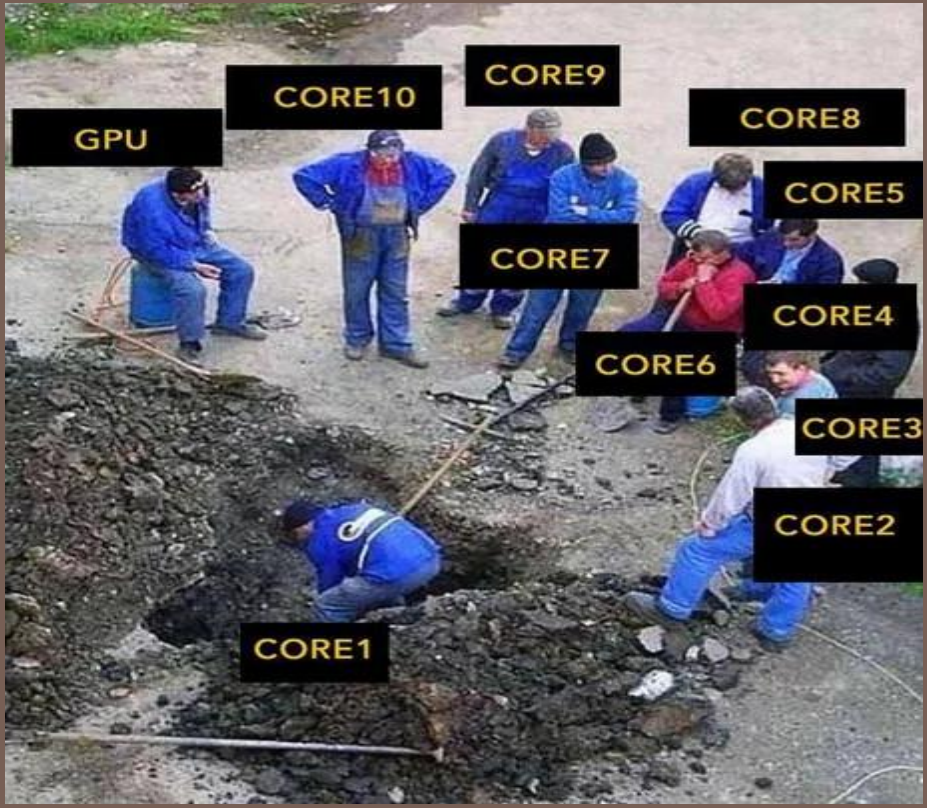
MULTICORE PROGRAMMING WITH OPENMP



26.03.2024

Philip Wiese
wiesep@iis.ee.ethz.ch

Zexin Fu
zexifu@iis.ee.ethz.ch

Original slides by Bjoern Forsberg (bjoernf@iis.ee.ethz.ch)
Based on slides by Dr. Andrea Marongiu (a.marongiu@unibo.it)

Parts of exercise

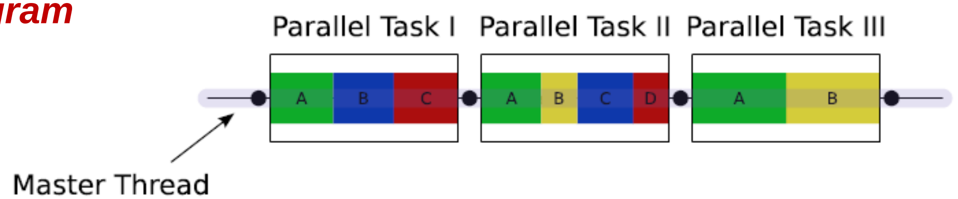
- Understand OpenMP parallelization
  - Annotations, Compiler transformation, library support
- Exercises on core concepts, such as
  - SPMD parallelization (OpenMP parallel for)
    - Work distribution and scheduling
  - MPMD parallelization (OpenMP sections and tasks)
  - Data races
- Parallelization of simple Convolution kernel

Programming model: OpenMP

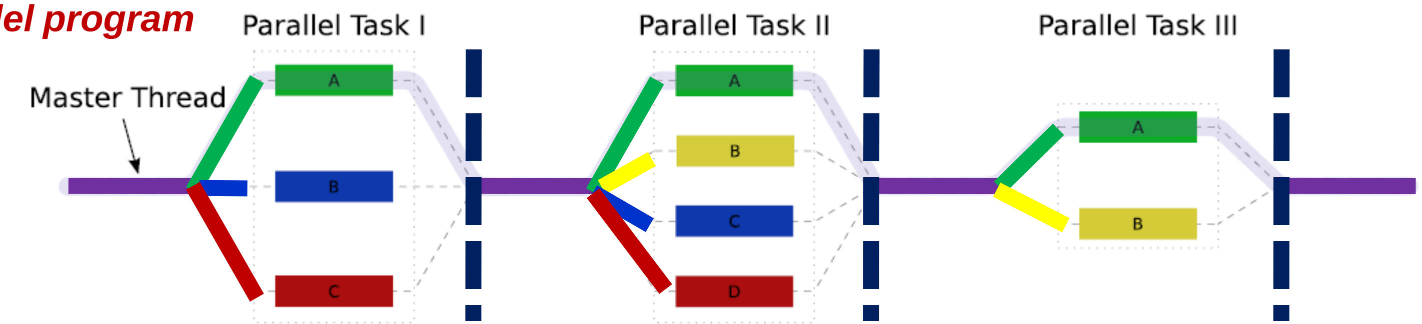
- De-facto standard for the shared memory programming model
- A collection of compiler directives, library routines and environment variables
- Easy to specify parallel execution within a serial code
- Requires special support in the compiler
- Generates calls to threading libraries (e.g. pthreads)
- Focus on loop-level parallel execution (still the case?)
- Popular in high-end embedded

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*>

Example:

#**pragma omp** parallel for num_threads(4)

How many threads

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

- 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*)

A sequential program..
..is easily parallelized

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

Outlining parallelism
The `parallel` directive

- 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*)

A sequential program..
..is easily parallelized

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

```
int main()
{
   omp_parallel_start(&parfun, …);
   parfun();
   omp_parallel_end();
}

int parfun(…)
{
 printf ("\nHello world!");
}
```

#pragma omp parallel

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

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

```
int main()
{
   omp_parallel_start(&parfun, …);
   parfun();
   omp_parallel_end();
}

int parfun(…)
{
 printf ("\nHello world!");
}
```

#pragma omp parallel

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

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

```
int main()
{
  omp_parallel_start(&parfun, …);
  parfun();
  omp_parallel_end();
}

int parfun(…)
{
 printf ("\nHello world!");
}
```

#pragma omp parallel

First we call the runtime to fork new threads, and pass them a pointer to the function to execute in parallel

```
int main()
{
  omp_parallel_start(&parfun, …);
  parfun();
  omp_parallel_end();
}

int parfun(…)
{
 printf ("\nHello world!");
}
```

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

#pragma omp parallel

Then the master itself calls the parallel function

```
int main()
{
  omp_parallel_start(&parfun, …);
  parfun();
  omp_parallel_end();
}

int parfun(…)
{
 printf ("\nHello world!");
}
```

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

#pragma omp parallel

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

```
int main()
{
  omp_parallel_start(&parfun, …);
  parfun();
  omp_parallel_end();
}

int parfun(…)
{
 printf ("\nHello world!");
}
```

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

#pragma omp parallel

Data scope attributes

```
int main()
{
  int id;
  int a = 5;
#pragma omp parallel
  {
    id = omp_get_thread_num();
    if (id == 0)
      printf ("Master: a = %d.", a*2);
    else
      printf ("Slave: a = %d.", a);
  }
}
```

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

```c
int main()
{
  int id;
  int a = 5;
#pragma omp parallel
  {
    id = omp_get_thread_num();
    if (id == 0)
      printf ("Master: a = %d.", a*2);
    else
      printf ("Slave: a = %d.", a);
  }
}
```

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

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

How to inform the compiler about these different behaviors?

