

# Parallel Computing

Academic year – 2020/21, spring semester  
Computer science

## Lecture 5

Lecturer, instructor:

**Balakshin Pavel Valerievich**

(pvbalakshin@itmo.ru; pvbalakshin@hdu.edu.cn)

Assistant:

**Liang Tingting**

(liangtt@hdu.edu.cn)

# #pragma omp for

Iterations inside canonical loops for can be divided between several threads for parallel execution by specifying only one directive:

```
#pragma omp parallel for num_threads(2)
for(i = 0; i < 20; i++)
    a[i] = i + b;
```

Thread 0 will fill elements from `a[0]` to `a[9]`. Thread 1 will fill the elements from `a[10]` to `a[19]`. OpenMP creates a local copy of the `i` variable for each thread (so threads do not interfere with each other when executing `i++`). Variables ***a*** and ***b*** are shared by the threads (which is not a problem since both are read-only and are not changed within a loop).

**If the loop is not canonical, OpenMP will either not parallelize it or will parallelize it with errors.**

# Canonical “for” loops

The loop for is called canonical, if it is possible to calculate the number of iterations in advance at its beginning. This is possible if the following conditions are met simultaneously:

- there must not have a break operation/statement inside the loop;
- there is no goto operation inside the loop, leading outside the loop;
- the loop control variable must be an integer;
- the loop variable (iterator) is not changed inside the loop;
- the initialization expression must be an integer assignment;
- the increment expression must have integer increments or decrements only;
- the representation of the loop has the following form:

$$\text{for} \left( \begin{array}{l} \text{index} = \text{start} \\ \text{index} < \text{end} \\ \text{index} \leq \text{end} \\ \text{index} \geq \text{end} \\ \text{index} > \text{end} \end{array} ; \begin{array}{l} \text{index}++ \\ ++\text{index} \\ \text{index}-- \\ --\text{index} \\ \text{index} += \text{incr} \\ \text{index} -= \text{incr} \\ \text{index} = \text{index} + \text{incr} \\ \text{index} = \text{incr} + \text{index} \\ \text{index} = \text{index} - \text{incr} \end{array} \right)$$

# Variable scope

The iterator variable is considered private by default, all other variables are considered shared by default, so sometimes it is possible not to specify the scope of variables.

**Example 1** (by default: i – private; x – shared)

```
#pragma omp parallel for num_threads(2)
for(i = 0; i < 20; i++)
    a[i] = sqrt(sin(x) + cos(x) / ln(x));
```

**Example 2** (by default: j and k are used to be shared, that is wrong)

```
#pragma omp parallel for num_threads(X) private(j, k)
for(i = 2; i <= N-1; ++i)
    for(j = 2; j <= i; ++j)
        for(k = 1; k <= M; ++k)
            b[i][j] += a[i - 1][j] / k + a[i + 1][j] / k;
```

# #pragma omp for: conflicts

If iteration with sequence number  $k$  affects iteration results of iteration with sequence number  $m$ , the loop cannot be parallelized as it is impossible to predict in advance the order of termination of iterations by several threads. A programmer is responsible for detecting such conflicts.

OpenMP will not detect iterations interdependence and will compile the next error program:

```
#pragma omp parallel for num_threads(2)
for(i = 1; i < 20; i++)
    a[i] = 2 * a[i - 1];
```

Thread #0 will fail to fill item  $a[9]$  by the moment when thread #1 calculates the value  $a[10] = 2 * a[9]$ .

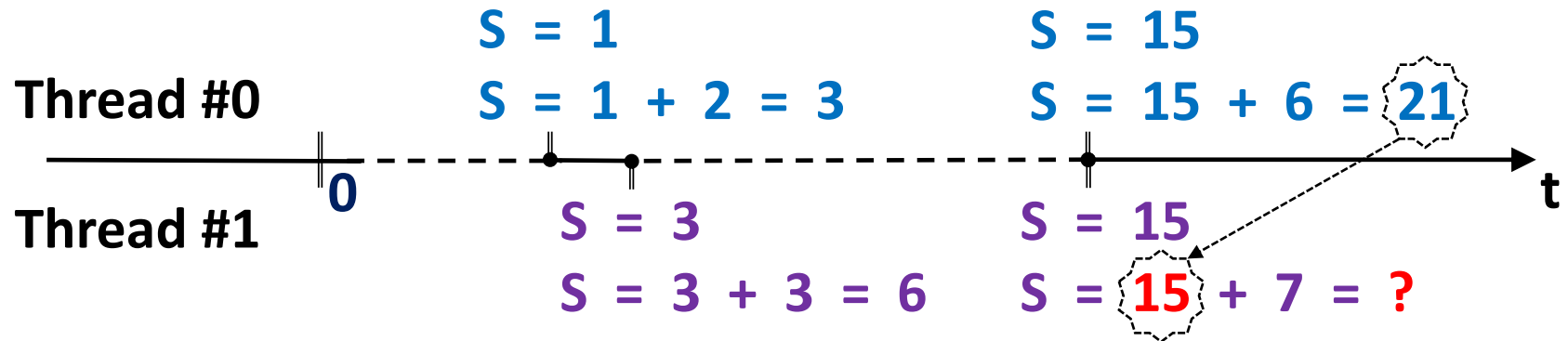
# #pragma omp for: conflicts (2)

If threads change the value of a shared variable in the loop body, the data in this variable may be distorted when writing is attempted at the same time.

