Introdução à Programação Paralela com OpenMP

Além das Diretivas de Compilação

Rogério A. Gonçalves[1], João M. de Queiroz Filho[1] e Alfredo Goldman[2]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento de Computação (DACOM)
Campo Mourão – PR – Brasil

[2]Universidade de São Paulo (USP)
Instituto de Matemática e Estatística (IME)
Centro de Competência em Software Livre (CCSL)
São Paulo – SP – Brasil

rogerioag@utfpr.edu.br, joaomfilho1995@gmail.com e gold@ime.usp.br

XVIII Simpósio em Sistemas Computacionais de Alto Desempenho

Agenda

1. Introdução
2. Implementações do OpenMP
3. Diretivas de Compilação
4. Aplicações
5. Considerações Finais
6. Referências

Material do Minicurso

Material

Slides e Código dos Exemplos

`https://github.com/rogerioag/minicurso-openmp-wscad-2017`

Objetivos

- Apresentar uma introdução ao `OpenMP`
- Mostrando o código pós expansão das diretivas de compilação

Programação Paralela

- Grande parte das linguagens de programação apresentam mecanismos para o uso de *threads*.
- Criação de *threads* (`spawn`) e sincronização do trabalho (`join, sync`).
- São recursos nativos ou por extensões das linguagens.
- As implementações podem ser de baixo nível ou de mais alto nível.
- A `pthread` disponível em sistemas `GNU/Linux`
  - criação de *threads*: *`pthread_create(...)`*
  - sincronização do trabalho: *`pthread_join()`*.
- `Cilk` fornece também funções:
  - para criar *threads*: *`cilk_spawn`*
  - sincronização: *`cilk_sync`*
  - paralelização de laços: *`cilk_for`*.

Padrão OpenMP I

- O `OpenMP`[1] é um padrão bem conhecido e amplamente utilizado em aplicações para plataformas *multicore* e *manycores*.
- A abordagem do `OpenMP` é de mais alto nível
  - Uso de *diretivas de compilação*.
  - As diretivas funcionam como anotações no código.
  - Código anotado e não reescrito.
- São implementadas usando-se as diretivas de pré-processamento `#pragma`, em `C/C++`, e sentinelas `!$`, no `Fortran`.

Padrão OpenMP II

- O `OpenMP` trabalha em sistemas com memória compartilhada e implementa o modelo `fork-join`

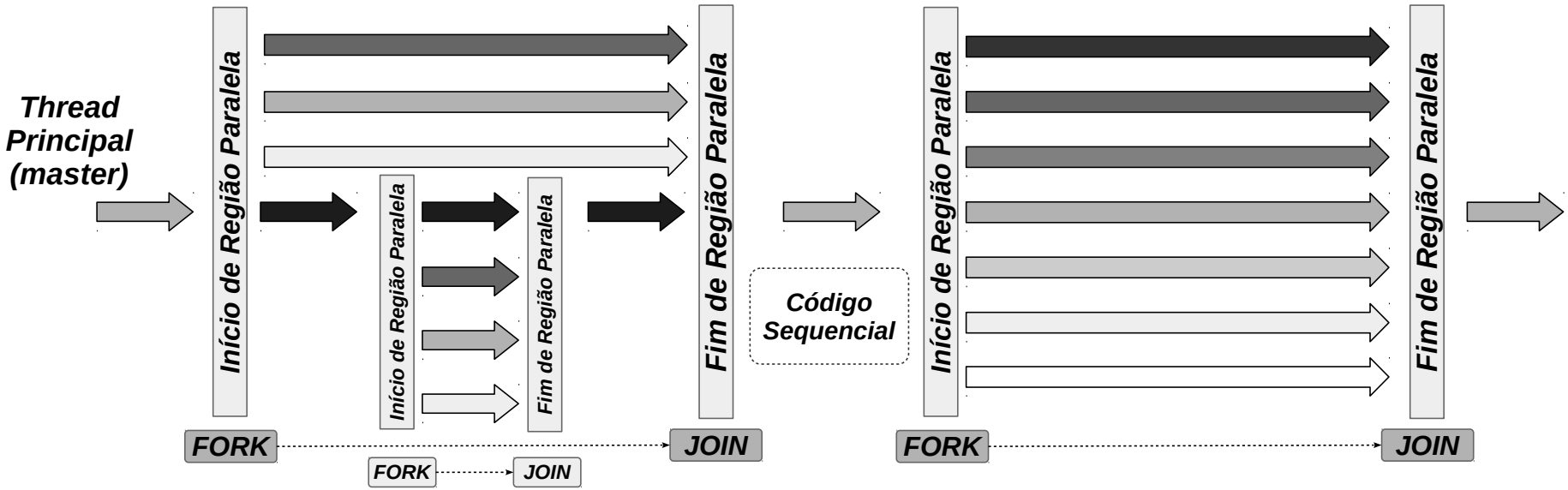


Figura 1: Modelo `Fork-join` baseado em regiões paralelas

- Múltiplas *threads* executam tarefas definidas implicitamente ou explicitamente.

Padrão OpenMP III

- As diretivas são substituídas pelo seu formato de código expandido com as chamadas para o *runtime* do `OpenMP`.
- Nosso estudo está baseado nas diretivas de compilação da biblioteca `libgomp`[2] do `GCC`.
- A motivação desse estudo foi a necessidade de interceptar código de aplicações `OpenMP` para decidir sobre *offloading* de código para aceleradores.

[1] Dagum and Menon (1998); OpenMP-ARB (2015); OpenMP Site (2017)
[2] GNU Libgomp (2016a,b)

Implementações do OpenMP I

- O `OpenMP` tem sido suportado por praticamente todos os compiladores atuais.
- Compiladores como `GCC`[3], Intel `icc`[4] e `LLVM clang`[5] tem implementações para `OpenMP`.
- Pelo menos duas implementações: `GNU GCC libgomp`[6] e a `Intel libomp` *(OpenMP* Runtime Library)* [7].
- A especificação do `OpenMP` atualmente cobre *offloading* de código para aceleradores.
- A `libgomp` é capaz de fazer *offloading* usando o padrão `OpenACC`[8].

[3] GCC (2015); GNU Libgomp (2015a)
[4] Intel (2016b)
[5] Lattner and Adve (2004); LLVM Clang (2015); LLVM OpenMP (2015)
[6] GNU Libgomp (2015b,c, 2016a,b)
[7] Intel (2016a)
[8] OpenACC (2015, 2017)

Diretivas de Compilação e Código OpenMP Expandido

(Formato pós expansão das diretivas)

LibGOMP: *GNU OpenMP Runtime Library*
(GNU Offloading and Multi Processing Runtime Library)

Diretivas de Compilação I

- As *diretivas de compilação* são formadas por construtores e cláusulas
- Durante a fase de pré-processamento, são substituídos por uma versão de código expandido.
- O formato de código estruturado é composto de chamadas às funções do *runtime* do `OpenMP`.
- Verificamos o formato de código gerado para os construtores:
  - Regiões paralelas (`parallel region`)
  - Laços (`for`)
  - Tarefas explícitas (`task`)
  - Tarefas com laços (`taskloop`)
  - Vetorização (`simd`)
  - Aceleradores (`target`)

Regiões paralelas
construtor parallel

Regiões paralelas: construtor `parallel` I

- **Diretiva:** `#pragma omp parallel`
- Esta é uma das mais importantes diretivas, pois ela é responsável pela demarcação de regiões paralelas.

Sintaxe

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

- Quando uma região paralela é encontrada, é criado um time de *threads* para executar o código da região.
- Porém, esse construtor não divide o trabalho entre as *threads*.

Regiões paralelas: construtor `parallel` I

- Quando uma região paralela é declarada:

```
1 #pragma omp parallel
2 {
3   // bloco
4 }
```

- O construtor `parallel` é implementado criando-se uma nova função com o código do `bloco` – *outlined function*.
- Chamadas às funções da `libgomp` são colocadas no código para delimitar a região paralela:

ABI da `libgomp`

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

Regiões paralelas: construtor `parallel` II

- Cláusulas permitidas com o construtor `parallel`:

Cláusulas

```
1 if(expression)
2 num_threads(integer-expression)
3 default(shared|none)
4 private(list)
5 firstprivate(list)
6 shared(list)
7 copyin(list)
8 reduction(reduction-identifier:list)
9 proc_bind(master|close|spread)
```

Regiões paralelas: construtor `parallel` III

- Após a expansão da diretiva o código gerado assume o formato:

```
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();
```

Regiões paralelas: construtor `parallel` IV

- Obtendo o código intermediário `GIMPLE` (GCC):

Terminal

```
$ gcc -fopenmp -fdump-tree-all parallel-region.c
$ gcc -fopenmp -fdump-tree-ompexp parallel-region.c
$
```

- Gerando a visualização do código `GIMPLE`:

Terminal

```
$ gcc -fopenmp -fdump-tree-ompexp-graph parallel-region.c
$
```

Regiões paralelas: construtor `parallel` V

- O código na representação intermediária `GIMPLE`:

```
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 }
```

Regiões paralelas: construtor `parallel` VI

- O código em *assembly*:

Terminal

```
$ gcc -fopenmp -S parallel-region.c parallel-region.S
$
```

```
 1  .file "parallel-region.c"
 2  .text
 3  .globl  main
 4  .type main, @function
 5
 6  ; Código da Função main.
 7  main:
 8    pushq %rbp
 9    movq  %rsp, %rbp
10    movl  $0, %edx
11    movl  $0, %esi
12    movl  $main._omp_fn.0, %edi
13    call  GOMP_parallel_start
14    movl  $0, %edi
15    call  main._omp_fn.0
16    call  GOMP_parallel_end
17    movl  $0, %eax
18    popq  %rbp
19  ret
20  .size main, .-main
21
22  ; Código da nova função.
23  .type main._omp_fn.0, @function
24  main._omp_fn.0:
25    pushq %rbp
26    movq  %rsp, %rbp
27    movq  %rdi, -8(%rbp)
28    popq  %rbp
29    ret
30  .size main._omp_fn.0, .-main._omp_fn.0
31  .ident  "GCC: (Debian 4.8.4-1) 4.8.4"
32  .section  .note.GNU-stack,"",@progbits
```

Regiões paralelas: construtor `parallel` VII

- 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     }
```

Regiões paralelas: construtor `parallel` VIII

```
1       #pragma omp single
2       {
3          printf("Thread[%d][%lu]: Código Executado por uma das threads.\n",
                id, (long int) pthread_self());
4       }
5
6       if(omp_get_thread_num() == 3){
7          printf("Thread[%d][%lu]: Código Executado pela thread de id: 3.\n"
                , id, (long int) pthread_self());
8       }
9
10      #pragma omp critical
11      {
12         printf("Thread[%d][%lu]: Executando a região crítica.\n", id, (
               long int) pthread_self());
13         printf("Thread[%d][%lu]: Antes... valor: %d\n", id, (long int)
               pthread_self(), valor);
14         valor = valor + id;
15         printf("Thread[%d][%lu]: Depois.. valor: %d\n", id, (long int)
               pthread_self(), valor);
16      }
```

Regiões paralelas: construtor `parallel` IX

```
1       printf("Thread[%d][%lu]: Barreira.\n", id, (long int)
            pthread_self());
2
3       #pragma omp barrier
4       printf("Thread[%d][%lu]: Depois da barreira.\n", id, (long
            int) pthread_self());
5     }
6
7     printf("Thread[%d][%lu]: Depois da Região Paralela.\n",
          omp_get_thread_num(), (long int) pthread_self());
8
9     return 0;
10  }
```

Regiões paralelas: construtor `parallel` X

Terminal

```
rogerio@chamonix:/src/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/parallel-with-clauses$
```

Loops
construtor for

Loops: Construtor `for` I

- **Diretiva:** `#pragma omp 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 – *outlined function*.
- É necessário compartilhar o trabalho e coordenar a execução paralela.
- O construtor `for` é usado para distribuir o trabalho entre as *threads*.

Construtor *for*:

```
#pragma omp for schedule ({auto, static, dynamic, guided,
    runtime}, {variable/expression | numerical value/constant})
```

Loops: Construtor `for` II

- Semelhante ao que ocorre no processamento do construtor *parallel* individualmente, um construtor `parallel` com um construtor `for` ou o formato combinado `parallel for` também é implementado com a criação de uma nova função.
- O Código das duas regiões paralelas são equivalentes.

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

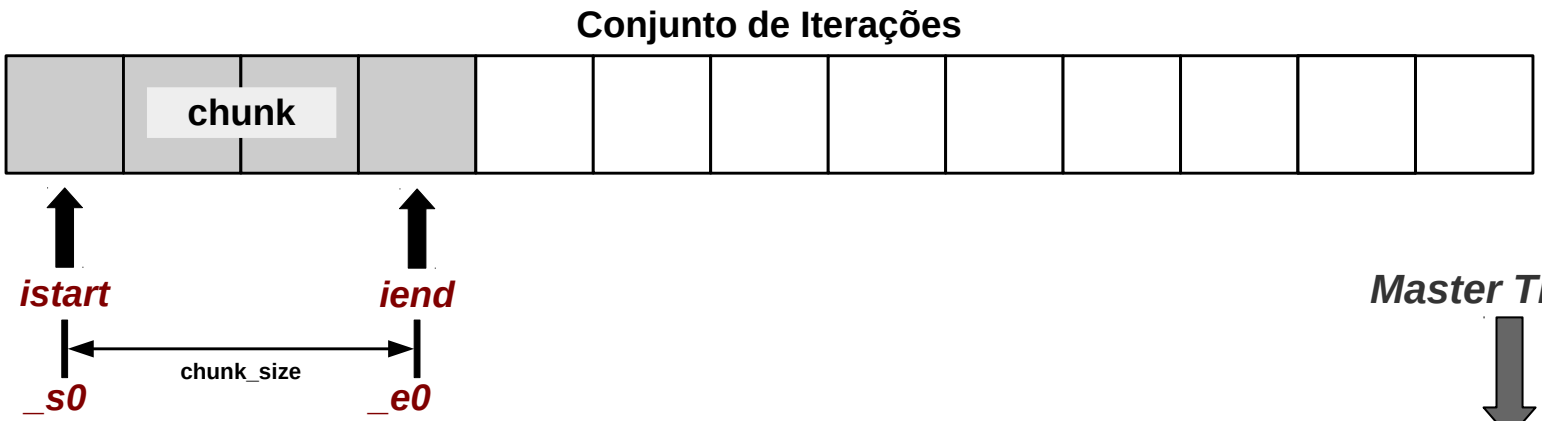
```
1 #pragma omp parallel for
2 for (i = lb; i <= ub; i++){
3   body;
4 }
```

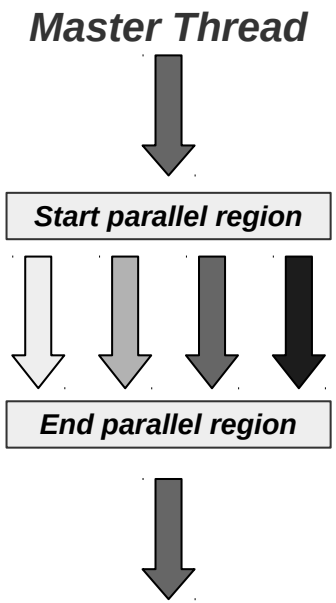
Loops: Construtor `for` III

- O conjunto de iterações é dividido de acordo com um algoritmo de escalonamento.
- O escalonamento é definido usando-se a cláusula `schedule`
- O tipos que estão disponíveis no `OpenMP` são:
  - *static*
  - *auto*
  - *runtime*
  - *dynamic*
  - *guided*
- Quando não se especifica um escalonamento o código é equivalente ao gerado para `schedule(static)`.

Loops: Construtor `for` IV

- Como as *threads* executam os subconjuntos de iterações *(chunks)* de um laço:






```
void subfunction (void *data){
  long _s0, _e0;
  while (GOMP_loop_<<schedule_type>>_next (&_s0, &_e0)){
    long _e1 = _e0, i;
    for (i = _s0; i < _e1; i++)
      body;
  }
  GOMP_loop_end_nowait ();
}

GOMP_parallel_loop_<<schedule_type>>_start (subfunction, NULL, 0, lb, ub+1, 1, 0);
subfunction (NULL);
GOMP_parallel_end ();
```

Loop Original

Loops: Construtor `for` V

- **Exemplo:** Quando não é especificado o algoritmo de escalonamento

```
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 }
```

Loops: Construtor `for` VI

- O particionamento do conjunto de iterações é estático.
- 16 iterações $\div$ 4 *threads*

Terminal

```
rogerio@chamonix:/src/for-construtor$ ./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/for-constructor$
```

Loops: Construtor `for` VII

- **Exemplo:** Definindo o escalonamento como `schedule(static,2)`:

```
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(static,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
17   printf("Thread[%d][%lu]: Depois da Região Paralela.\n",
         omp_get_thread_num(), (long int) pthread_self());
18
19   return 0;
20 }
```

Loops: Construtor `for` VIII

- O particionamento do conjunto de iterações é estático.
- 16 iterações $\div$ 4 *threads*
- Atribuição das iterações segue o *chunk_size* em um *round-robin* nas *threads*.

Terminal

