The role of modern Compilers
Supporting parallel programming models

Prof. Andrea Marongiu
(andrea.marongiu@unimore.it)

Outline

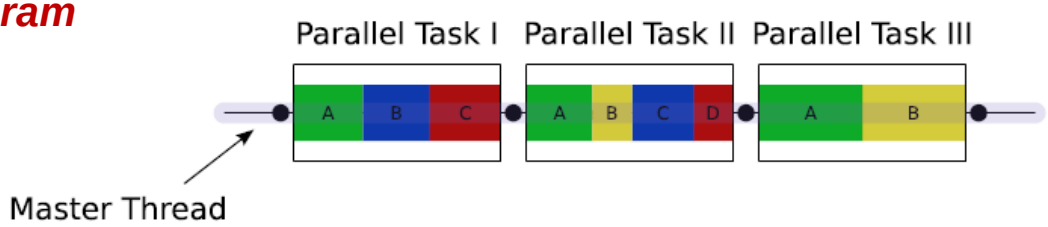
- Compiler structure
- Intermediate Representations
- Optimization

Programming model: OpenMP

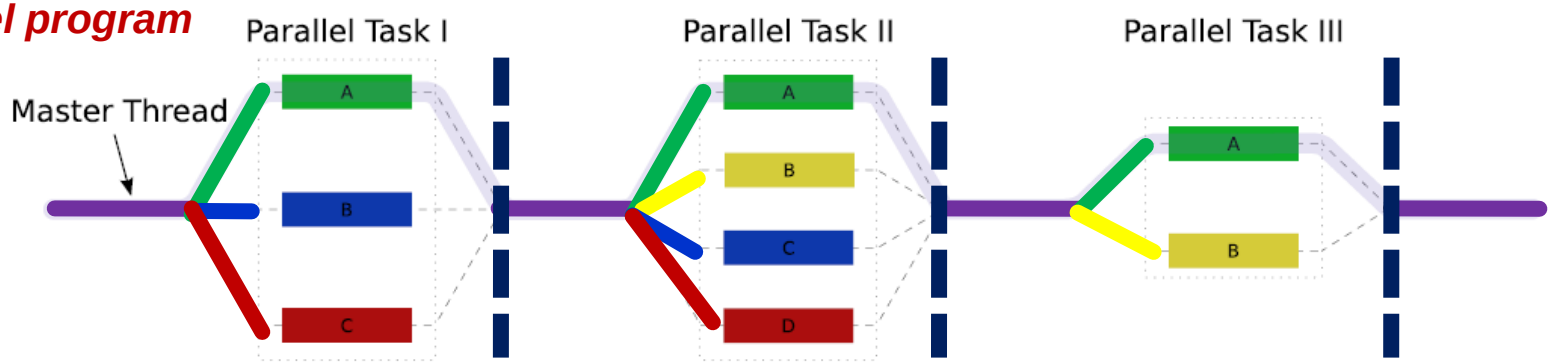
- De-facto standard for the shared memory programming model
- A collection of compiler directives, library routines and environment variables
- Easy to specify parallel execution within a serial code
- Targets several types of parallelism (data, task, heterogeneous)
- Popular in high-end embedded
- Requires special support in the compiler
- Generates calls to threading libraries (e.g. pthreads)

Fork/Join Parallelism

Sequential program



Parallel program



- Initially only master thread is active
- Master thread executes sequential code
- Fork: Master thread creates or awakens additional threads to execute parallel code
- Join: At the end of parallel code created threads are suspended upon **barrier** synchronization

Pragmas

- Pragma: a compiler directive in C or C++
- Stands for “pragmatic information”
- A way for the programmer to communicate with the compiler
- Compiler free to ignore pragmas: original sequential semantic is not altered
- Syntax:

    #**pragma omp** <*rest of pragma*>

Components of OpenMP
a subset of the directives

Directives

- Parallel regions
  - ***#pragma omp parallel***
- Work sharing
  - ***#pragma omp for***
  - ***#pragma omp sections***
- Synchronization
  - ***#pragma omp barrier***
  - ***#pragma omp critical***
  - ***#pragma omp atomic***

Clauses

- Data scope attributes
  - ***private***
  - ***shared***
  - ***reduction***
- Loop scheduling
  - ***static***
  - ***dynamic***

Runtime Library

- Thread Forking/Joining
  - ***omp_parallel_start()***
  - ***omp_parallel_end()***
- Loop scheduling
- Thread IDs
  - ***omp_get_thread_num()***
  - ***omp_get_num_threads()***

Outlining parallelism
The `parallel` directive

```
void ex1 ()
{
  <bb 2> [0.00%]:
  __builtin_GOMP_parallel (
       ex1._omp_fn.0, 0B, 0, 0);
}
```

```
ex1._omp_fn.0 (void * .omp_data_i)
{
  <bb 3> [0.00%]:
  printf ("Hello World!\n");
  return;
}
```

hello.c.012t.ompexp

```
<CROSS-COMPILE>-gcc -O0 -DEX1
-fopenmp -fdump-tree-all main.c
```

Code originally contained within the scope of the pragma is outlined to a new function within the compiler

```
void ex1 ()
{
#pragma omp parallel
   {
    printf ("\nHello world!");
   }
}
```

A sequential program..
..is easily parallelized

- Fundamental construct to outline parallel computation within a sequential program
- Code within its scope is replicated among threads
- Defers implementation of parallel execution to the runtime (machine-specific, e.g. *pthread_create*)

Outlining parallelism
The `parallel` directive

```
void ex1 ()
{
  <bb 2> [0.00%]:
  __builtin_GOMP_parallel (
       ex1._omp_fn.0, 0B, 0, 0);
}


ex1._omp_fn.0 (void * .omp_data_i)
{
  <bb 3> [0.00%]:
  printf ("Hello World!\n");
  return;
}
```

hello.c.012t.ompexp

```
<CROSS-COMPILE>-gcc -O0 -DEX1
-fopenmp -fdump-tree-all main.c
```

The #pragma construct in the **main** function is replaced with function calls to the runtime library

```
void ex1 ()
{
#pragma omp parallel
   {
     printf ("\nHello world!");
   }
}
```

NOTE: The **master** thread is the only thread that executes this code

#pragma omp parallel

```
void ex1 ()
{
  <bb 2> [0.00%]:
  __builtin_GOMP_parallel (
       ex1._omp_fn.0, 0B, 0, 0);
}


ex1._omp_fn.0 (void * .omp_data_i)
{
  <bb 3> [0.00%]:
  printf ("Hello World!\n");
  return;
}
```

hello.c.012t.ompexp

Within the runtime we first call a function to fork new threads,

GOMP_parallel_start

```
void
GOMP_parallel (void (*fn) (void *),
               void *data,
               unsigned num_threads,
               unsigned int flags)
{
  num_threads = ... num_threads ...;
  gomp_team_start (fn, data,
                   num_threads, ...);
  fn (data);
  ialias_call (GOMP_parallel_end) ();
}
```

```
void ex1 ()
{
#pragma omp parallel
  {
    printf ("\nHello world!");
  }
}
```

#pragma omp parallel

```
void ex1 ()
{
  <bb 2> [0.00%]:
  __builtin_GOMP_parallel (
       ex1._omp_fn.0, 0B, 0, 0);
}


ex1._omp_fn.0 (void * .omp_data_i)
{
  <bb 3> [0.00%]:
  printf ("Hello World!\n");
  return;
}
```

hello.c.012t.ompexp

Within the runtime we first call a function to fork new threads, **passing a pointer to the outlined function**

GOMP_parallel_start

```
void
GOMP_parallel (void (*fn) (void *),
               void *data,
               unsigned num_threads,
               unsigned int flags)
{
  num_threads = ... num_threads ...;
  gomp_team_start (fn, data,
                   num_threads, ...);
  fn (data);
  ialias_call (GOMP_parallel_end) ();
}
```

```
void ex1 ()
{
#pragma omp parallel
   {
     printf ("\nHello world!");
   }
}
```

…and **data**, which we will be looking at in a moment…

#pragma omp parallel

```
void ex1 ()
{
  <bb 2> [0.00%]:
  __builtin_GOMP_parallel (
        ex1._omp_fn.0, 0B, 0, 0);
}


ex1._omp_fn.0 (void * .omp_data_i)
{
  <bb 3> [0.00%]:
  printf ("Hello World!\n");
  return;
}
```

hello.c.012t.ompexp

```
void
GOMP_parallel (void (*fn) (void *),
               void *data,
               unsigned num_threads,
               unsigned int flags)
{
  num_threads = ... num_threads ...;
  comp_team_start (fn, data,
                   num_threads, ...);
  fn (data);
  ialias_call (GOMP_parallel_end) ();
}
```

Then the master itself calls the parallel function

```
void ex1 ()
{
#pragma omp parallel
   {
     printf ("\nHello world!");
   }
}
```

REMEMBER: The **master** thread is the only thread that executes this library code

#pragma omp parallel

```
void ex1 ()
{
  <bb 2> [0.00%]:
  __builtin_GOMP_parallel (
        ex1._omp_fn.0, 0B, 0, 0);
}


ex1._omp_fn.0 (void * .omp_data_i)
{
  <bb 3> [0.00%]:
  printf ("Hello World!\n");
  return;
}
```

hello.c.012t.ompexp

```
void ex1 ()
{
#pragma omp parallel
   {
     printf ("\nHello world!");
   }
}
```

