Um Estudo sobre a Expansão das Diretivas de Compilação para Interceptação de Código OpenMP

Rogério A. Gonçalves[1,2] 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)
Laboratório de Sistemas de Software (LSS)
São Paulo – SP – Brasil

rogerioag@utfpr.edu.br, {rag, gold}@ime.usp.br

Agenda

Objetivos

- Apresentar alguns conceitos sobre a implementação das diretivas de compilação do `OpenMP`
- Que podem ser utilizados para o desenvolvimento de bibliotecas de interceptação.
- Técnica de hooking – `LD_PRELOAD`
- Alguns exemplos de aplicações

Material

- Slides e Exemplos

github

`https://github.com/rogerioag/minicurso-erad-sp-2016`
ou
git clone https://github.com/rogerioag/minicurso-erad-sp-2016.git

Padrão OpenMP I

- O `OpenMP`[1] é um padrão bem conhecido e amplamente utilizado em aplicações para plataformas *multicore*.
- O uso de *diretivas de compilação*.
- O `OpenMP` implementa o modelo `fork-join`.
- Múltiplas threads executam tarefas definidas implicitamente ou explicitamente.
- As diretivas funcionam como anotações.
- São implementadas usando-se as diretivas de pré-processamento `#pragma`, em `C/C++`, e sentinelas `!$`, no `Fortran`.

Padrão OpenMP II

- As diretivas são substituídas pelo seu formato de código expandido com as chamadas para o runtime do `OpenMP`.
- Nosso estudo foi 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 fazer *offloading* de código para aceleradores.

[1] Dagum and Menon (1998); OpenMP-ARB (2011, 2013, 2015)
[2] GNU Libgomp (2015b,c, 2016a,b)

Implementações 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 (2012, 2015b, 2011, 2013, 2015a)

Diretivas de Compilação
e
Código OpenMP Expandido
(Formato pós expansão das diretivas)

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

Estudo das Diretivas de Compilação I

- Verificou-se o formato de código gerado pelo `GCC` + `libgomp` para os construtores de regiões paralelas *(parallel region)* e compartilhamento de trabalho *(parallel for)*.
- O código foi gerado nas versões do `GCC` (4.8, 4.9, 5.3, 6.1) para verificarmos o formato interceptável.
- Adotamos o `GCC 4.8`, porque esta versão apresentar um formato mais consistente e definido que possibilita a interceptação.

Regiões paralelas: construtor *parallel* I

- Esta é uma das mais importantes diretivas, pois ela é responsável pela demarcação de regiões paralelas.

Regiões paralelas são criadas usando-se o construtor:

```
#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 a diretiva de região paralela é utilizada:

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

- A diretiva *parallel* é implementada com a criação de uma nova função *(outlined function)* usando o código contido em `body`.
- A `libgomp` usa funções para delimitar a região. Chamadas a essas funções são colocadas no código para indicar o início e o fim de uma região paralela:

The `libgomp` ABI

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

Regiões paralelas: construtor *parallel* II

- O código expandido gerado assume o formato:

```
1  void subfunction (void *data){
2    use data;
3    body;
4  }
5
6  // replace the annotated parallel region.
7  setup data;
8
9  GOMP_parallel_start(subfunction, &data, num threads);
10 subfunction(&data);
11 GOMP_parallel_end();
```

Regiões paralelas: construtor *parallel* III

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

```
1 main._omp_fn.0 (void * .omp_data_i) {
2   return;
3 }
4
5 main () {
6 int D.1803;
7
8 <bb 2>:
9   __builtin_GOMP_parallel_start (main._omp_fn.0, 0B,
      0);
10  main._omp_fn.0 (0B);
11  __builtin_GOMP_parallel_end ();
12  D.1803 = 0;
13
14 <L0>:
15  return D.1803;
16 }
```

Regiões paralelas: construtor *parallel* IV

- O código em *assembly*:

```
 1 .file "parallel-region.c"
 2 .text
 3 .globl main
 4 .type main, @function
 5 main:
 6   pushq %rbp
 7   movq %rsp, %rbp
 8   movl $0, %edx
 9   movl $0, %esi
10   movl $main._omp_fn.0, %edi
11   call GOMP_parallel_start
12   movl $0, %edi
13   call main._omp_fn.0
14   call GOMP_parallel_end
15   movl $0, %eax
16   popq %rbp
17 ret
```

```
18 .size main, .-main
19
20
21 .type main._omp_fn.0, @function
22 main._omp_fn.0:
23   pushq %rbp
24   movq %rsp, %rbp
25   movq %rdi, -8(%rbp)
26   popq %rbp
27   ret
28 .size main._omp_fn.0, .-
       main._omp_fn.0
29 .ident "GCC: (Debian 4.8.4-1) 4
       .8.4"
30 .section .note.GNU-stack,"",
       @progbits
```

Loops: Construtor `for` I

- Um time de threads é criado quando uma região paralela é alcançada.
- Mas 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 parallel for num_threads (number_of_threads)
    schedule ({auto, static, dynamic, guided, runtime}, {variable
    /expression | numerical value/constant})
```

Loops: Construtor `for` II

- O Código com uma região paralela com um laço é equivalente ao que apresenta os construtores em modo combinado.

```
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` I

- Semelhante ao processamento do construtor *parallel*, a diretiva `parallel for` também é implementada com a criação de uma nova função *(outlined function)* com o código do *loop*.
- Quando não se especifica um escalonamento o código é equivalente ao gerado para *schedule(static)*.
- A `libgomp` utiliza as funções para delimitar a região paralela e na construção do formato do *loop*, com escalonamento estático:

ABI da `libgomp` – Funções usadas a diretiva *parallel for*

```
void GOMP_parallel_loop_static(void (*)(void *), void *,unsigned
    , long, long, long, long, unsigned)
bool GOMP_loop_static_next(long *, long *)
void GOMP_loop_end_nowait(void)
void GOMP_parallel_end(void)
```

Loops: Construtor `for` II

- O código expandido que substitui a declaração do *loop* paralelo é composto de uma função *outlined* e de chamadas para criar a região paralela. O *loop* original está nas *linhas 5* e *6*.

```
1  void subfunction (void *data) {
2    long _s0, _e0;
3    while (GOMP_loop_static_next (&_s0, &_e0)){
4      long _e1 = _e0, i;
5      for (i = _s0; i < _e1; i++)
6        body;
7    }
8    GOMP_loop_end_nowait ();
9  }
10
11 GOMP_parallel_loop_static (subfunction, NULL, 0, lb,
       ub+1, 1, 0);
12 subfunction (NULL);
13 GOMP_parallel_end ();
```

Loops: Construtor `for` III

- Os códigos diferem apenas na definição do limite superior dos laços.

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

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

Loops: Construtor `for` IV

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

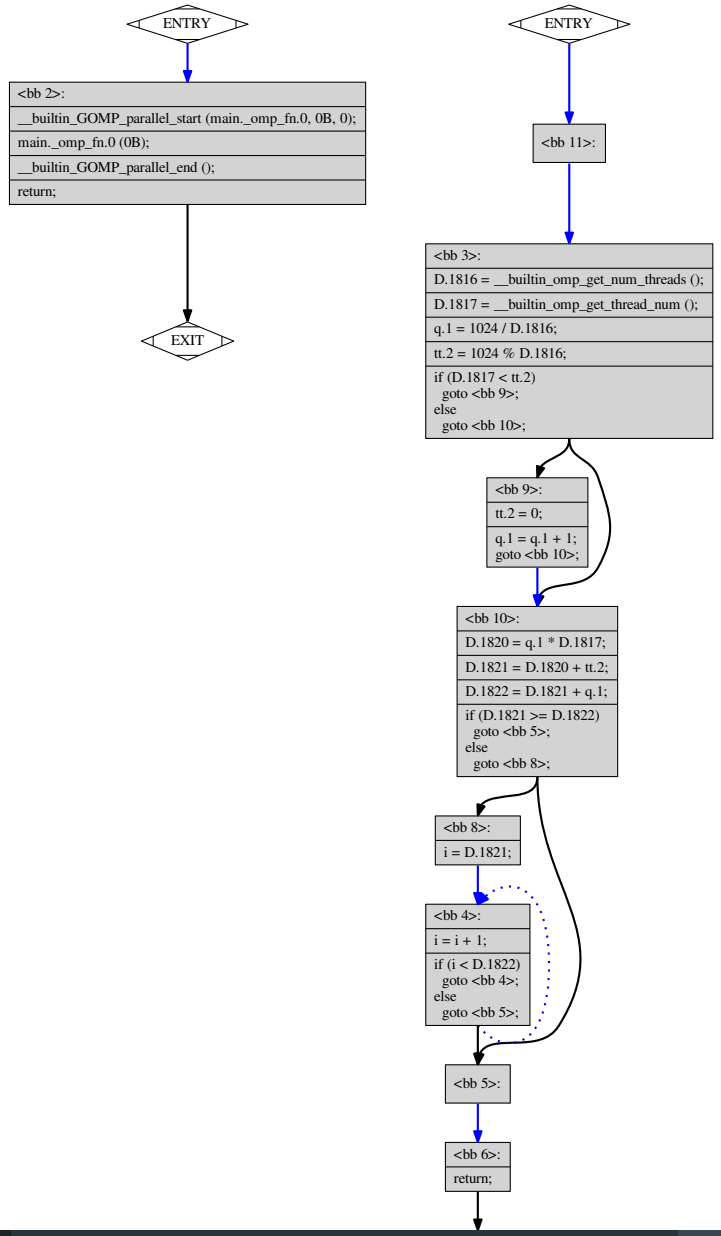
```
 1 main () {
 2
 3 <bb 2>:
 4   __builtin_GOMP_parallel_start (
       main._omp_fn.0, 0B, 0);
 5   main._omp_fn.0 (0B);
 6   __builtin_GOMP_parallel_end ();
 7   return;
 8 }
 9
10 main._omp_fn.0(void* .omp_data_i){
11
12 <bb 11>:
13
14 <bb 3>:
15   D.1816 =
       __builtin_omp_get_num_threads
       ();
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 V

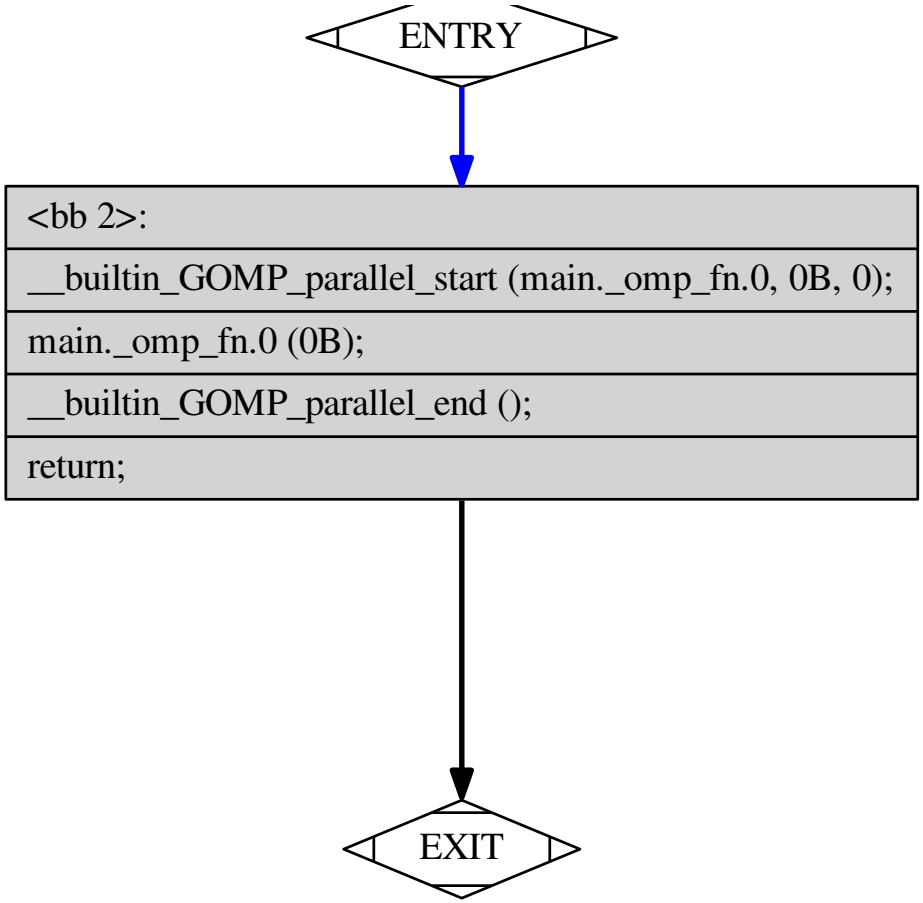
- O código em *assembly*:

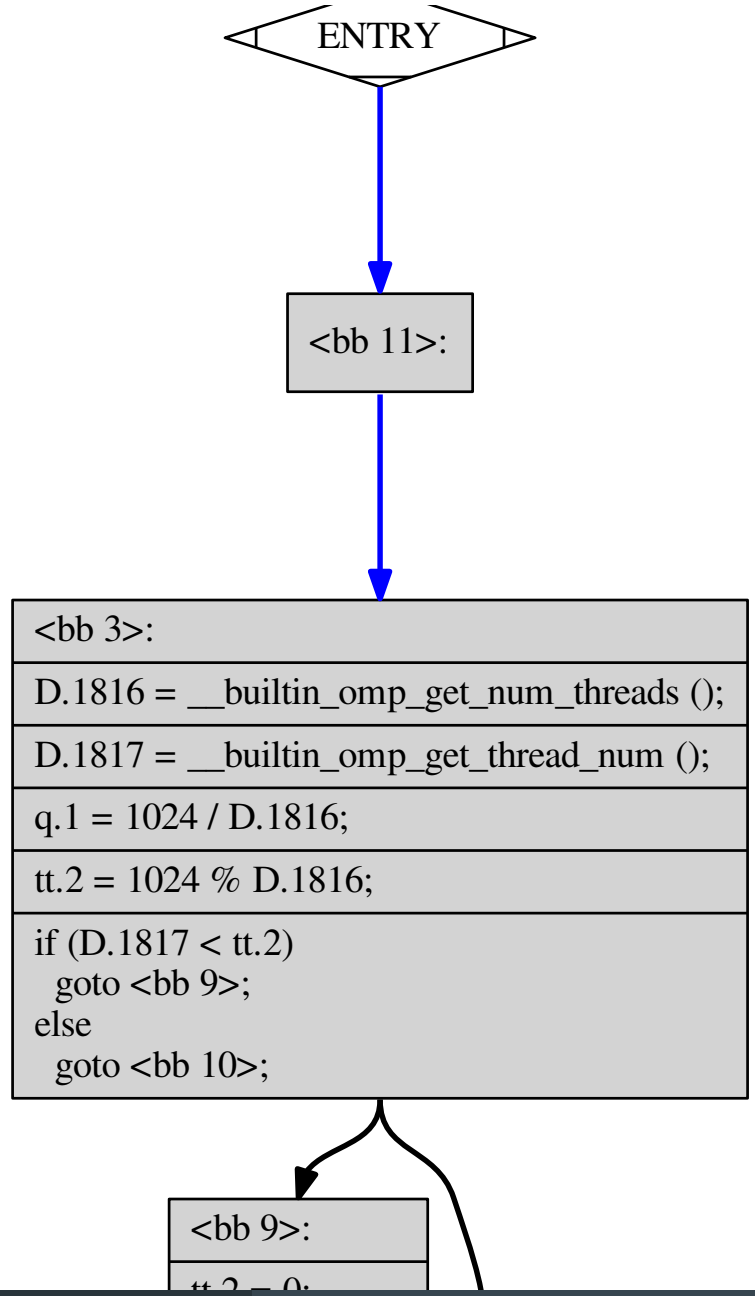
```
 1     .file "for-schedule-static-upper
           -bound-value.c"
 2     .text
 3     .globl main
 4     .type main, @function
 5 main:
 6     pushq %rbp
 7     movq %rsp, %rbp
 8     movl $0, %edx
 9     movl $0, %esi
10     movl $main._omp_fn.0, %edi
11     call GOMP_parallel_start
12     movl $0, %edi
13     call main._omp_fn.0
14     call GOMP_parallel_end
15     popq %rbp
16     ret
17     .size main, .-main
18     .type main._omp_fn.0, @function
19 main._omp_fn.0:
20     pushq %rbp
21     movq %rsp, %rbp
22     pushq %rbx
23     subq $40, %rsp
24     movq %rdi, -40(%rbp)
25     call omp_get_num_threads
26     movl %eax, %ebx
27     call omp_get_thread_num
28     movl %eax, %esi
29     movl $1024, %eax
30     cltd
34     cltd
35     idivl %ebx
36     movl %edx, %eax
37     cmpl %eax, %esi
38     jl .L3
39 .L6:
40     imull %ecx, %esi
41     movl %esi, %edx
42     addl %edx, %eax
43     leal (%rax,%rcx), %edx
44     cmpl %edx, %eax
45     jge .L2
46     movl %eax, -20(%rbp)
47 .L5:
48     addl $1, -20(%rbp)
49     cmpl %edx, -20(%rbp)
50     jl .L5
51     jmp .L2
52 .L3:
53     movl $0, %eax
54     addl $1, %ecx
55     jmp .L6
56 .L2:
57     addq $40, %rsp
58     popq %rbx
59     popq %rbp
60     ret
61     .size main._omp_fn.0, .-
           main._omp_fn.0
62     .ident "GCC: (Debian 4.8.4-1) 4
           .8.4"
63     .section .note.GNU-stack,"",
```