```
rogerio@chamonix:/src/example-for$ ./example-for-constructor-static-with-chunk.exe
Thread[0][18446744073543477120]: Antes da Região Paralela.
Thread[0][18446744073543477120]: Trabalhando na iteração 0.
Thread[0][18446744073543477120]: Trabalhando na iteração 1.
Thread[0][18446744073543477120]: Trabalhando na iteração 8.
Thread[0][18446744073543477120]: Trabalhando na iteração 9.
Thread[2][18446744073522579200]: Trabalhando na iteração 4.
Thread[2][18446744073522579200]: Trabalhando na iteração 5.
Thread[2][18446744073522579200]: Trabalhando na iteração 12.
Thread[2][18446744073522579200]: Trabalhando na iteração 13.
Thread[3][18446744073514186496]: Trabalhando na iteração 6.
Thread[3][18446744073514186496]: Trabalhando na iteração 7.
Thread[3][18446744073514186496]: Trabalhando na iteração 14.
Thread[3][18446744073514186496]: Trabalhando na iteração 15.
Thread[1][18446744073530971904]: Trabalhando na iteração 2.
Thread[1][18446744073530971904]: Trabalhando na iteração 3.
Thread[1][18446744073530971904]: Trabalhando na iteração 10.
Thread[1][18446744073530971904]: Trabalhando na iteração 11.
Thread[0][18446744073543477120]: Depois da Região Paralela.
rogerio@chamonix:/src/example-for$
```

Loops: Construtor `for` IX

Terminal

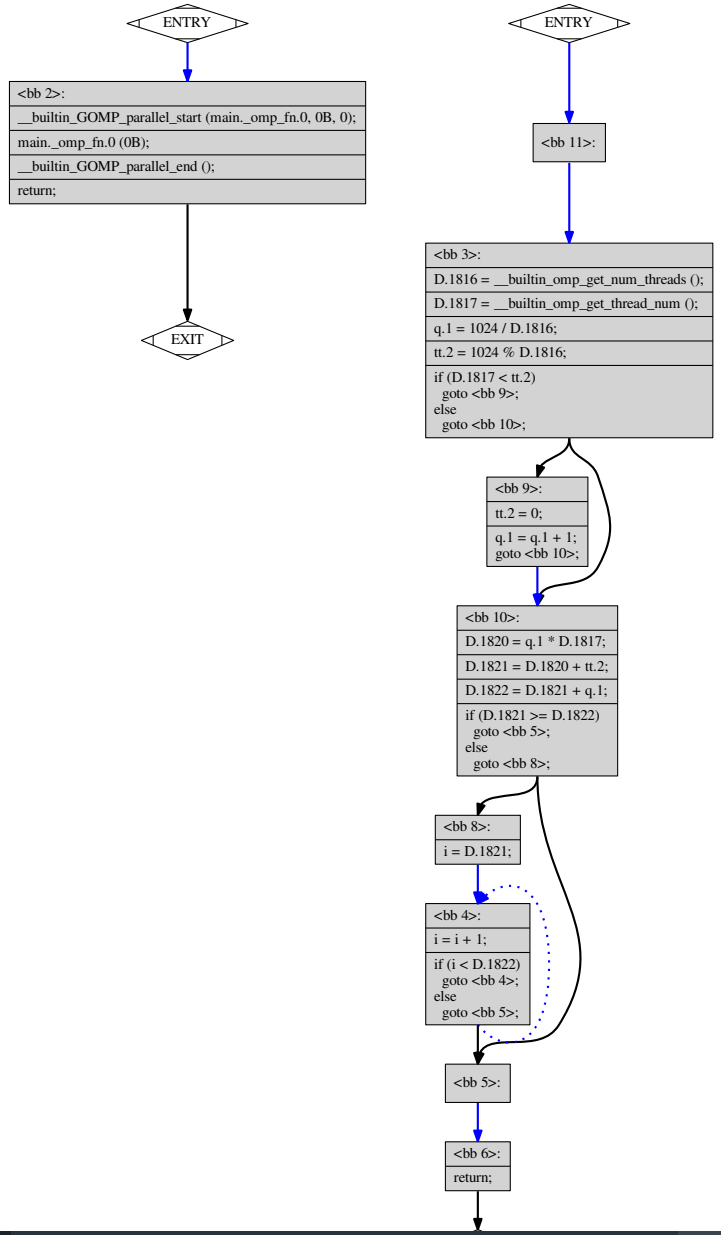
```
$ gcc -fopenmp -fdump-tree-ompexp for.c
$ gcc -fopenmp -fdump-tree-ompexp-graph for.c
```

Loops: Construtor `for` X

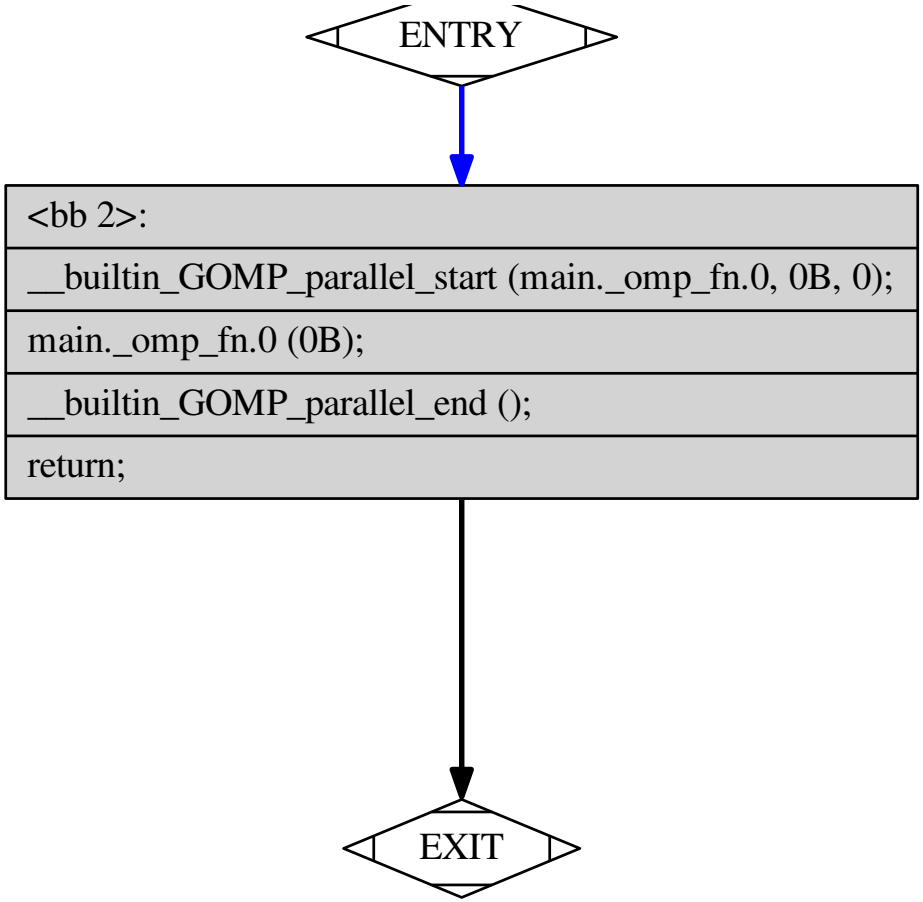
- O código em `GIMPLE`, que é a representação intermediária do `GCC`:

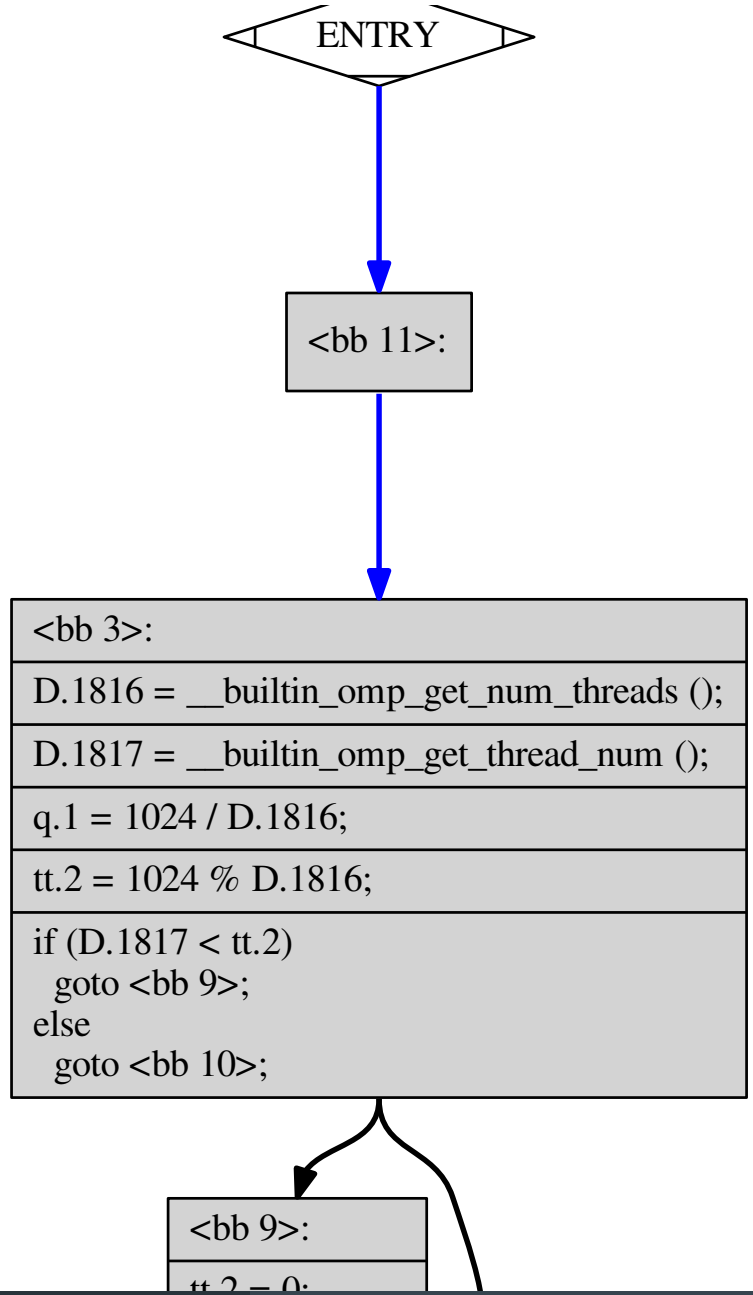
```
1  main () {
2
3  <bb 2>:
4    __builtin_GOMP_parallel_start (main.
         _omp_fn.0, 0B, 0);
5    main._omp_fn.0 (0B);
6    __builtin_GOMP_parallel_end ();
7    return;
8  }
9
10 main._omp_fn.0(void* .omp_data_i){
11
12 <bb 11>:
13
14 <bb 3>:
15   D.1816 =
         __builtin_omp_get_num_threads
         ();
16   D.1817 =
         __builtin_omp_get_thread_num()
         ;
17   q.1 = 1024 / D.1816;
18   tt.2 = 1024 % D.1816;
19   if (D.1817 < tt.2)
20     goto <bb 9>;
21   else
22     goto <bb 10>;
23
24 <bb 10>:
25   D.1820 = q.1 * D.1817;
26   D.1821 = D.1820 + tt.2;
27   D.1822 = D.1821 + q.1;
28   if (D.1821 >= D.1822)
29     goto <bb 5>;
30   else
31     goto <bb 8>;
32
33 <bb 8>:
34   i = D.1821;
35
36 <bb 4>:
37   i = i + 1;
38   if (i < D.1822)
39     goto <bb 4>;
40   else
41     goto <bb 5>;
42
43 <bb 5>:
44
45 <bb 6>:
46   return;
47
48 <bb 9>:
49   tt.2 = 0;
50   q.1 = q.1 + 1;
51   goto <bb 10>;
52 }
```

Loops: Construtor `for`

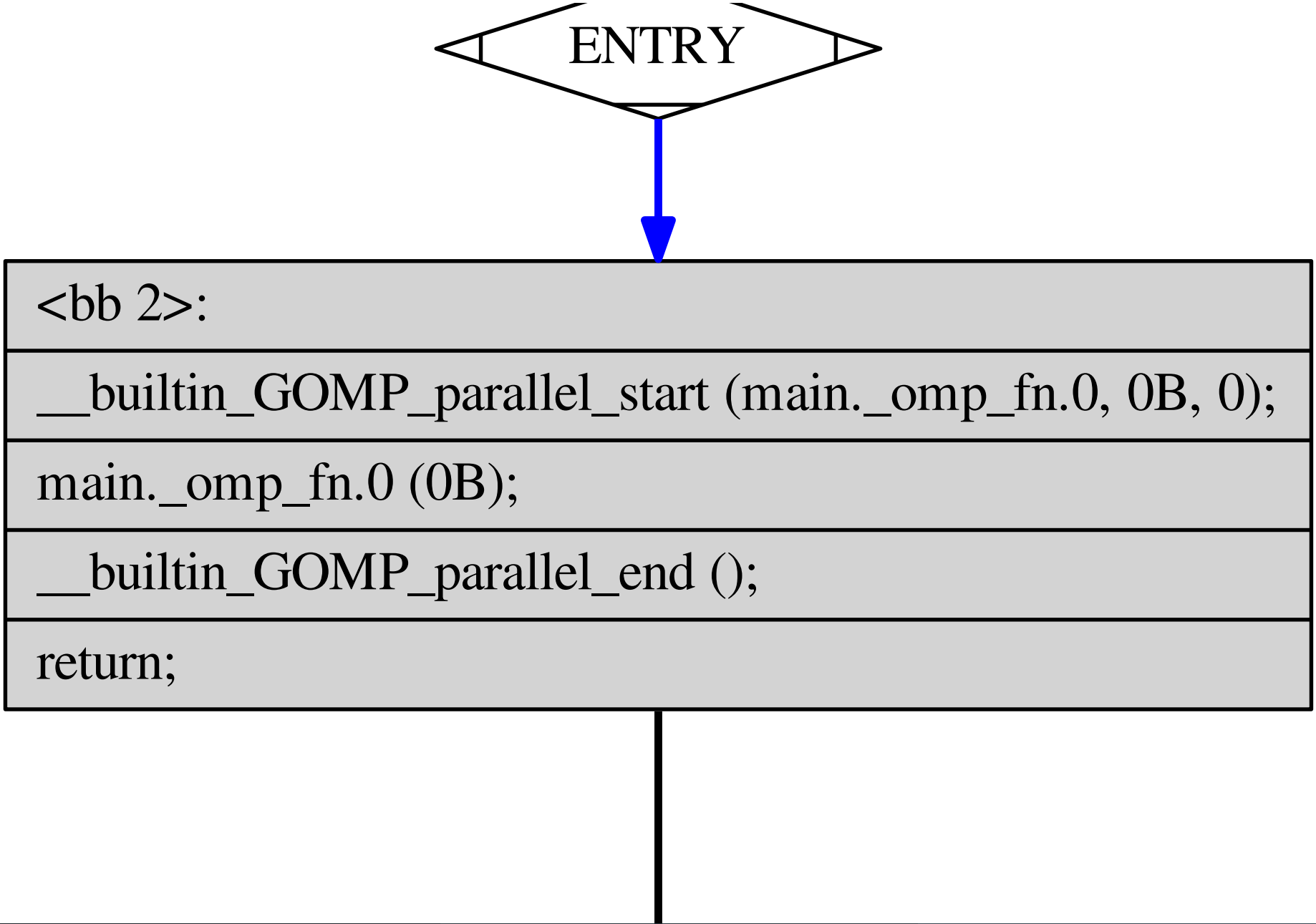


Loops: Construtor `for`

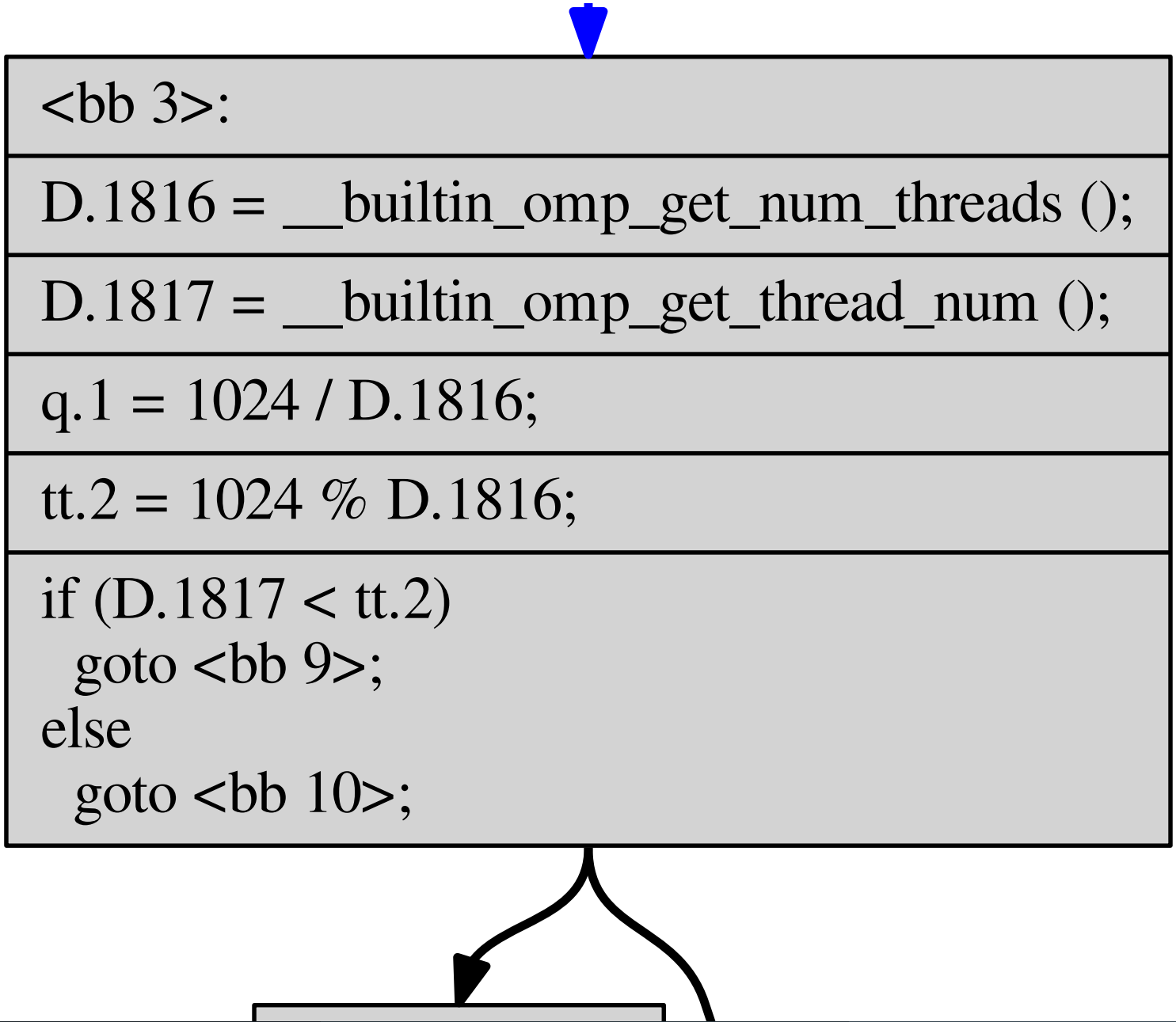




Loops: Construtor `for`



Loops: Construtor `for`



Loops: Construtor `for` I

- **Exemplo:** Laço anotado com o construtor `for` e com a cláusula `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      // Todas as threads executam esse código.
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   return 0;
18 }
```