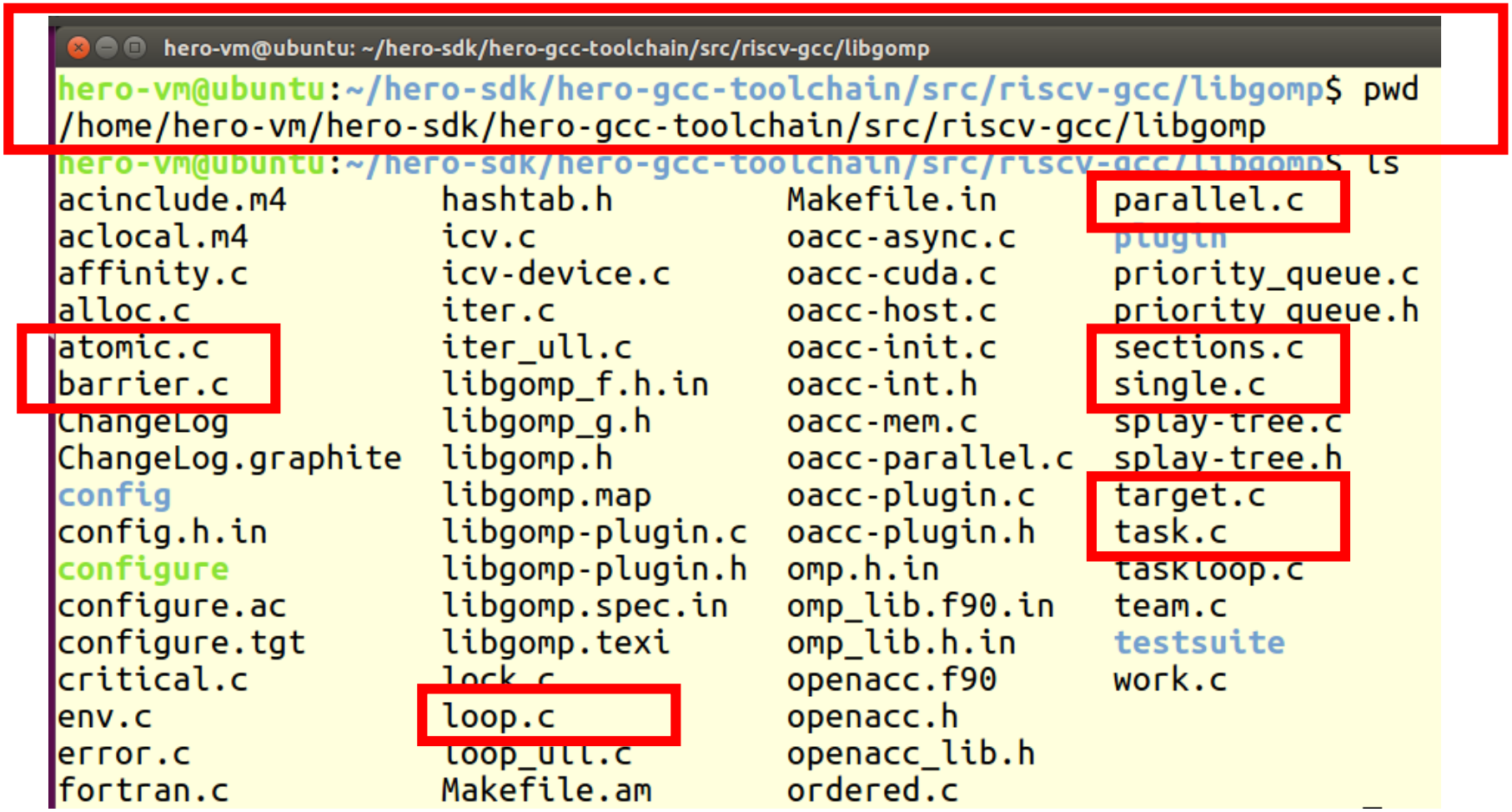
```
void
GOMP_parallel (void (*fn) (void *),
               void *data,
               unsigned num_threads,
               unsigned int flags)
{
  num_threads = ... num_threads ...;
  gomp_team_start (fn, data,
                   num_threads, ...);

  fn (data);
  ialias_call (GOMP_parallel_end) ();
}
```

Finally we call the runtime to synchronize threads with a barrier and suspend them

REMEMBER: The **master** thread is the only thread that executes this library code

GOMP: The OpenMP runtime library

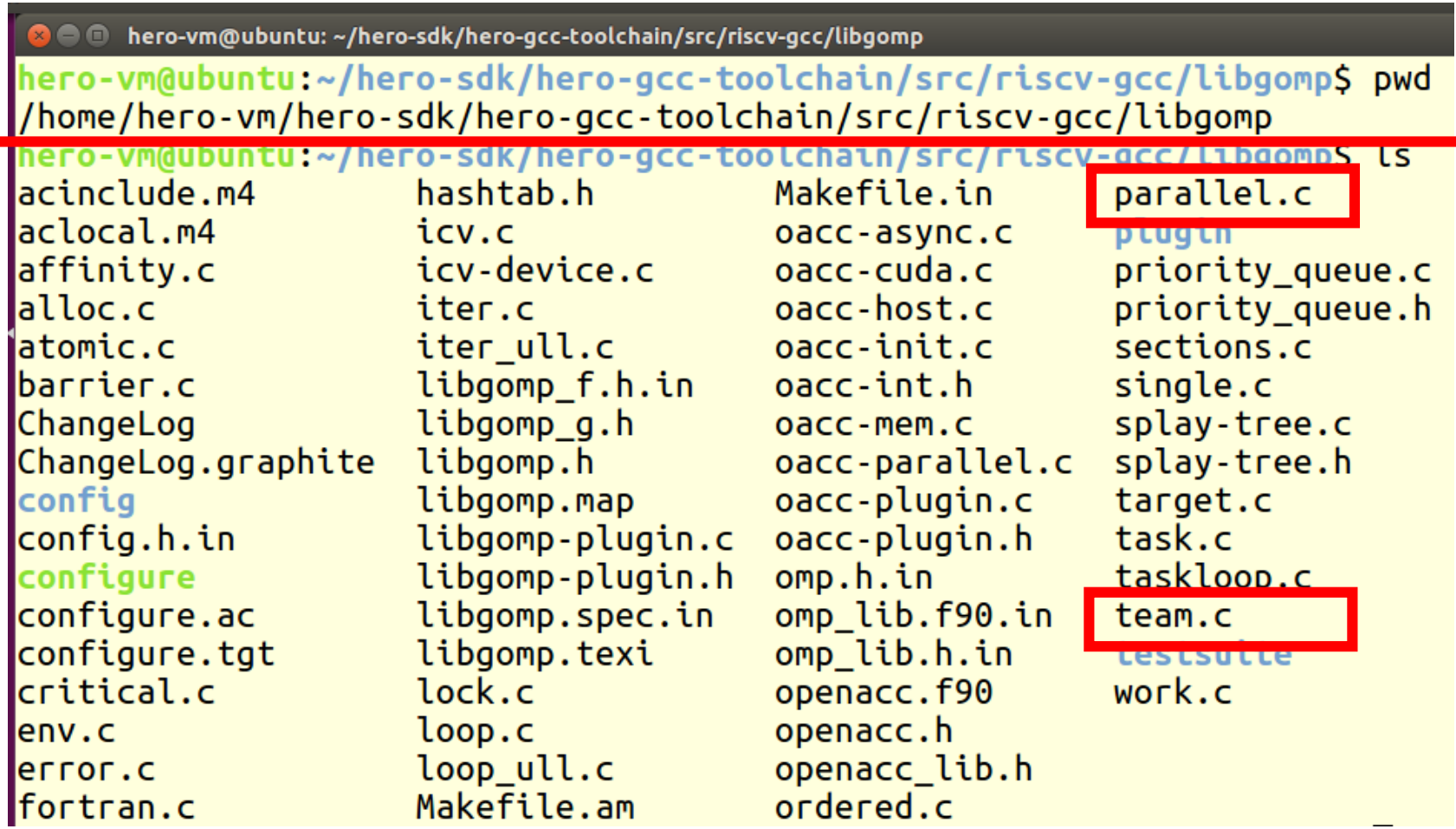
- Where does this library code live?



Easy to spot at least one file per #pragma construct

GOMP: The OpenMP runtime library

- Where does this library code live?



Let's have a look at the implementation of the PARALLEL construct

#pragma omp parallel

Data scope attributes

```
void ex2 ()
{
  int id;
  int base = 5;
  int slaves = 0;

  #pragma omp parallel
  {
    id = omp_get_thread_num();
    slaves = omp_get_num_threads() - 1;

    if (id == 0)
      printf ("Master: base = %d.", base*slaves);
    else
      printf ("Slave: base = %d.", base*id);
  }
}
```

Call runtime to get thread ID:
Every thread sees a different value

Master and slave threads access the same variable **a**

A slightly more complex example

#pragma omp parallel

Data scope attributes

```
void ex2 ()
{
  int id;
  int base = 5;
  int slaves = 0;

  #pragma omp parallel
  {
    id = omp_get_thread_num();
    slaves = omp_get_num_threads()

    if (id == 0)
      printf ("Master: base = %d.", base*slaves);
    else
      printf ("Slave: base = %d.", base*id);
  }
}
```

Call runtime to get thread ID:
Every thread sees a different value

How to inform the compiler about these different behaviors?