A programmer is responsible for detecting such conflicts. OpenMP will not detect the conflict and will compile the next error program:


```
int s = 0;
#pragma omp parallel for num_threads(2) shared(a, s) private(i)
    for (i = 0; i < 20; i++) {
        a[i] = i;
        s = s + a[i];
    }
```

# #pragma omp for: conflicts (3)

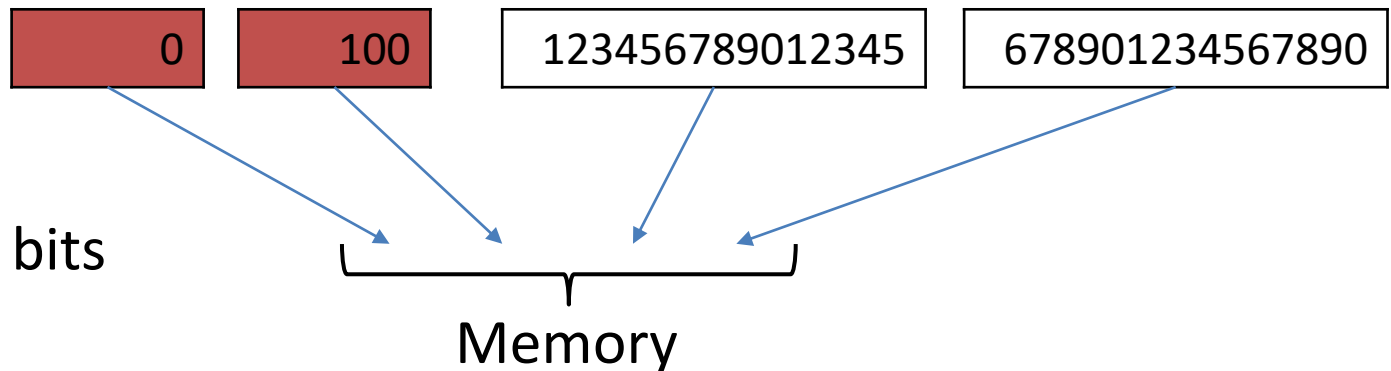


```
int s = 0;
#pragma omp parallel for num_threads(2) shared(a, s) private(i)
    for (i = 0; i < 20; i++) {
        a[i] = i;
        s = s + a[i];
    }
```

# #pragma omp for: conflicts (4)



Time	Thread #0	Thread #1
1	S = 0	
2		S = 1
...	...	...
15	S = 10	
...	...	...
100		S = 24
...	...	...
150	S = 100	S = 123456789012345678901234567890





# Shared variables defense

If threads change the value of the shared variable in the loop body, the assignment operation should be protected from simultaneous changes.

```
s = 0;
#pragma omp parallel for num_threads(2) private(i)
for(i = 0; i < 20; i++) {
    a[i] = i;
    #pragma omp critical
    s = s + a[i];
}
```

As a result  $s$  will be calculated correctly because the assignment operation will be executed sequentially instead of parallel by all the threads. This allows to solve the conflict between threads but **will negatively influence parallel speedup** since the corresponding part of the loop's instructions is marked as un-paralleled.

# #pragma omp atomic

Protection of the shared variable assignment operation is also possible with the help of the faster atomic directive, which can be used only for atomic hardware-accelerated commands/instructions of the "load-modify-store" kind:

- `x <instruction> = <expression>;` where `<instruction>` can be `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, `>>`;
- `x++;`
- `++x;`
- `x--;`
- `--x;`

```
#pragma omp atomic
```

```
Counter += 10;
```

```
#pragma omp atomic
```

```
Counter += a++; // error: instruction a++ will not be defend
```

# reduction parameter

```
#pragma omp parallel for num_threads(2) reduction(+:s)
for(i = 0; i < 20; ++i) {
    a[i] = i;
    s = s + a[i];
}
```

- As a result, *s* will be calculated correctly and the *s* modification instruction will be executed by threads in parallel (simultaneously) since OpenMP will create local copies of *s* for each thread. At the end of the loop, OpenMP will accumulate all the local copies and place them into the *s* shared variable.
- Besides the *+* operation, the reduction parameter can handle other operations: *-*, *\**, */*.
- OpenMP itself initializes local variables *s* with value 0 or 1 (depending on the instruction: *0 for sum, 1 for multiplication*) ignoring the initial value of *s* variable.