Loops: Construtor `for` II

- A saída produzida pela execução com escalonamento *dynamic*:

Terminal

```
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$
```

Loops: Construtor `for` – Formatos I

- Os códigos diferem apenas na definição do limite superior dos laços.
- «schedule_type»: *dynamic*, *runtime* e *guided*

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

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

Loops: Construtor `for` – Primeiro formato I

- Para laços que utilizam escalonamentos dos tipos *dynamic, runtime* e *guided* – «schedule_type»:

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

- A `libgomp` usa as funções para delimitar a região paralela de código e criar o *primeiro* formato do *loop*:

ABI `libgomp` – funções usadas no primeiro formato do *parallel for*

```
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);
```

Loops: Construtor `for` – Primeiro formato II

- O código expandido para o *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 GOMP_parallel_loop_<<schedule_type>>_start (subfunction,
       NULL, 0, lb, ub+1, 1, 0);
13 subfunction (NULL);
14 GOMP_parallel_end ();
```

Loops: Construtor `for` – Primeiro formato III

- O código `GIMPLE` para o *primeiro formato*:

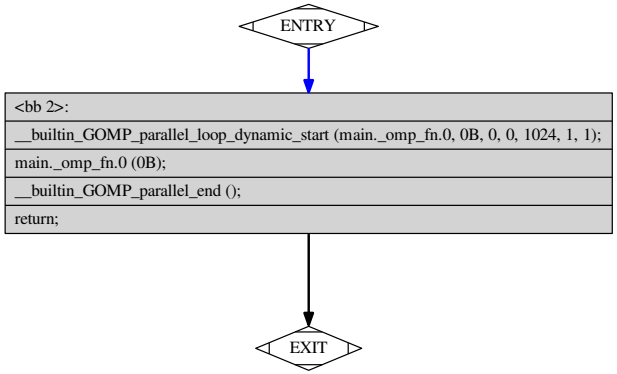
```
1  main () {
2
3  <bb 2>:
4    __builtin_GOMP_parallel_loop_dynamic_start (main._omp_fn.0, 0B, 0, 0,
         1024, 1, 1);
5    main._omp_fn.0 (0B);
6    __builtin_GOMP_parallel_end ();
7    return;
8  }
9
10 main._omp_fn.0 (void * .omp_data_i) {
11
12 <bb 10>:
13
14 <bb 3>:
15   D.1818 = __builtin_GOMP_loop_dynamic_next (&.istart0.1, &.iend0.2);
16   if (D.1818 != 0)
17     goto <bb 8>;
18   else
19     goto <bb 5>;
20
21 <bb 8>:
22   .istart0.3 = .istart0.1;
23   i = (int) .istart0.3;
24   .iend0.4 = .iend0.2;
25   D.1822 = (int) .iend0.4;
```

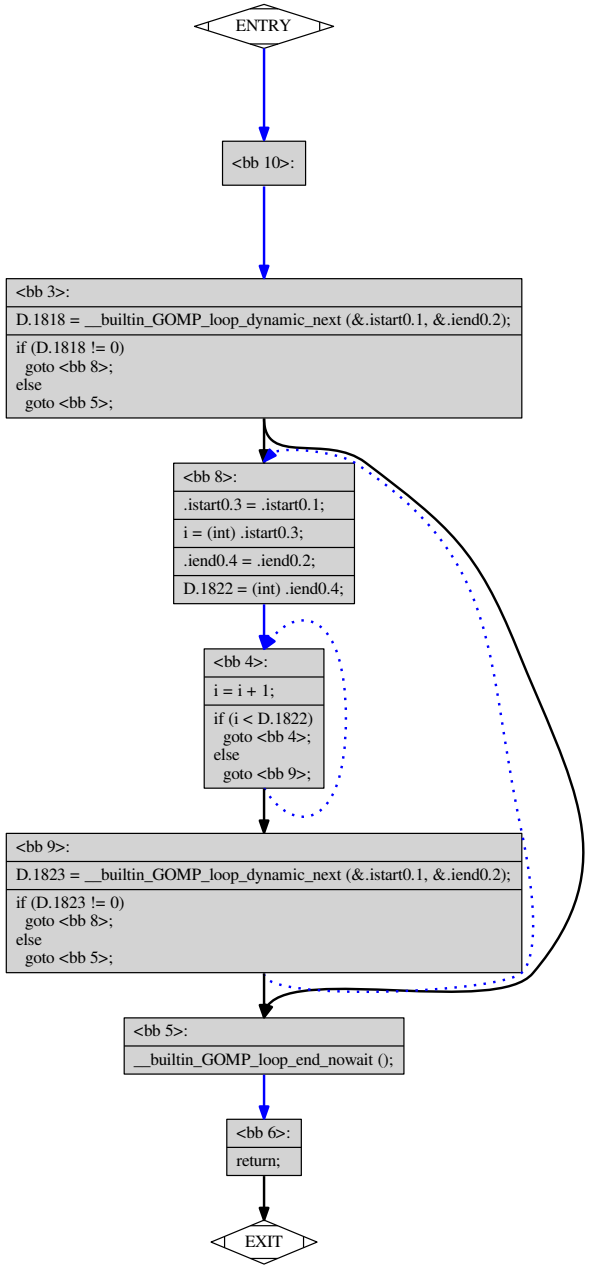
Loops: Construtor `for` – Primeiro formato IV

- O código `GIMPLE` para o *primeiro formato*:

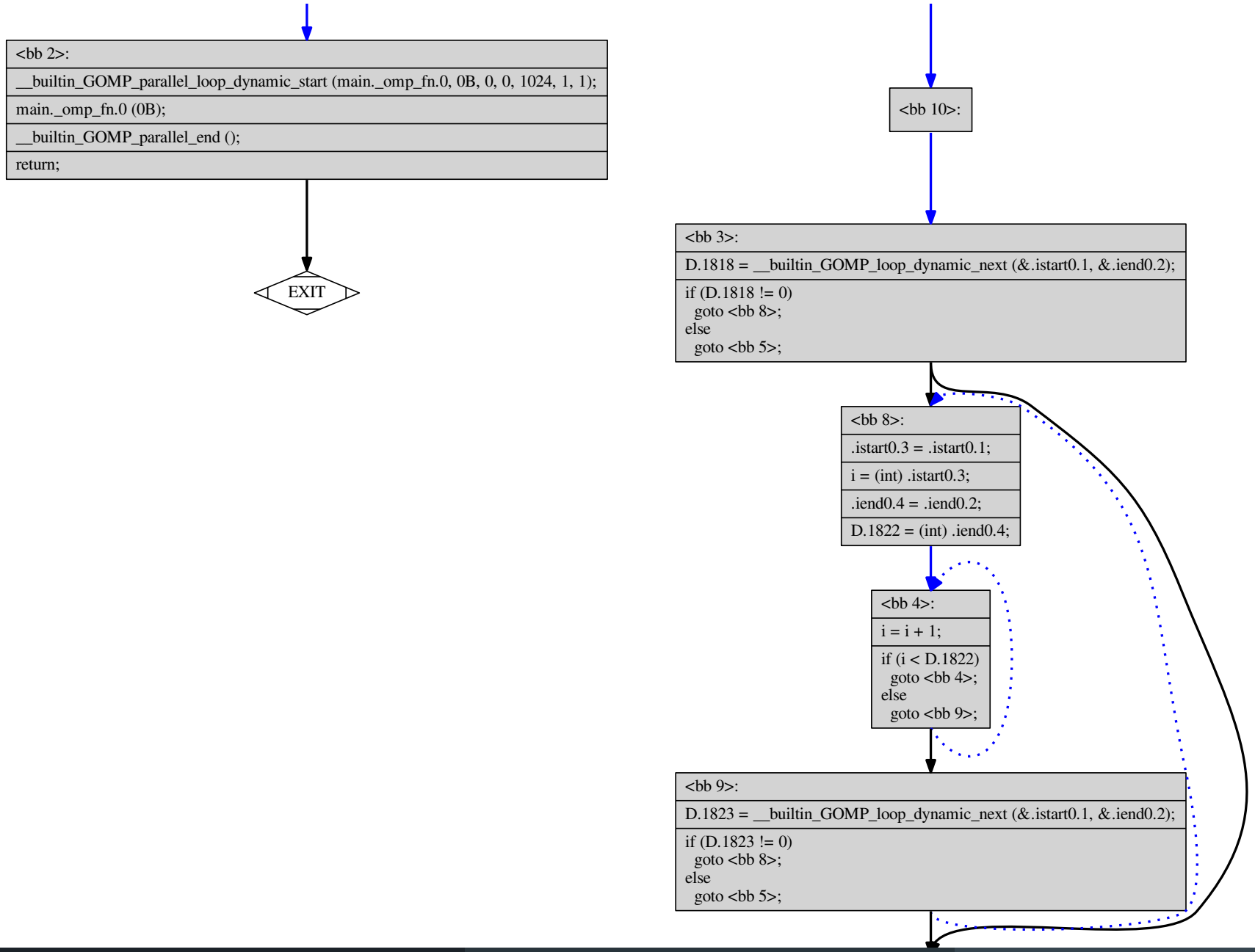
```
1  <bb 4>:
2    i = i + 1;
3    if (i < D.1822)
4      goto <bb 4>;
5    else
6      goto <bb 9>;
7
8  <bb 9>:
9    D.1823 = __builtin_GOMP_loop_dynamic_next (&.istart0.1, &.iend0.2);
10   if (D.1823 != 0)
11     goto <bb 8>;
12   else
13     goto <bb 5>;
14
15 <bb 5>:
16   __builtin_GOMP_loop_end_nowait ();
17
18 <bb 6>:
19   return;
20 }
```

Loops: Construtor `for` – Primeiro formato

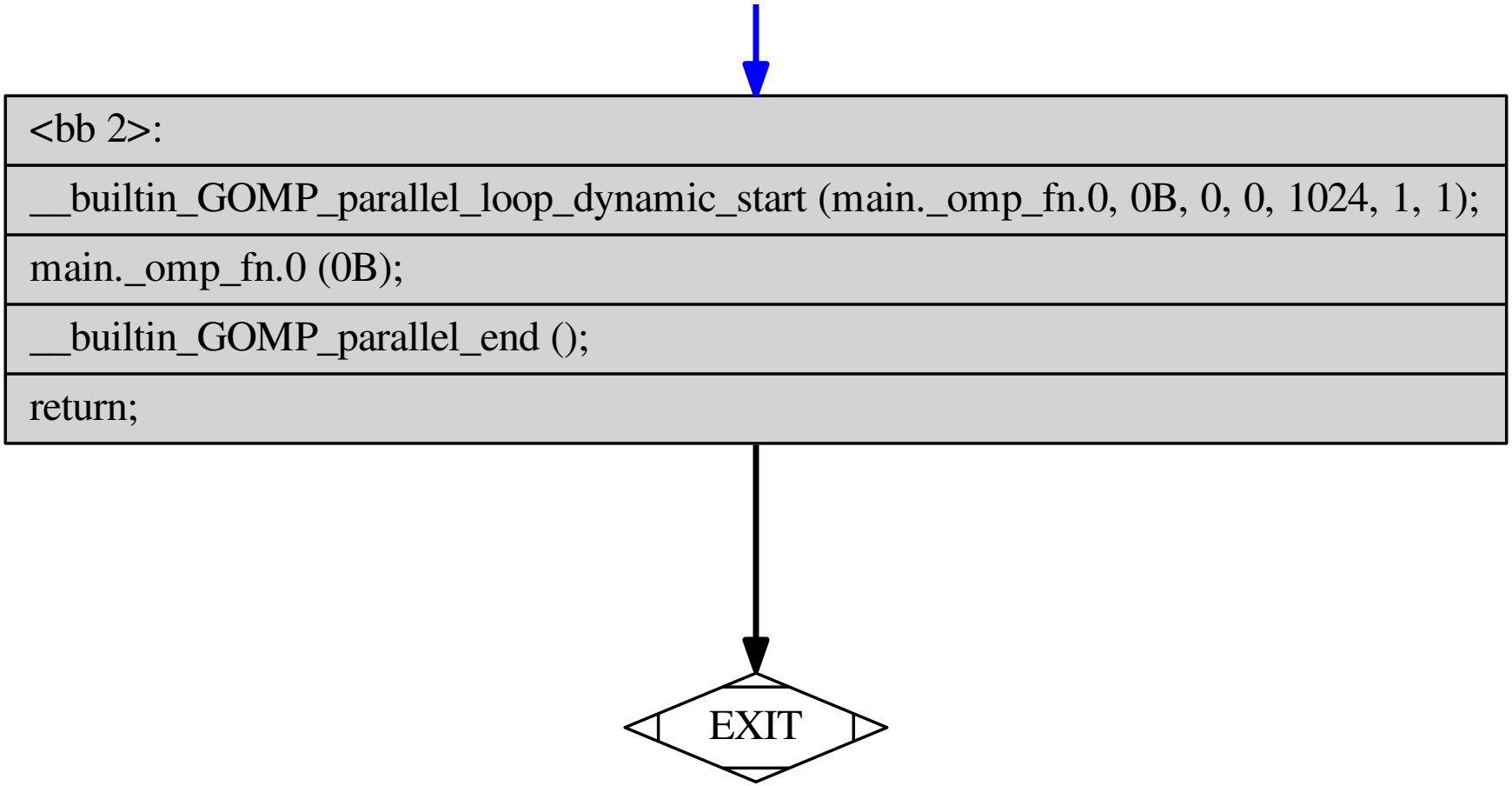




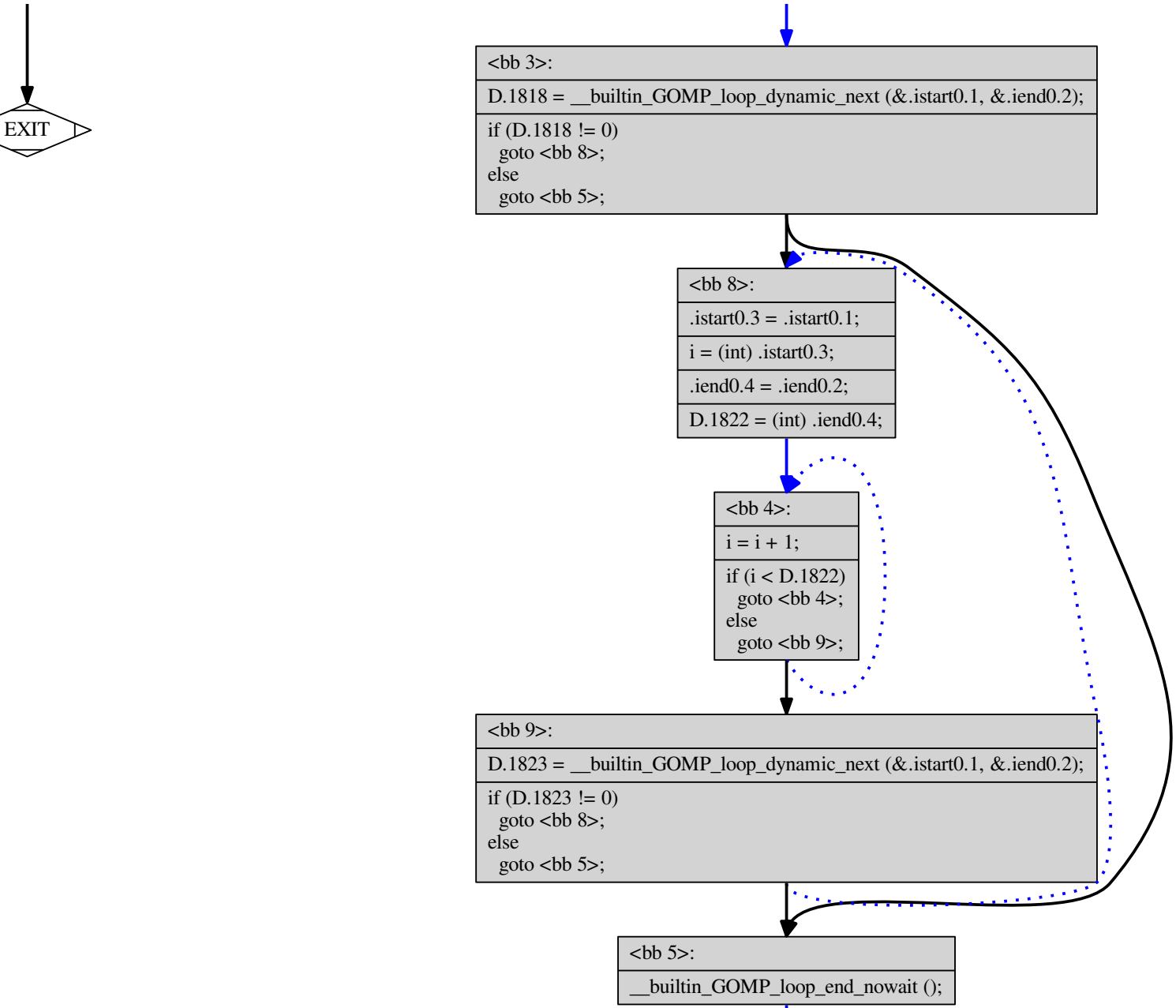
Loops: Construtor `for` – Primeiro formato



Loops: Construtor `for` – Primeiro formato



Loops: Construtor `for` – Primeiro formato



Loops: Construtor `for` – Segundo formato I

- Para laços que utilizam escalonamentos dos tipos *dynamic*, *runtime* e *guided* – «schedule_type»:

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

- A `libgomp` usa as funções para delimitar a região paralela de código e criar o *segundo* formato do *loop*:

ABI `libgomp` – funções usadas no segundo formato do *parallel for*