A slightly more complex example

#pragma omp parallel

Data scope attributes

What is the view of memory among different threads in a parallel region?

```
int main()
{
  int id;
  int a = 5;
#pragma omp parallel shared (a) private (id)
  {
    id = omp_get_thread_num();
    if (id == 0)
      printf ("Master: a = %d.", a*2);
    else
      printf ("Slave: a = %d.", a);
  }
}
```

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

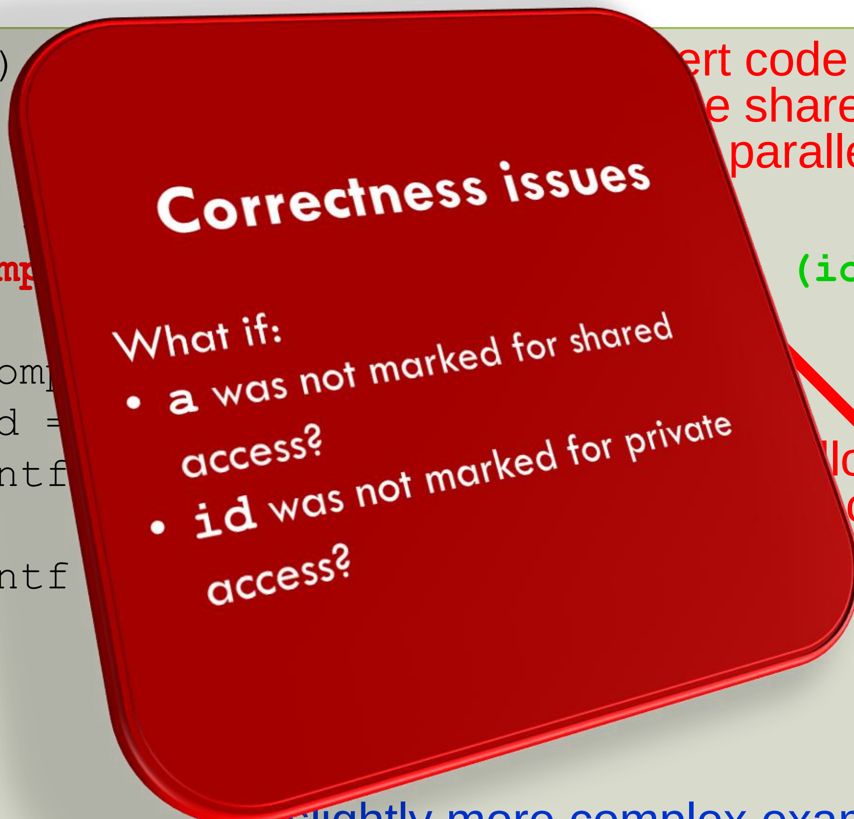
#pragma omp parallel

Data scope attributes

```
int main()
{
  int id;
  int a =
#pragma omp                                   (id)
  {
    id = om
    if (id =
      printf
    else
      printf
  }
}
```

ert code to retrieve the address
e shared object from within
parallel thread

llow symbol privatization:
ch thread contains a
vate copy of this variable



A slightly more complex example

More data sharing clauses

- firstprivate
  - copyin, private storage
- lastprivate
  - copyout, private storage

SPMD VS MPMD

Recall..

- SPMD (single program, multiple data)
  - Processors execute the same stream of instructions over different data
  - `#pragma omp for`
- MPMD (multiple program, multiple data)
  - Processors execute different streams of instructions over (possibly) different data
  - `#pragma omp sections`
  - `#pragma omp task`