# #pragma omp for schedule

#pragma omp for schedule(static, chunk\_size)

#pragma omp for schedule(dynamic, chunk\_size)

#pragma omp for schedule(guided, chunk\_size)

#pragma omp for schedule(runtime)

#pragma omp for schedule(auto)

(runtime is taken from environmental variable OMP\_SCHEDULE)

```
#pragma omp parallel num_threads(8) {
```

```
    #pragma omp for schedule(static, 1)
```

```
        for (i = 0; i < 8; i++)
```

```
            printf("[1] iter %d, tid %d\n", i, omp_get_thread_num());
```

```
    #pragma omp for schedule(dynamic, 1)
```

```
        for (i = 0; i < 8; i++)
```

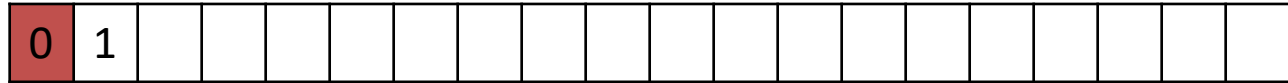
```
            printf("[2] iter %d, tid %d\n", i, omp_get_thread_num());
```

```
}
```

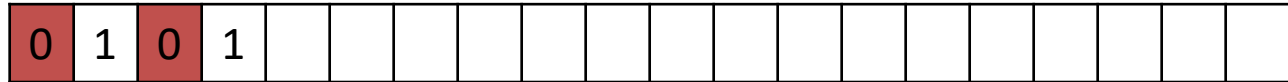
# schedule (static, 1)

num\_threads(2)

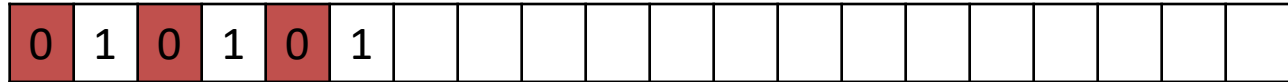
Time = 1



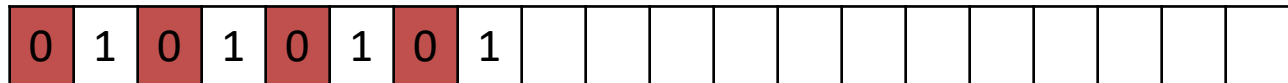
Time = 2



Time = 3

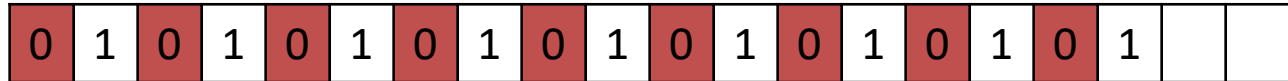


Time = 4

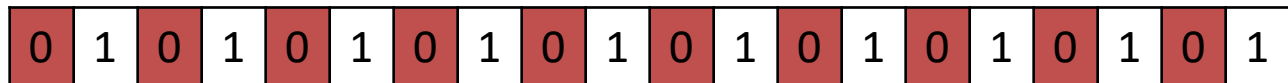


.....