```
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);
```

Loops: Construtor `for` – Segundo formato II

- O código expandido para o *segundo formato*:

```
1  void subfunction (void *data){
2    long i, _s0, _e0;
3    if (GOMP_loop_runtime_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_runtime_next (&_s0, &_e0));
10   }
11   GOMP_loop_end ();
12 }
13 /* The annoted loop is replaced. */
14 GOMP_parallel_loop_static (subfunction, NULL, 0, lb,
       ub+1, 1, 0);
15 subfunction (NULL);
16 GOMP_parallel_end ();
```

Loops: Construtor `for` – Segundo formato III

- O código GIMPLE para o *segundo formato*:

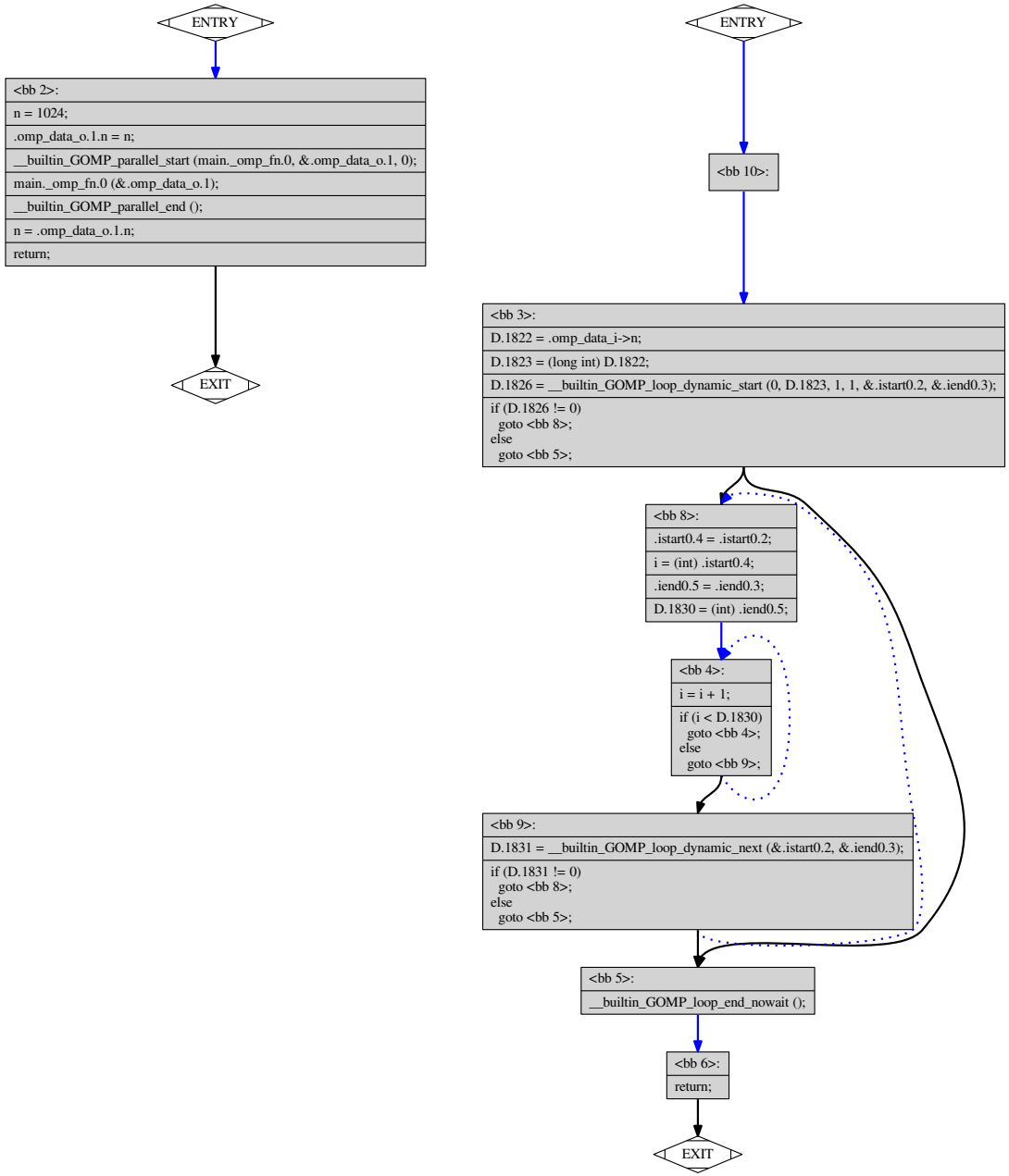
```
1  main () {
2    struct .omp_data_s.0 .omp_data_o.1;
3
4  <bb 2>:
5    n = 1024;
6    .omp_data_o.1.n = n;
7    __builtin_GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 0);
8    main._omp_fn.0 (&.omp_data_o.1);
9    __builtin_GOMP_parallel_end ();
10   n = .omp_data_o.1.n;
11   return;
12 }
13
14 main._omp_fn.0 (struct .omp_data_s.0 * .omp_data_i) {
15
16 <bb 10>:
17
18 <bb 3>:
19   D.1822 = .omp_data_i->n;
20   D.1823 = (long int) D.1822;
21   D.1826 = __builtin_GOMP_loop_dynamic_start (0, D.1823, 1, 1, &
         .istart0.2, &.iend0.3);
22   if (D.1826 != 0)
23     goto <bb 8>;
24   else
25     goto <bb 5>;
```

Loops: Construtor `for` – Segundo formato IV

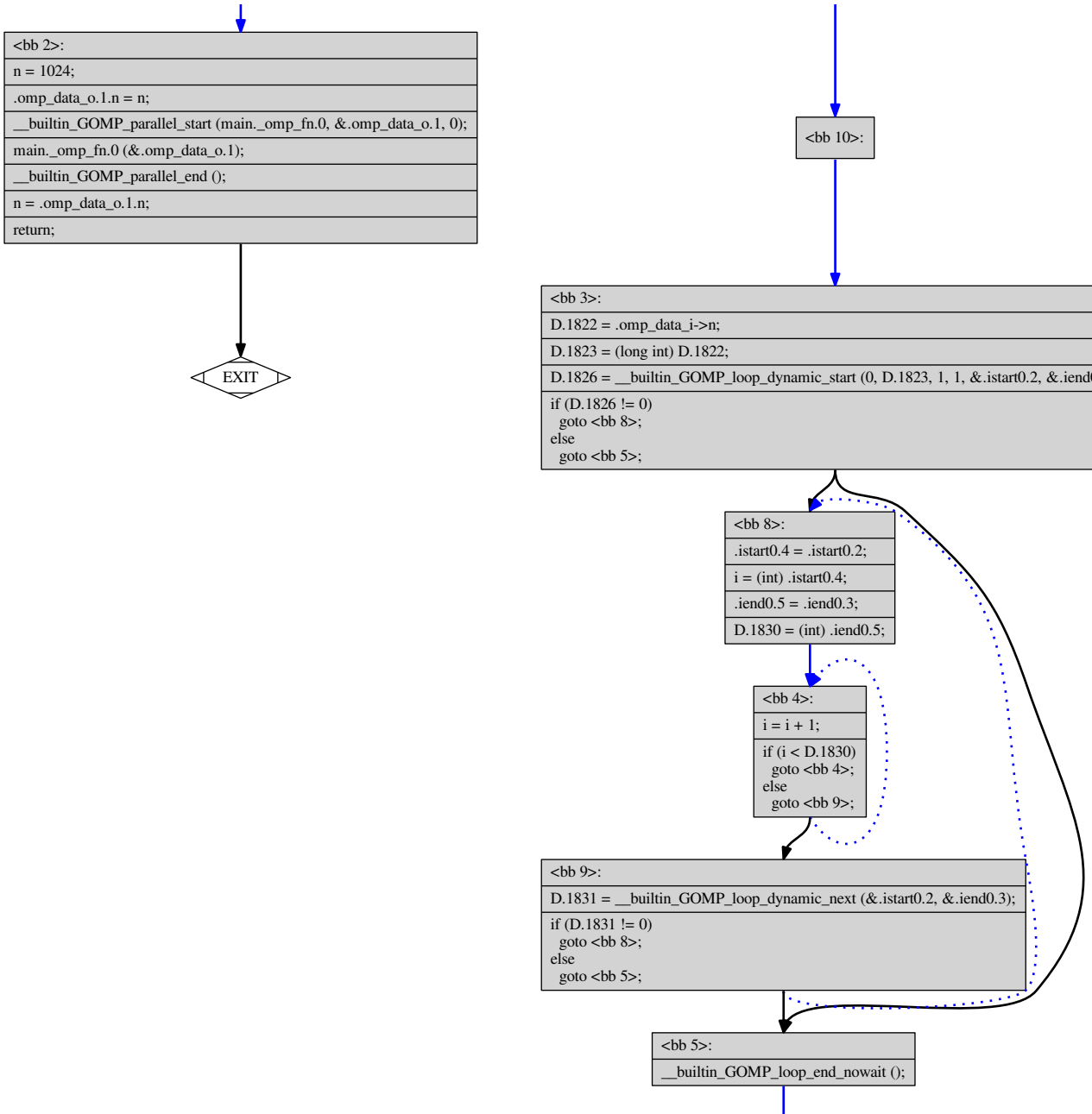
- O código `GIMPLE` para o segundo formato:

```
1  <bb 8>:
2    .istart0.4 = .istart0.2;
3    i = (int) .istart0.4;
4    .iend0.5 = .iend0.3;
5    D.1830 = (int) .iend0.5;
6
7  <bb 4>:
8    i = i + 1;
9    if (i < D.1830)
10     goto <bb 4>;
11   else
12     goto <bb 9>;
13
14 <bb 9>:
15   D.1831 = __builtin_GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);
16   if (D.1831 != 0)
17     goto <bb 8>;
18 else
19     goto <bb 5>;
20
21 <bb 5>:
22   __builtin_GOMP_loop_end_nowait ();
23
24 <bb 6>:
25   return;
26 }
```

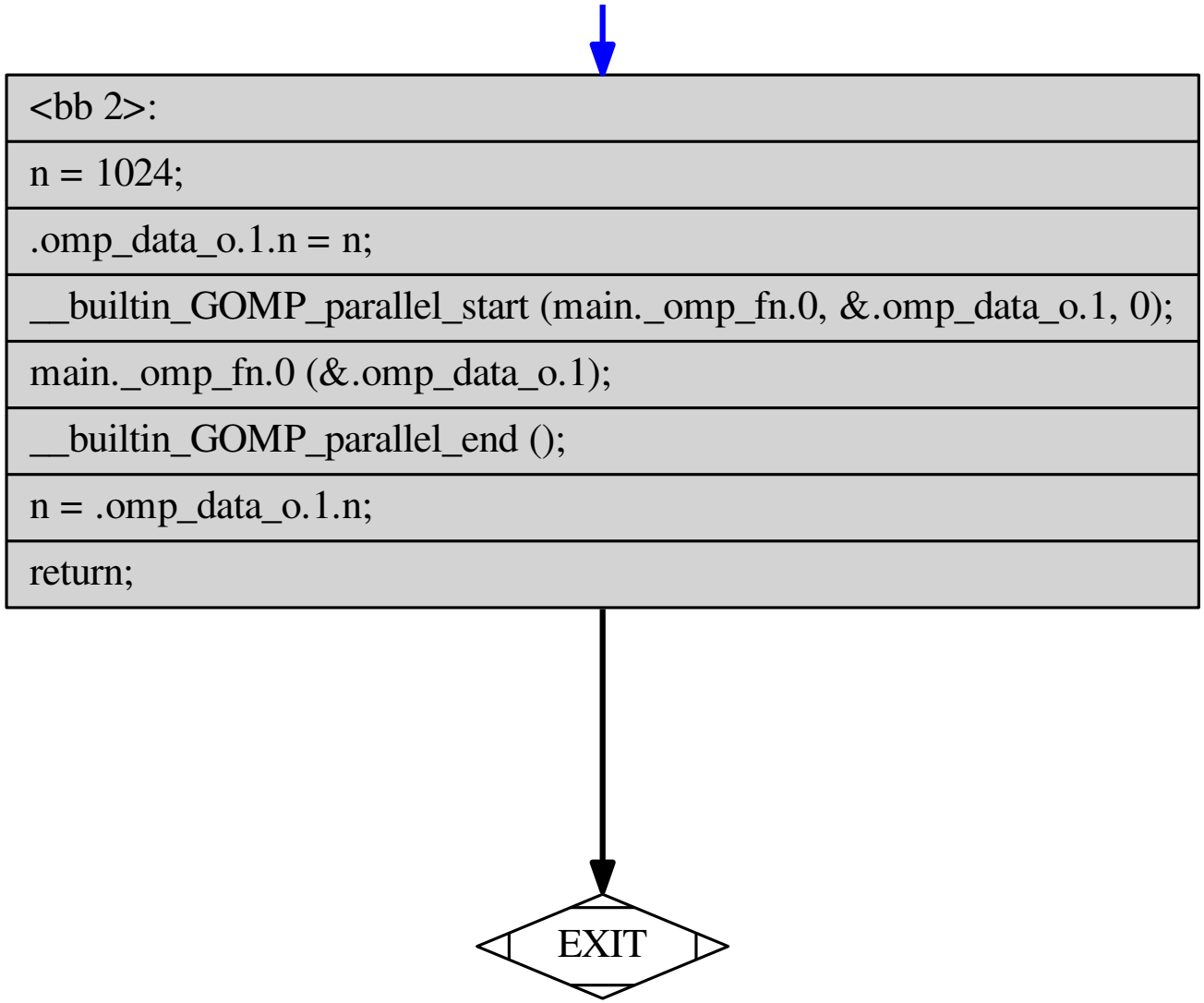
Loops: Construtor `for` – Segundo formato



Loops: Construtor `for` – Segundo formato

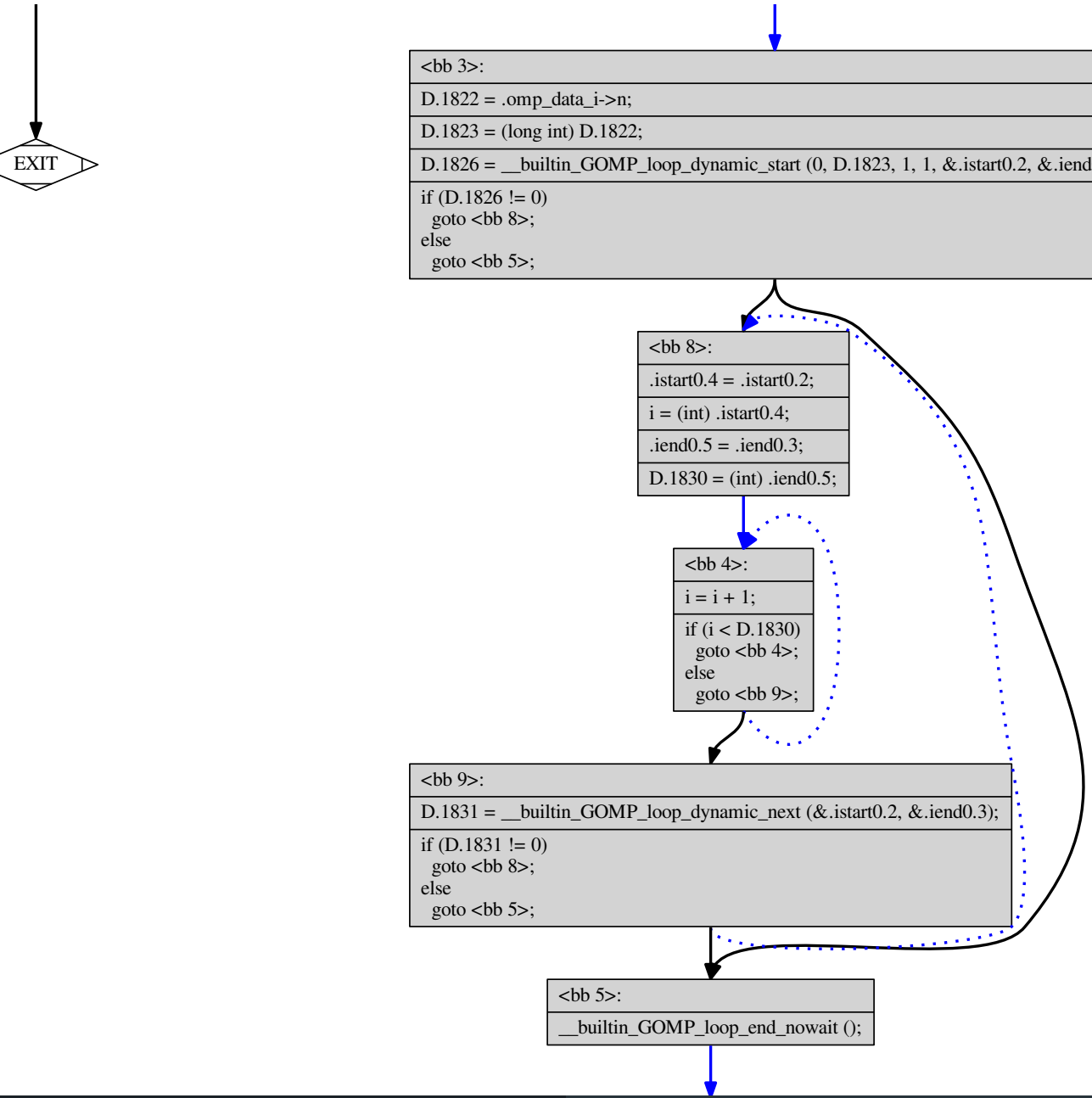


Loops: Construtor `for` – Segundo formato



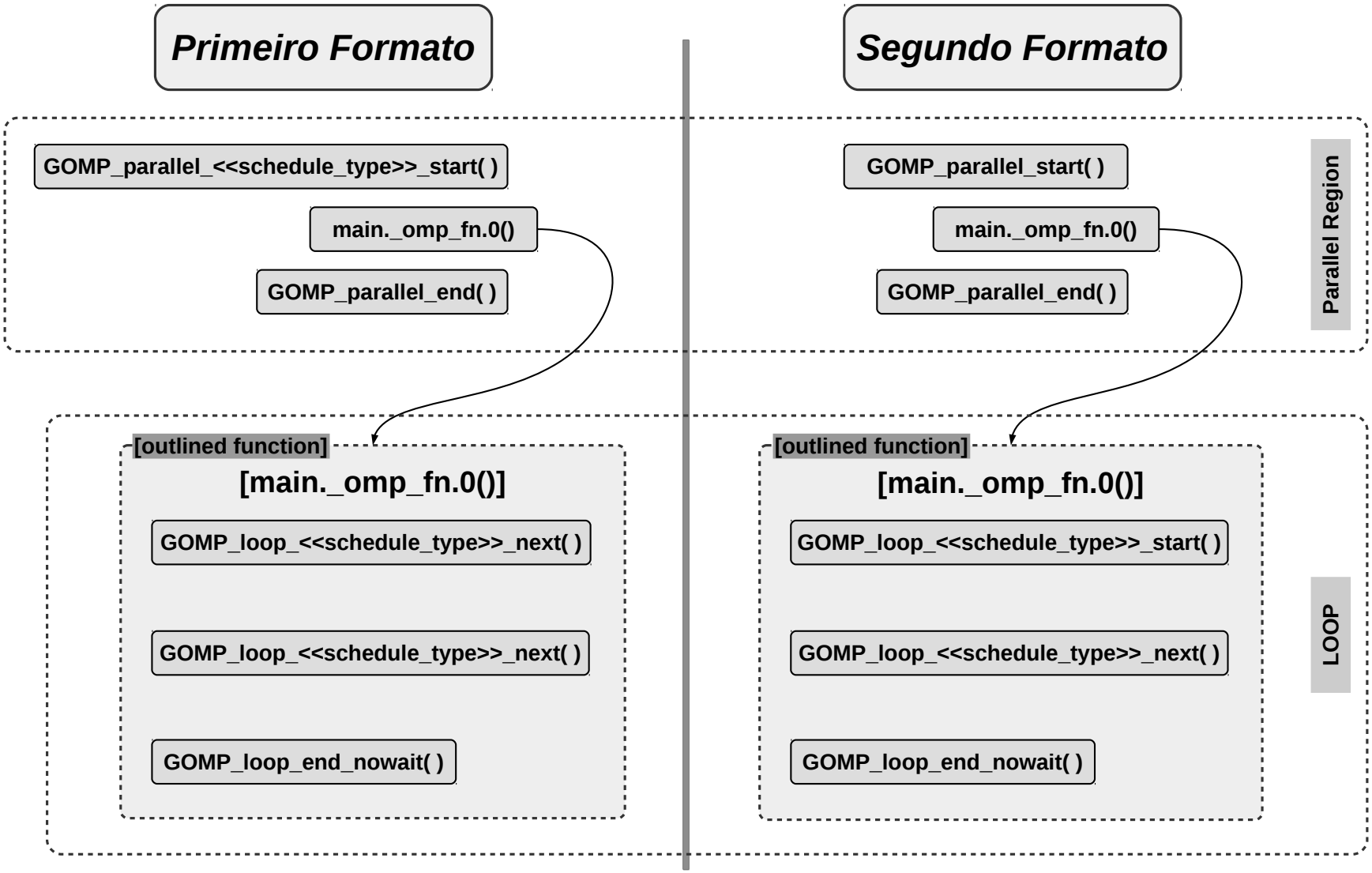
<bb 3>:
D.1822 = .
D.1823 = (
D.1826 = _
if (D.1826
goto <bb
else
goto <bb

Loops: Construtor `for` – Segundo formato



Loops: Construtor `for` – Resumo

- Dois formatos de código que são gerados para laços:



Seções
construtor sections

Seções: Construtor *sections* I

- O *sections* é um construtor de compartilhamento de trabalho não iterativo
- Define um conjunto de blocos estruturados que são distribuídos entre as *threads* de um time.

Sintaxe

```
1 #pragma omp sections [clause[ [,] clause] ... ] new-line
2 {
3   [#pragma omp section new-line]
4   structured-block
5
6   [#pragma omp section new-line
7   structured-block]
8   ...
9 }
```

Seções: Construtor *sections* II

- Cláusulas permitidas para `sections`:

Cláusulas

```
1 private(list)
2 firstprivate(list)
3 lastprivate(list)
4 reduction(reduction-identifier: list)
5 nowait
```

Seções: Construtor *sections* I

- **Diretiva:** `#pragma omp sections`
- É utilizada para dividir tarefas entre as *threads* em blocos de código que não possuem iterações.
- Cada *thread* irá executar um bloco de código diferente especificado por `#pragma omp section`.

Sintaxe

```
1 #pragma omp sections
2 {
3   #pragma omp section
4   bloco_1;
5   #pragma omp section
6   bloco_2;
7   #pragma omp section
8   bloco_3;
9 }
```

Seções: Construtor *sections* II

- Como são implementadas as *sections*:

```
1  for (i = GOMP_sections_start (3); i != 0; i =
       GOMP_sections_next ())
2    switch (i) {
3      case 1:
4        bloco_1;
5        break;
6      case 2:
7        bloco_2;
8        break;
9      case 3:
10       bloco_3;
11       break;
12   }
13 GOMP_barrier ();
```

Seções: Construtor *sections* III

- Exemplo utilizando *sections* com a cláusula *reduction*:

```
1  int main(int argc, char *argv[]) {
2    int i, id;
3    int sum = 0;
4
5    fprintf(stdout, "Thread[%d][%lu]: Antes da Região Paralela.\n", omp_get_thread_num
         (), (long int)pthread_self());
6
7    #pragma omp parallel num_threads(8) private(id)
8    {
9      id = omp_get_thread_num();
10     #pragma omp sections reduction(+:sum)
11     {
12       #pragma omp section
13       {
14         fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na seção 1.\n", id, (long
               int) pthread_self());
15         for(i=0; i<1024;i++){
16           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   }
```

Seções: Construtor *sections* IV

```
1     ...
2     fprintf(stdout, "Thread[%d][%lu]: Depois da Região Paralela.\n",
            omp_get_thread_num(), (long int)pthread_self());
3     fprintf(stdout, "Thread[%d][%lu]: sum: %d\n", omp_get_thread_num(), (long int)
            pthread_self(), sum);
4
5     return 0;
6  }
```

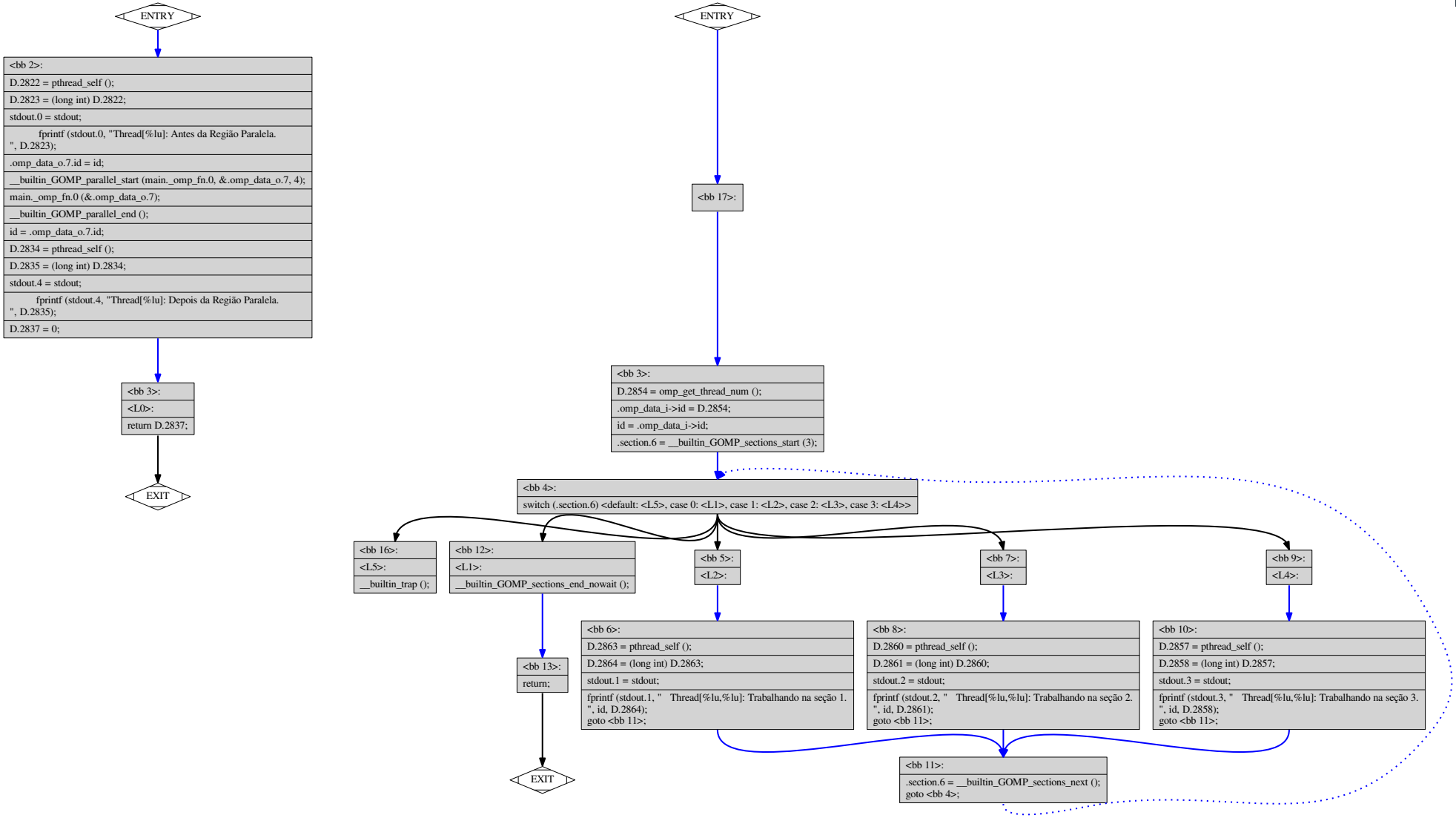
Seções: Construtor *sections* V

- Saída do exemplo utilizando *sections*:

Terminal

```
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$
```

Seções: Construtor *sections* VI



Tarefas
construtor task

Tarefas: Construtor `task` I

- **Diretiva:** `#pragma omp task`
- O construtor *task* permite a criação de tarefas explícitas.
- Disponível a partir das especificações 3.0 e 3.1
- Implementado pela `libgomp` do `GCC 4.4` e `GCC 4.7`, respectivamente.

Sintaxe

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

- Quando uma *thread* encontra um construtor *task*, uma nova tarefa é gerada para executar o bloco associado ao construtor.

Tarefas: Construtor `task` II

- 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*.