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

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

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

```
int main()
{
  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

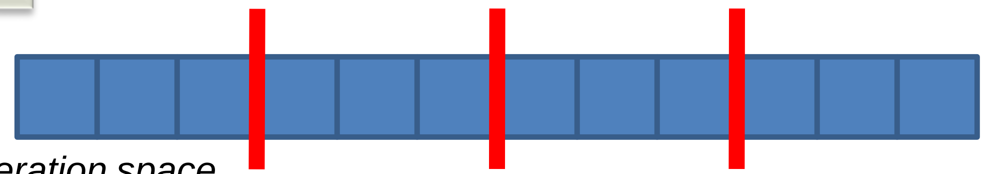
Es. 12 iterations (N), 4 threads (Nthr)

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

DATA CHUNK

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

$$\frac{\textbf{3 iterations}}{\textbf{thread}}$$



Iteration space

Useful for:

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

		0	1	2	3
	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

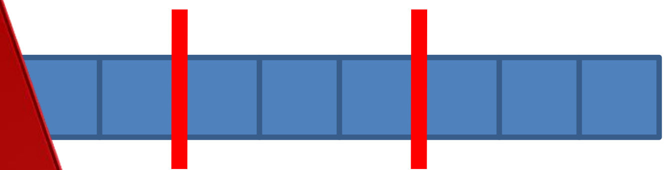
Es. 12 iterations (N), 4 threads (Nthr)

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

DATA CHUNK

...il (N / Nthr)

3 iterations / thread



Useful for:

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

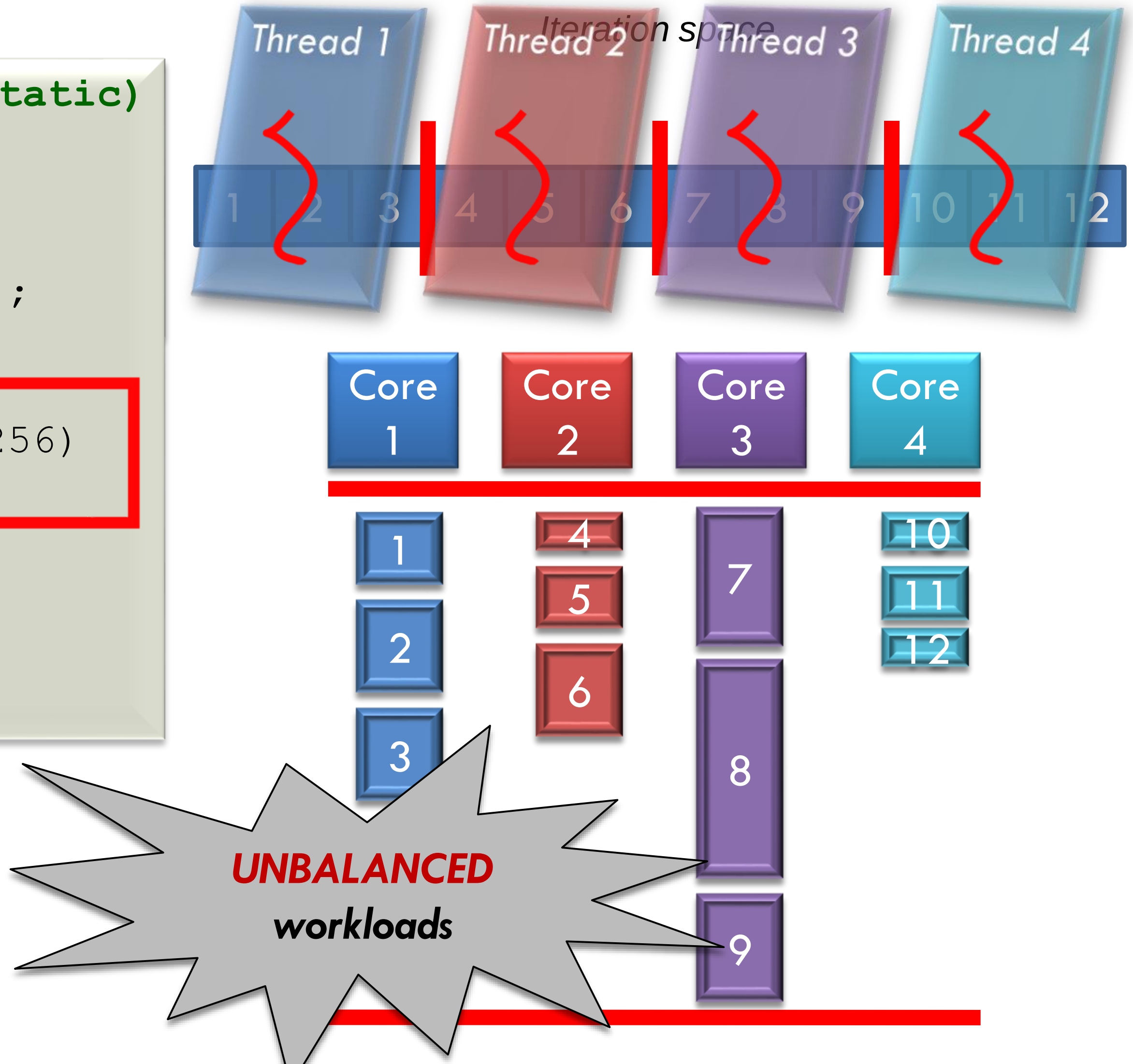
What happens with static scheduling when iterations have different duration?

	0	1	2	3
LOWER BOU...	0	3	6	9
UPPER BOUND ...[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++)
      {
        int start = rand();
        int count = 0;

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

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

A variable amount of work in each iteration

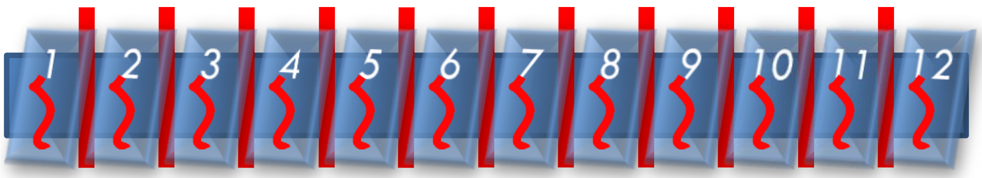


The `schedule` clause
Dynamic Loop Partitioning

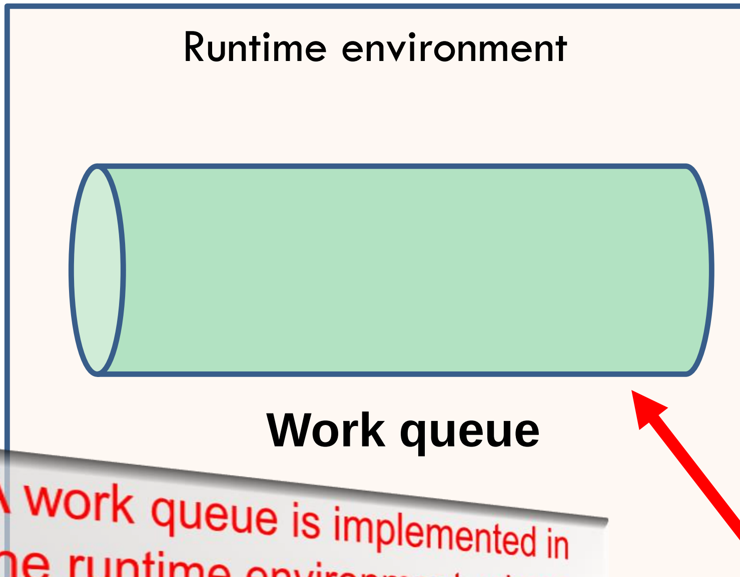
```
#pragma omp for schedule(dynamic)
  {
    for (i=0; i<12; i++)
      {
        int start = rand();
        int count = 0;