Master and slave threads access the same variable **a**

A slightly more complex example

#pragma omp parallel

Data scope attributes

```
void ex2 ()
{
  int id;
  int base = 5;
  int slaves = 0;

  #pragma omp parallel shared (base, slaves) private (id)
  {
    id = omp_get_thread_num();
    slaves = omp_get_num_threads() - 1;

    if (id == 0)
      printf ("Master: base = %d.", base*slaves);
    else
      printf ("Slave: base = %d.", base*id);
  }
}
```

Insert code to retrieve the address of the shared object from within each parallel thread

Allow symbol privatization: Each thread contains a private copy of this variable

A slightly more complex example

More data sharing clauses

- **firstprivate**
  - copyin, private storage
- **lastprivate**
  - Copyout, private storage

- **Exercise** – Try to add these clauses to your code, and see how the generated code differs.

OpenMP lowering

```
Introduced new external node (ex2._omp_fn.1/6).
ex2 ()
{
  int id;
  int base;
  int slaves;

  base = 5;
  slaves = 0;
  {
    .omp_data_o.2.base = base;
    .omp_data_o.2.slaves = slaves;
    #pragma omp parallel private(id) shared(slaves) firstprivate(base) [child fn: ex2._omp_fn.1 (.omp_data_o.2)]
      {
        .omp_data_i = (struct .omp_data_s.1 & restrict) &.omp_data_o.2;
        base = .omp_data_i->base;
        id = omp_get_thread_num ();
        D.6171 = omp_get_num_threads ();
        D.6189 = D.6171 + -1;
        .omp_data_i->slaves = D.6189;
        if (id == 0) goto <D.6186>; else goto <D.6187>;
        <D.6186>:
        D.6190 = .omp_data_i->slaves;
        D.6174 = base * D.6190;
        printf ("Master: base = %d.", D.6174);
        goto <D.6188>;
        <D.6187>:
        D.6176 = base * id;
        printf ("Slave: base = %d.", D.6176);
        <D.6188>:
        #pragma omp return
      }
    slaves = .omp_data_o.2.slaves;
    .omp_data_o.2 = {CLOBBER};
  }
}
```

<CROSS-COMPILE>-gcc -O0 -DEX2 -fopenmp -fdump-tree-all main.c

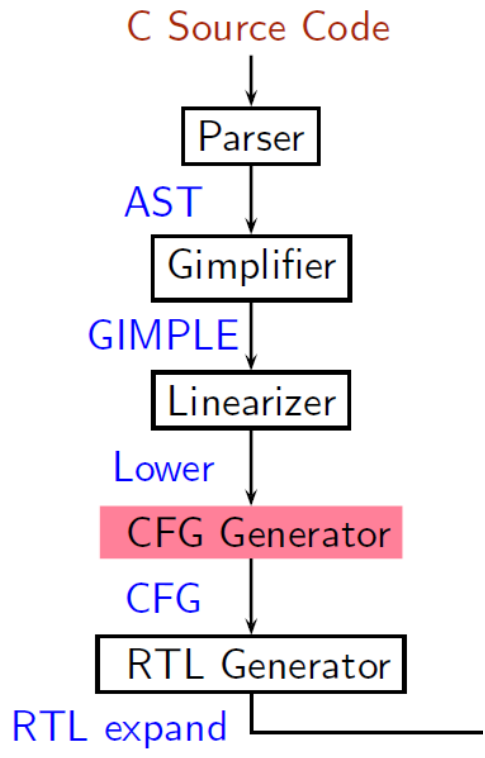
First of two stages of OpenMP processing:

1. **Lowering**
2. **Expansion**

Operates in high GIMPLE form

```
hello.c
hello.c.001t.tu
hello.c.002t.class
hello.c.003t.original
hello.c.004t.gimple
hello.c.006t.omplower
hello.c.007t.lower
hello.c.010t.eh
hello.c.011t.cfg
hello.c.012t.ompexp
```

ls -la



hello.c.006t.omplower

OpenMP lowering

```
Introduced new external node (ex2._omp_fn.1/6).
ex2 ()
{
  int id;
  int base;
  int slaves;

  base = 5;
  slaves = 0;
  {
    .omp_data_o.2.base = base;
    .omp_data_o.2.slaves = slaves;
    #pragma omp parallel private(id) shared(slaves) firstprivate(base) [child fn: ex2._omp_fn.1 (.omp_data_o.2)]
      {
        .omp_data_i = (struct .omp_data_s.1 & restrict) &.omp_data_o.2;
        base = .omp_data_i->base;
        id = omp_get_thread_num ();
        D.6171 = omp_get_num_threads ();
        D.6189 = D.6171 + -1;
        .omp_data_i->slaves = D.6189;
        if (id == 0) goto <D.6186>; else goto <D.6187>;
        <D.6186>:
        D.6190 = .omp_data_i->slaves;
        D.6174 = base * D.6190;
        printf ("Master: base = %d.", D.6174);
        goto <D.6188>;
        <D.6187>:
        D.6176 = base * id;
        printf ("Slave: base = %d.", D.6176);
        <D.6188>:
        #pragma omp return
      }
    slaves = .omp_data_o.2.slaves;
    .omp_data_o.2 = {CLOBBER};
  }
}
```

In this stage we create a new function node…

…and implement **data marshalling** for sharing data among parallel threads

(but outlining doesn't happen just yet)

hello.c.006t.omplower

OpenMP lowering

```
typedef struct
{
  int base;
  int slaves;
} omp_data_t
```

```
Introduced new external node (ex2._omp_fn.1/6).
ex2 ()
{
  int id;
  int base;
  int slaves;

  base = 5;
  slaves = 0;
  {
    .omp_data_o.2.base = base;
    .omp_data_o.2.slaves = slaves;
    #pragma omp parallel private(id) shared(slaves) firstprivate(base) [child fn: ex2._omp_fn.1 (.omp_data_o.2)]
    {
      .omp_data_i = (struct .omp_data_s.1 & restrict) &.omp_data_o.2;
      base = .omp_data_i->base;
      id = omp_get_thread_num ();
      D.6171 = omp_get_num_threads ();
      D.6189 = D.6171 + -1;
      .omp_data_i->slaves = D.6189;
      if (id == 0) goto <D.6186>; else goto <D.6187>;
      <D.6186>:
      D.6190 = .omp_data_i->slaves;
      D.6174 = base * D.6190;
      printf ("Master: base = %d.", D.6174);
      goto <D.6188>;
      <D.6187>:
      D.6176 = base * id;
      printf ("Slave: base = %d.", D.6176);
      <D.6188>:
      #pragma omp return
    }
    slaves = .omp_data_o.2.slaves;
    .omp_data_o.2 = {CLOBBER};
  }
}
```

marshalling IN/OUT (`omp_data_t omp_data_o;`)

…and implement **data marshalling** for sharing data among parallel threads

Replace uses (`omp_data_t omp_data_i;`)

hello.c.006t.omplower

OpenMP expansion

```
ex2._omp_fn.1 (struct .omp_data_s.1 & restrict .omp_data_i)
{
  int slaves [value-expr: .omp_data_i->slaves];
  int D.6198;
  int D.6197;
  int D.6196;
  int D.6195;
  int D.6194;
  int id;
  int base;

  <bb 3> [0.00%]:
  base = .omp_data_i->base;
  id = omp_get_thread_num ();
  D.6194 = omp_get_num_threads ();
  D.6195 = D.6194 + -1;
  .omp_data_i->slaves = D.6195;
  if (id == 0)
    goto <bb 4>; [0.00%]
  else
    goto <bb 5>; [0.00%]

  <bb 6> [0.00%]:
  return;

  <bb 5> [0.00%]:
  D.6196 = base * id;
  printf ("Slave: base = %d.", D.6196);
  goto <bb 6>; [0.00%]

  <bb 4> [0.00%]:
  D.6197 = .omp_data_i->slaves;
  D.6198 = base * D.6197;
  printf ("Master: base = %d.", D.6198);
  goto <bb 6>; [0.00%]

}
```

hello.c.012t.ompexp

<CROSS-COMPILE>-gcc -O0 -DEX2 -fopenmp -fdump-tree-all main.c

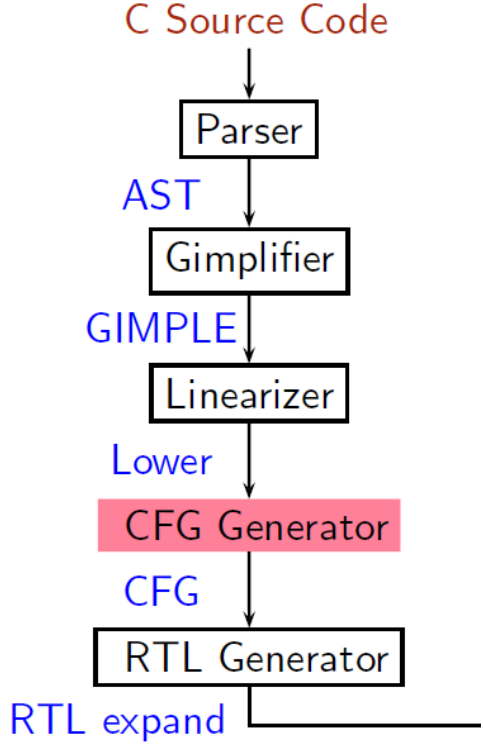
Second of two stages of OpenMP processing:

1. **Lowering**
2. **Expansion**

Operates in low GIMPLE form

```
hello.c
hello.c.001t.tu
hello.c.002t.class
hello.c.003t.original
hello.c.004t.gimple
hello.c.006t.omplower
hello.c.007t.lower
hello.c.010t.eh
hello.c.011t.cfg
hello.c.012t.ompexp
```

ls -la



OpenMP expansion

```
ex2._omp_fn.1 (struct .omp_data_s.1 & restrict .omp_data_i)
{
  int slaves [value-expr: .omp_data_i->slaves];
  int D.6198;
  int D.6197;
  int D.6196;
  int D.6195;
  int D.6194;
  int id;
  int base;

  <bb 3> [0.00%]:
  base = .omp_data_i->base;
  id = omp_get_thread_num ();
  D.6194 = omp_get_num_threads ();
  D.6195 = D.6194 + -1;
  .omp_data_i->slaves = D.6195;
  if (id == 0)
    goto <bb 4>; [0.00%]
  else
    goto <bb 5>; [0.00%]

  <bb 6> [0.00%]:
  return;

  <bb 5> [0.00%]:
  D.6196 = base * id;
  printf ("Slave: base = %d.", D.6196);
  goto <bb 6>; [0.00%]

  <bb 4> [0.00%]:
  D.6197 = .omp_data_i->slaves;
  D.6198 = base * D.6197;
  printf ("Master: base = %d.", D.6198);
  goto <bb 6>; [0.00%]

}
```

marshalling IN/OUT (`omp_data_t omp_data_o;`)

```
ex2 ()
{
  int slaves;
  int base;
  int id;
  struct .omp_data_s.1 .omp_data_o.2;
  int D.6190;
  int D.6189;

  <bb 2> [0.00%]:
  base = 5;
  slaves = 0;
  .omp_data_o.2.base = base;
  .omp_data_o.2.slaves = slaves;
  __builtin_GOMP_parallel (ex2._omp_fn.1, &.omp_data_o.2, 0, 0);
  slaves = .omp_data_o.2.slaves;
  .omp_data_o.2 = {CLOBBER};
  return;

}
```

Call runtime by passing a pointer to the outlined function and the marshalled data

Replace uses (`omp_data_t omp_data_i;`)

hello.c.012t.ompexp

Sharing work among threads
The `for` directive

- The **parallel** pragma instructs every thread to execute all of the code inside the block
- If we encounter a **for** loop that we want to divide among threads, we use the **for** pragma

```
#pragma omp for
```

#pragma omp for

#pragma omp for can be placed everywhere inside a **parallel** construct, or combined with it, as in the example

The code of the **for** loop is moved inside the outlined function.

```
void ex3 (int *a)
{
  int i;

#pragma omp parallel for
  for (i=0; i<10; i++)
    a[i] = i;
}
```

```
void ex3(...)
{
  omp_parallel_start(&parfun, ...);
  parfun();
  omp_parallel_end();
}

int parfun(...)
{
  int LB = ...;
  int UB = ...;

  for (i=LB; i<UB; i++)
      a[i] = i;
}
```

#pragma omp for

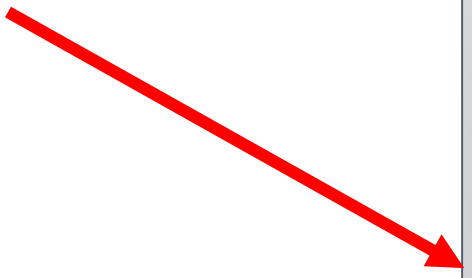
Every thread works on a different subset of the iteration space..

```
void ex3 (int *a)
{
  int i;

#pragma omp parallel for
  for (i=0; i<10; i++)
    a[i] = i;
}
```

```
void ex3(...)
{
  omp_parallel_start(&parfun, ...);
  parfun();
  omp_parallel_end();
}

int parfun(...)
{
  int LB = ...;
  int UB = ...;

  for (i=LB; i<UB; i++)
      a[i] = i;
}
```

#pragma omp for

..since lower and upper bounds (**LB**, **UB**) are computed locally as a function of the thread ID



```
void ex3 (int *a)
{
  int i;

#pragma omp parallel for
  for (i=0; i<10; i++)
    a[i] = i;
}
```

```
void ex3(...)
{
  omp_parallel_start(&parfun, ...);
  parfun();
  omp_parallel_end();
}

int parfun(...)
{
  int LB = ...;
  int UB = ...;

  for (i=LB; i<UB; i++)
      a[i] = i;
}
```

The `schedule` clause
Static Loop Partitioning

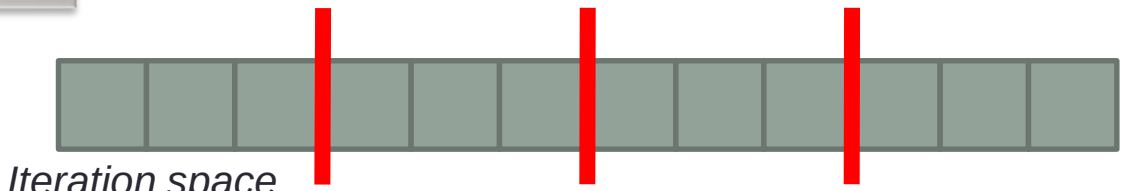
```
#pragma omp for schedule(static)
  {
    for (i=0; i<12; i++)
      a[i] = i;
  }
```

Es. 12 iterations (N), 4 threads (Nthr)

DATA CHUNK

$$C = ceil\left(\frac{N}{Nthr}\right)$$

3 iterations / thread



Iteration space

Useful for:

- *Simple, regular loops*
- *Iterations with equal duration*

	Thread ID (TID)	0	1	2	3
LOWER BOUND	**LB** = C * TID	0	3	6	9
UPPER BOUND	**UB** = *min* { [C * (TID + 1)], N}	3	6	9	12

The `schedule` clause
Static Loop Partitioning

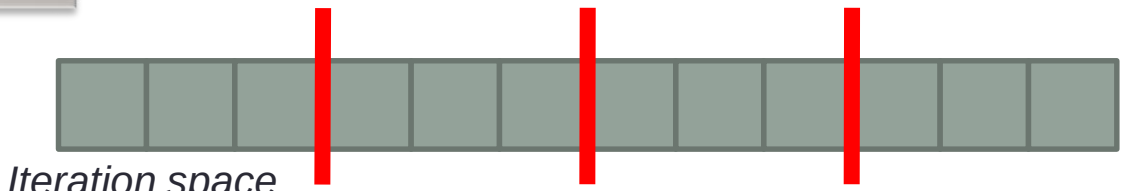
```
#pragma omp for schedule(static)
  {
    for (i=0; i<12; i++)
      a[i] = i;
  }
```

Es. 12 iterations (N), 4 threads (Nthr)

DATA CHUNK

$$C = ceil\left(\frac{N}{Nthr}\right)$$

3 iterations / thread



Iteration space

Useful for:

- *Simple, regular loops*
- *Iterations with equal duration*

	Thread ID (TID)	0	1	2	3
LOWER BOUND	**LB** = C * TID	0	3	6	9
UPPER BOUND	**UB** = *min* { [C * (TID + 1)], N}	3	6	9	12

Static scheduling clauses

- **EXERCISE** – Generate and analyze the IR dumps for the OpenMP lowering and expansion passes for the `#pragma omp for` directive, playing with static chunking

- `schedule (static, N)`

```
<CROSS-COMPILE>-gcc -O0 -DEX3 -fopenmp -fdump-tree-all main.c
```

The `schedule` clause
Static Loop Partitioning

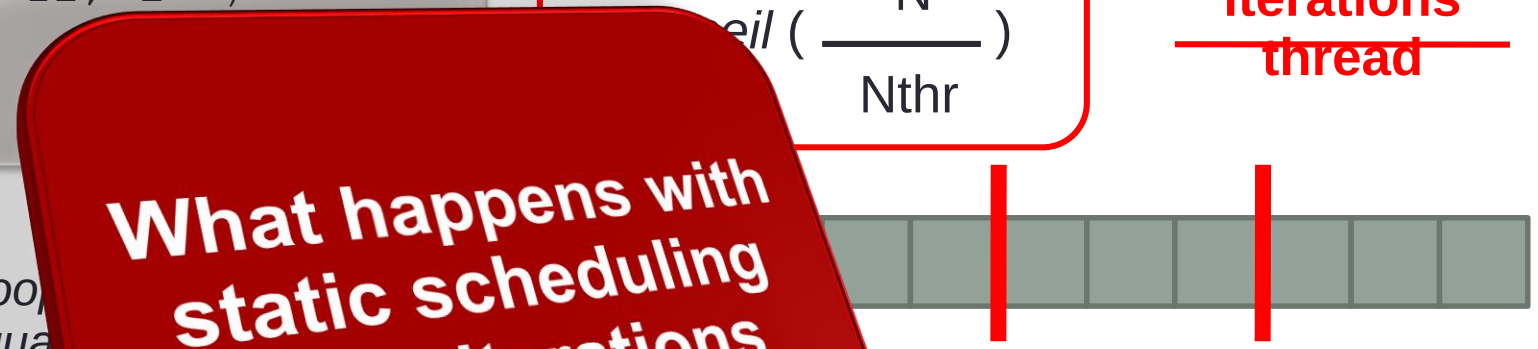
```
#pragma omp for schedule(static)
  {
    for (i=0; i<12; i++)
      a[i] = i;
  }
```

Es. 12 iterations (N), 4 threads (Nthr)

DATA CHUNK

$$\textit{ceil}\left(\frac{N}{Nthr}\right)$$

3 iterations / thread



Useful for:

- *Simple, regular loop*
- *Iterations with equa*

What happens with static scheduling when iterations have different duration?