Time = 9



Time = 10

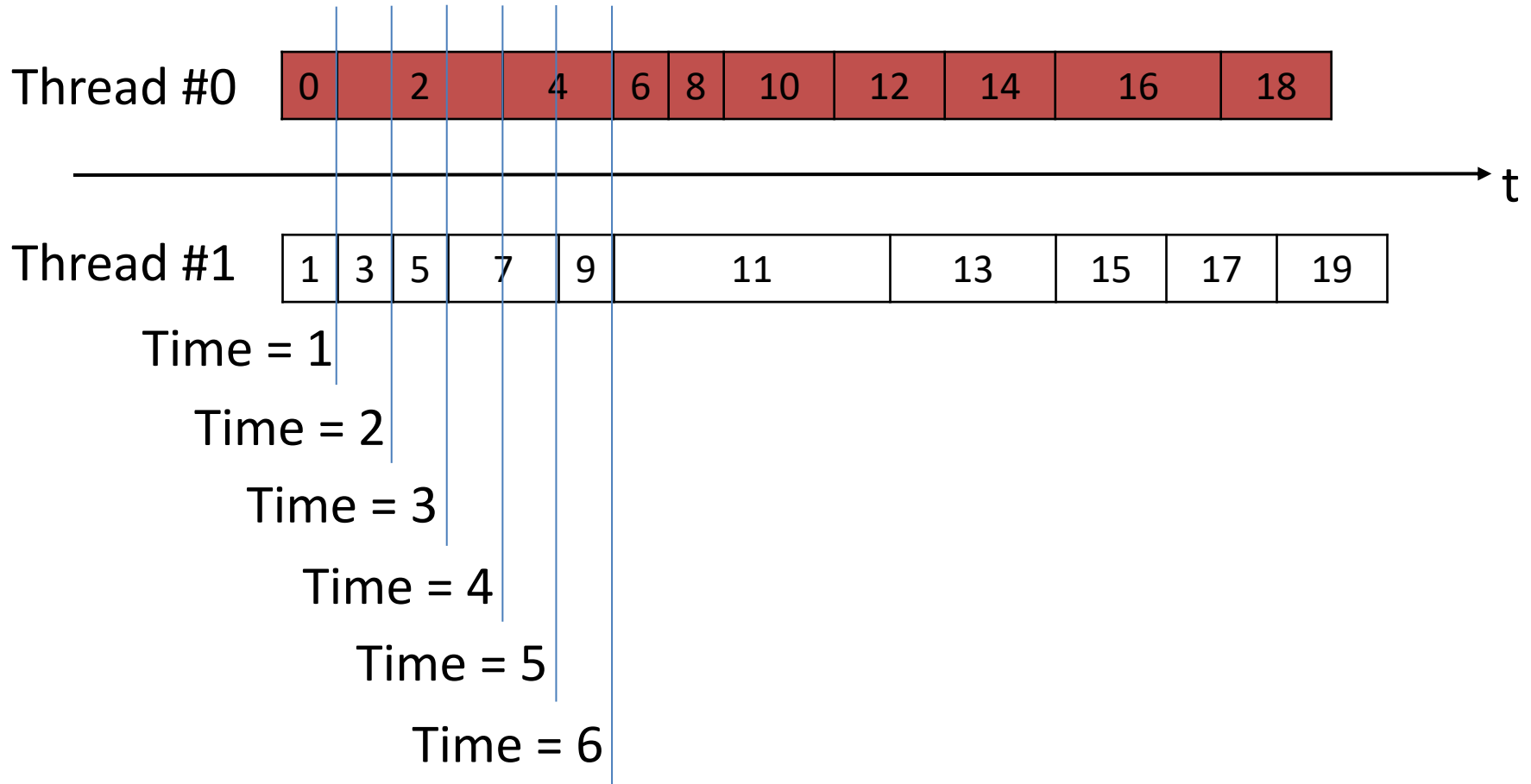


t

The iterations of thread #0 are marked in red, the iterations of thread #1 are marked in white.

# schedule (static, 1)

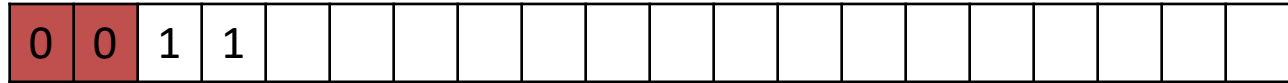
num\_threads(2)



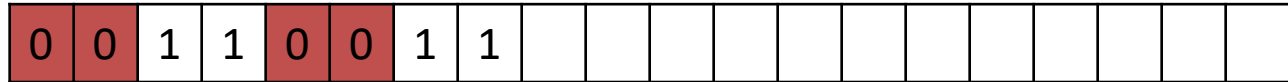
# schedule (static, 2)

num\_threads(2)

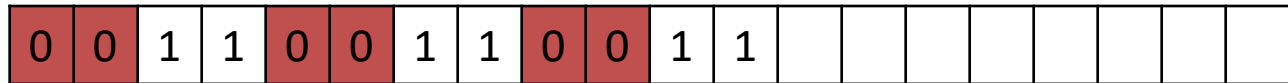
Time = 2



Time = 4

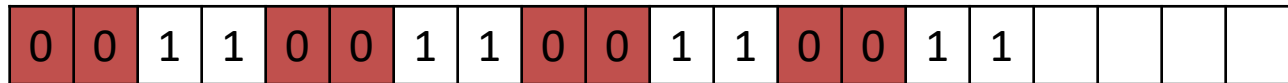


Time = 6

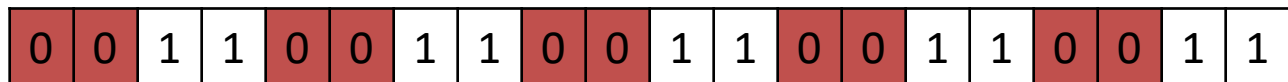


.....

Time = 8



Time = 10

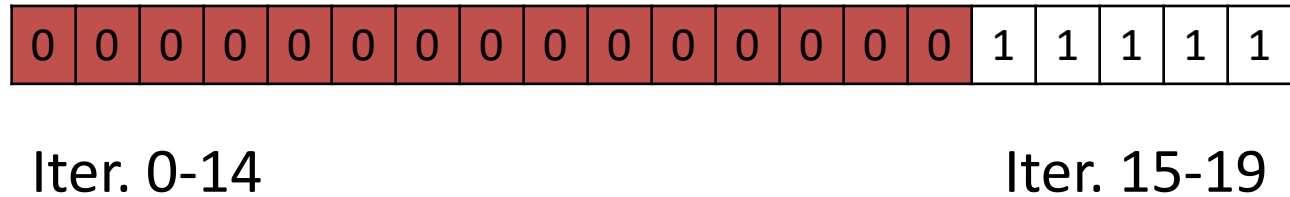


t

The iterations of thread #0 are marked in red, the iterations of thread #1 are marked in white.

# schedule (static, 15)

```
num_threads(2)
```



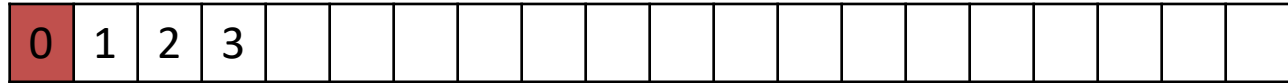
The iterations of thread #0 are marked in red, the iterations of thread #1 are marked in white.