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

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

A thread is generated for every single iteration

Iteration space

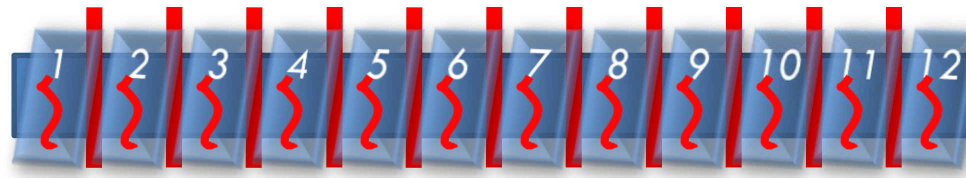


The `schedule` clause
Dynamic Loop Partitioning



A work queue is implemented in the runtime environment, where tasks are stored

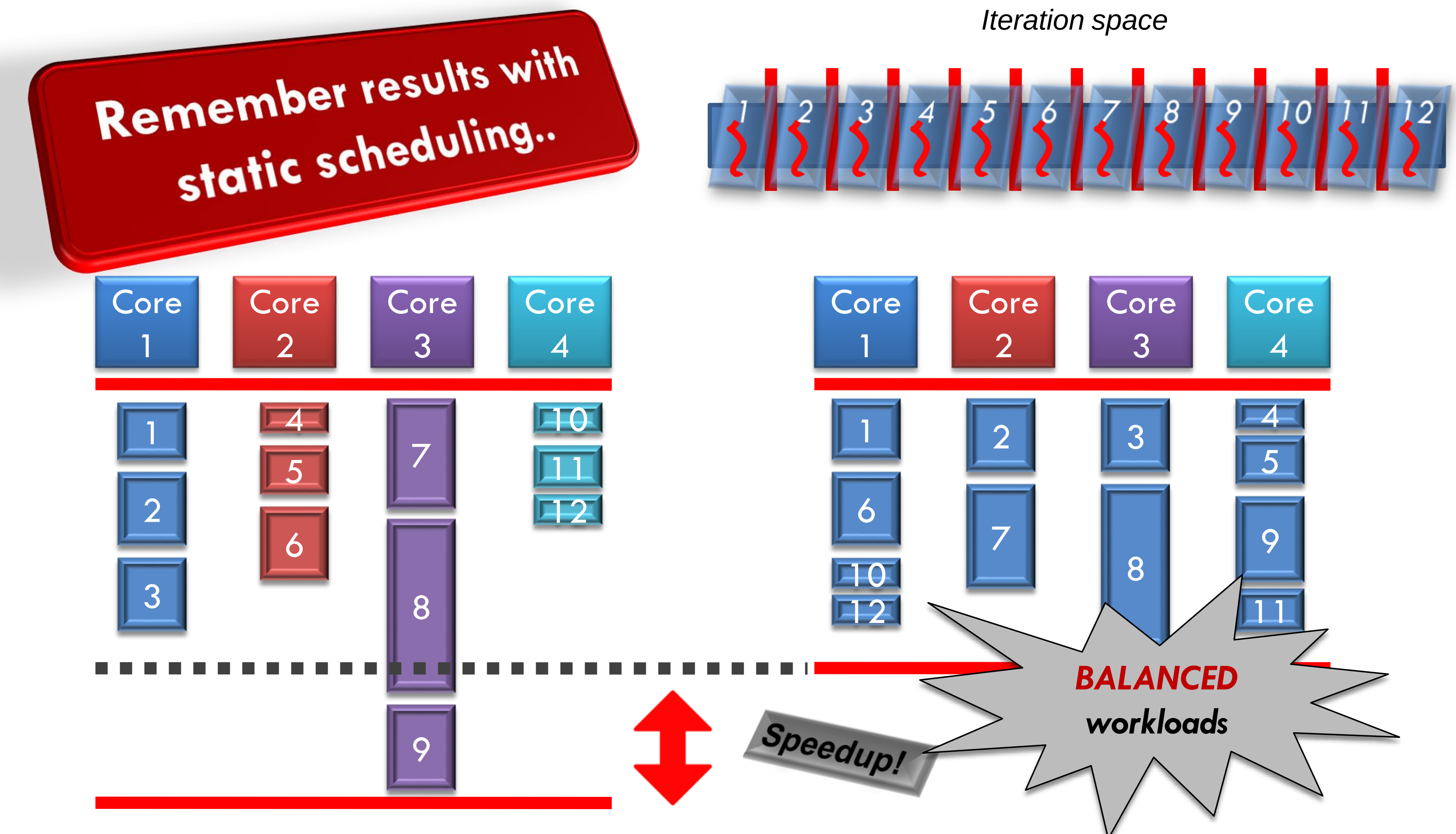
Iteration space



Work is fetched by threads from the runtime environment through synchronized accesses to the **work queue**