	0	1	2	3
LOWER BO	0	3	6	9
UPPER BOUND	3	6	9	12

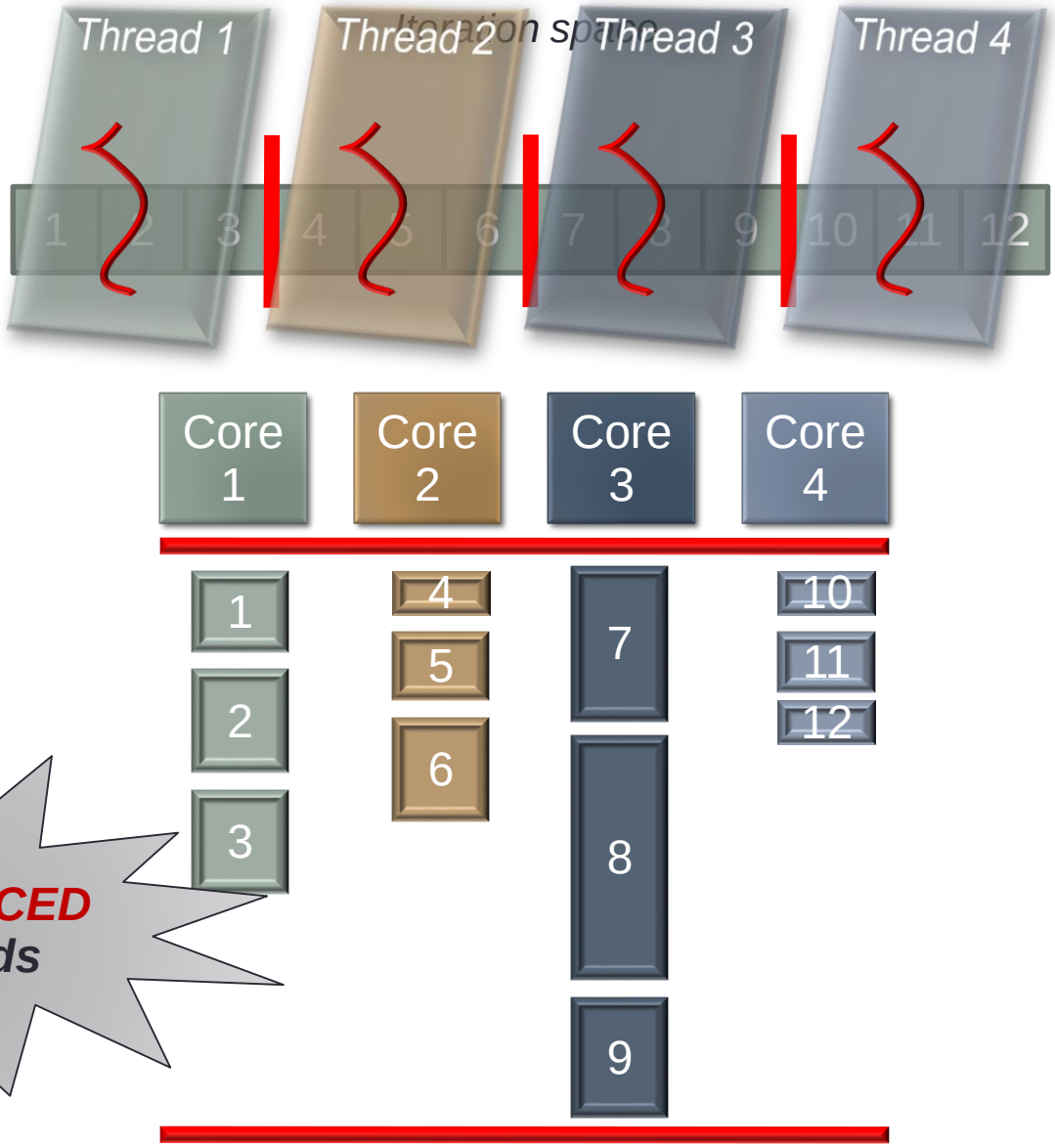
(TID + 1)], N}

The `schedule` clause
Static Loop Partitioning

```
#pragma omp for schedule(static)
  {
    for (i=0; i<12; i++)
      {
        int start = rand();
        int count = 0;

        while (start++ < 256)
          count++;

        a[count] = foo();
      }
  }
```

A variable amount of work in each iteration

UNBALANCED workloads



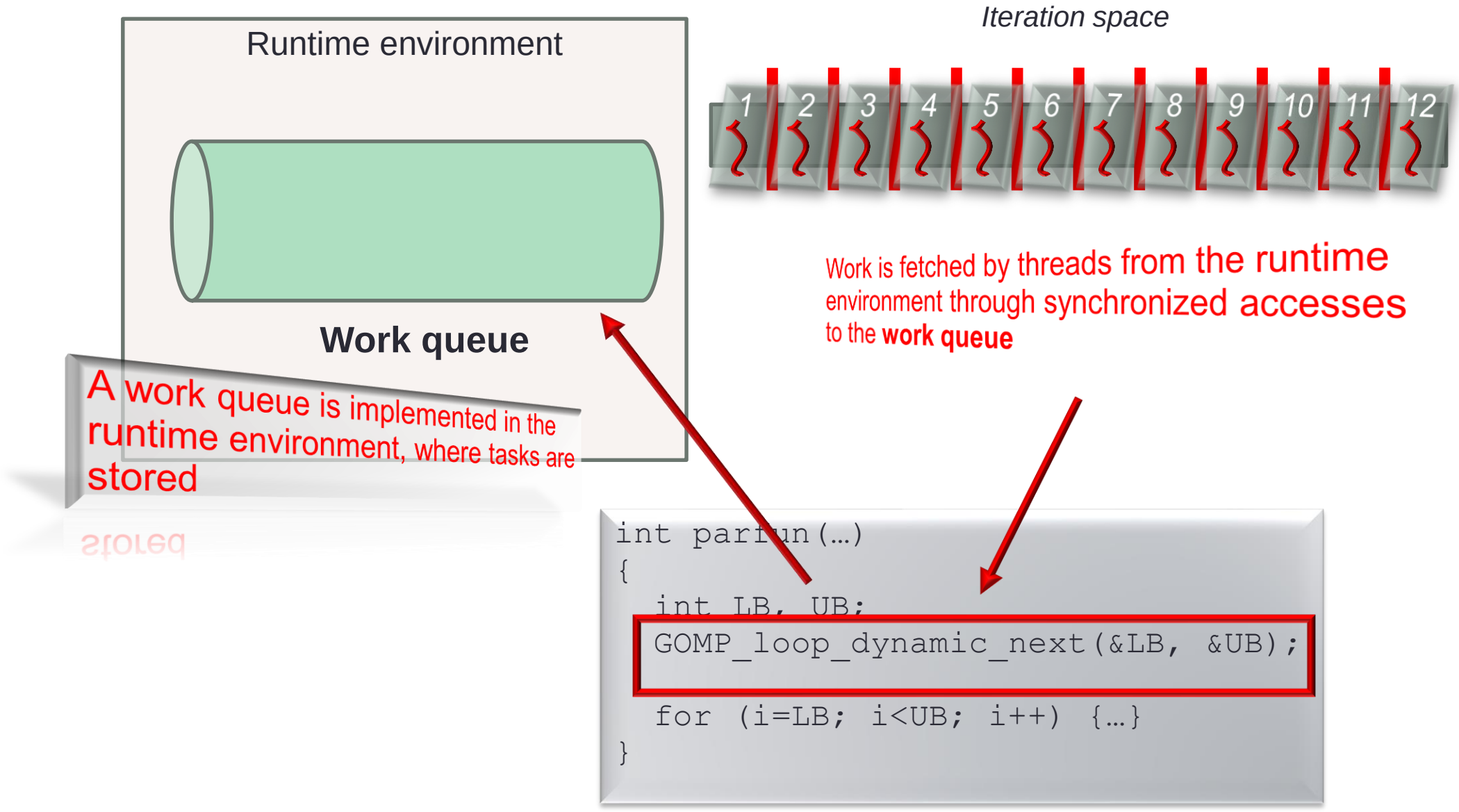
The `schedule` clause
Dynamic Loop Partitioning

```
#pragma omp for schedule(dynamic)
  {
    for (i=0; i<12; i++)
      {
        int start = rand();
        int count = 0;

        while (start++ < 256)
          count++;

        a[count] = foo();
      }
  }
```