```
1  #pragma omp parallel
2  {
3    #pragma omp single
4    {
5      #pragma omp task
6      {
7        /* Bloco estruturado. */
8      }
9    }
10 }
```

Tarefas: Construtor `task` III

- Para cada construtor *task* é criada uma nova função *(outlined function)* com o código do seu bloco.
- A `libgomp` utiliza as funções para a criação do formato de código para *tasks*.

ABI da `libgomp` – Funções usadas a diretiva *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);
```

Tarefas: Construtor `task` IV

- Cláusulas permitidas para o construtor *task*:

Cláusulas

```
1  if([task :] scalar-expression)
2  final(scalar-expression)
3  untied
4  default(shared|none)
5  mergeable
6  private(list)
7  firstprivate(list)
8  shared(list)
9  depend(dependence-type:list)
10 priority(priority-value)
```

Tarefas: Construtor task V

```
1  int main(int argc, char *argv[]) {
2    int id = 0;
3    int x = atoi(argv[1]);
4
5    fprintf(stdout, "Thread[%lu,%lu]: Antes da região paralela.\n
         ", omp_get_thread_num(), (long int) pthread_self());
6
7    #pragma omp parallel num_threads(8) firstprivate(x) private(
         id)
8    {
9      id = omp_get_thread_num();
10     fprintf(stdout, "  Thread[%lu,%lu]: Todas as threads
           executam.\n", id, (long int) pthread_self());
11
12     #pragma omp single
13     {
14       fprintf(stdout, "  Thread[%lu,%lu]: Antes de criar tasks
             .\n", id, (long int) pthread_self());
15       #pragma omp task if(x > 10)
16       {
17         fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na
               task 1.\n", omp_get_thread_num(), (long int)
               pthread_self());
18       }
```

Tarefas: Construtor task VI

```
1          #pragma omp task if(x > 20)
2          {
3             fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na
                  task 2.\n", omp_get_thread_num(), (long int)
                  pthread_self());
4          }
5
6          fprintf(stdout, "    Thread[%lu,%lu]: Antes do taskwait.\
               n", id, (long int) pthread_self());
7          #pragma omp taskwait
8          fprintf(stdout, "    Thread[%lu,%lu]: Depois do taskwait
               .\n", id, (long int) pthread_self());
9
10         #pragma omp task
11         {
12            fprintf(stdout, "    Thread[%lu,%lu]: Trabalhando na
                  task 3.\n", omp_get_thread_num(), (long int)
                  pthread_self());
13         }
14      }
15   }
16   fprintf(stdout, "Thread[%lu,%lu]: Depois da região paralela.\
         n", omp_get_thread_num(), (long int) pthread_self());
17
18   return 0;
```

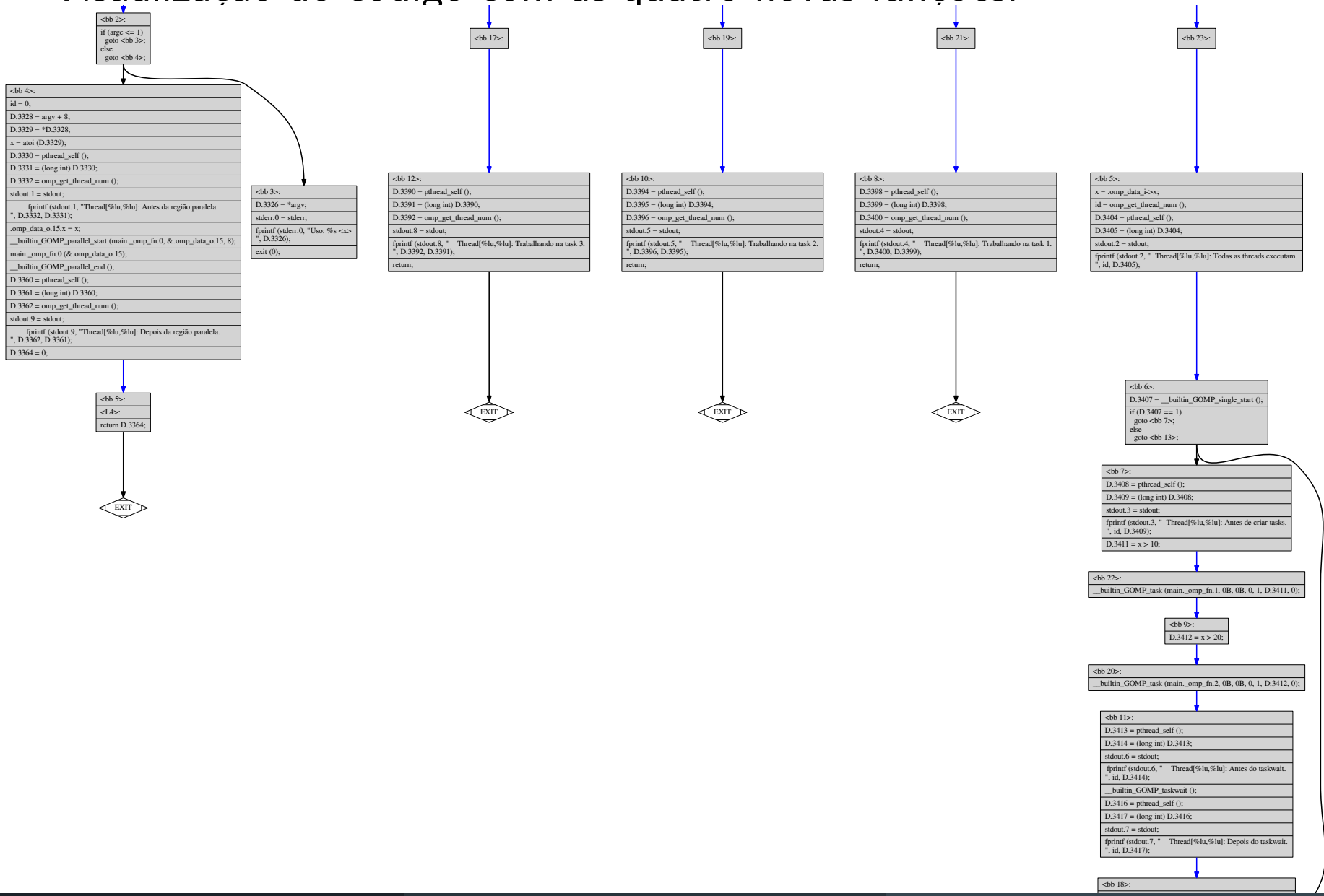
Tarefas: Construtor `task` VII

Terminal

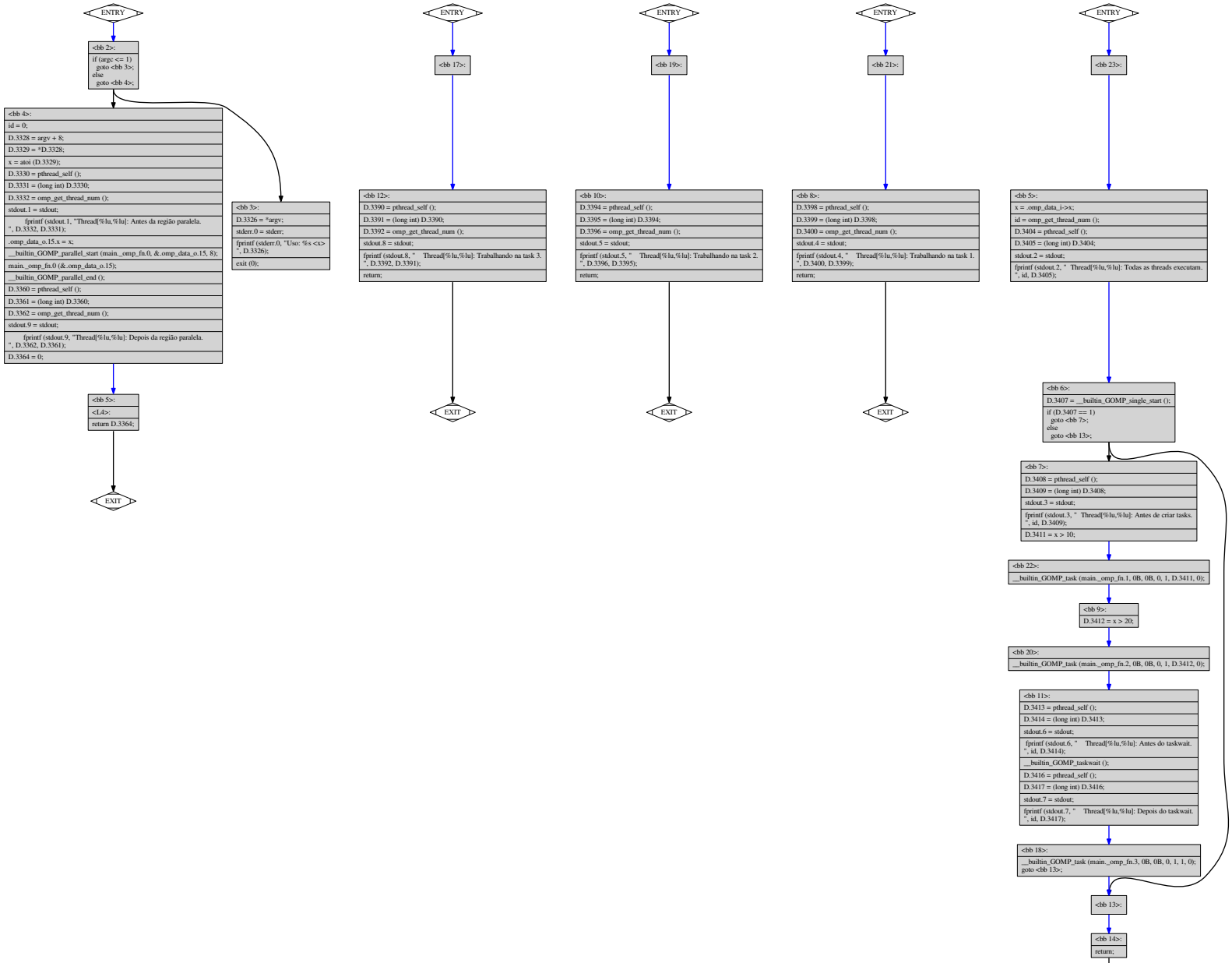
```
rogerio@chamonix:/src/example-tasks$ ./example-tasks.exe 1024
Thread[0,140369357629312]: Antes da região paralela.
Thread[0,140369357629312]: Todas as threads executam.
Thread[0,140369357629312]: Antes de criar tasks.
Thread[7,140369294767872]: Todas as threads executam.
Thread[0,140369357629312]: 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$
```

Tarefas: Construtor `task` VIII

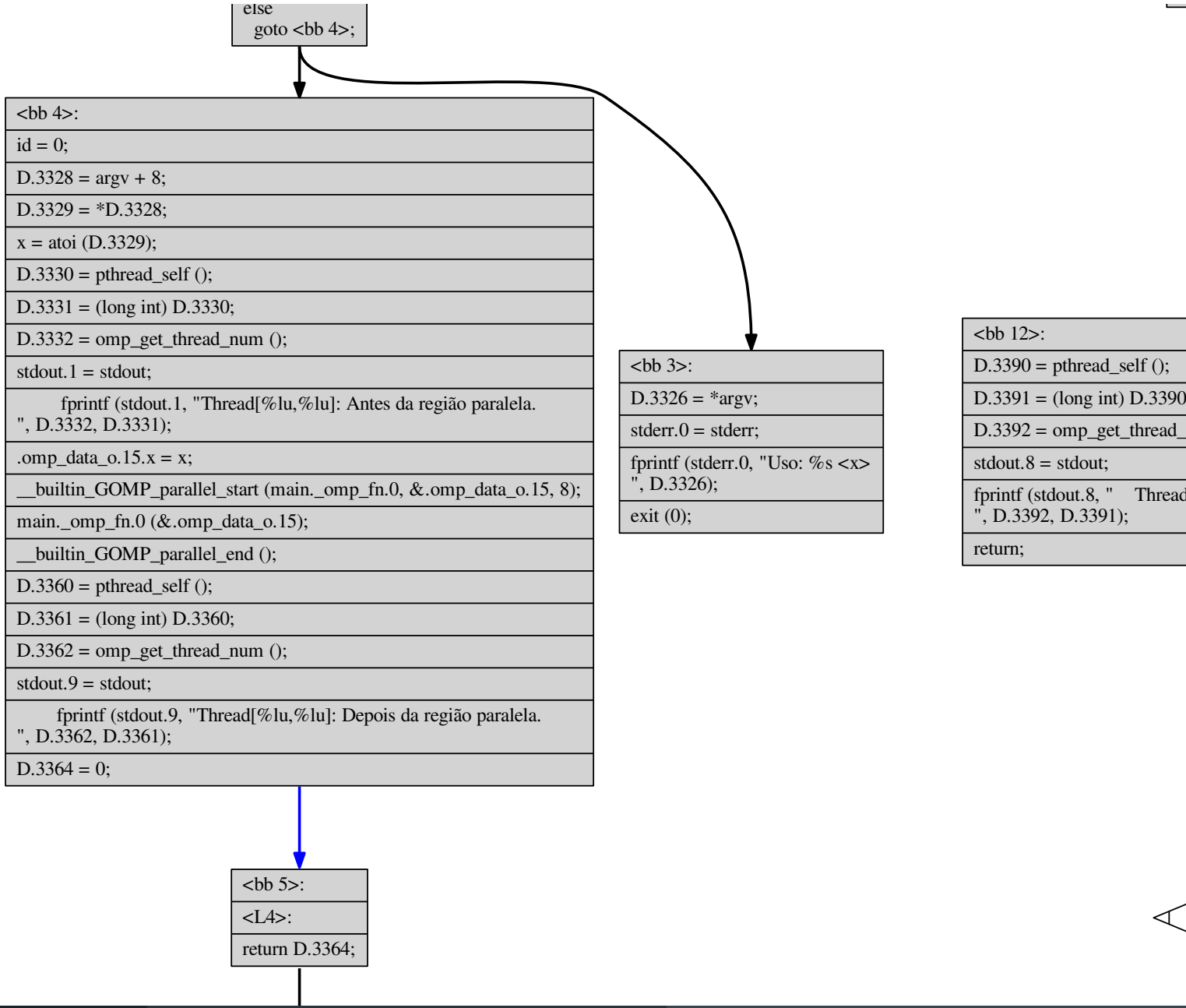
- Visualização do código com as quatro novas funções:



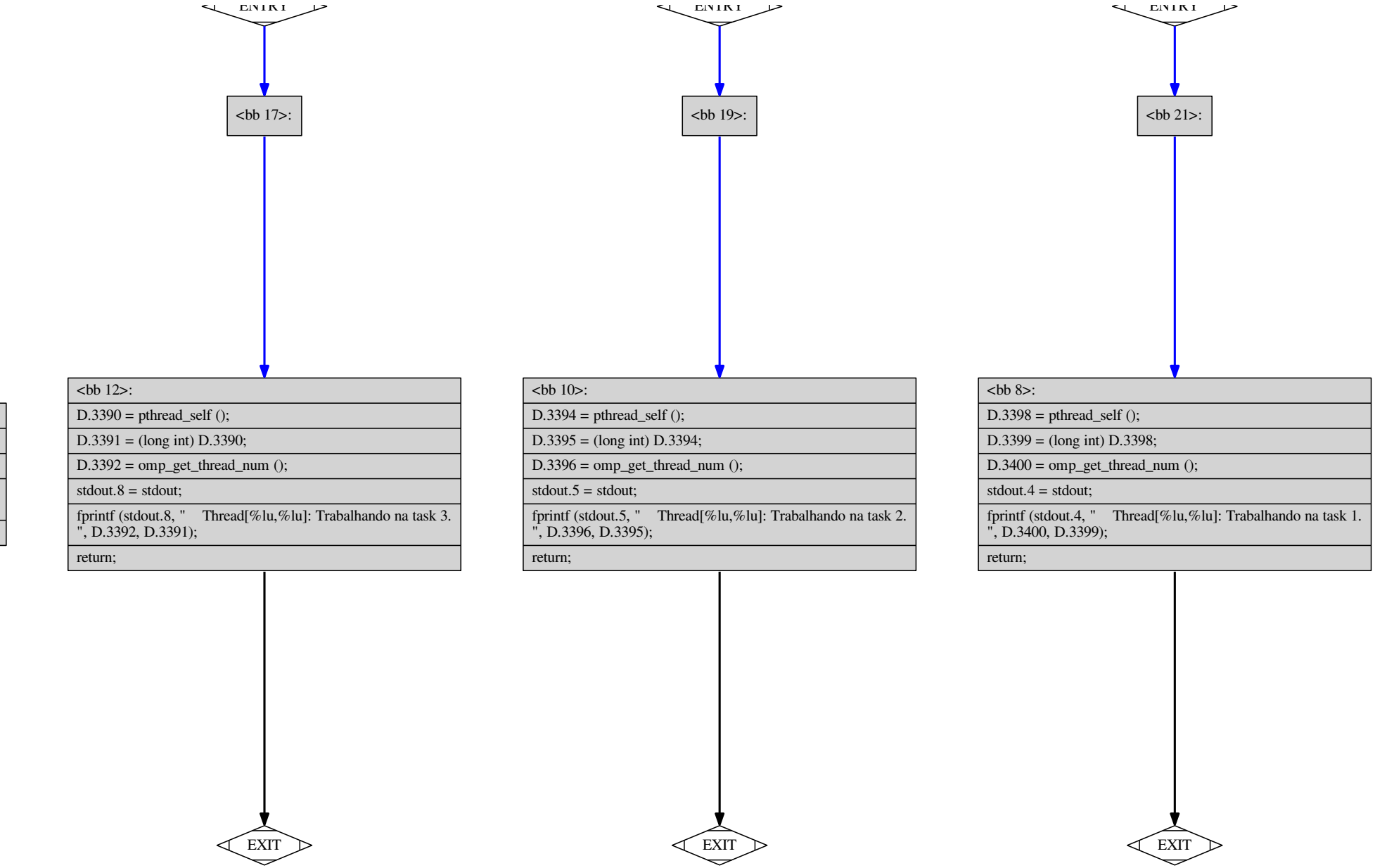
Tarefas: Construtor `task`



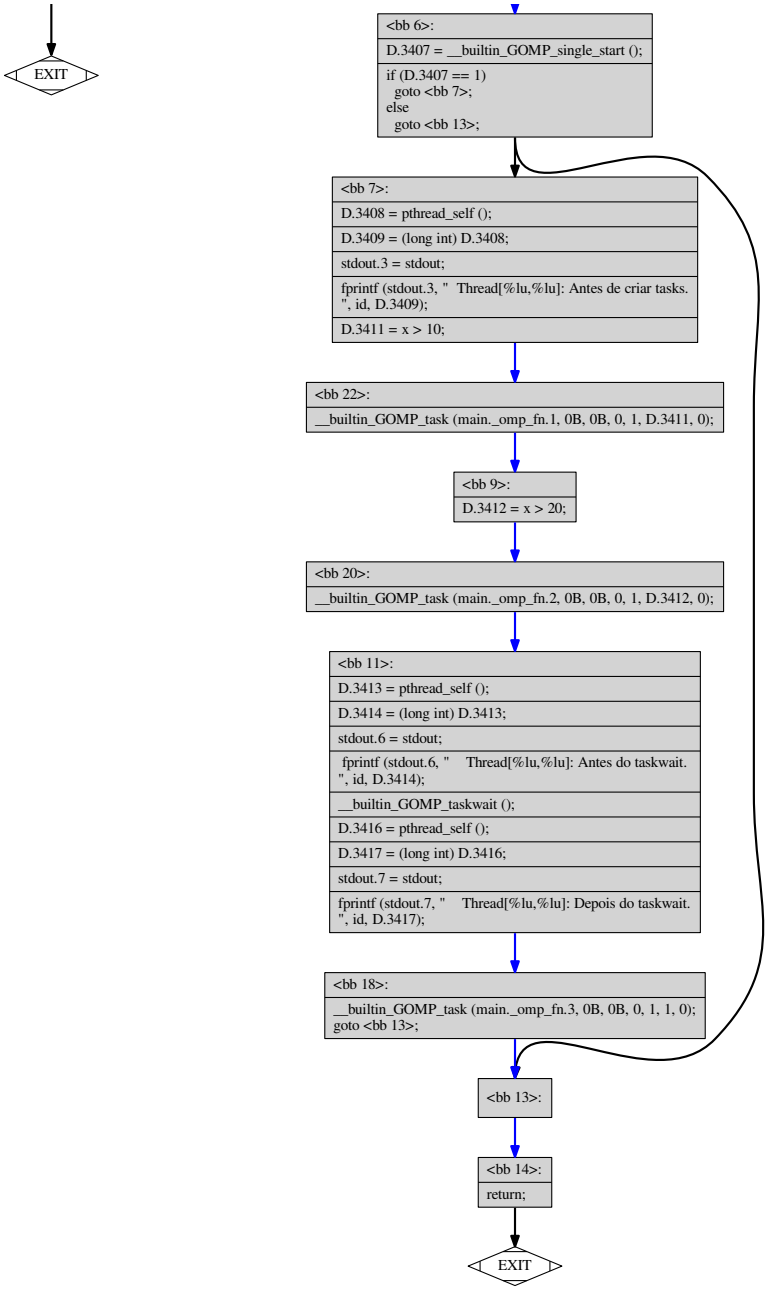
Tarefas: Construtor `task`



Tarefas: Construtor `task`



Tarefas: Construtor `task`



Tarefas com loops
construtor taskloop

Tarefas com *loops*: construtor `taskloop` I

- **Diretiva:** `#pragma omp taskloop`
- O construtor *taskloop* permite distribuir as iterações de um ou mais laços aninhados para tarefas.
- O *taskloop* é um construtor que especifica que as iterações de um ou mais *loops* associados serão executadas em paralelo usando `OpenMP` *tasks*. As iterações são distribuídas entre as tarefas criadas pelo construtor e escalonadas para serem executadas.

Tarefas com *loops*: construtor `taskloop` II

- Quando um construtor *taskloop* é encontrado uma nova função é criada.
- As tarefas serão criadas e a elas serão associadas um subconjunto de iterações do laço.
- Disponível a partir do `GCC 6.x`.

Construtor *taskloop*:

```
#pragma omp taskloop [clause[[,] clause] ...] new-line
{
  // for-loops.
}
```

Tarefas com *loops*: construtor `taskloop` III

Cláusulas

```
1  if([taskloop:]scalar-expression)
2  shared(list)
3  private(list)
4  firstprivate(list)
5  lastprivate(list)
6  default(shared |none)
7  grainsize(grain-size)
8  num_tasks(num-tasks)
9  collapse(n)
10 final(scalar-expr)
11 priority(priority-value)
12 untied
13 mergeable
14 nogroup
```

Tarefas com *loops*: construtor `taskloop` IV

- Para cada construtor *task* é criada uma nova função *(outlined function)* com o código do seu bloco.
- A `libgomp` utiliza as funções para a criação do formato de código para tasks.

ABI da `libgomp` – Funções usadas a diretiva *taskloop*

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

Tarefas com *loops*: construtor `taskloop` V

- Exemplo: Distribuindo as iterações do laço com o *taskloop*.

```
1  int main(int argc, char *argv[]) {
2    fprintf(stdout, "Thread[%lu,%lu]: Antes da Região Paralela.\n", (long int)
         omp_get_thread_num(), (long int) pthread_self());
3
4    #pragma omp parallel num_threads(4)
5    {
6      #pragma omp single
7      {
8        fprintf(stdout, "  Thread[%lu,%lu]: Antes das tasks.\n", (long int)
             omp_get_thread_num(), (long int) pthread_self());
9        #pragma omp taskgroup
10       {
11         #pragma omp task
12         {
13           fprintf(stdout, "Thread[%lu,%lu]: Trabalhando na task avulsa.\n",
                 omp_get_thread_num(), (long int) pthread_self());
14         }
15
16         #pragma omp task
17         {
18           fprintf(stdout, "Thread[%lu,%lu]: Trabalhando na task func().\n",
                 omp_get_thread_num(), (long int) pthread_self());
19           func();
20         }
21       }
22     }
23   }
24   fprintf(stdout, "Thread[%lu,%lu]: Depois da Região Paralela.\n", (long int)
         omp_get_thread_num(), (long int) pthread_self());
25
26   return 0;
```

Tarefas com *loops*: construtor `taskloop` VI

- Exemplo: Distribuindo as iterações do laço com o *taskloop*.

```
1  void func(){
2    int i, j;
3    fprintf(stdout, "Thread[%lu,%lu]: taskloop.\n",
         omp_get_thread_num(), (long int) pthread_self());
4    #pragma omp taskloop num_tasks(8) private(j) grainsize(2)
5    for (i = 0; i < 16; i++) {
6      for (j = 0; j < i; j++) {
7        fprintf(stdout, "Thread[%lu,%lu]: Trabalhando na iteração
             (%d,%d).\n", omp_get_thread_num(), (long int)
             pthread_self(),i,j);
8      }
9    }
10 }
```

Tarefas com *loops*: construtor `taskloop` VII

- O código em `GIMPLE`, que é a representação intermediária do `GCC`:

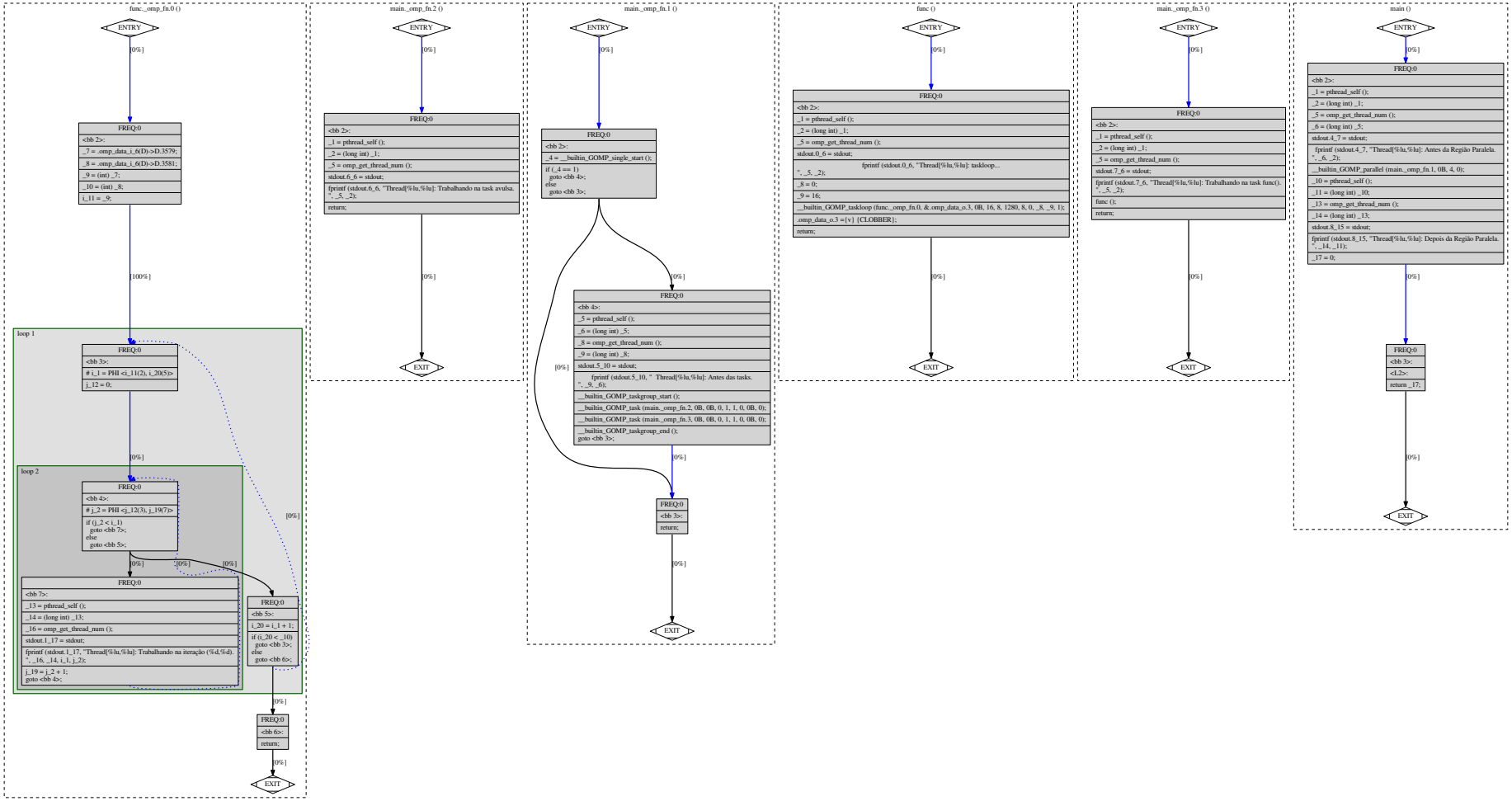


Figura 5:

Tarefas com *loops*: construtor `taskloop` VIII

Terminal

```
rogerio@chamonix:/src/example-taskloop$ ./example-taskloop.exe
Thread[0,139960402667392]: Antes da Região Paralela.
  Thread[1,139960390162176]: Antes das tasks.
Thread[2,139960381769472]: Trabalhando na task avulsa.
Thread[0,139960402667392]: Trabalhando na task func().
Thread[0,139960402667392]: taskloop...
Thread[3,139960373376768]: Trabalhando na iteração (1,0).
Thread[3,139960373376768]: Trabalhando na iteração (4,0).
Thread[3,139960373376768]: Trabalhando na iteração (4,1).
Thread[3,139960373376768]: Trabalhando na iteração (4,2).
Thread[3,139960373376768]: Trabalhando na iteração (4,3).
Thread[0,139960402667392]: Trabalhando na iteração (14,0).
Thread[0,139960402667392]: Trabalhando na iteração (14,1).
Thread[0,139960402667392]: Trabalhando na iteração (14,2).
Thread[0,139960402667392]: Trabalhando na iteração (14,3).
...
Thread[2,139960381769472]: Trabalhando na iteração (2,0).
Thread[2,139960381769472]: Trabalhando na iteração (2,1).
Thread[2,139960381769472]: Trabalhando na iteração (3,0).
Thread[2,139960381769472]: Trabalhando na iteração (3,1).
...
Thread[3,139960373376768]: Trabalhando na iteração (5,0).
Thread[3,139960373376768]: Trabalhando na iteração (5,1).
Thread[3,139960373376768]: Trabalhando na iteração (5,2).
Thread[3,139960373376768]: Trabalhando na iteração (5,3).
Thread[3,139960373376768]: Trabalhando na iteração (5,4).
Thread[2,139960381769472]: Trabalhando na iteração (3,2).
Thread[0,139960402667392]: Depois da Região Paralela.
rogerio@chamonix:/src/example-taskloop$
```

Suporte à Vetorização
construtor simd

Suporte à Vetorização: `construtor simd` I

- **Diretiva:** `#pragma omp simd`
- O construtor `simd` pode ser aplicado a um laço diretamente indicando que múltiplas iterações do laço podem ser executadas concorrentemente usando instruções `SIMD`.
- E 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`.

Sintaxe

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

Suporte à Vetorização: `construtor simd` II

- **Exemplo:** Utilização do construtor `simd` em um laço que faz a multiplicação de dois *arrays*.

```
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 }
```

Suporte à Vetorização: `construtor simd` III

- O construtor `simd` não apresenta um formato específico.
- Código do corpo do laço terá instruções `SIMD`.

```
1  .L6:
2      movl    -20(%rbp), %eax
3      cltq
4      movsd   h_c(,%rax,8), %xmm1
5      movl    -20(%rbp), %eax
6      cltq
7      movsd   h_a(,%rax,8), %xmm2
8      movl    -20(%rbp), %eax
9      cltq
10     movsd   h_b(,%rax,8), %xmm0
11     mulsd   %xmm2, %xmm0
12     addsd   %xmm0, %xmm1
13     movq    %xmm1, %rax
14     movl    -20(%rbp), %edx
15     movslq  %edx, %rdx
16     movq    %rax, h_c(,%rdx,8)
17     addl    $1, -20(%rbp)
```

Suporte à Vetorização: `construtor simd` IV

- **Exemplo:** O construtor `simd` combinado com o `for`:

```
1  int main(int argc, char *argv[]) {
2    int i;
3    /* Inicialização dos vetores. */
4    init_array();
5
6    #pragma omp parallel for simd schedule(dynamic, 32)
         num_threads(4)
7    for (i = 0; i < N; i++) {
8      h_c[i] = h_a[i] * h_b[i];
9    }
10
11   /* Resultados. */
12   print_array();
13   check_result();
14
15   return 0;
16 }
```

Suporte à Vetorização: `construtor simd` V

- **Exemplo:** O construtor `simd` combinado com o `for`.
- O *assembly* gerado para a função de execução do laço com instruções `SIMD` em seu corpo.

```
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. */
```

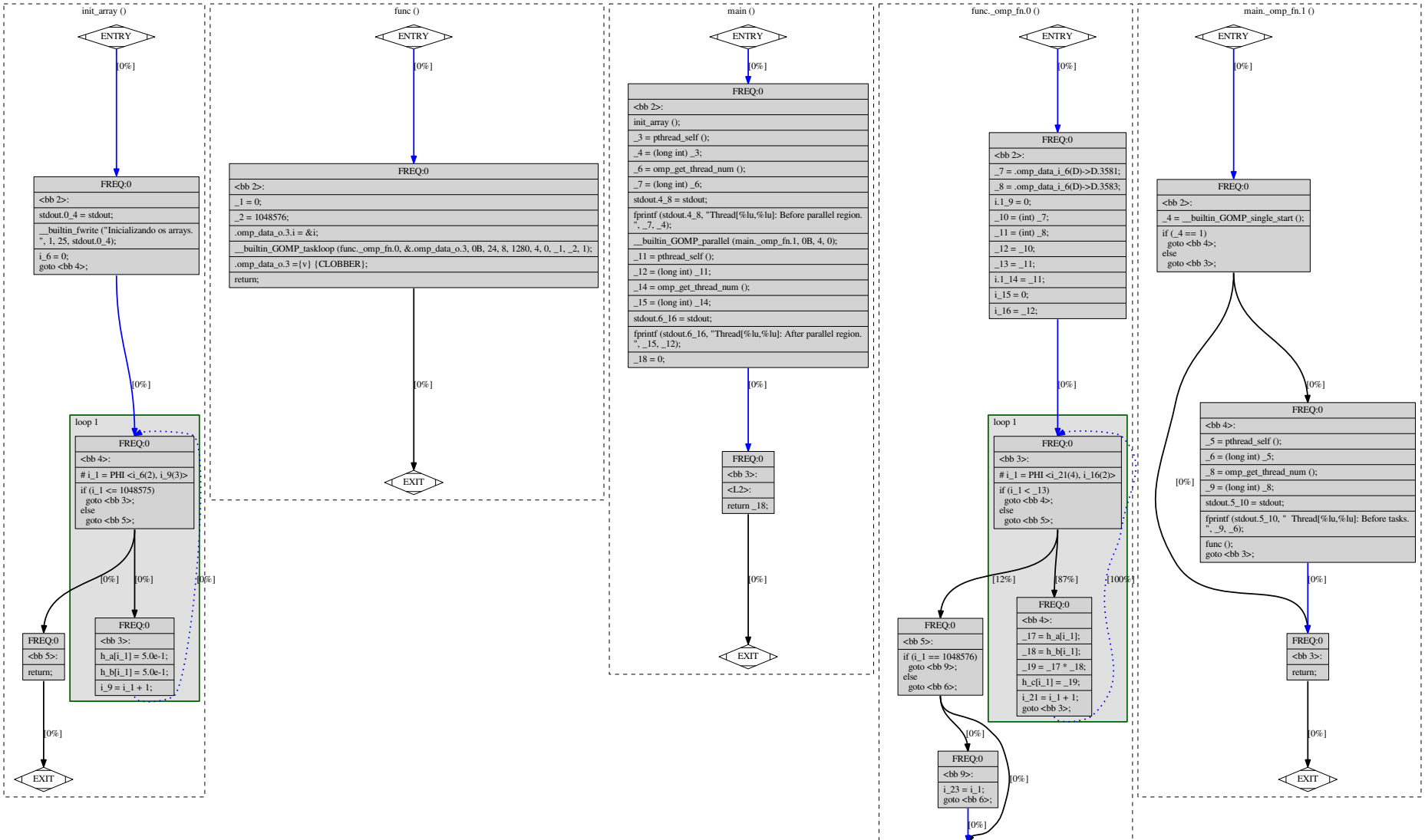
Suporte à Vetorização: `construtor simd` VI

- O construtor `simd` pode também ser combinado com o `taskloop`.
- O exemplo 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`.

```
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 }
```

Suporte à Vetorização: `construtor simd` VII

- Toda a estrutura de execução do construtor `taskloop` e dos construtores utilizados na região paralela é criada.