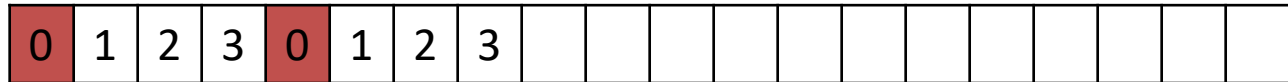
# schedule (static, 1)

num\_threads(4)

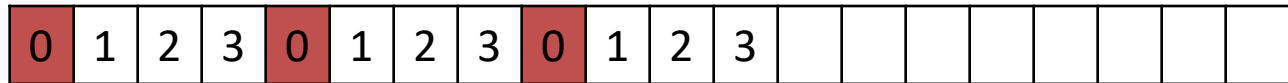
Time = 1



Time = 2

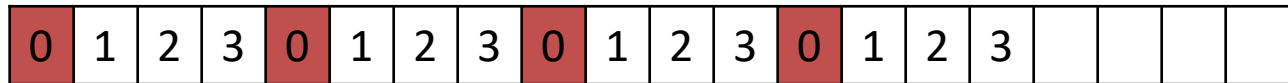


Time = 3

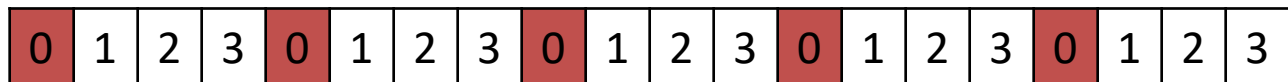


.....

Time = 4



Time = 5



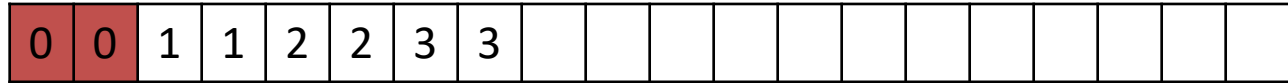
t

The iterations of thread #0 are marked in red, the iterations of other threads are marked in white.

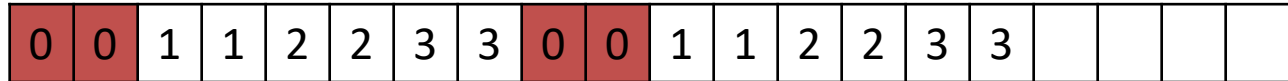
# schedule (static, 2)

num\_threads(4)

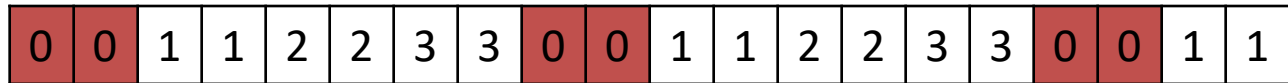
Time = 1



Time = 2



Time = 3

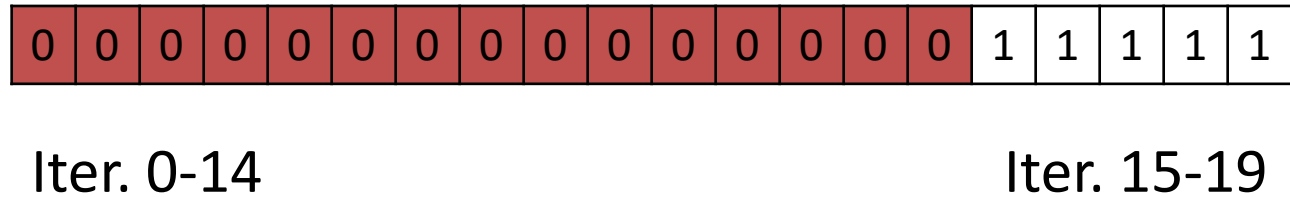


t

The iterations of thread #0 are marked in red, the iterations of other threads are marked in white.

# schedule (static, 15)

```
num_threads(4)
```



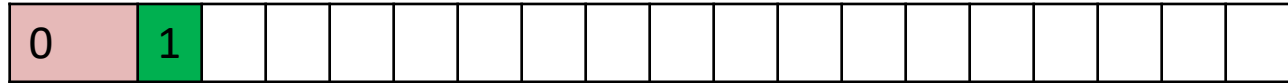
t

The iterations of thread #0 are marked in red, the iterations of other threads are marked in white.

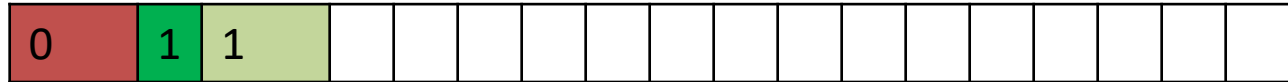
# schedule (dynamic, 1)

num\_threads(2)

