15   /* Código a ser executado depois. */
16   POST_GOMP_task();
17 }
```

Parte da saída da execução do exemplo do uso do construtor `task` com a biblioteca de interceptação é apresentada no Código 3.9.

Terminal 3.9

```
rogerio@chamonix:/src/simple-omp-hook/tests/parallel-region-with-tasks$ LD_PRELOAD=./
    libhookomp.so ./parallel-region-with-tasks.exe 1024
Thread[0,139859015989184]: Antes da região paralela.
TRACE: [ hookomp.c:0000753] Calling [GOMP_parallel_start()]
TRACE: [prepostfunctions.c:0000024] Calling [PRE_GOMP_parallel_start()]
TRACE: [prepostfunctions.c:0000033] Calling [POST_GOMP_parallel_start()]
Thread[1,139858992330496]: Todas as threads executam.
Thread[2,139858983937792]: Todas as threads executam.
TRACE: [ hookomp.c:0000987] Calling [GOMP_single_start()]
TRACE: [ hookomp.c:0000987] Calling [GOMP_single_start()]
...
Thread[1,139858992330496]: Antes de criar tasks.
TRACE: [ hookomp.c:0000830] Calling [GOMP_task()]
TRACE: [ hookomp.c:0000771] Calling [GOMP_parallel_end()]
TRACE: [prepostfunctions.c:0000029] Calling [PRE_GOMP_parallel_end()]
TRACE: [prepostfunctions.c:0000062] Calling [PRE_GOMP_task()]
TRACE: [prepostfunctions.c:0000067] Calling [POST_GOMP_task()]
Thread[3,139858975545088]: Trabalhando na task 1.
TRACE: [ hookomp.c:0000830] Calling [GOMP_task()]
TRACE: [prepostfunctions.c:0000062] Calling [PRE_GOMP_task()]
TRACE: [prepostfunctions.c:0000067] Calling [POST_GOMP_task()]
Thread[2,139858983937792]: Trabalhando na task 2.
Thread[1,139858992330496]: Antes do taskwait.
TRACE: [ hookomp.c:0000848] Calling [GOMP_taskwait()]
Thread[1,139858992330496]: Depois do taskwait.
TRACE: [ hookomp.c:0000830] Calling [GOMP_task()]
TRACE: [prepostfunctions.c:0000062] Calling [PRE_GOMP_task()]
TRACE: [prepostfunctions.c:0000067] Calling [POST_GOMP_task()]
Thread[0,139859015989184]: Trabalhando na task 3.
TRACE: [prepostfunctions.c:0000037] Calling [POST_GOMP_parallel_end()]
Number of parallel regions: 1
Number of tasks: 3
Number of Finished tasks: 3
Thread[0,139859015989184]: Depois da região paralela.
rogerio@chamonix:/src/simple-omp-hook/tests/parallel-region-with-tasks$
```

3.5. Considerações Finais

Pelo fato do `OpenMP` ser um padrão amplamente utilizado em aplicações paralelas para sistemas *multicore* e com aceleradores, é importante conhecer sobre o seu funcionamento. É fundamental ter conhecimentos que vão além do uso das diretivas, ter ideia de como o código final é gerado, do seu formato e de como é executado. Pois em alguns casos não é simplesmente anotar o código, é necessário saber se o mesmo é paralelizável, um laço de repetição é um bom exemplo disso.

Mas ainda assim o uso de diretivas de compilação tem uma grande vantagem com relação ao uso de bibliotecas para criação de aplicações *multithreading* como a `pthreads`. A quantidade de código a ser escrito inserindo anotações nos devidos lugares é muito menor. Sem a preocupação de alterar o código de maneira que não seja mais compilado pelas ferramentas originais, pois se o compilador não reconhecer as diretivas elas são simplesmente ignoradas.

Existem diversas outras diretivas de compilação do `OpenMP` que não foram abordadas neste texto, mas que podem ser consultadas na documentção do `OpenMP` e utilizadas com outras implementações e ferramentas de compilação [OpenMP Site 2017] [OpenMP-ARB 2015].