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

   for (i=LB; i<UB; i++) {…}
}
```

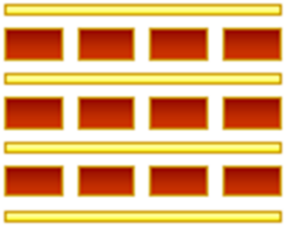
The `schedule` clause
Dynamic Loop Partitioning



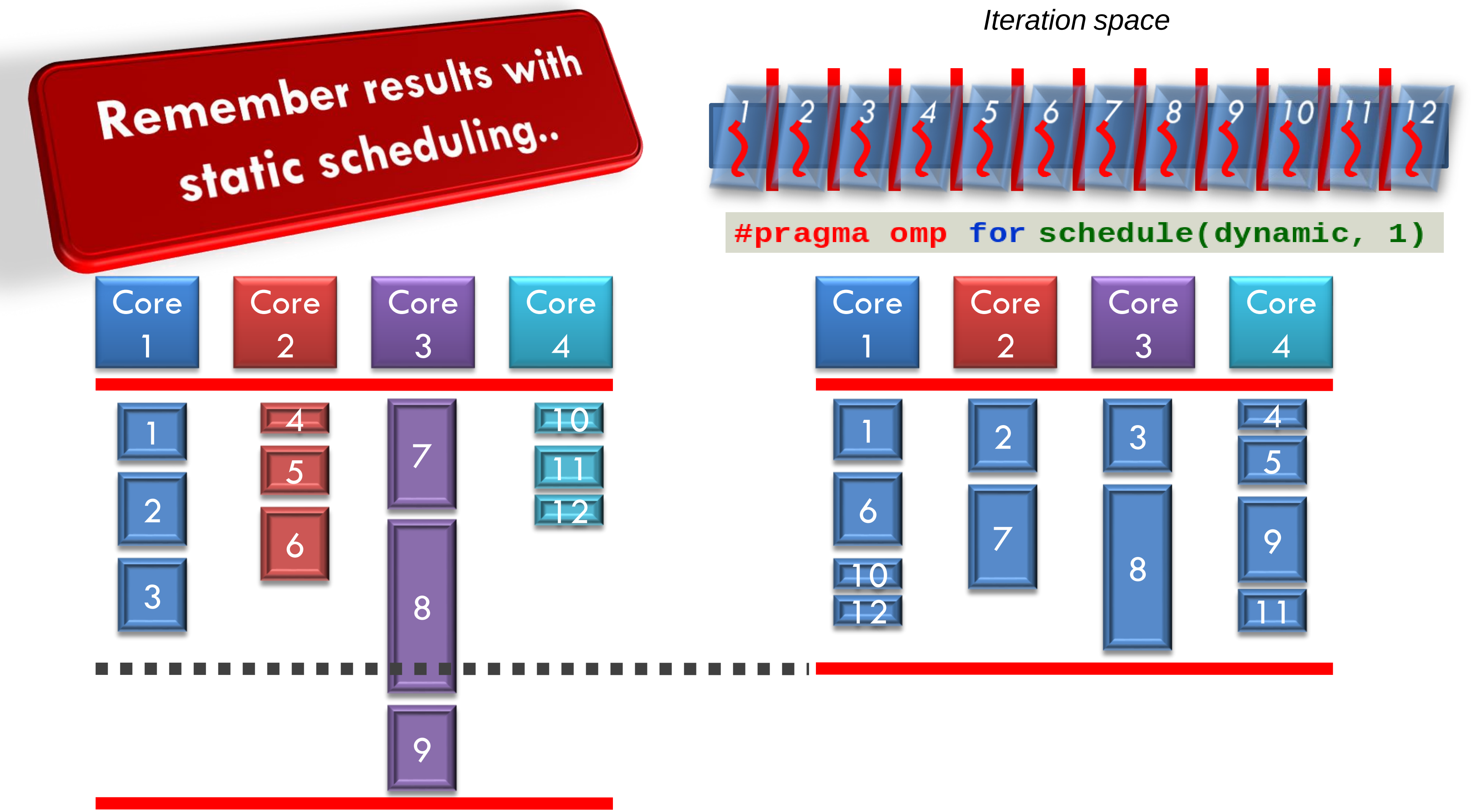
Parallelization granularity

Iteration chunking

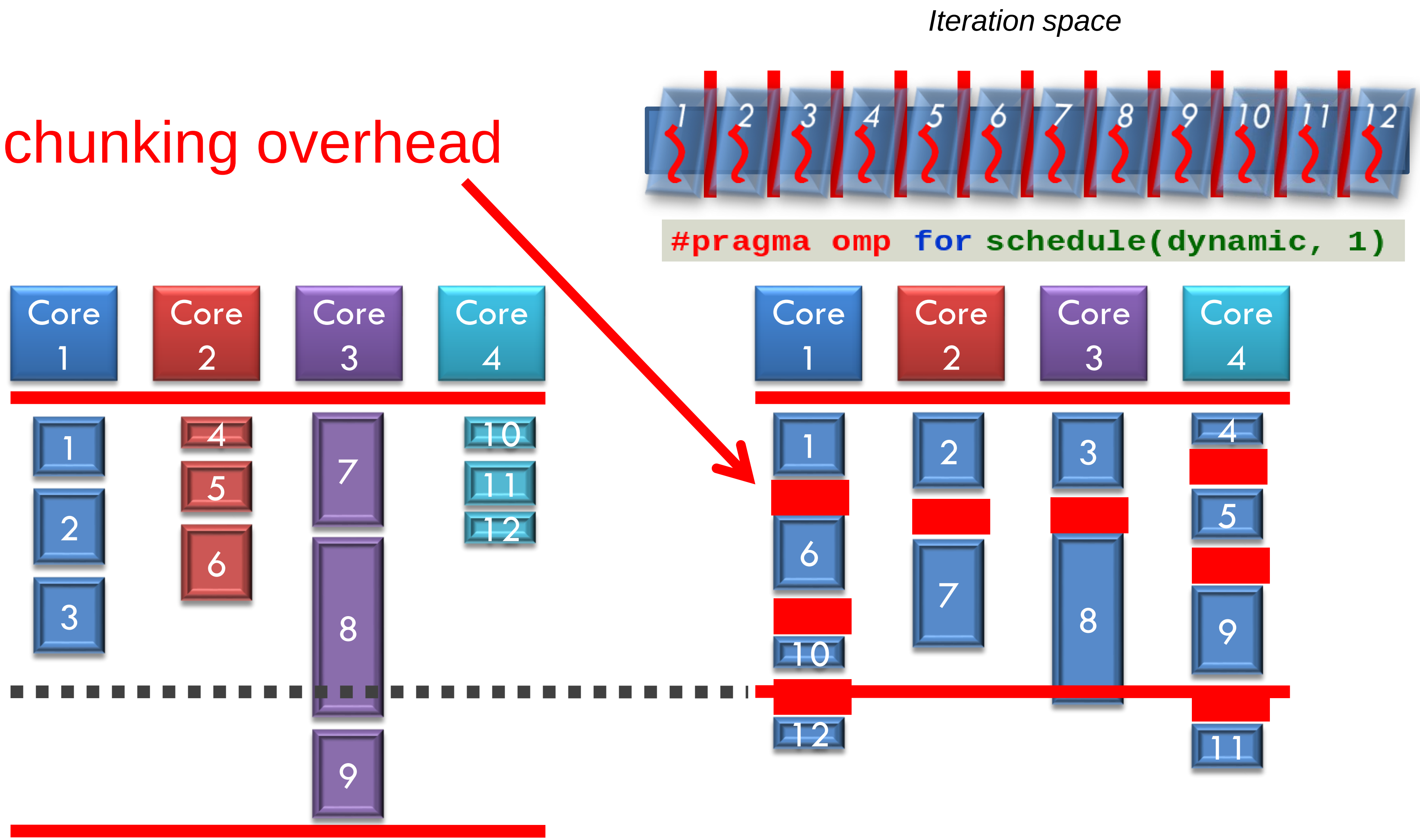
- Fine-grain Parallelism
  - Best opportunities for load balancing, but..
  - Small amounts of computational work between parallelism computation stages
  - Low computation to parallelization ratio ➔ High parallelization overhead
- Coarse-grain Parallelism
  - Harder to load balance efficiently, but..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



The `schedule` clause
Dynamic Loop Partitioning

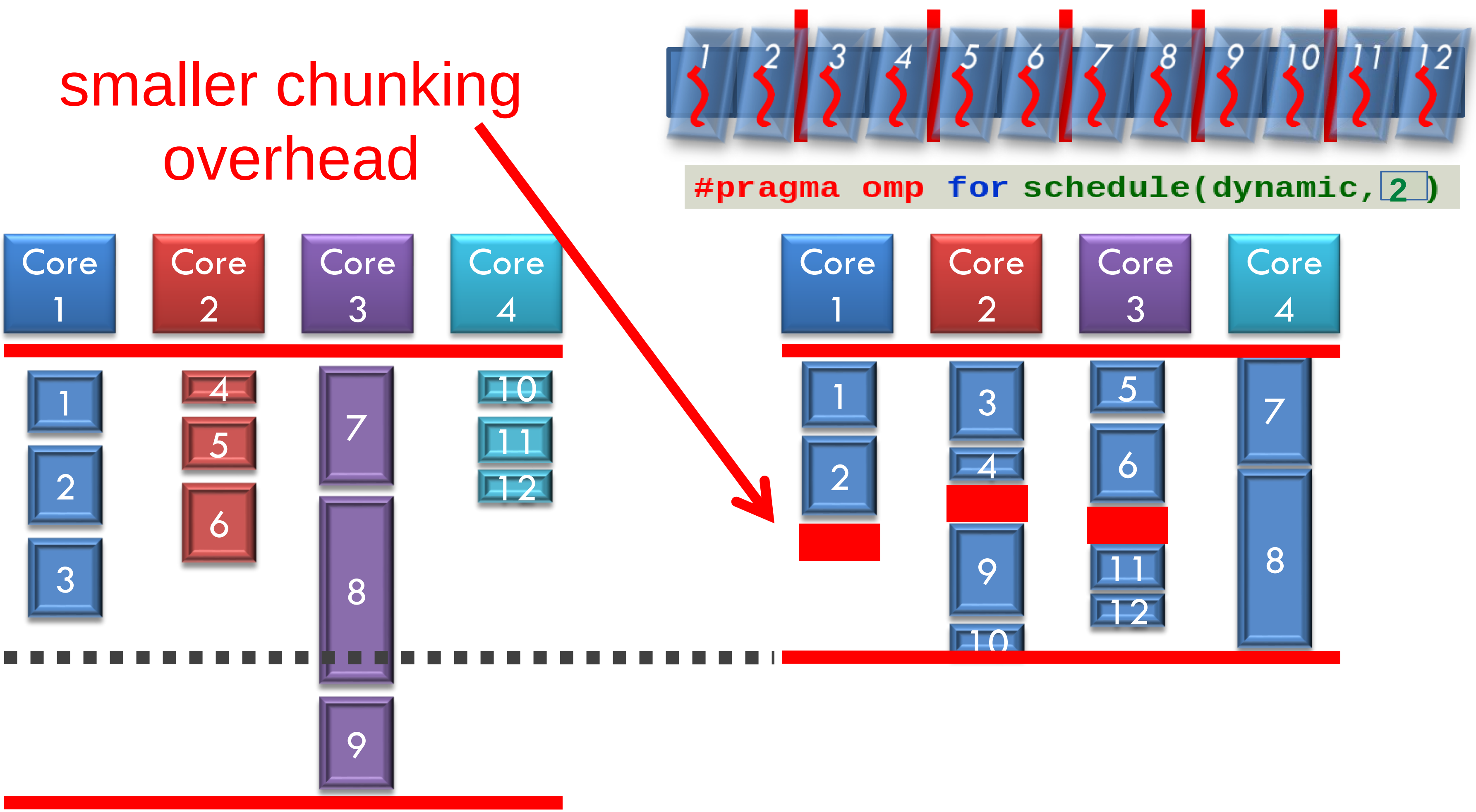


The `schedule` clause
Dynamic Loop Partitioning



The `schedule` clause
Dynamic Loop Partitioning

Iteration space