Loops: Construtor `for`

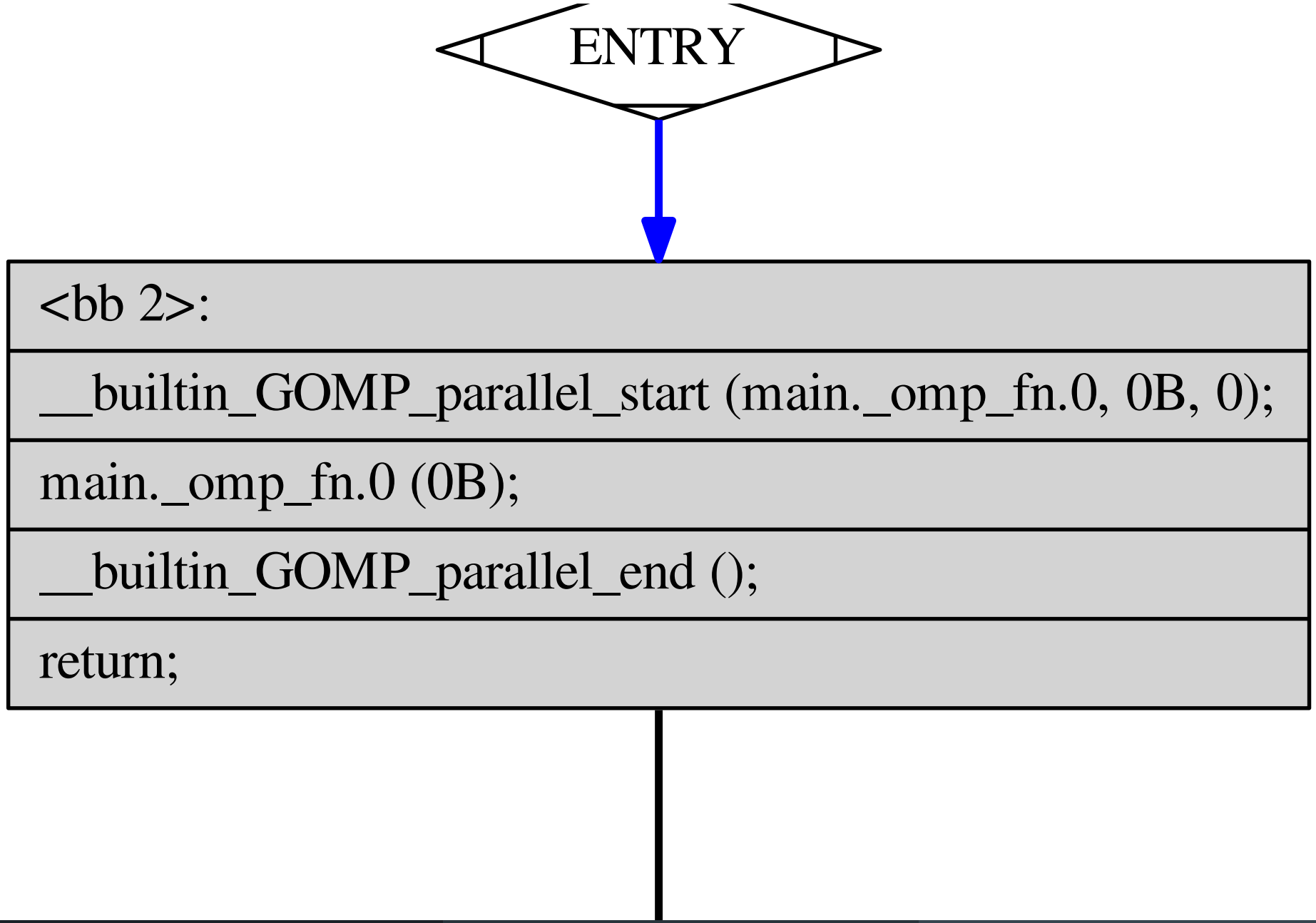


Loops: Construtor `for`

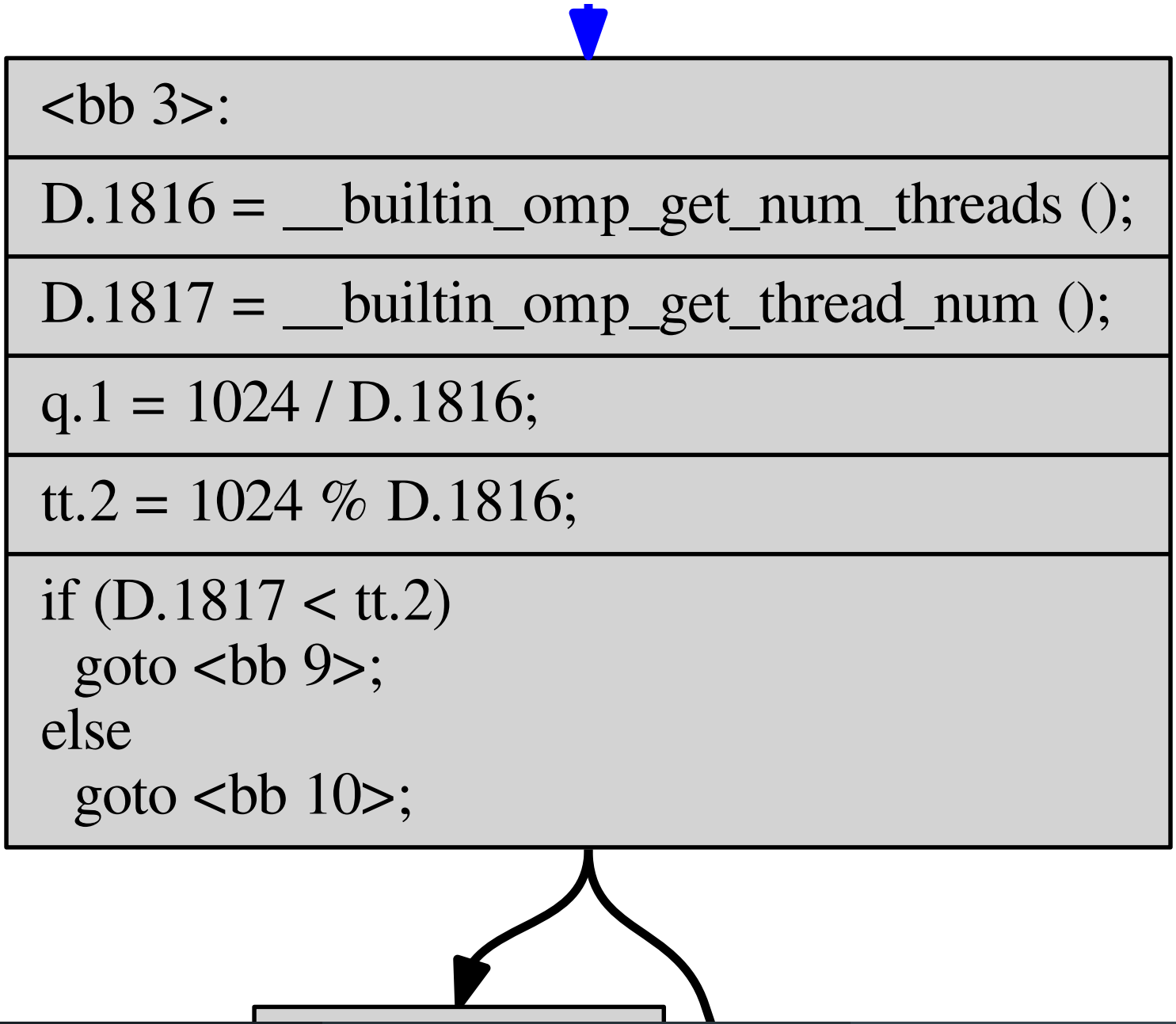




Loops: Construtor `for`



Loops: Construtor `for`



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(dynamic|runtime|
      guided)
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 da *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);
```

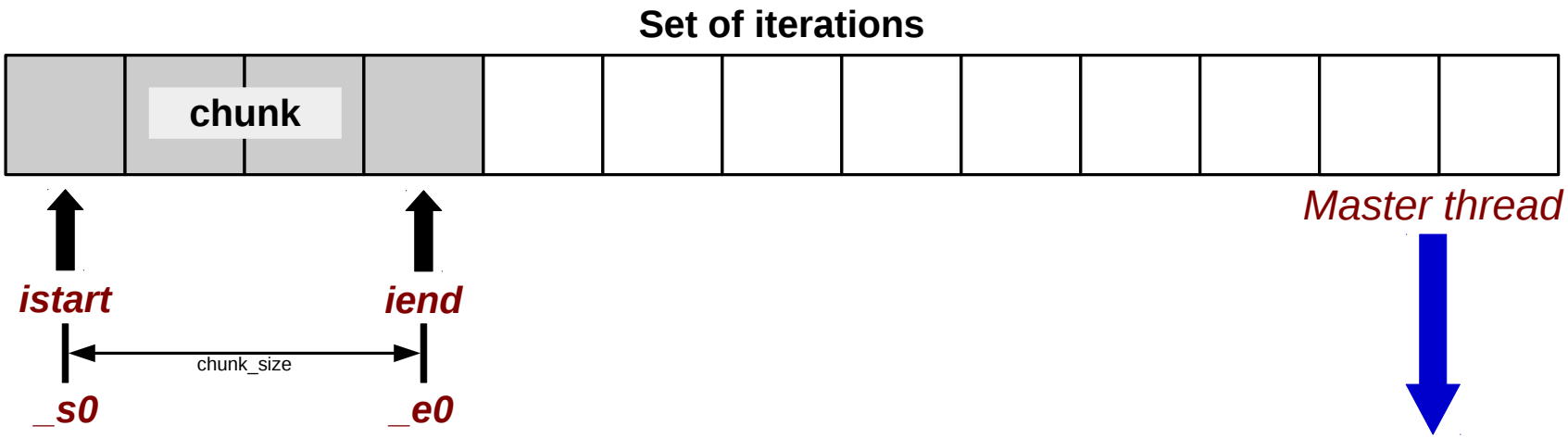
Loops: Construtor `for` – Primeiro formato II

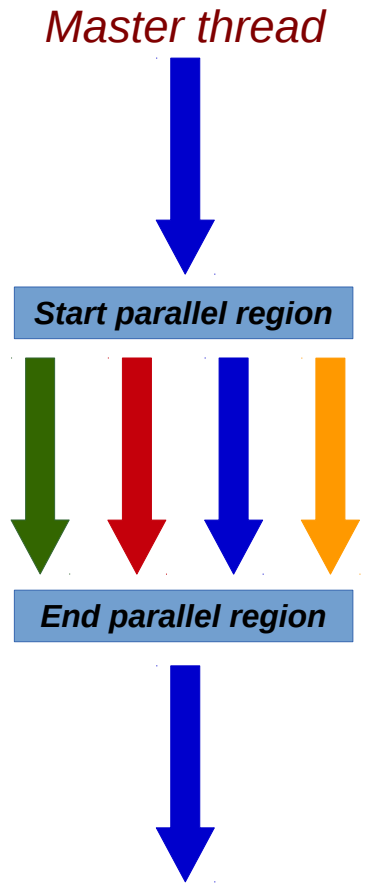
- O código expandido para o *primeiro formato*:

```
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

- Execução de Chunks pelas Threads






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

GOMP_parallel_loop_runtime_start (subfunction, NULL, 0, lb, ub+1, 1, 0);
subfunction (NULL);
GOMP_parallel_end ();
```

Original Loop

Loops: Construtor `for` – Primeiro formato IV

- 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 V

- O código `GIMPLE` para o primeiro formato:

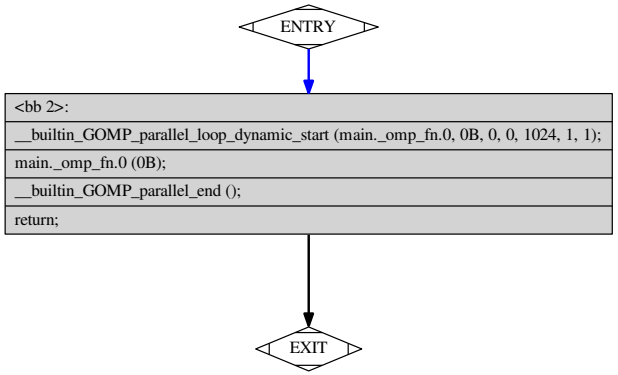
```
 1 <bb 4>:
 2   i = i + 1;
 3   if (i < D.1822)
 4     goto <bb 4>;
 5   else
 6     goto <bb 9>;
 7
 8 <bb 9>:
 9   D.1823 = __builtin_GOMP_loop_dynamic_next (&.istart0.1, &.iend0.2);
10   if (D.1823 != 0)
11     goto <bb 8>;
12   else
13     goto <bb 5>;
14
15 <bb 5>:
16   __builtin_GOMP_loop_end_nowait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

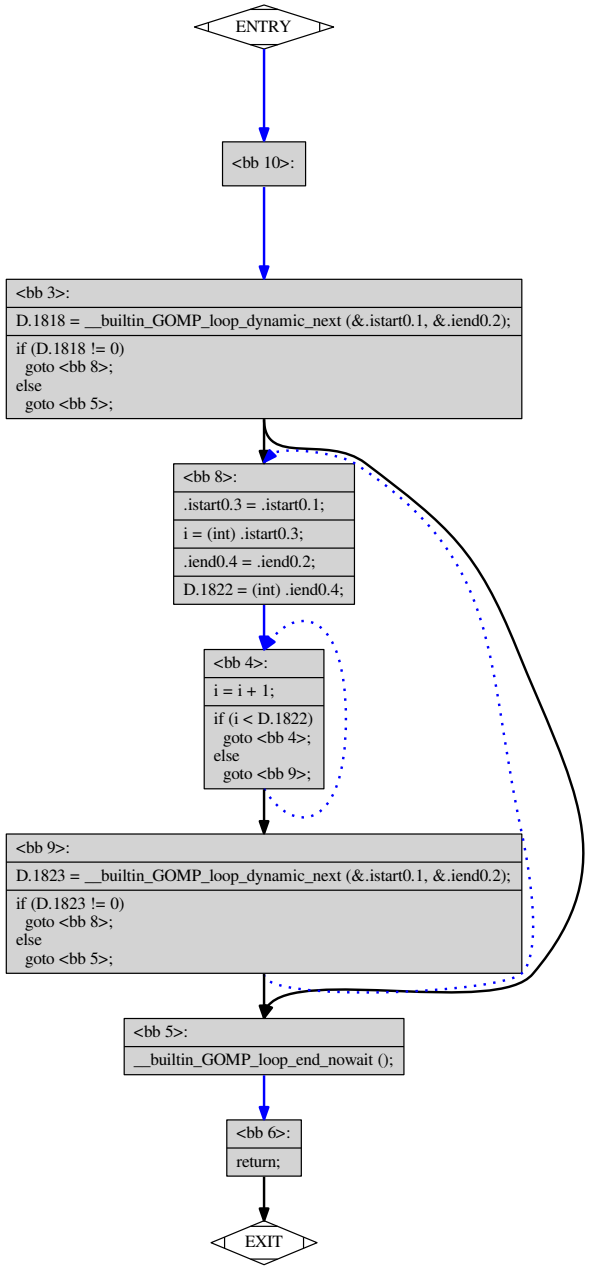
Loops: Construtor `for` – Primeiro formato VI

- O código em *assembly* para o primeiro formato:

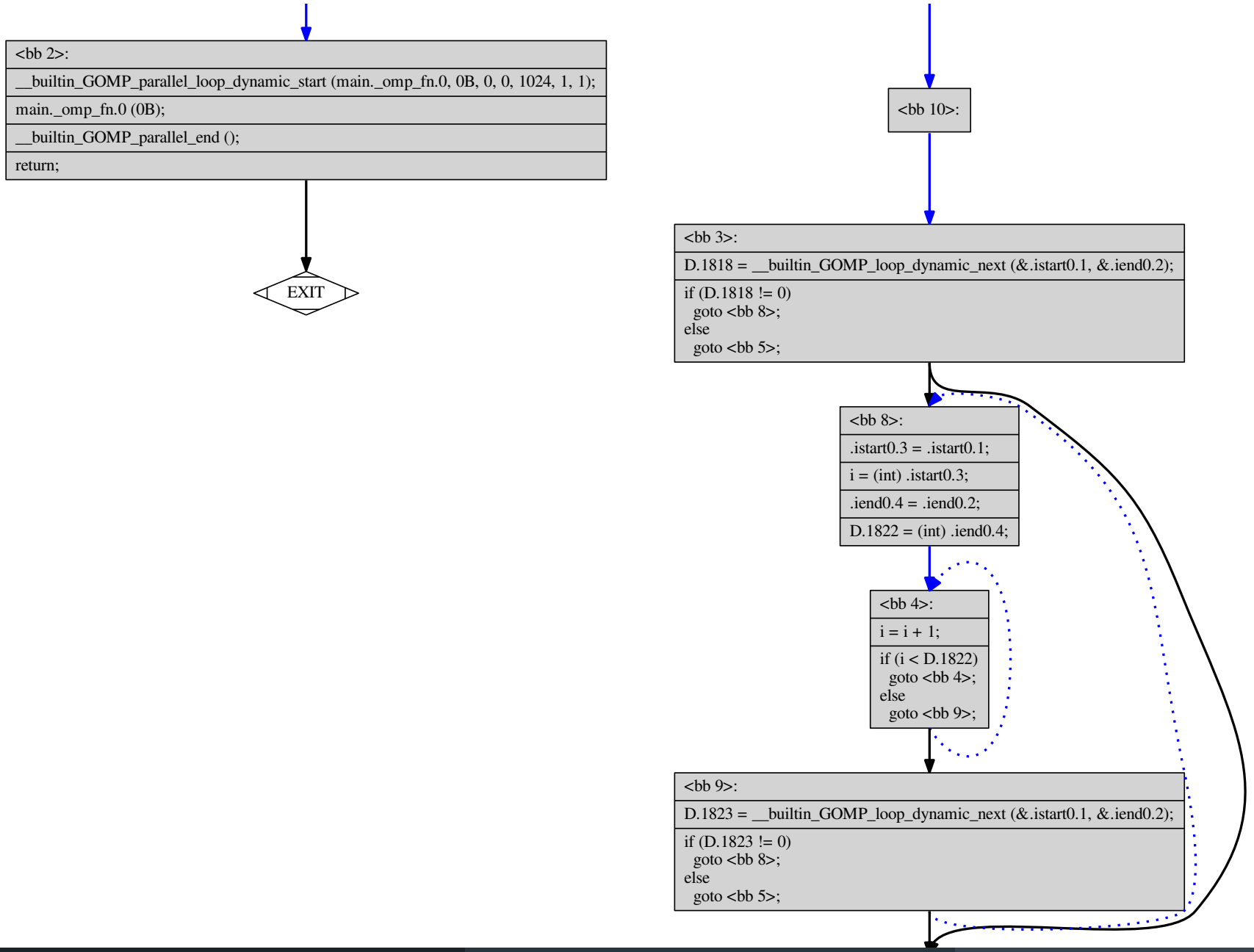
```
 1     .file "for-schedule-dynamic-
           upper-bound-value.c"
 2     .text
 3     .globl main
 4     .type main, @function
 5 main:
 6     pushq %rbp
 7     movq %rsp, %rbp
 8     subq $32, %rsp
 9     movq $1, (%rsp)
10     movl $1, %r9d
11     movl $1024, %r8d
12     movl $0, %ecx
13     movl $0, %edx
14     movl $0, %esi
15     movl $main._omp_fn.0, %edi
16     call
           GOMP_parallel_loop_dynamic_start

17     movl $0, %edi
18     call main._omp_fn.0
19     call GOMP_parallel_end
20     leave
21     ret
22     .size main, .-main
23     .type main._omp_fn.0, @function
24 main._omp_fn.0:
25     pushq %rbp
26     movq %rsp, %rbp
27     subq $48, %rsp
28     movq %rdi, -40(%rbp)
30     leaq -24(%rbp), %rax
31     movq %rdx, %rsi
32     movq %rax, %rdi
33     call GOMP_loop_dynamic_next
34     testb %al, %al
35     je .L3
36 .L5:
37     movq -24(%rbp), %rax
38     movl %eax, -4(%rbp)
39     movq -16(%rbp), %rax
40 .L4:
41     addl $1, -4(%rbp)
42     cmpl %eax, -4(%rbp)
43     jl .L4
44     leaq -16(%rbp), %rdx
45     leaq -24(%rbp), %rax
46     movq %rdx, %rsi
47     movq %rax, %rdi
48     call GOMP_loop_dynamic_next
49     testb %al, %al
50     jne .L5
51 .L3:
52     call GOMP_loop_end_nowait
53     leave
54     ret
55     .size main._omp_fn.0, .-
           main._omp_fn.0
56     .ident "GCC: (Debian 4.8.4-1) 4
           .8.4"
57     .section .note.GNU-stack,"",
           @progbits
```

Loops: Construtor `for`

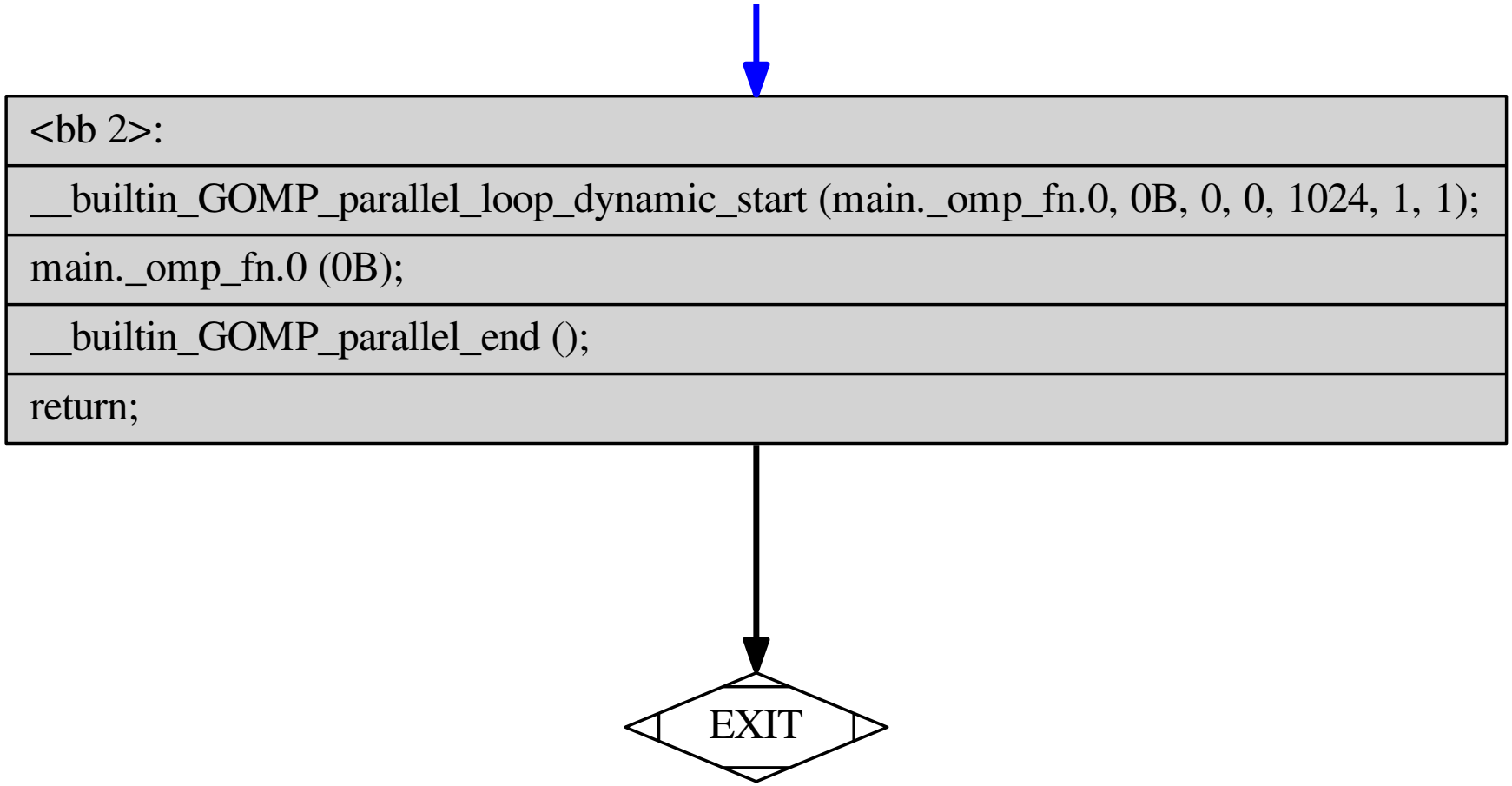




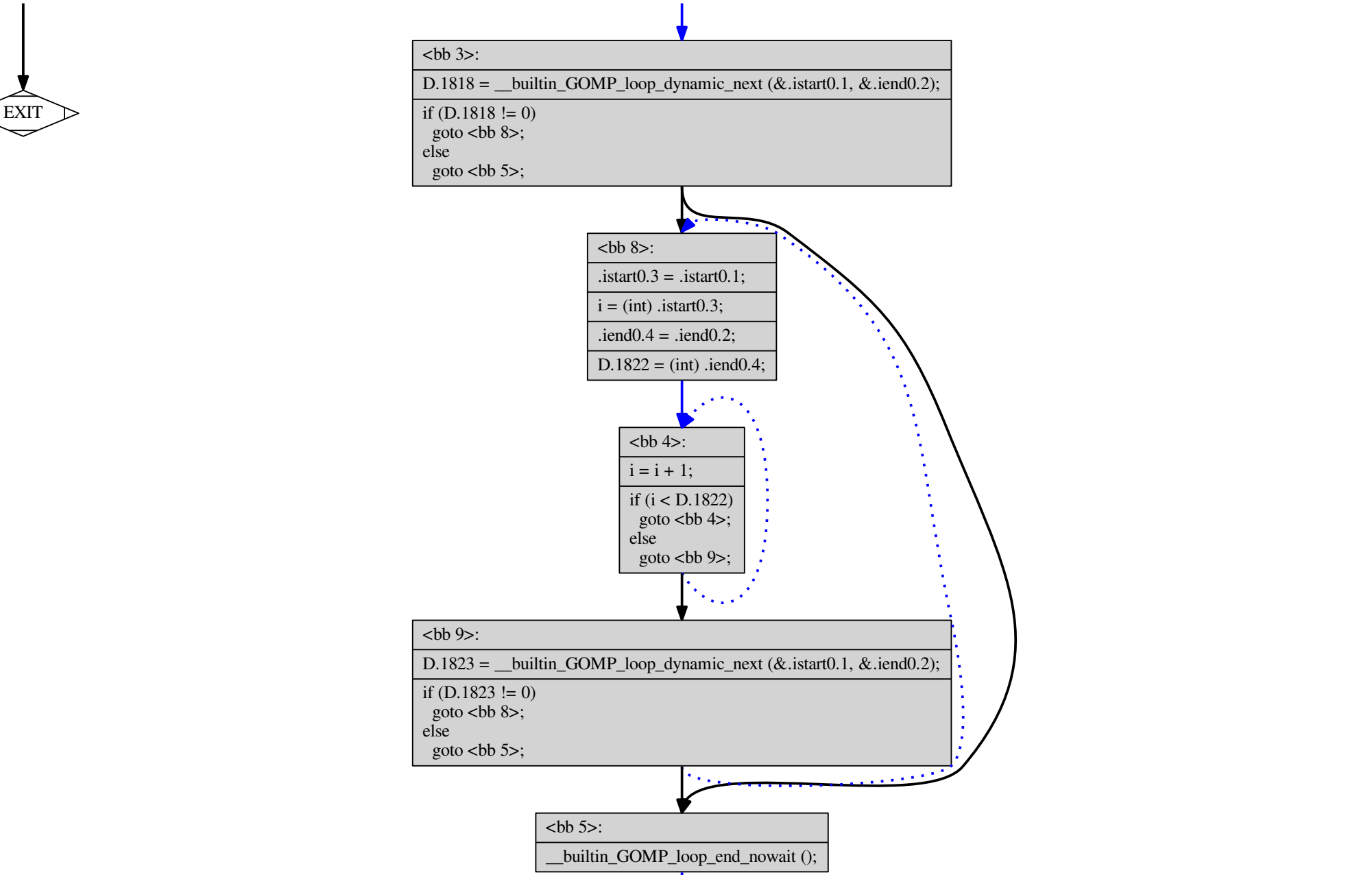
Loops: Construtor `for`



Loops: Construtor `for`



Loops: Construtor `for`



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(dynamic|runtime|
      guided)
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 da *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:

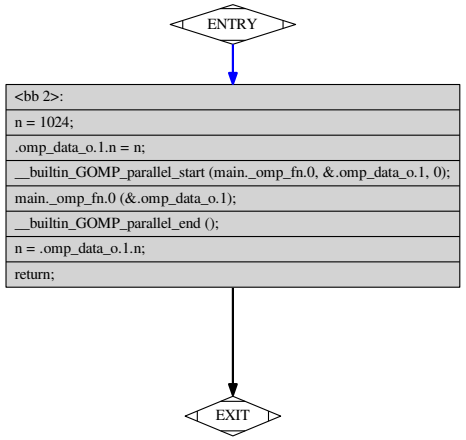
```
 1 <bb 8>:
 2   .istart0.4 = .istart0.2;
 3   i = (int) .istart0.4;
 4   .iend0.5 = .iend0.3;
 5   D.1830 = (int) .iend0.5;
 6
 7 <bb 4>:
 8   i = i + 1;
 9   if (i < D.1830)
10     goto <bb 4>;
11   else
12     goto <bb 9>;
13
14 <bb 9>:
15   D.1831 = __builtin_GOMP_loop_dynamic_next (&.istart0.2, &.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 }
```

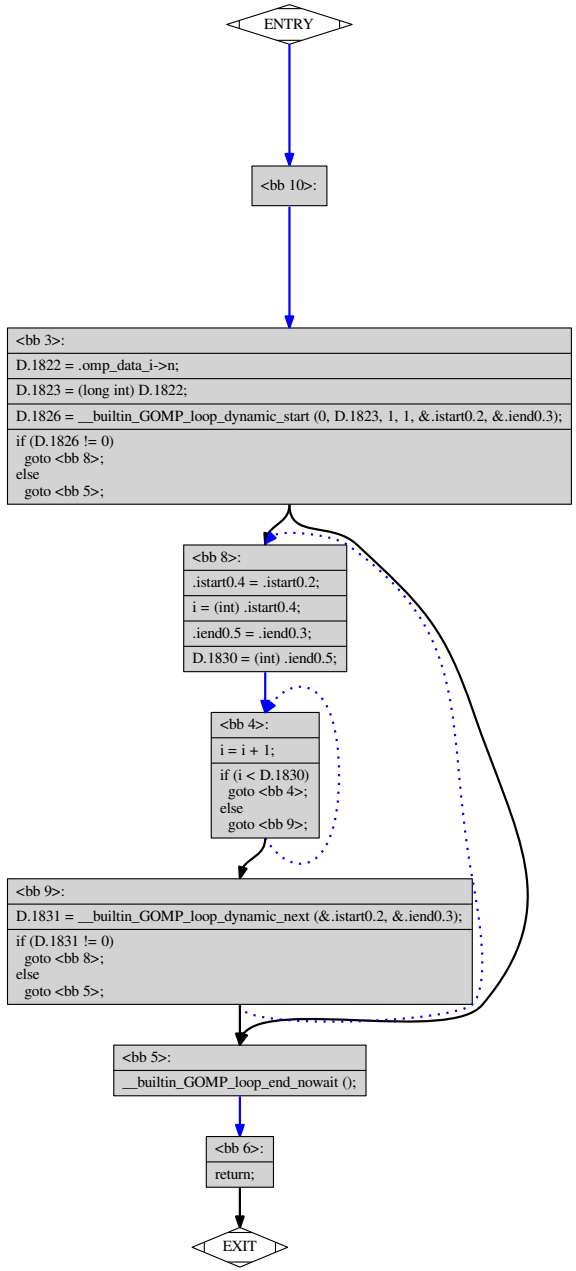
Loops: Construtor `for` – Segundo formato V

- O código em *assembly* para o segundo formato:

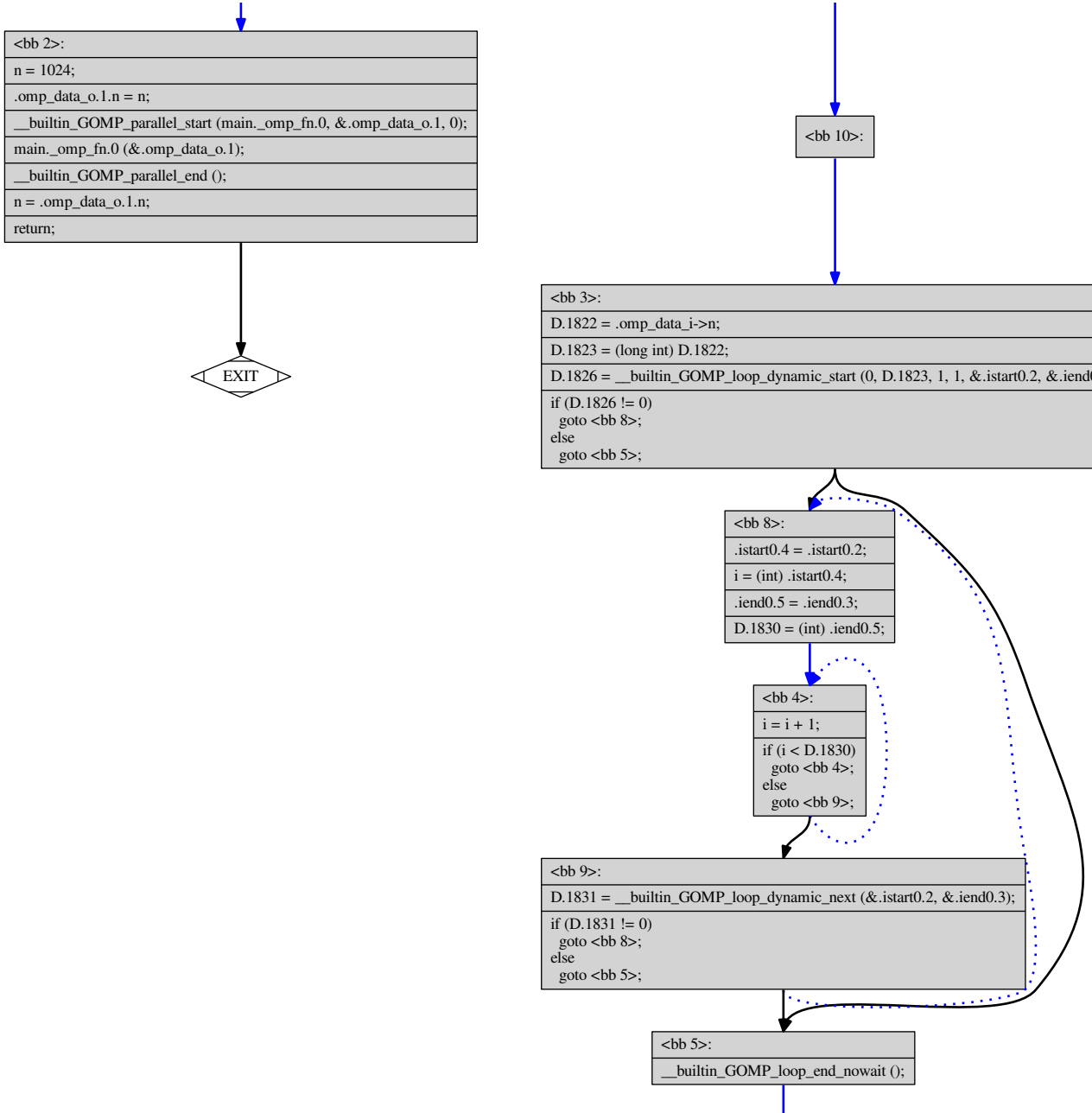
```
 1     .file "for-schedule-dynamic-
           upper-bound-variable.c"
 2     .text
 3     .globl main
 4     .type main, @function
 5 main:
 6     pushq %rbp
 7     movq %rsp, %rbp
 8     subq $32, %rsp
 9     movl $1024, -4(%rbp)
10     movl -4(%rbp), %eax
11     movl %eax, -32(%rbp)
12     leaq -32(%rbp), %rax
13     movl $0, %edx
14     movq %rax, %rsi
15     movl $main._omp_fn.0, %edi
16     call GOMP_parallel_start
17     leaq -32(%rbp), %rax
18     movq %rax, %rdi
19     call main._omp_fn.0
20     call GOMP_parallel_end
21     movl -32(%rbp), %eax
22     movl %eax, -4(%rbp)
23     leave
24     ret
25     .size main, .-main
26     .type main._omp_fn.0, @function
27 main._omp_fn.0:
28     pushq %rbp
29     movq %rsp, %rbp
30     subq $48, %rsp
36     leaq -24(%rbp), %rdx
37     movq %rcx, %r9
38     movq %rdx, %r8
39     movl $1, %ecx
40     movl $1, %edx
41     movq %rax, %rsi
42     movl $0, %edi
43     call GOMP_loop_dynamic_start
44     testb %al, %al
45     je .L3
46 .L5:
47     movq -24(%rbp), %rax
48     movl %eax, -4(%rbp)
49     movq -16(%rbp), %rax
50 .L4:
51     addl $1, -4(%rbp)
52     cmpl %eax, -4(%rbp)
53     jl .L4
54     leaq -16(%rbp), %rdx
55     leaq -24(%rbp), %rax
56     movq %rdx, %rsi
57     movq %rax, %rdi
58     call GOMP_loop_dynamic_next
59     testb %al, %al
60     jne .L5
61 .L3:
62     call GOMP_loop_end_nowait
63     leave
64     ret
65     .size main._omp_fn.0, .-
           main._omp_fn.0
66     .ident "GCC: (Debian 4.8.4-1) 4
```