Time = 1



Time = 2



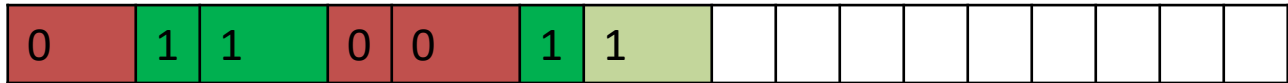
Time = 3



Time = 4



Time = 5



Time = 6

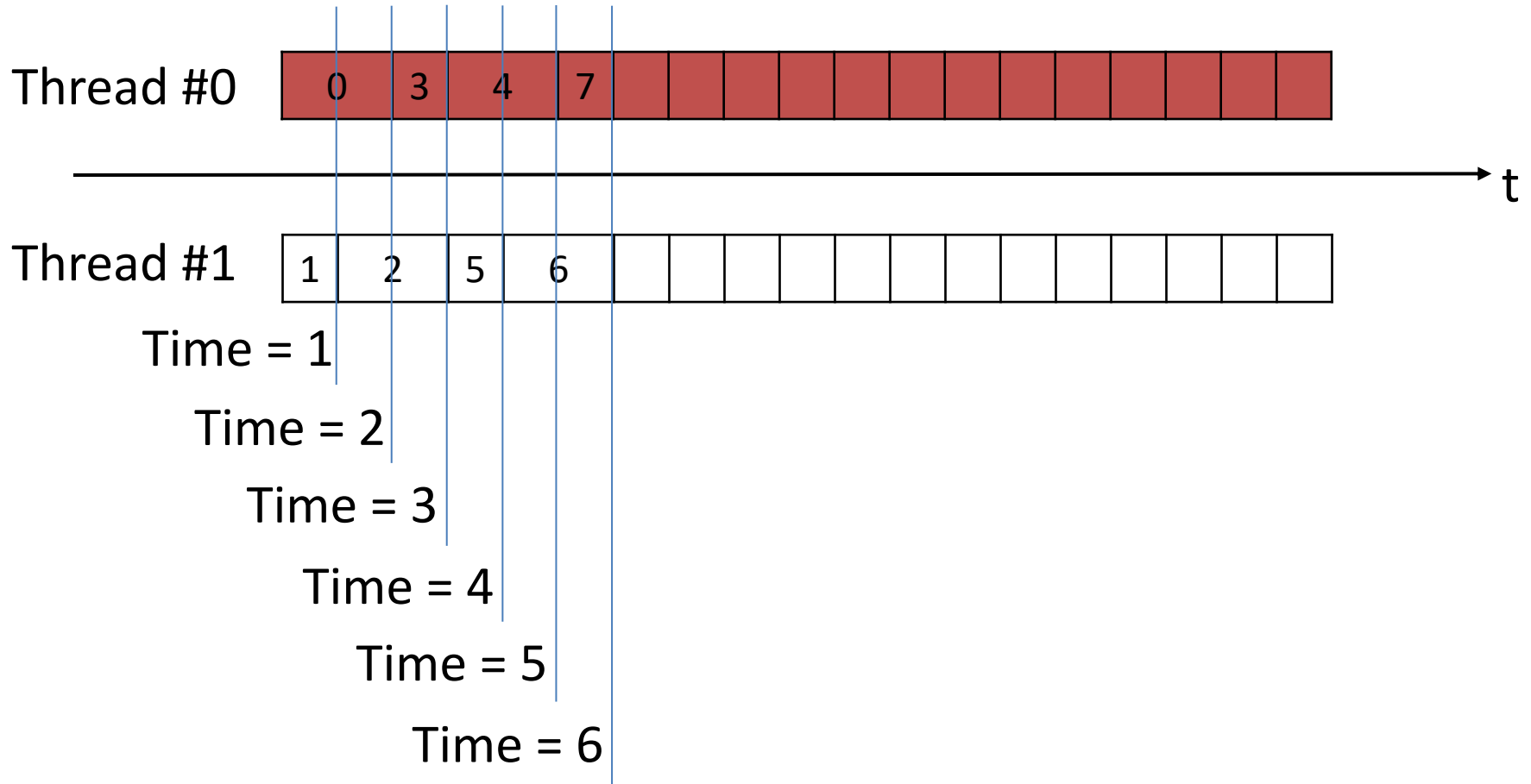


t

The iterations of thread #0 are marked in red, the iterations of thread #1 are marked in green.

# schedule (dynamic, 1)

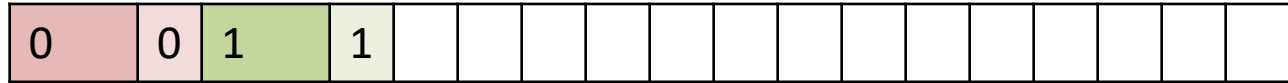
```
num_threads(2)
```



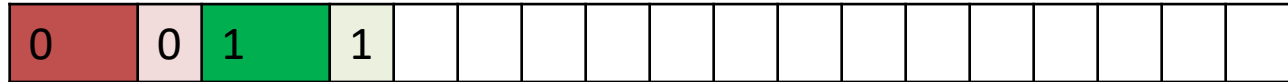
# schedule (dynamic, 2)

num\_threads(2)