```
#pragma omp for schedule(dynamic,2)
```

#pragma omp barrier

- Most important synchronization mechanism in shared memory fork/join parallel programming
- All threads participating in a parallel region wait until everybody has finished before computation flows on
- This prevents later stages of the program to work with inconsistent shared data
- It is implied at the end of **parallel** constructs, as well as **for** and **sections** (unless a ***nowait*** clause is specified)

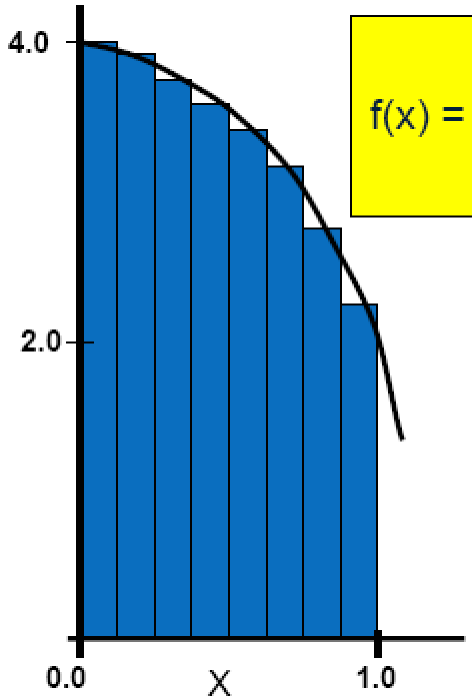
#pragma omp critical

- Critical Section: a portion of code that only one thread at a time may execute
- We denote a critical section by putting the pragma

```
#pragma omp critical
```

in front of a block of C code

π-finding code example



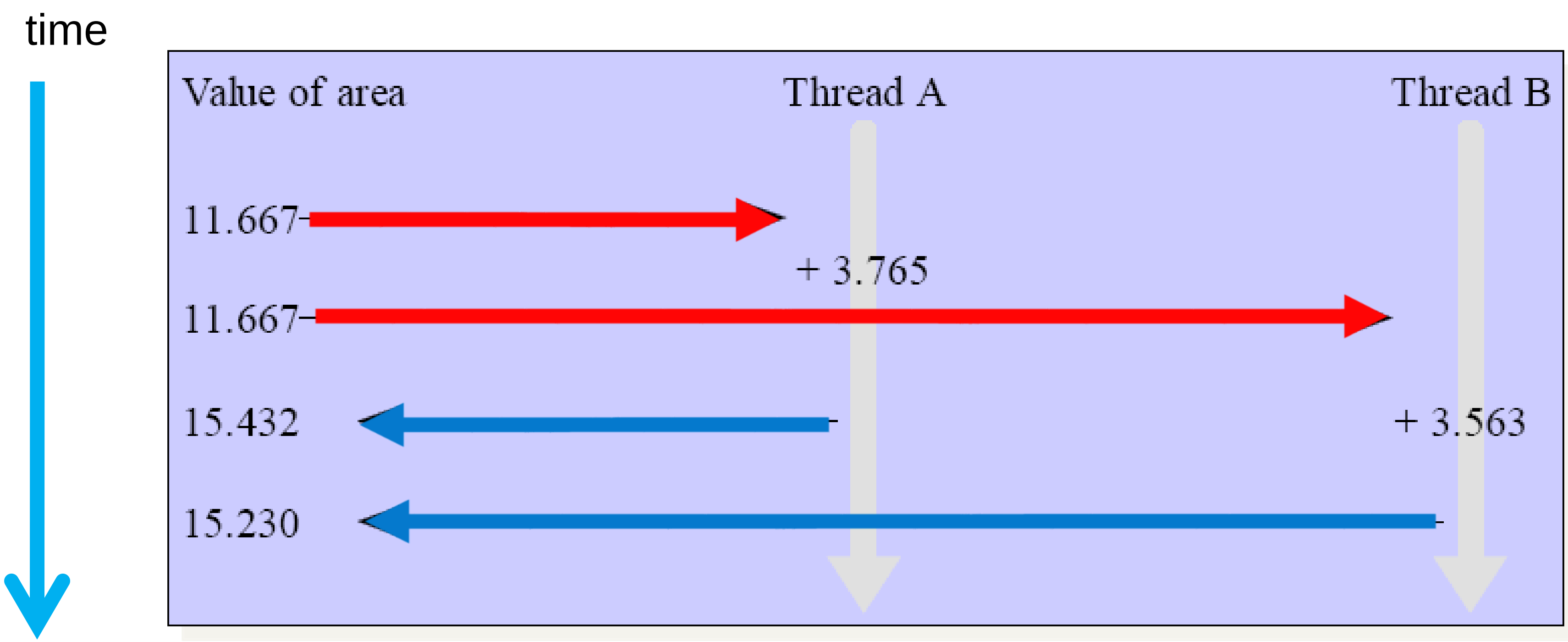
$$f(x) = \frac{4.0}{(1+x^2)}$$

```
double area, pi, x;
int i, n;
#pragma omp parallel for private(x) \
                          shared(area)
{
  for (i=0; i<n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x);
  }
}
  pi = area/n;
```

Synchronize accesses to shared variable **area** to avoid inconsistent results

Race condition (Cont'd)

- *Thread A reads "11.667" into a local register*
- *Thread B reads "11.667" into a local register*
- *Thread A updates area with "11.667+3.765"*
- *Thread B ignores write from thread A and updates area with "11.667 + 3.563"*



π-finding code example

```
double area, pi, x;
int i, n;
#pragma omp parallel for private(x) shared(area)
{
  for (i=0; i<n; i++) {
    x = (i +0.5)/n;
#pragma omp critical
    area += 4.0/(1.0 + x*x);
  }
}
  pi = area/n;
```

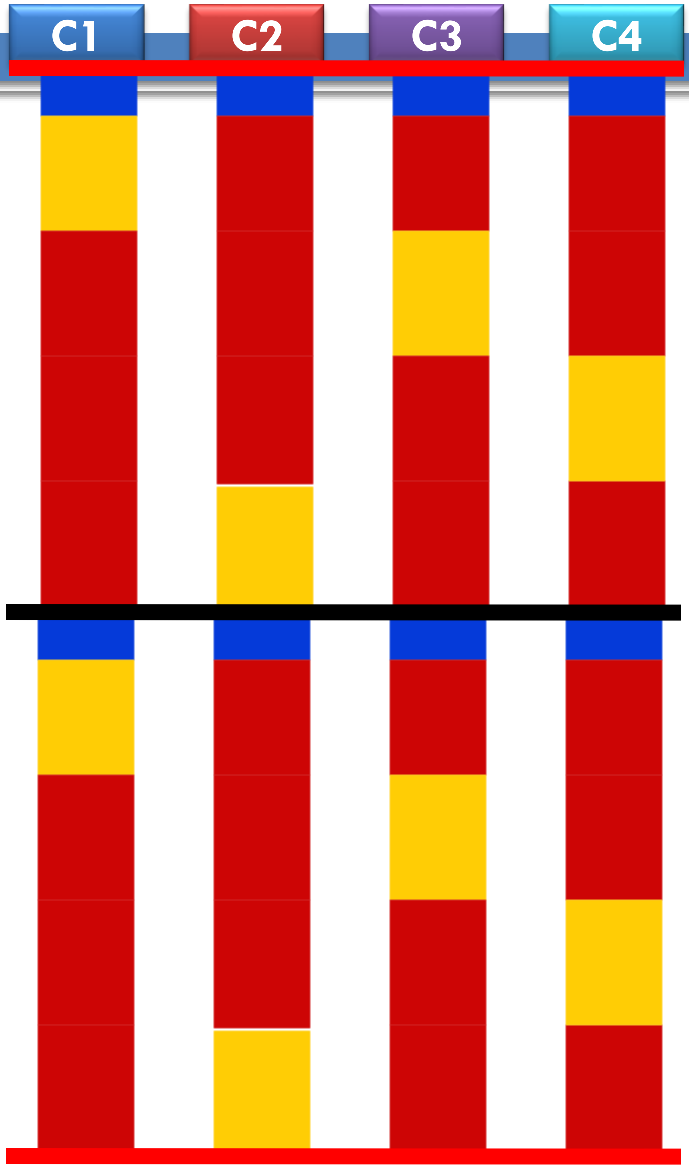
#pragma omp critical protects the code within its scope by acquiring a lock before entering the critical section and releasing it after execution

π-finding code example