Loops: Construtor `for`

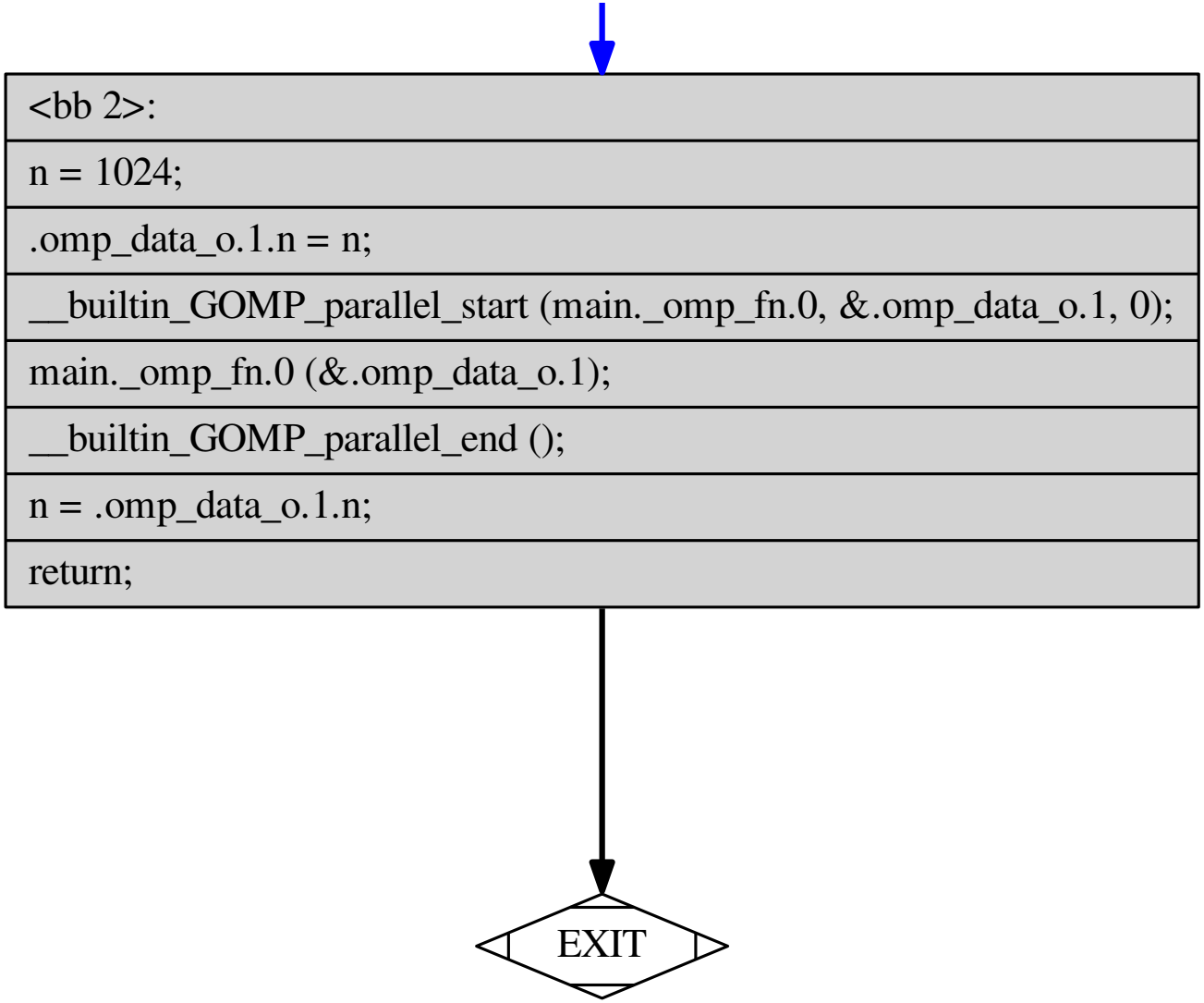




Loops: Construtor `for`



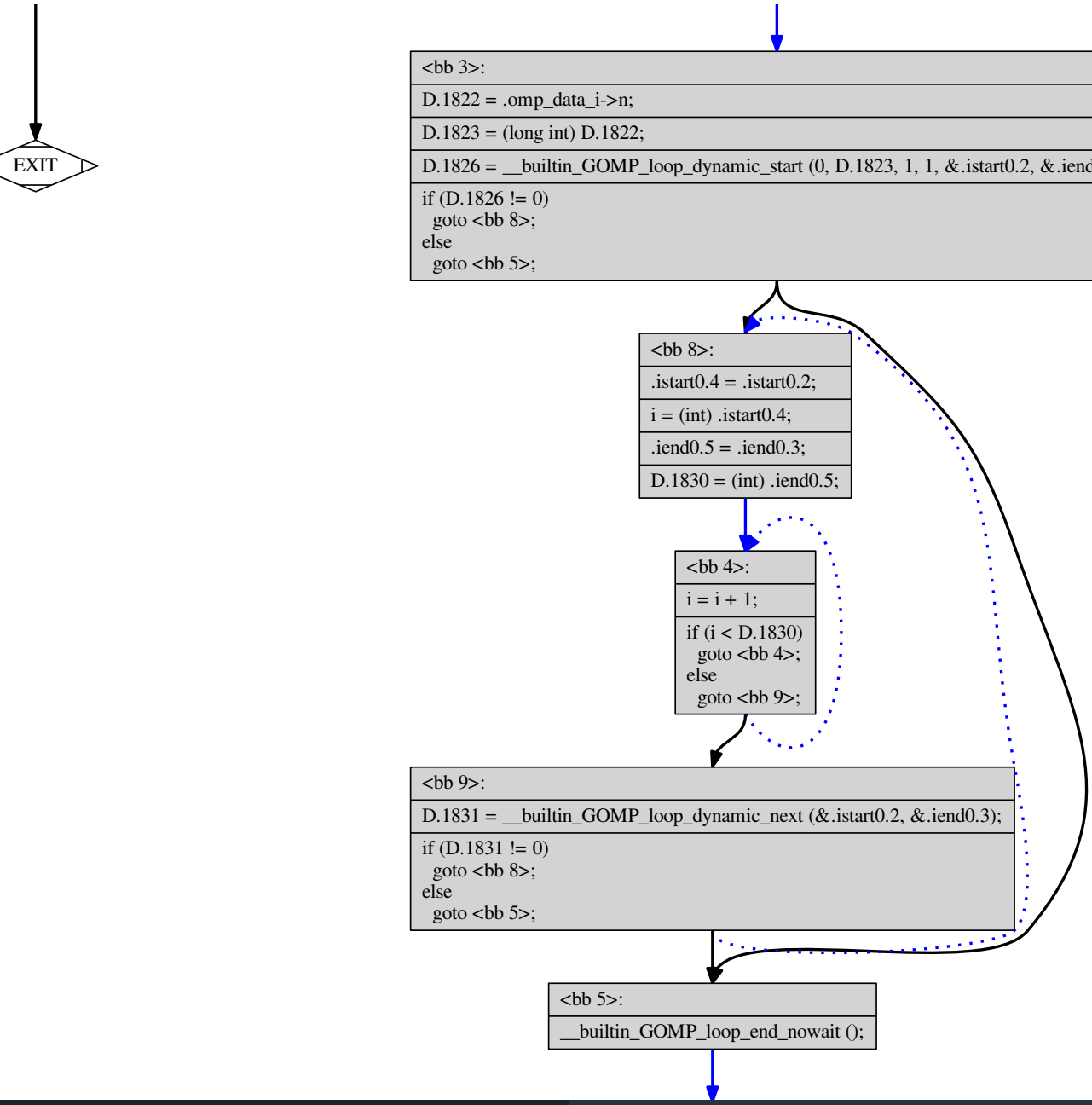
Loops: Construtor `for`



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

if (D.1826
goto <bb
else
goto <bb

Loops: Construtor `for`



Loops: Construtor `task` I

- O construtor *task* permite a criação de tarefas explícitas.

Construtor *task*:

```
#pragma omp task
{
  // bloco de codigo.
}
```

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

Loops: Construtor `task` I

- 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);
```

Loops: Construtor task II

- Neste exemplo duas *tasks* são criadas dentro de uma região paralela.
- A diretiva *single* é utilizada para garantir que o código seja executado por apenas uma das threads do time.
- Caso contário todas as threads criadas executariam o mesmo código criando cada uma delas duas *tasks*.

```
1  #pragma omp parallel num_threads(8)
2  {
3    #pragma omp single
4    {
5      printf("ESCOLA REGIONAL DE ");
6      #pragma omp task
7      {
8        printf("ALTO ");
9      }
10     #pragma omp task
11     {
12       printf("DESEMPENHO ");
13     }
14     #pragma omp taskwait
15
16     printf("DE SAO PAULO.\n");
17   }
18 }
```

Loops: Construtor task III

- 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, 8);
 5   main._omp_fn.0 (0B);
 6   __builtin_GOMP_parallel_end ();
 7   D.2259 = 0;
 8
 9 <L2>:
10   return D.2259;
11 }
12
13 main._omp_fn.0 (void * .omp_data_i
       ) {
14
15 <bb 18>:
16
17 <bb 3>:
18
19 <bb 4>:
20   D.2277 =
         __builtin_GOMP_single_start
         ();
21   if (D.2277 == 1)
22     goto <bb 5>;
23   else
24     goto <bb 10>;
25
26 <bb 10>:
```

```
34 <bb 17>:
35   __builtin_GOMP_task (main.
         _omp_fn.1, 0B, 0B, 0, 1, 1,
         0);
36
37 <bb 7>:
38
39 <bb 15>:
40   __builtin_GOMP_task (main.
         _omp_fn.2, 0B, 0B, 0, 1, 1,
         0);
41
42 <bb 9>:
43   __builtin_GOMP_taskwait ();
44   __builtin_puts (&"DE SAO PAULO."
         [0]);
45   goto <bb 10>;
46 }
47
48 main._omp_fn.2 (void * .omp_data_i
       ) {
49 <bb 14>:
50
51 <bb 8>:
52   printf ("DESEMPENHO ");
53   return;
54 }
55
56 main._omp_fn.1 (void * .omp_data_i
       ) {
57 <bb 16>:
```

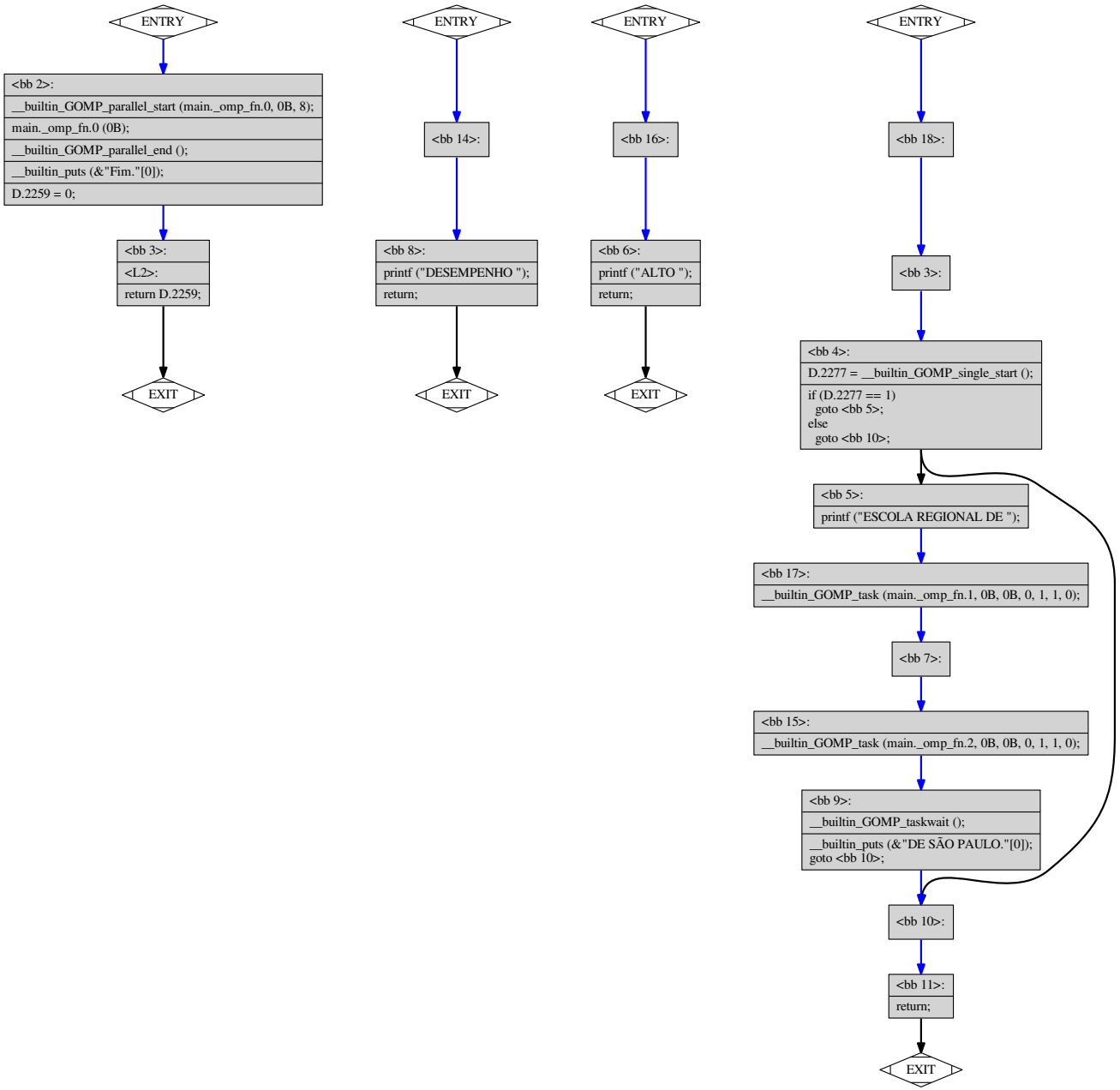
Loops: Construtor task IV

- O código em *assembly*:

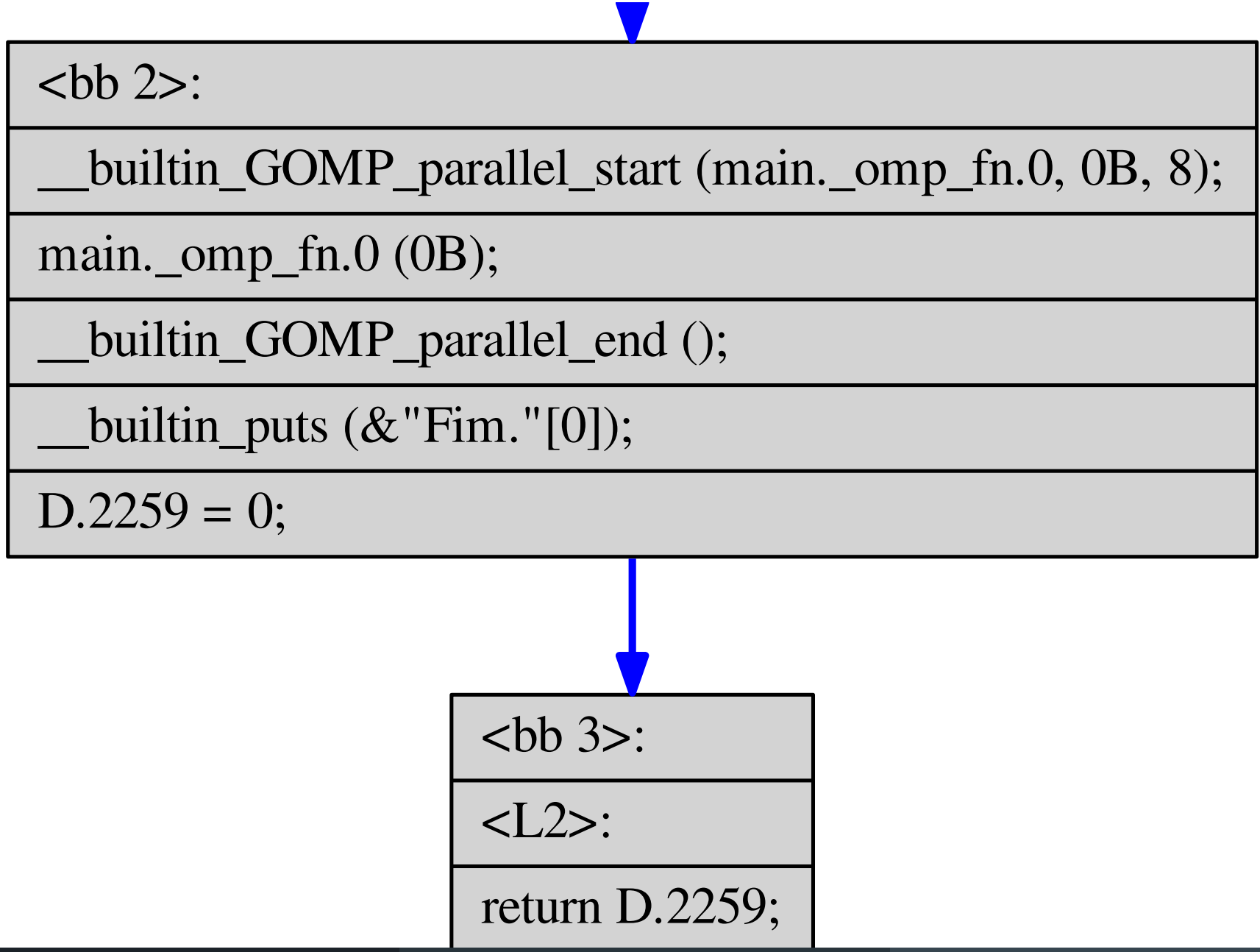
```
 1     .file "task-01.c"
 2     .section .rodata
 3 .LC0:
 4     .string "Fim."
 5     .text
 6     .globl main
 7     .type main, @function
 8 main:
 9     pushq %rbp
10     movq %rsp, %rbp
11     subq $16, %rsp
12     movl %edi, -4(%rbp)
13     movq %rsi, -16(%rbp)
14     movl $8, %edx
15     movl $0, %esi
16     movl $main._omp_fn.0, %edi
17     call GOMP_parallel_start
18     movl $0, %edi
19     call main._omp_fn.0
20     call GOMP_parallel_end
21     movl $.LC0, %edi
22     call puts
23     movl $0, %eax
24     leave
25     ret
26     .size main, .-main
27     .section .rodata
28     .LC1:
29     .string "ESCOLA REGIONAL DE "
30 .LC2:
31     .string "DE SAO PAULO."
```

```
55     movl $1, %r8d
56     movl $0, %ecx
57     movl $0, %edx
58     movl $0, %esi
59     movl $main._omp_fn.2, %edi
60     call GOMP_task
61     call GOMP_taskwait
62     movl $.LC2, %edi
63     call puts
64 .L3:
65     leave
66     ret
67     .size main._omp_fn.0, .-
           main._omp_fn.0
68     .section .rodata
69 .LC3:
70     .string "ALTO "
71     .text
72     .type main._omp_fn.1, @function
73 main._omp_fn.1:
74     pushq %rbp
75     movq %rsp, %rbp
76     subq $16, %rsp
77     movq %rdi, -8(%rbp)
78     movl $.LC3, %edi
79     movl $0, %eax
80     call printf
81     leave
82     ret
83     .size main._omp_fn.1, .-
           main._omp_fn.1
84     .section .rodata
```