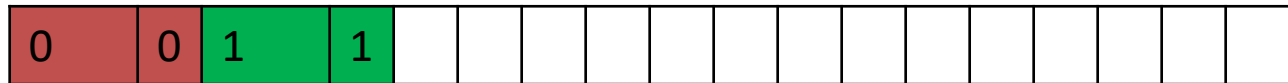
Time = 1



Time = 2



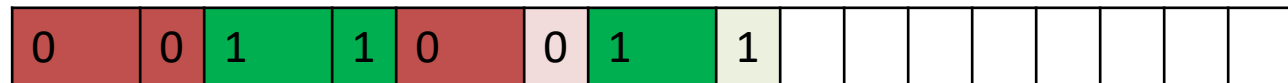
Time = 3



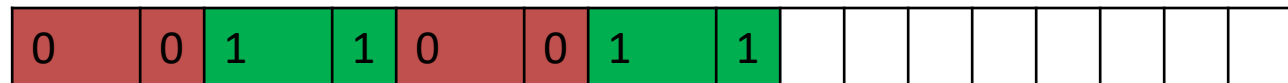
Time = 4



Time = 5



Time = 6



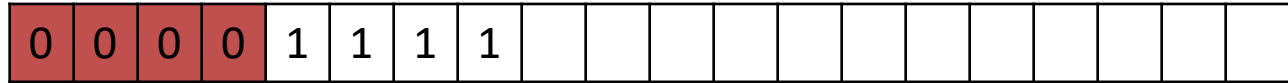
t

The iterations of thread #0 are marked in red, the iterations of thread #1 are marked in green.

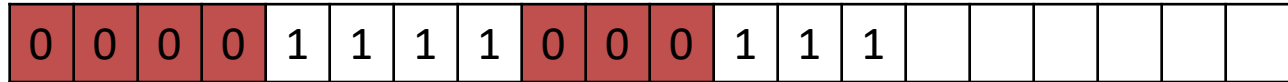
# schedule (guided, 1)

num\_threads(2)

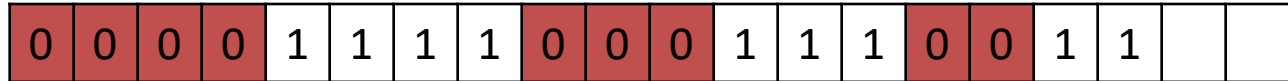
Iter. 0-3



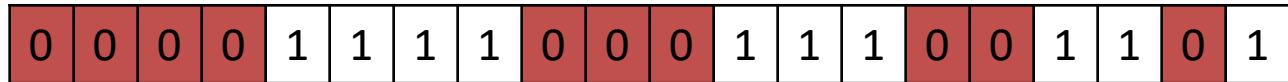
Iter. 4-6



Iter. 7-8



Iter. 9



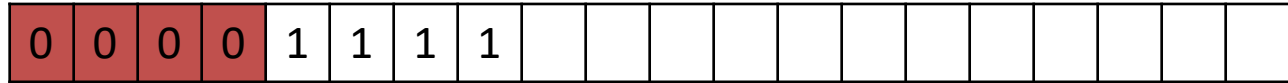
t

The iterations of thread #0 are marked in red, the iterations of thread #1 are marked in white.

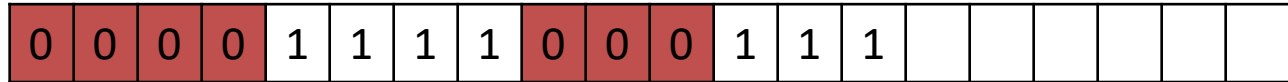
# schedule (guided, 3)

num\_threads(2)

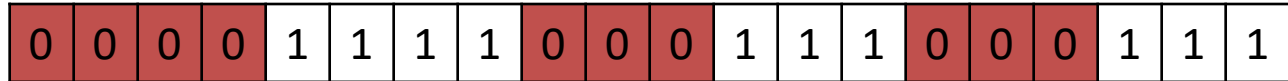
Iter. 0-3



Iter. 4-6



Iter. 7-9

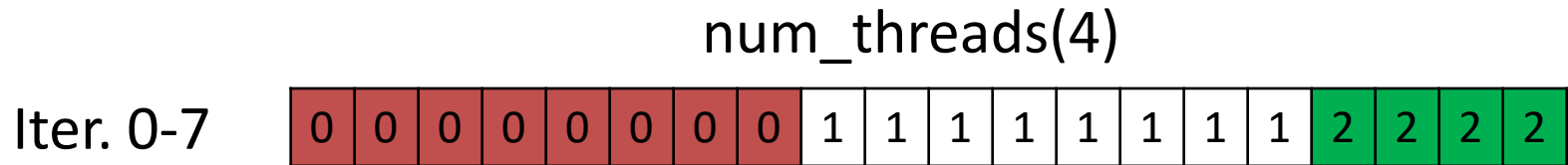


t

The iterations of thread #0 are marked in red, the iterations of thread #1 are marked in white.



# schedule (guided, X)



The iterations of thread #0 are marked in red, the iterations of thread #1 are marked in white, the iterations of thread #2 are marked in green.