Offloading para Aceleradores

construtor target

Offloading para Aceleradores: construtor `target` I

- 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.

```
1 __global__ void vecAdd(float *a, float *b, float *c, int n)
2 {
3   int id = blockIdx.x * blockDim.x + threadIdx.x;
4   if (id < n)
5     c[id] = a[id] + b[id];
6 }
```

Offloading para Aceleradores: construtor `target` II

- Declaração dos dados e ponteiros no dispositivo:

```
1  int main( int argc, char* argv[] ){
2    float *h_a;
3    float *h_b;
4    float *h_c;
5
6    // Declaração dos vetores de entrada na memória da GPU.
7    float *d_a;
8    float *d_b;
9    // Declaração do vetor de saída do dispositivo.
10   float *d_c;
11
12   // Tamanho em bytes de cada vetor.
13   size_t bytes = n * sizeof(float);
14
15   // Alocação de memória para os vetores do host.
16   h_a = (float*) malloc(bytes);
17   h_b = (float*) malloc(bytes);
18   h_c = (float*) malloc(bytes);
19
20   // Alocação de memória para cada vetor na GPU.
21   cudaMalloc(&d_a, bytes);
22   cudaMalloc(&d_b, bytes);
23   cudaMalloc(&d_c, bytes);
```

Offloading para Aceleradores: construtor `target` III

- `CUDA` fornece função para realizar transferências de dados entre a memória principal e a memória do dispositivo.

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

Offloading para Aceleradores: construtor `target` IV

- Então é feita a chamada à função *kernel*.
- 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.

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

Offloading para Aceleradores: construtor `target` V

- Cópia do resultado (`d_c`) da soma de vetores realizada no dispositivo para (`h_c`) na memória do *host*.
- Liberação da memória alocada.

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

Offloading para Aceleradores: construtor `target` VI

- No contexto de *diretivas de compilação* o padrão `OpenACC` fornece um conjunto de diretivas.
- Laços paralelizáveis anotados para serem transformados em *kernel* para um dispositivo.
- Cada diretiva em `C/C++` inicia com `#pragma acc` e existem construtores e cláusulas para a criação de *kernels* com base em laços.

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

Offloading para Aceleradores: construtor `target` VII

- O exemplo soma de vetores escrito com as diretivas do `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 }
```

Código 4: Exemplo de Soma de Vetores anotado com diretivas `OpenACC`

Offloading para Aceleradores: construtor `target` VIII

- A saída gerada pelo compilador pgcc:

Terminal

```
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$
```

Offloading para Aceleradores: construtor `target` IX

- Para *offloading* de código para dispositivos aceleradores, no `OpenMP` temos o construtor `target`.
- **Diretiva:** `#pragma omp target`

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

Offloading para Aceleradores: construtor `target` X

- O Código 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)`.

Offloading para Aceleradores: construtor `target` XI

- **Exemplo:** Soma de vetores.

```
1 void vecaddgpu(float *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 }
```

Offloading para Aceleradores: construtor `target` XII

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

```
1 void vecaddgpu(float *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 }
```

Offloading para Aceleradores: construtor `target` XIII

- O construtor `target` também permite a escolha de fazer o *offloading* do código para o dispositivo ou não, utilizando a cláusula `if`.
- 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*.

```
1  #define THRESHOLD 1024
2
3  void vecaddgpu(float *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 }
```

Offloading para Aceleradores: construtor `target` XIV

- Especificar uma região de dados pode ser útil para múltiplos *kernels*

```
1  #define THRESHOLD 1048576
2
3  void vecaddgpu(float *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 }
```

Offloading para Aceleradores: construtor `target` XV

- Atualização dos dados entre as execuções dos *kernels* é utilizando o construtor `target update`.

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

Offloading para Aceleradores: construtor `target` XVI

- O exemplo de soma de vetores feito `OpenMP` utilizando o construtor `target` e suas combinações vistas nos exemplos anteriores.

```
1  #define THRESHOLD 1024
2
3  float *h_a;
4  float *h_b;
5  float *h_c;
6  int n = 0;
7  /* Código Suprimido. */
8
9  void vecaddgpu(float *c, float *a, float *b){
10   #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:
         n]) if(n>THRESHOLD)
11   {
12     #pragma omp target if(n>THRESHOLD)
13     #pragma omp parallel for if(n>THRESHOLD)
14     for( int i = 0; i < n; ++i ){
15       c[i] = a[i] + b[i];
16     }
17   }
18 }
```

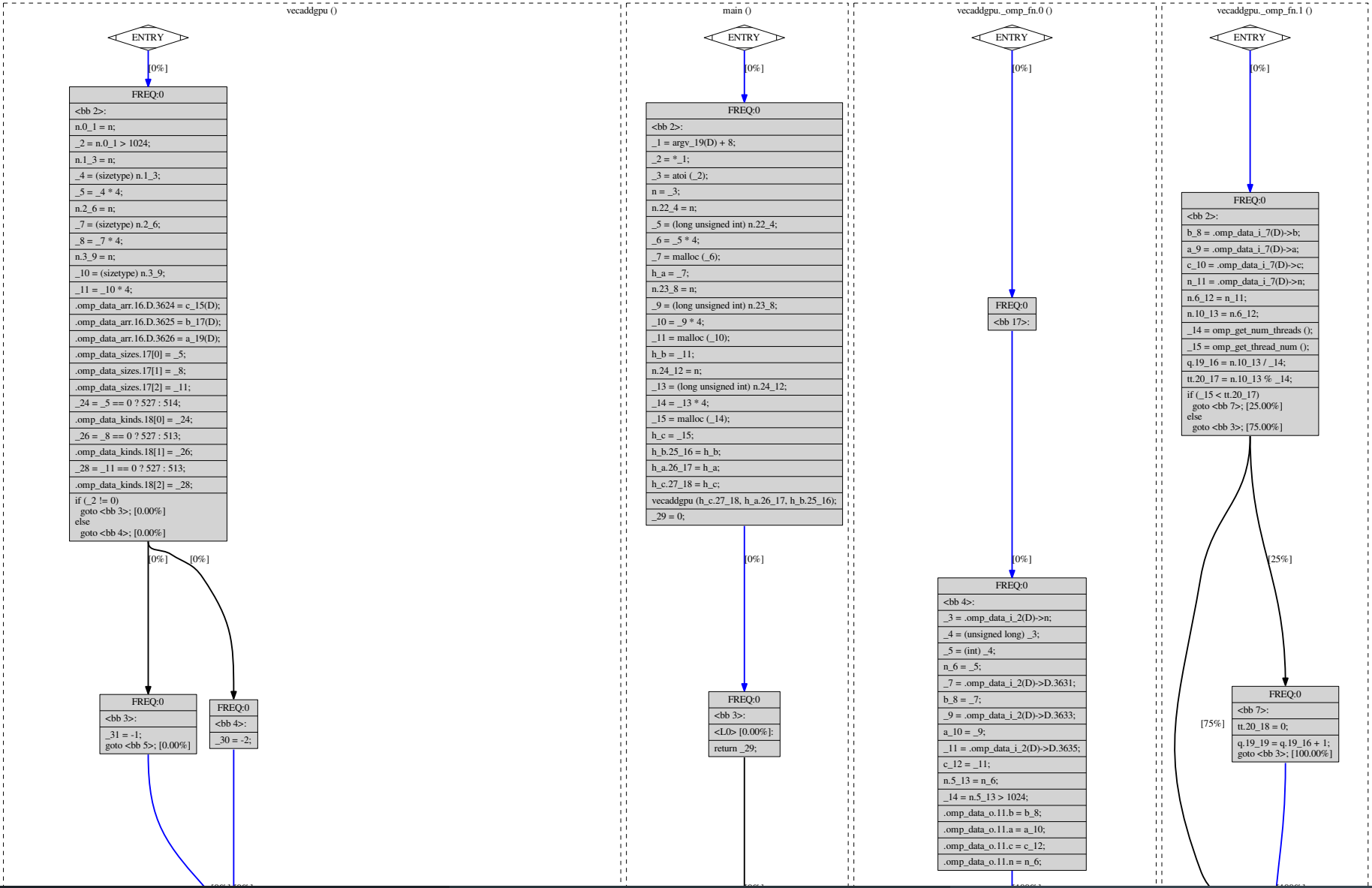
Offloading para Aceleradores: construtor `target` XVII

- O exemplo de soma de vetores feito `OpenMP` utilizando o construtor `target` e suas combinações vistas nos exemplos anteriores.

```
1  int main(int argc, char *argv[]) {
2    int i;
3    n = atoi(argv[1]);
4
5    h_a = (float*) malloc(n*sizeof(float));
6    h_b = (float*) malloc(n*sizeof(float));
7    h_c = (float*) malloc(n*sizeof(float));
8
9    init_array();
10
11   vecaddgpu(h_c, h_a, h_b);
12
13   return 0;
14 }
```

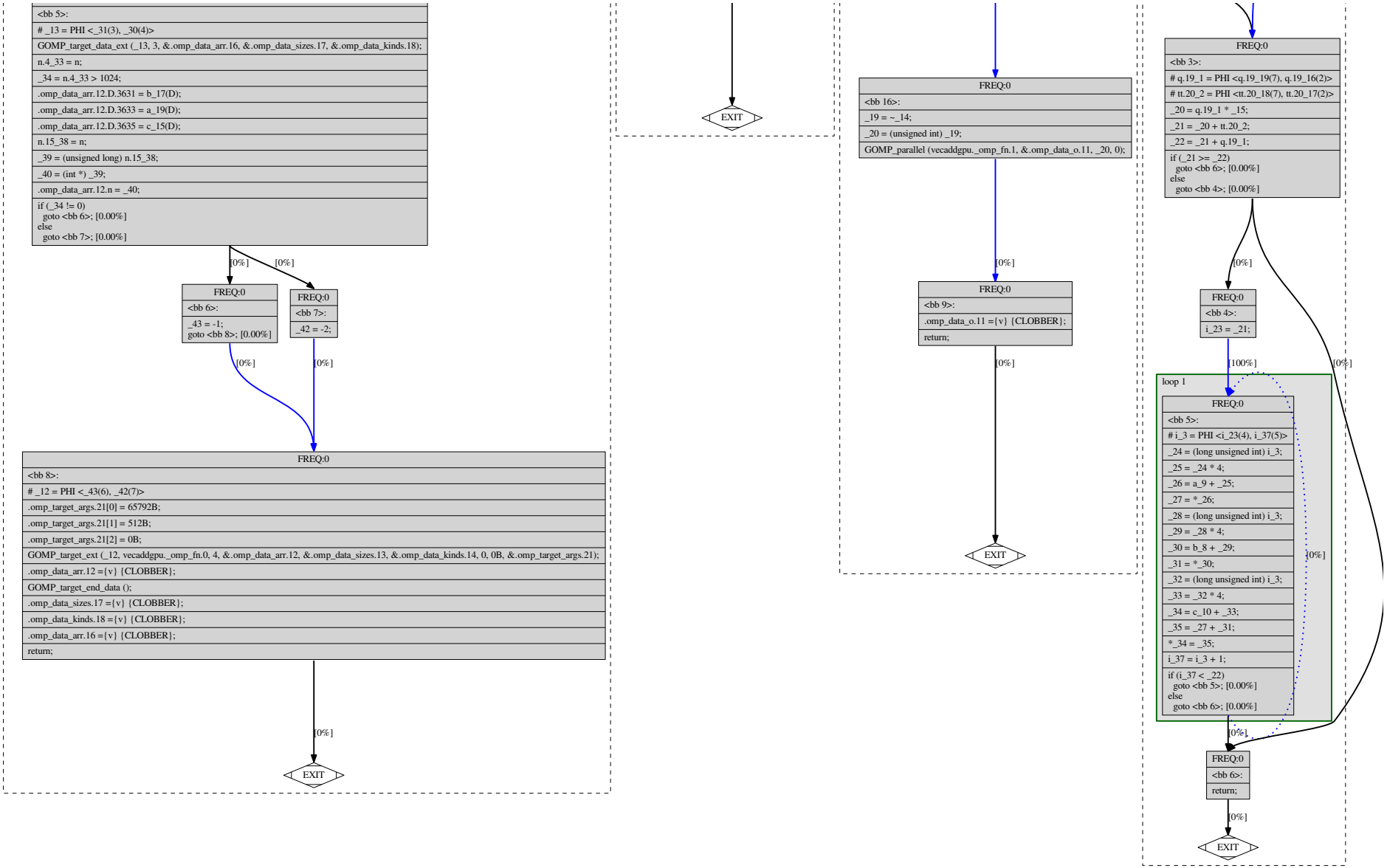
Offloading para Aceleradores: construtor `target` XVIII

- Estrutura do Código gerado para o construtor `target`:



Offloading para Aceleradores: construtor `target` XIX

- Estrutura do Código gerado para o construtor `target`:



Offloading para Aceleradores: construtor `target` XX

- As funções relacionadas com a geração de código para o construtor `target` que identificamos na `ABI` da `libgomp`:

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)
```

Offloading para Aceleradores: construtor `target` XXI

- Como o código utiliza a cláusula `if` para decidir se deve ou não fazer o *offloading* com base no tamanho dos dados.

Terminal

```
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
```

Offloading para Aceleradores: construtor `target` XXII

- Da mesma forma o exemplo foi executado com $n = 512$ e podemos verificar com o `nvprof` que o *offloading* de código não foi feito.

Terminal

```
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
```

- Nenhuma operação relacionada ao dispositivo (transferências de dados e lançamento da execução de *kernels*) que caracterizaria o *offloading* de código aconteceu.

Aplicações

Aplicações I

- 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*.
- O que pode cobrir desde *logging*, criação de *traces*[a], monitoramento[b] 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] Trahay et al. (2011)
[b] Mohr et al. (2002)

Aplicações II

```
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 }
```

Aplicações III

- Uma macro pode ser definida para recuperar os ponteiros para as funções originais do runtime `OpenMP`:

```
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)
8
9  #if defined(VERBOSE) && VERBOSE > 0
10   #define PRINT_FUNC_NAME fprintf(stderr, "TRACE-FUNC-NAME:
         [%10s:%07d] Thread [%lu] is calling [%s()]\n",__FILE__,
         __LINE__, (long int) pthread_self(), __FUNCTION__)
11 #else
12   #define PRINT_FUNC_NAME (void) 0
13 #endif
```

Aplicações IV

- Função proxy para a função original utilizando a macro.
- Chamadas de funções para executar algum código antes (`PRE_`) ou algum código depois (`POST_`).

```
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 }
```

Aplicações V

- Função *proxy* para a função de inicialização de laço com escalonamento do tipo *dynamic*.

```
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 }
```

Aplicações VI

- A função *proxy* para a função de término de laços de repetição.

```
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 }
```

Aplicações VII

- Função *proxy* capaz de interceptar a função de criação de *tasks*.

```
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 }
```

Aplicações VIII

- Parte da saída da execução do exemplo do uso do construtor `task` com a biblioteca de interceptação:

Terminal

```
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 [
```

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.
- 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.
- Existem diversas outras diretivas de compilação do `OpenMP` e implementações que não foram abordadas neste curso.

Referências I

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

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

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

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

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

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

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

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

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

Referências II

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 Clang (2015). clang: a C language family frontend for llvm.

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

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.

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

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

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

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

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.

Fim

Obrigado!

Contato

- **Rogério Aparecido Gonçalves:**
  rogerioag@utfpr.edu.br
- **João M. de Queiroz Filho:**
  joaomfilho1995@gmail.com
- **Alfredo Goldman:**
  gold@ime.usp.br

Informações

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

Material disponível em:
https://github.com/rogerioag/minicurso-openmp-wscad-2017