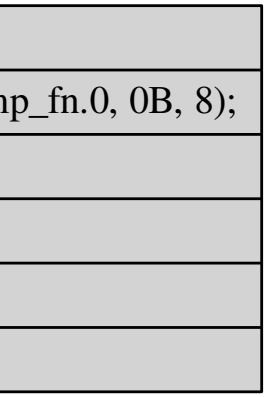
Loops: Construtor task

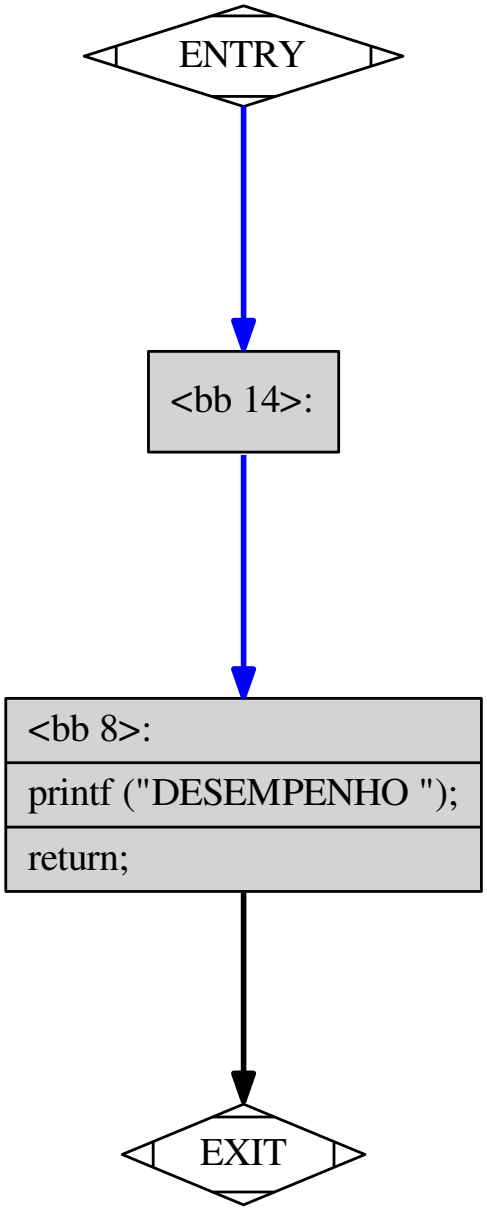


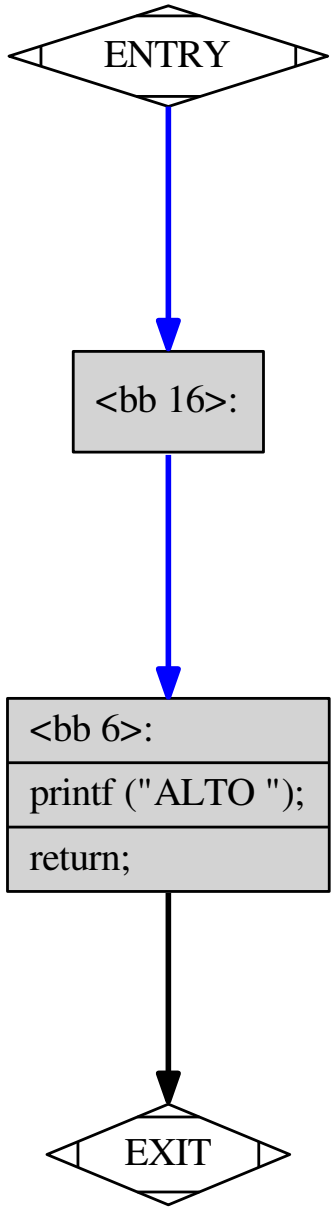
Loops: Construtor `task`

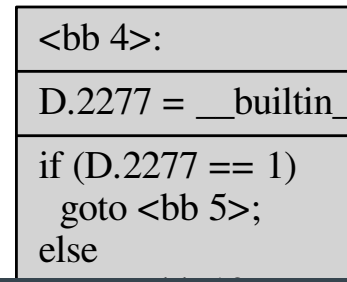


Loops: Construtor task

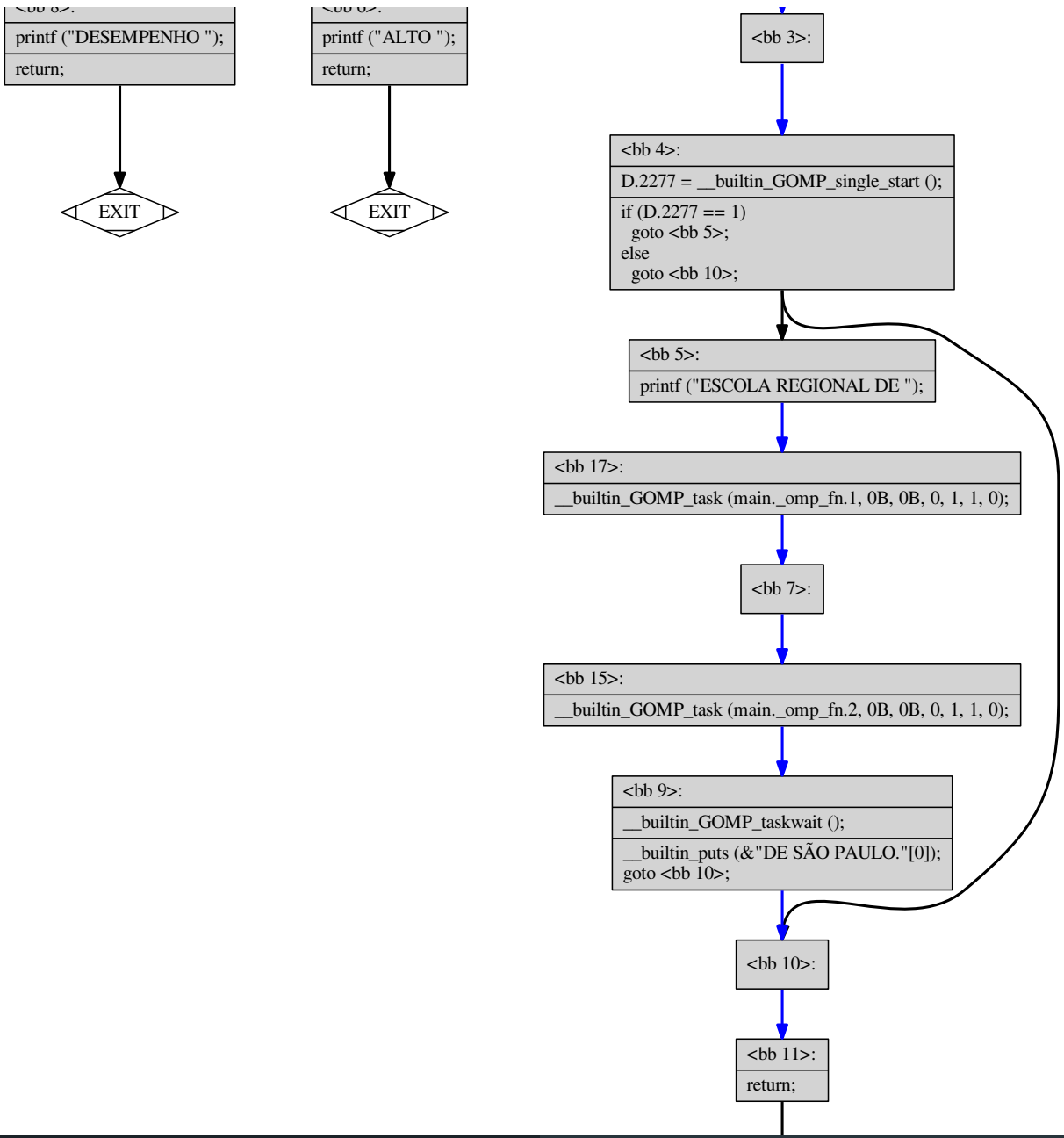








Loops: Construtor task



Formato de Código Interceptável

Formato de Código OpenMP Interceptável I

```
1 #pragma omp parallel for schedule(dynamic) chunk_size(N/num_threads)
2 for (i = 0; i < n; i++) {...}
```

Application	Call function/Operation	First format (chunk is a variable or one expression)
Parallel Region Start		`GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.1, 4);`
		>>> Create and start the team. *gomp_team_start(...)*
Call function	***main._omp_fn.0 (&.omp_data_o.1)***	***Function Context***
Loop Start	***Initial Chunk***	`GOMP_loop_dynamic_start (0, 1025, 1, D.1753, &.istart0.2, &.iend0.3);`
		>>> Create and start work share: Loop initialization. ***>>> Get the first set of iterations.*** *gomp_loop_init(…), gomp_iter_dynamic_next_*(...)*
	Execution	`<bb 9>:` `.istart0.4 = .istart0.2;` `i = (int) .istart0.4;` `.iend0.5 = .iend0.3;` `D.1760 = (int) .iend0.5;` `<bb 4>:` `D.1761 = (long unsigned int) i;` `D.1762 = D.1761 * 4;` `D.1763 = .omp_data_i->a;` `D.1764 = D.1763 + D.1762;` `*D.1764 = 1;` `i = i + 1;` `if (i < D.1760)` `goto <bb 4>;` `else` `goto <bb 10>;`
	Next Chunk	`GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);`
		>>> Get the next set of iterations. *gomp_iter_dynamic_next_*(...)*
Loop End	***Return of Function Call***	`GOMP_loop_end_nowait ();`
		>>> Finish the work share. *gomp_work_share_end_nowait()*
Parallel Region End		`GOMP_parallel_end ();`
		>>> Finish the parallel region. *gomp_team_end ();*

Formato de Código OpenMP Interceptável II

```
1 #pragma omp parallel for schedule(dynamic) chunk_size(64)
2 for (i = 0; i < n; i++) {...}
```

Application	Call function/Operation	Second format (chunk is a value or a constant)
Parallel Region Start		`GOMP_parallel_loop_dynamic_start (main._omp_fn.0, &.omp_data_o.1, 4, 0, 1025, 1, 4);`
		>>> Create and start the team, with Initialization of loop. *gomp_new_team(…), gomp_loop_init(...), gomp_team_start(...)*
Call function	***main._omp_fn.0(&.omp_data_o.1)***	
Loop Start	***Initial Chunk***	`GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);`
		>>> Get the first set of iterations. *gomp_iter_dynamic_next_*(...)*

	Execution	`<bb 9>:`
		`.istart0.4 = .istart0.2;`
		`i = (int) .istart0.4;`
		`.iend0.5 = .iend0.3;`
		`D.1760 = (int) .iend0.5;`
		
		`<bb 4>:`
		`D.1761 = (long unsigned int) i;`
		`D.1762 = D.1761 * 4;`
		`D.1763 = .omp_data_i->a;`
		`D.1764 = D.1763 + D.1762;`
		`*D.1764 = 1;`
		`i = i + 1;`
		`if (i < D.1760)`
		`goto <bb 4>;`
		`else`
		`goto <bb 10>;`

	Next Chunk	`GOMP_loop_dynamic_next (&.istart0.2, &.iend0.3);`
		>>> Get the next set of iterations. *gomp_iter_dynamic_next_*(...)*
Loop End	***Return of Function Call***	`GOMP_loop_end_nowait ();`
		>>> Finish the work share. *gomp_work_share_end_nowait()*
Parallel Region End		`GOMP_parallel_end ();`
		>>> Finish the parallel region. *gomp_team_end ();*

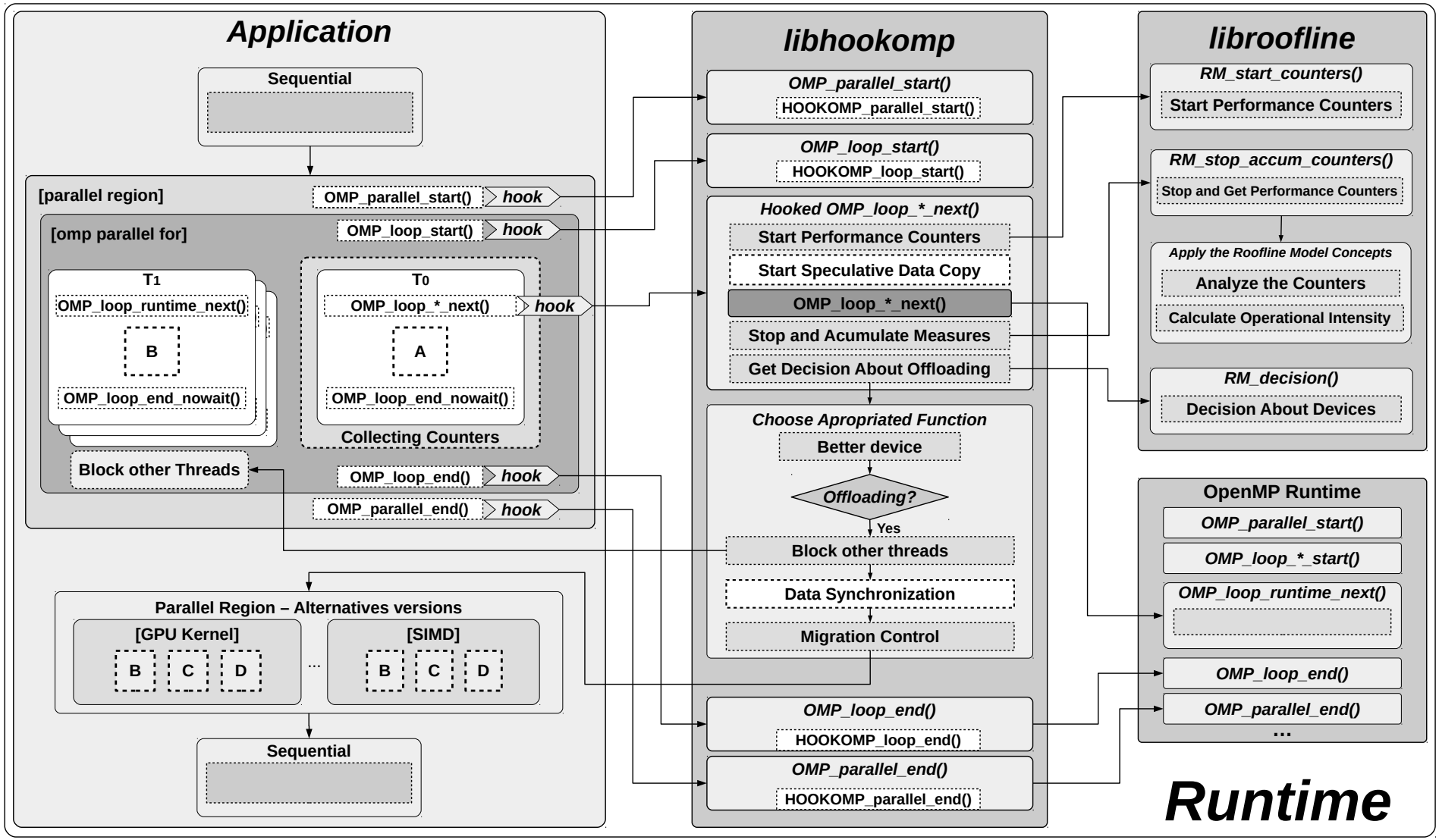
Hook para OpenMP

Hook para OpenMP I

- O conceito de interceptação por *hooking* pode ser utilizado no desenvolvimento de bibliotecas.
- Para criar um *hooks* para funções da `libgomp` é necessário criar uma biblioteca que tenha funções com o mesmo nome das funções disponibilizadas via ABI.
- Bibliotecas que podem ser pré-carregadas para alterarem o comportamento da execução de aplicações.
- Essa técnica pode ser utilizada para a execução de código pré ou pós chamada ao runtime `OpenMP`.
- O que pode cobrir desde *logging*, criação de *traces*[a] , monitoramento para avaliação de desempenho[b] ou *offloading* de código para dispositivos aceleradores.

[a] Trahay et al. (2011)
[b] Mohr et al. (2002)

Interação entre a Aplicação e as bibliotecas do runtime



Interceptando por hooking I

- 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*.

```
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 }
```

Interceptando por hooking II

- 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
```

Interceptando por hooking I

- 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    /* Recupera o ponteiro para a funcao OpenMP. */
5    GET_RUNTIME_FUNCTION(lib_GOMP_parallel_start, "
         GOMP_parallel_start");
6
7    /* Codigo a ser executado antes. */
8    PRE_GOMP_parallel_start();
9
10   /* Chamada a funcao original. */
11   lib_GOMP_parallel_start(fn, data, num_threads);
12
13   /* Codigo a ser executado depois. */
14   POST_GOMP_parallel_start();
15 }
```

Fim

Obrigado!

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.

Intel (2016b). Openmp* support.

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 (2011). OpenACC Application Programming Interface. Version 1.0.

OpenACC (2012). OpenACC Directives for Accelerators Site.

OpenACC (2013). OpenACC Application Programming Interface. Version 2.0.

OpenACC (2015a). OpenACC Application Programming Interface. Version 2.5.

OpenACC (2015b). OpenACC Directives for Accelerators.

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

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

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

Referências III

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.

Contatos

- Rogério Gonçalves: rogerioag@utfpr.edu.br, rag@ime.usp.br
- Alfredo Goldman: gold@ime.usp.br

Agradecimentos

Os autores agradecem à CAPES (BEX 3401/15-4, CAPES-COFECUB Projeto No. 828/15, *Choosing: Cooperation on Hybrid Computing Clouds for Energy Saving*) pela bolsa e a Fundação Araucária, Departamento de Ciência, Tecnologia e de Ensino Superior do Estado do Paraná (SETI-PR) e o Governo do Estado do Paraná pelo financiamento que viabilizou o DINTER e a realização deste trabalho.