```
double area, pi, x;
int i, n;
#pragma omp parallel for \
            private(x)   \
            shared(area)
{
  for (i=0; i<n; i++) {
    x = (i +0.5)/n;
  #pragma omp critical
    area += 4.0/(1.0 + x*x);
  }
}
  pi = area/n;
```

Parallel

Sequential



Waiting for lock

Correctness, not performance!

- A programming pattern such as `area += 4.0/(1.0 + x*x);` in which we:
  - *Fetch the value of an operand*
  - *Add a value to it*
  - *Store the updated value*

  is called a reduction, and is commonly supported by parallel programming APIs

- OpenMP takes care of storing partial results in private variables and combining partial results after the loop

Correctness, not performance!

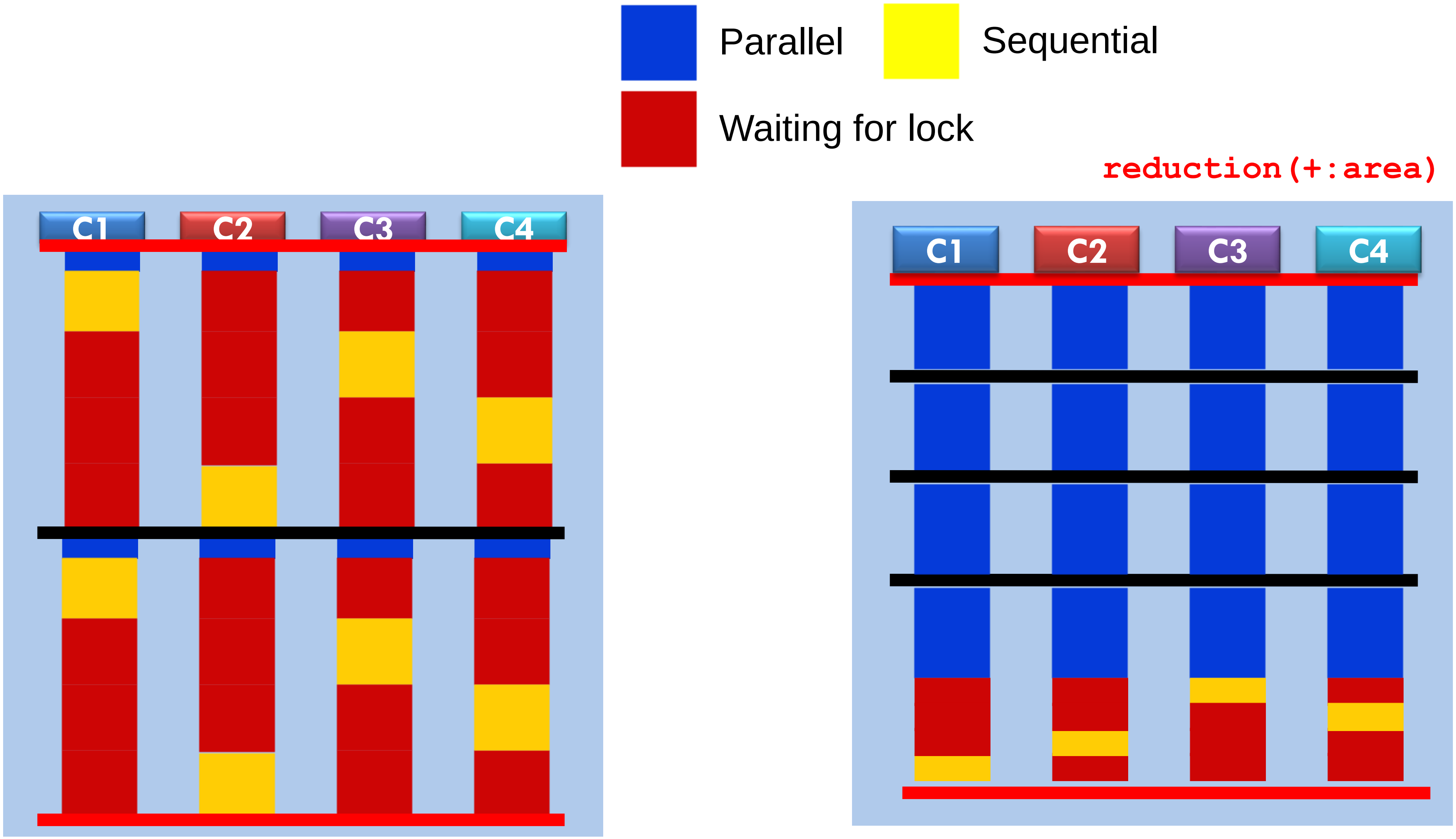
- As a matter of fact, using locks makes execution sequential
- To dim this effect we should try use fine grained locking (i.e. make critical sections as small as possible)
- A simple instruction to compute the value of area in the previous example is translated into many more simpler instructions within the compiler!
- The programmer is not aware of the real granularity of the critical section

Correctness, not performance!

```
double area, pi, x;
int i, n;
#pragma omp parallel for private(x) shared(area) reduction(+:area)
{
   for (i=0; i<n; i++) {
      x = (i +0.5)/n;

      area += 4.0/(1.0 + x*x);
   }
}
   pi = area/n;
```

The **reduction** clause instructs the compiler to create **private** copies of the **area** variable for every thread. At the end of the loop partial sums are combined on the shared **area** variable

Correctness, not performance!



SPMD VS MPMD

Recall..

- SPMD (single program, multiple data)
  - Processors execute the same stream of instructions over different data
  - `#pragma omp for`
- MPMD (multiple program, multiple data)
  - Processors execute different streams of instructions over (possibly) different data
  - `#pragma omp sections`
  - `#pragma omp task`

Sharing work among threads
The `sections` directive

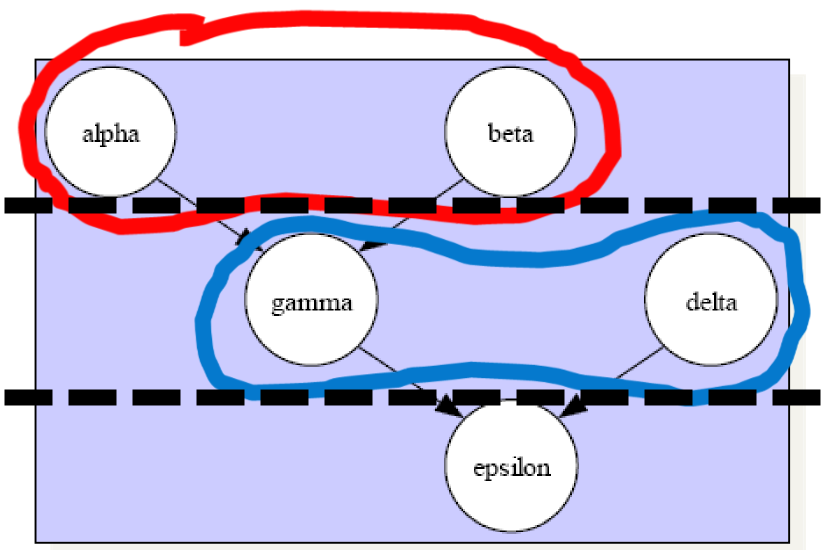
- The **for** pragma allows to exploit data parallelism in loops
- OpenMP also provides directives to exploit task parallelism