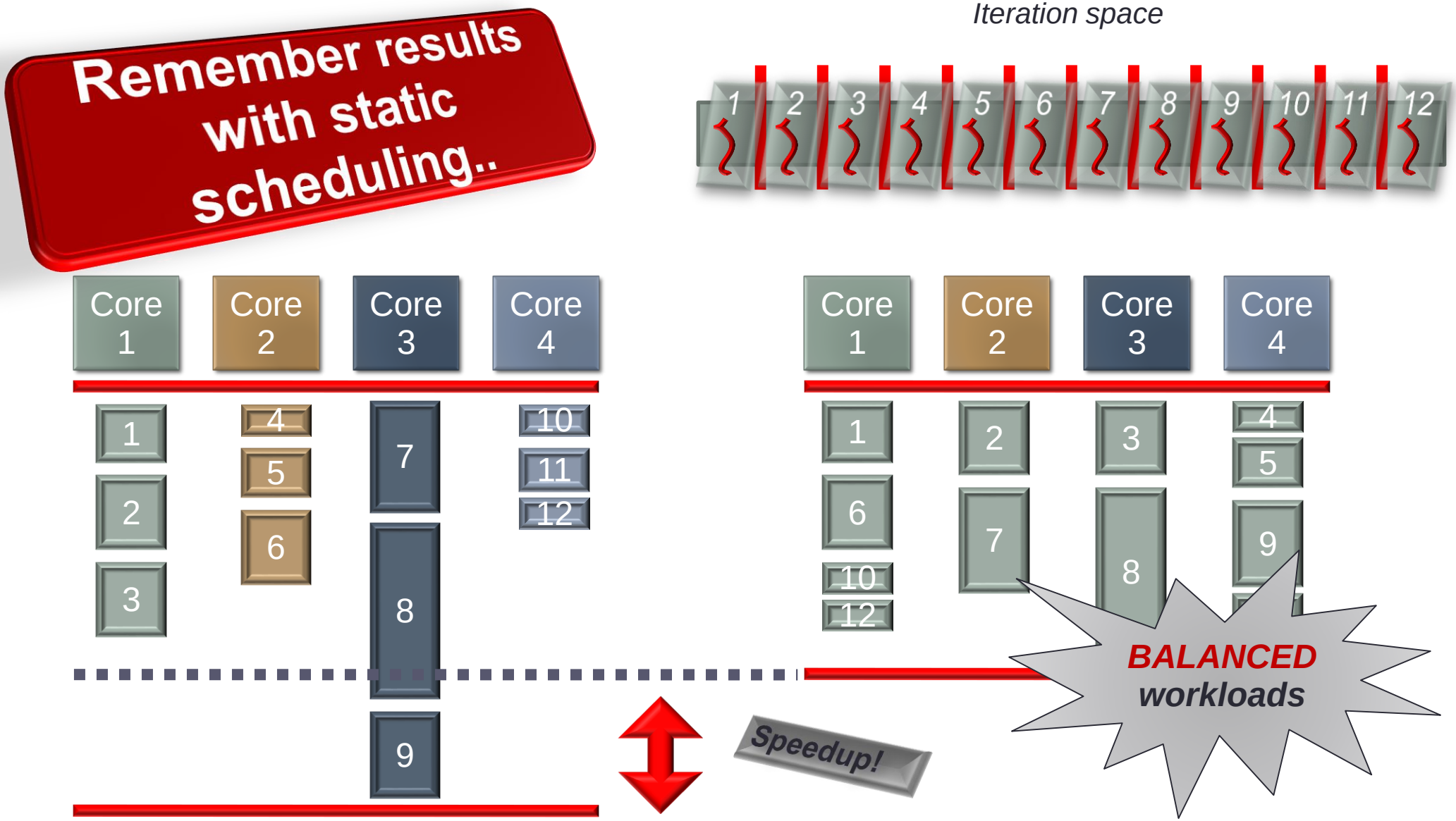
A thread is generated for every single iteration

Iteration space

The `schedule` clause
Dynamic Loop Partitioning

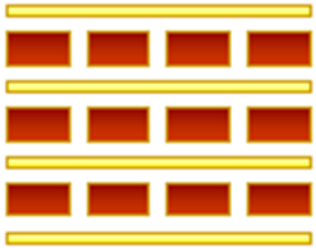


The `schedule` clause
Dynamic Loop Partitioning



Parallelization granularity
Iteration chunking

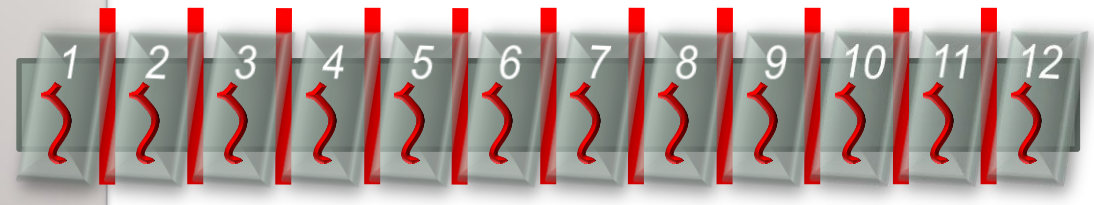
- Fine-grain Parallelism
  - Best opportunities for load balancing, but..
  - Small amounts of computational work between parallelism computation stages
  - Low computation to parallelization ratio ➔ High parallelization overhead
- Coarse-grain Parallelism
  - Harder to load balance efficiently, but..
  - Large amounts of computational work between parallelism computation stages
  - High computation to parallelization ratio ➔ Low parallelization overhead



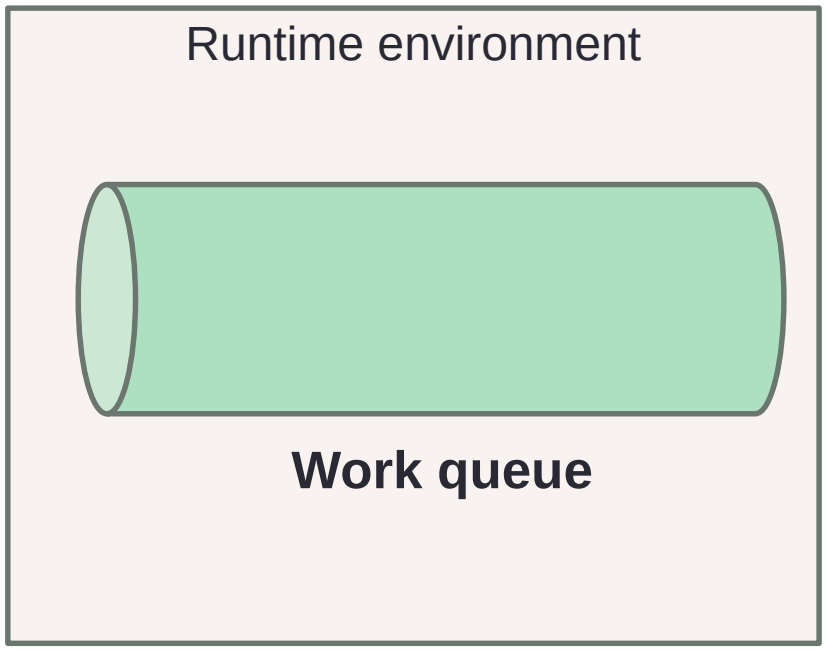
The `schedule` clause
Dynamic Loop Partitioning

```
#pragma omp for schedule(dynamic, 1)
  {
    for (i=0; i<12; i++)
      {
        int start = rand();
        int count = 0;

        while (start++ < 256)
          count++;

        a[count] = foo();
      }
  }
```

Iteration space



The `schedule` clause
Dynamic Loop Partitioning



Iteration space

Runtime scheduling primitive is invoked at every iteration

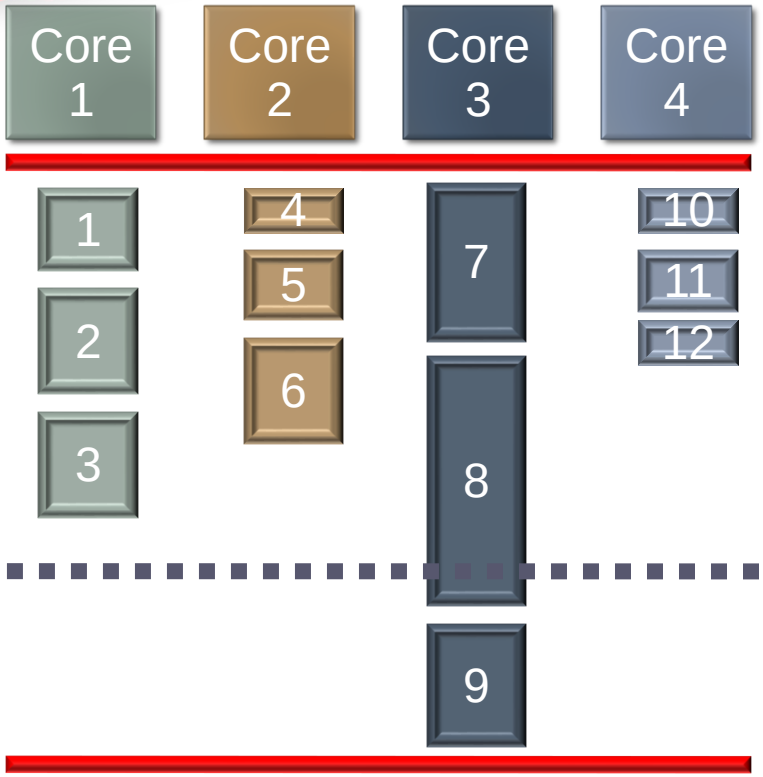
If granularity of WORK is very small the overhead for fine chunking is significant