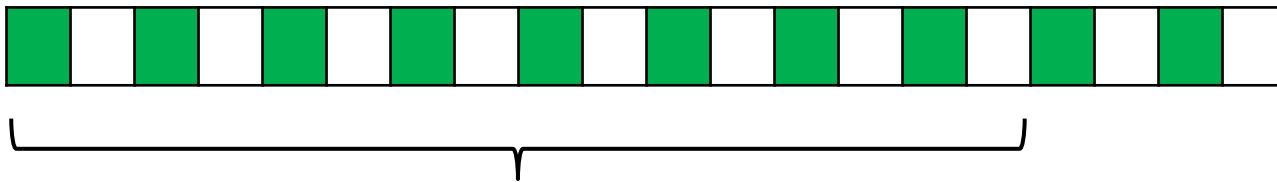
# Memory usage effect

```
1  int a[1000000];  
2  int j = 1, step = 1;  
3  for (int i = 0; i < 10000; ++i) {  
4      a[j] = 1;  
5      j += step;  
6  }
```

int = 4 bytes, cache line = 64 bytes

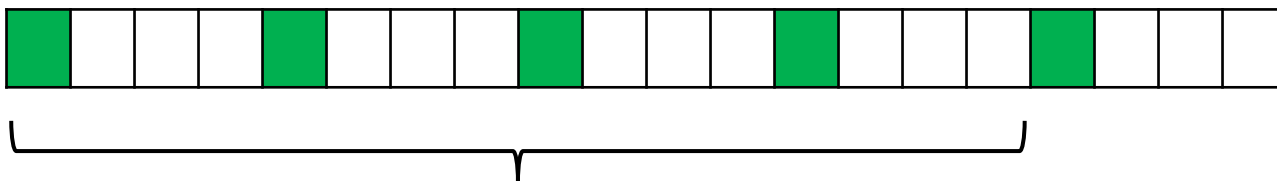
Step = 2

Cache miss = 12,5 %



Step = 4

Cache miss = 25 %



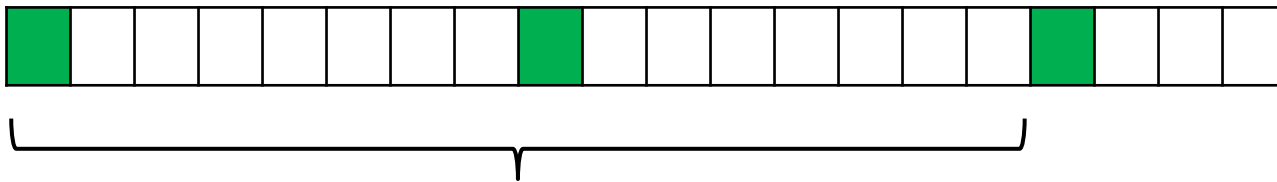
# Memory usage effect (2)

```
1  int a[1000000];  
2  int j = 1, step = 1;  
3  for (int i = 0; i < 10000; ++i) {  
4      a[j] = 1;  
5      j += step;  
6  }
```

int = 4 bytes, cache line = 64 bytes

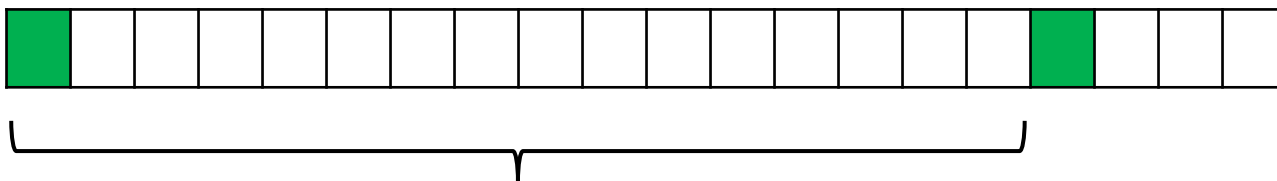
Step = 8

Cache miss = 50 %



Step = 16

Cache miss = 100 %



# How to identify cache miss

```
Terminal - vs@vs-debian: ~/workspace/test/sandbox
File Edit View Terminal Go Help
vs@vs-debian:~/workspace/test/sandbox$ gcc test.c -o test
vs@vs-debian:~/workspace/test/sandbox$ valgrind --tool=cachegrind ./test
==19678==
==19678== I    refs:      181,295
==19678== I1  misses:      783
==19678== LLi misses:      781
==19678== I1  miss rate:    0.43%
==19678== LLi miss rate:    0.43%
==19678==
==19678== D    refs:      98,854 (77,621 rd + 21,233 wr)
==19678== D1  misses:      2,469 ( 1,321 rd +  1,148 wr)
==19678== LLd misses:      2,111 ( 1,027 rd +  1,084 wr)
==19678== D1  miss rate:    2.4% (  1.7% +  5.4% )
==19678== LLd miss rate:    2.1% (  1.3% +  5.1% )
==19678==
==19678== LL refs:      3,252 ( 2,104 rd +  1,148 wr)
==19678== LL misses:      2,892 ( 1,808 rd +  1,084 wr)
==19678== LL miss rate:    1.0% (  0.6% +  5.1% )
vs@vs-debian:~/workspace/test/sandbox$
```