Agradecimentos

O material desse minicurso foi preparado no âmbito dos projetos de Extensão *"Escola de Computação Paralela"* (`UTFPR DIREC Nº 028/2017`) e de Pesquisa *"Estudo Exploratório sobre Técnicas e Mecanismos para Paralelização Automática e Offloading de Código em Sistemas Heterogêneos"* (`UTFPR PDTI Nº 916/2017`).

Referências

[Blumofe et al. 1995] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216.

[Dagum and Menon 1998] Dagum, L. and Menon, R. (1998). OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

[Denise Stringhini 2012] Denise Stringhini, Rogério Aparecido Gonçalves, A. G. (2012). Introdução à Computação Heterogênea. In de Souza; Renata Galante; Roberto Cesar Junior; Aurora Pozo, L. C. A. A. F., editor, *XXXI Jornadas de Atualização em Informática (JAI)*, volume 21 of *1*, chapter 7, pages 262–309. SBC, 1 edition. http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=12580.

[GCC 2015] GCC (2015). GCC, the GNU Compiler Collection.

[GNU Libgomp 2015a] GNU Libgomp (2015a). GNU libgomp, GNU Offloading and Multi Processing Runtime Library documentation (Online manual).

[GNU Libgomp 2015b] GNU Libgomp (2015b). GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation. Technical report, GNU.

[GNU Libgomp 2016] GNU Libgomp (2016). GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation. Technical report, GNU libgomp.

[Gonçalves et al. 2016] Gonçalves, R., Amaris, M., Okada, T., Bruel, P., and Goldman, A. (2016). Openmp is not as easy as it appears. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5742–5751.

[Intel 2016a] Intel (2016a). Intel® OpenMP* Runtime Library Interface. Technical report, Intel. OpenMP* 4.5, https://www.openmprtl.org.

[Intel 2016b] Intel (2016b). OpenMP* Support. https://software.intel.com/pt-br/node/522678.

[Lattner and Adve 2004] Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, number c in CGO '04, pages 75–86, Palo Alto, California. IEEE Computer Society.

[LLVM OpenMP 2015] LLVM OpenMP (2015). OpenMP®: Support for the OpenMP language.

[Mohr et al. 2002] Mohr, B., Malony, A. D., Shende, S., and Wolf, F. (2002). Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128.

[Nichols et al. 1996] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads programming - a POSIX standard for better multiprocessing.* O'Reilly.

[NVIDIA 2017] NVIDIA (2017). CUDA C Best Practices Guide. Technical report, NVIDIA. DG-05603-001_v9.0, Version v9.0.176, http://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[OpenACC 2015] OpenACC (2015). OpenACC Application Programming Interface. Version 2.5. http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf.

[OpenACC 2017] OpenACC (2017). OpenACC – More Science, Less Programming. http://www.openacc.org/.

[OpenMP 2017] OpenMP (2017). OpenMP Compilers. http://www.openmp.org/resources/openmp-compilers/.

[OpenMP-ARB 2011] OpenMP-ARB (2011). OpenMP Application Program Interface Version 3.1. Technical report, OpenMP Architecture Review Board (ARB).

[OpenMP-ARB 2013] OpenMP-ARB (2013). OpenMP Application Program Interface Version 4.0. Technical report, OpenMP Architecture Review Board (ARB).

[OpenMP-ARB 2015] OpenMP-ARB (2015). OpenMP Application Program Interface Version 4.5. Technical report, OpenMP Architecture Review Board (ARB). Version 4.5.

[OpenMP Site 2017] OpenMP Site (2017). OpenMP® – Enabling HPC since 1997: The OpenMP API specification for parallel programming.

[PGROUP 2015] PGROUP (2015). PGI Accelerator Compilers with OpenACC Directives.

[Trahay et al. 2011] Trahay, F., Rue, F., Faverge, M., Ishikawa, Y., Namyst, R., and Dongarra, J. (2011). EZTrace: a generic framework for performance analysis. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Newport Beach, CA, United States. Poster Session.