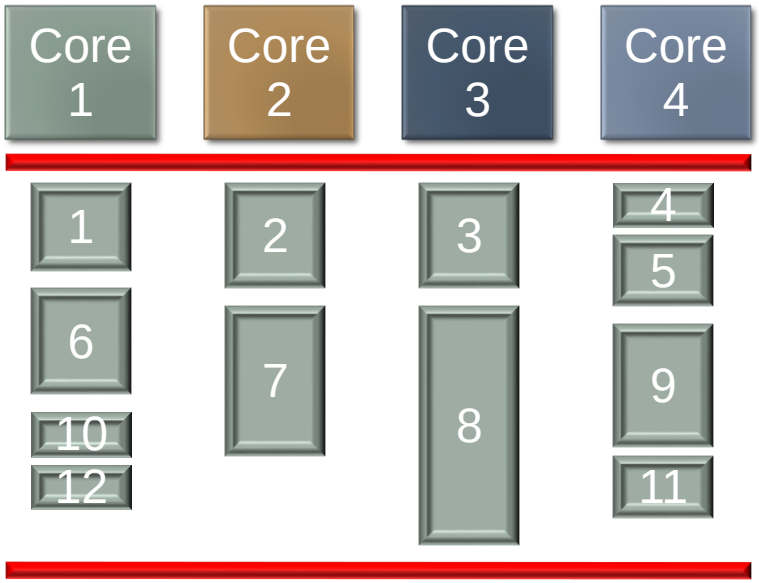
```
int parfun(…)
{
  int LB, UB;
  GOMP_loop_dynamic_next(&LB, &UB);

  for (i=LB; i<UB; i++) {WORK}
}
```

The `schedule` clause
Dynamic Loop Partitioning

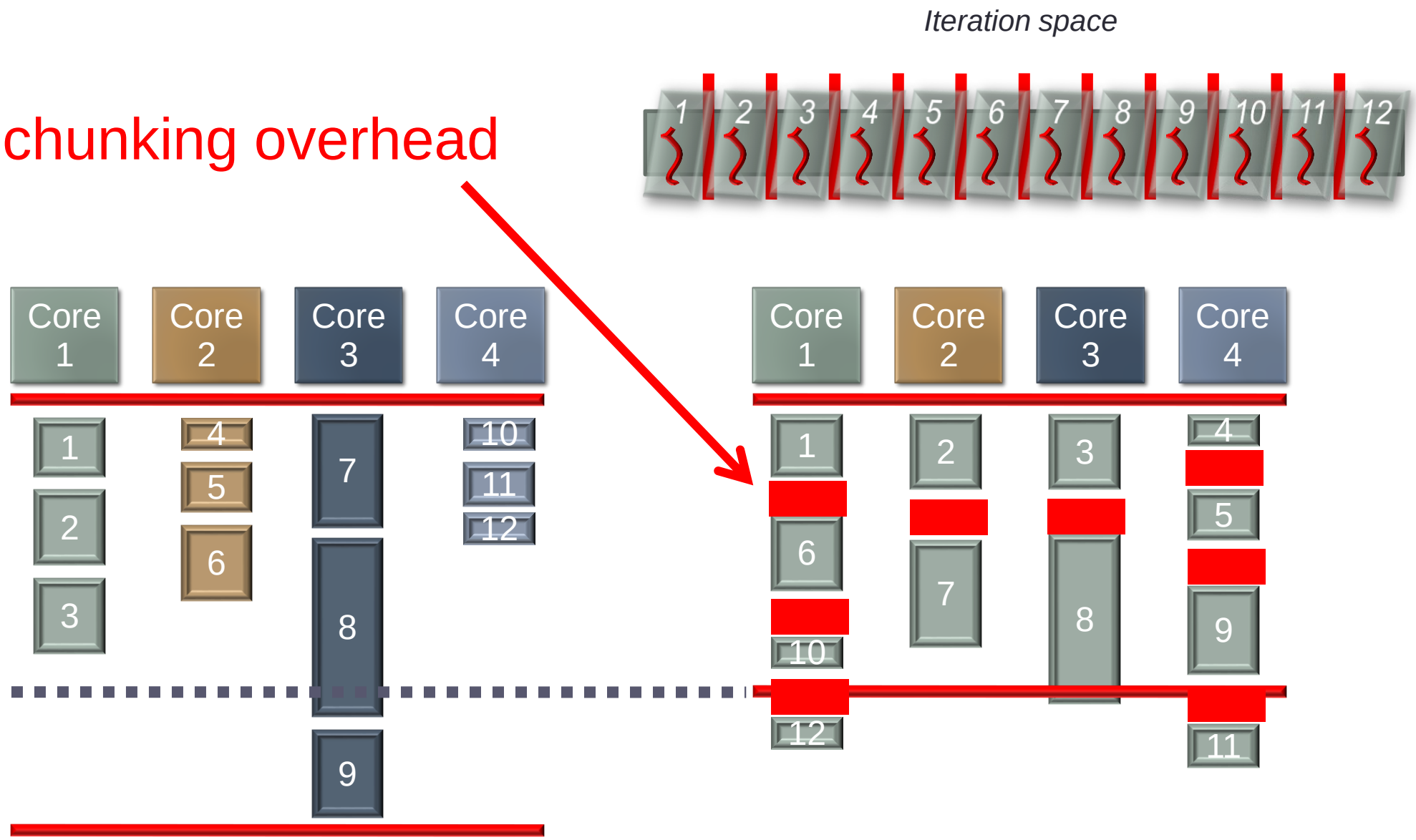
Remember results with static scheduling..

Iteration space





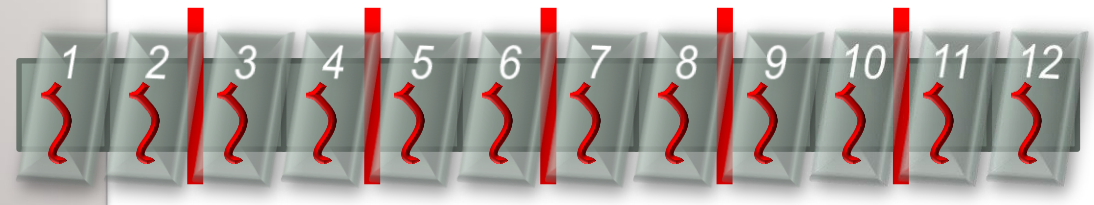
The `schedule` clause
Dynamic Loop Partitioning



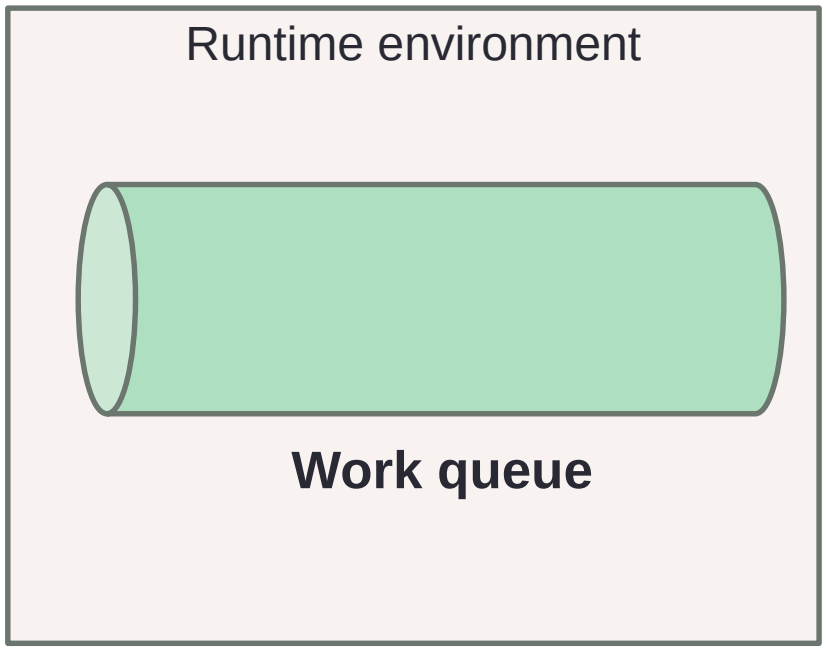
The `schedule` clause
Dynamic Loop Partitioning

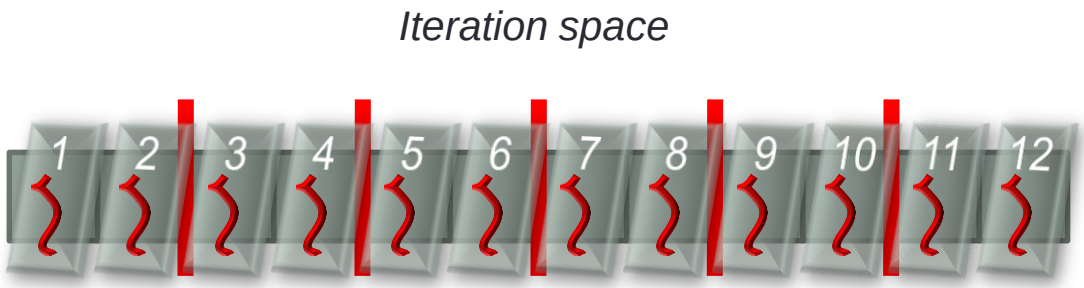
```
#pragma omp for schedule(dynamic, 2)
  {
    for (i=0; i<12; i++)
      {
        int start = rand();
        int count = 0;

        while (start++ < 256)
          count++;

        a[count] = foo();
      }
  }
```

Iteration space



The `schedule` clause
Dynamic Loop Partitioning





Runtime scheduling primitive is invoked every two iterations (half the times of previous case)

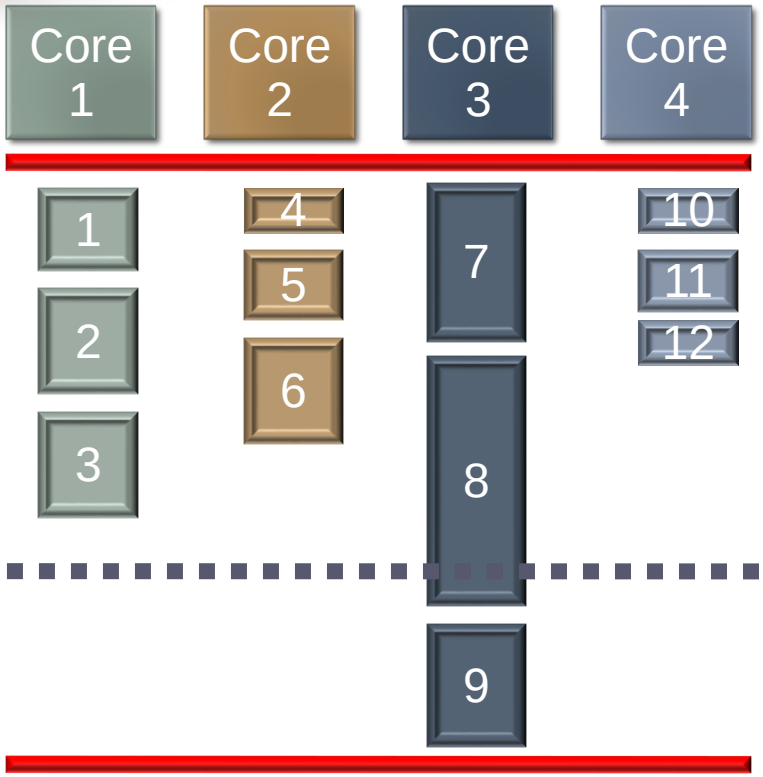
WORK is repeated twice among two scheduling events. Overhead gets amortized

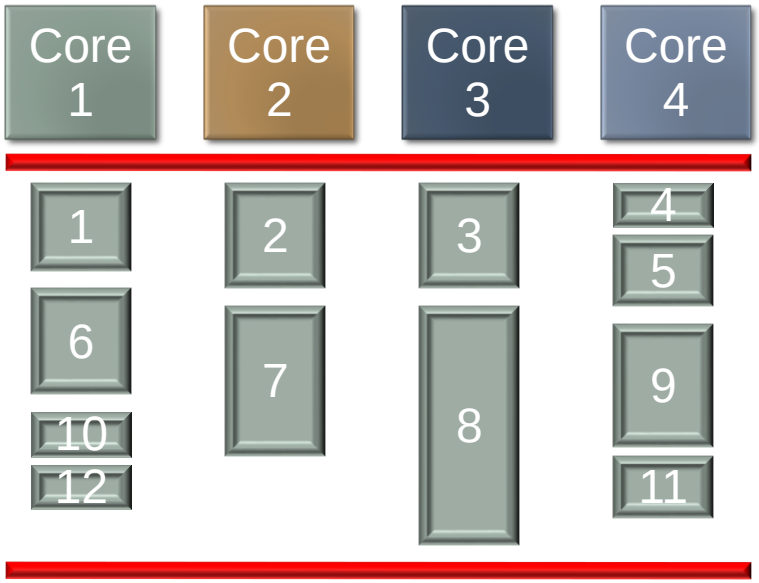
```
int parfun(…)
{
  int LB, UB;
  GOMP_loop_dynamic_next(&LB, &UB);

  for (i=LB; i<UB; i++) {WORK}
}
```

The `schedule` clause
Dynamic Loop Partitioning

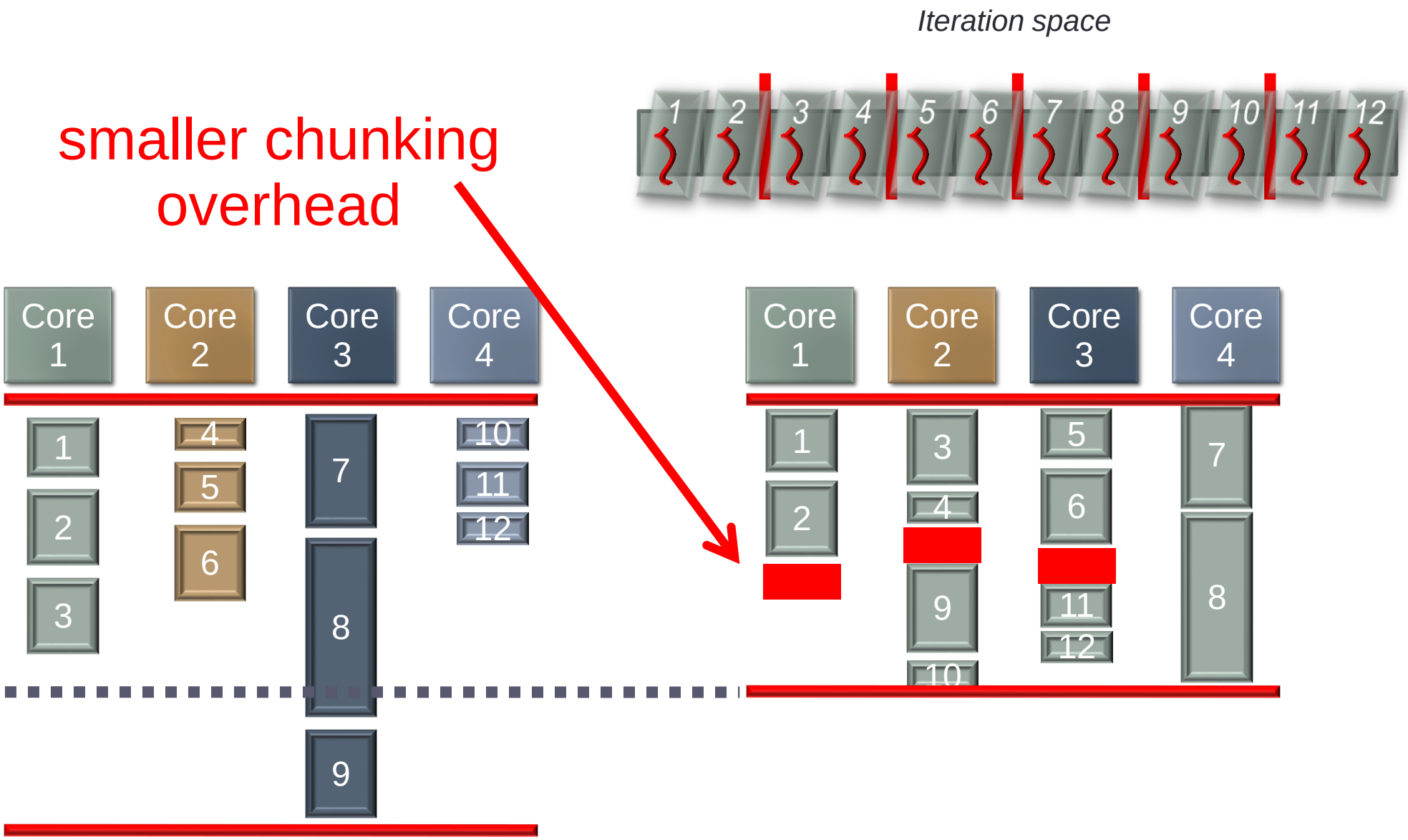
Remember results with static scheduling..

Iteration space





The `schedule` clause
Dynamic Loop Partitioning



More scheduling clauses

- **`schedule (guided[, chunk])`**
  - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
- **`schedule (runtime[, chunk])`**
  - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0)

- **EXERCISE** – Generate the IR dumps for the OpenMP lowering and expansion passes for different scheduling clauses