```
#pragma omp sections
```

Task Parallelism Example

```
int main()
{

      v = alpha();
      w = beta ();

      y = delta ();
      x = gamma (v, w);
      z = epsilon (x, y));

   printf ("%f\n", z);
}
```

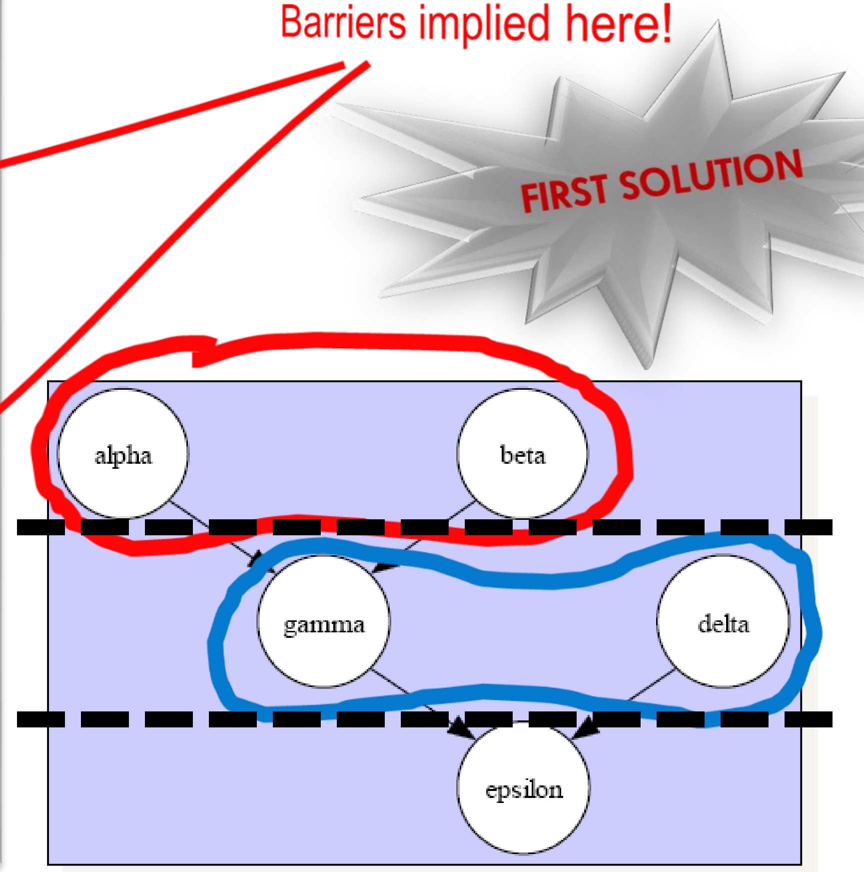
Identify independent nodes in the task graph, and outline parallel computation with the **sections** directive



Task Parallelism Example

```
int main()
{
#pragma omp parallel sections {

      v = alpha();

      w = beta ();
 }
#pragma omp parallel sections {

      y = delta ();

      x = gamma (v, w);
 }
      z = epsilon (x, y));


  printf ("%f\n", z);
}
```

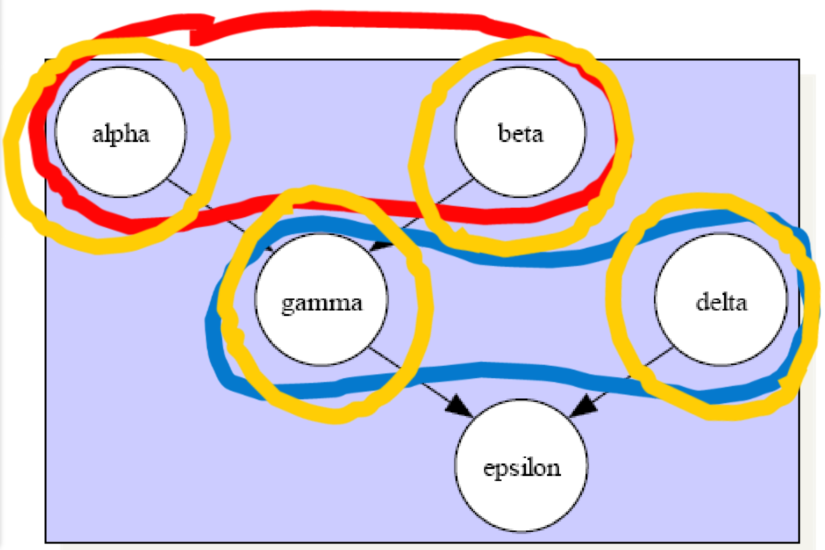
Barriers implied here!



Task Parallelism Example

```
int main()
{
#pragma omp parallel sections {
   #pragma omp section
        v = alpha();
   #pragma omp section
        w = beta ();
 }
#pragma omp parallel sections {
   #pragma omp section
        y = delta ();
   #pragma omp section
        x = gamma (v, w);
 }
        z = epsilon (x, y));


   printf ("%f\n", z);
}
```

Each parallel task within a **sections** block identifies a **section**

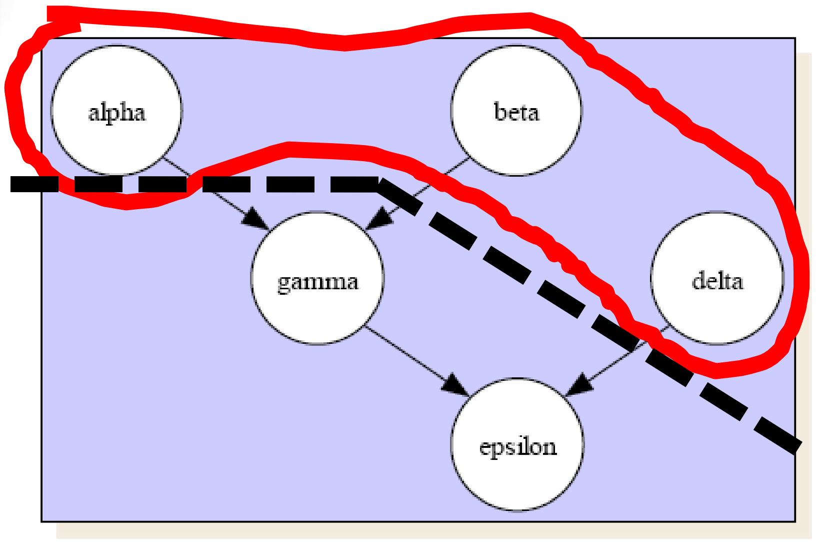


Task Parallelism Example

```
int main()
{


    v = alpha();

    w = beta ();

    y = delta ();


    x = gamma (v, w);

    z = epsilon (x, y));


  printf ("%f\n", z);
}
```

SECOND SOLUTION

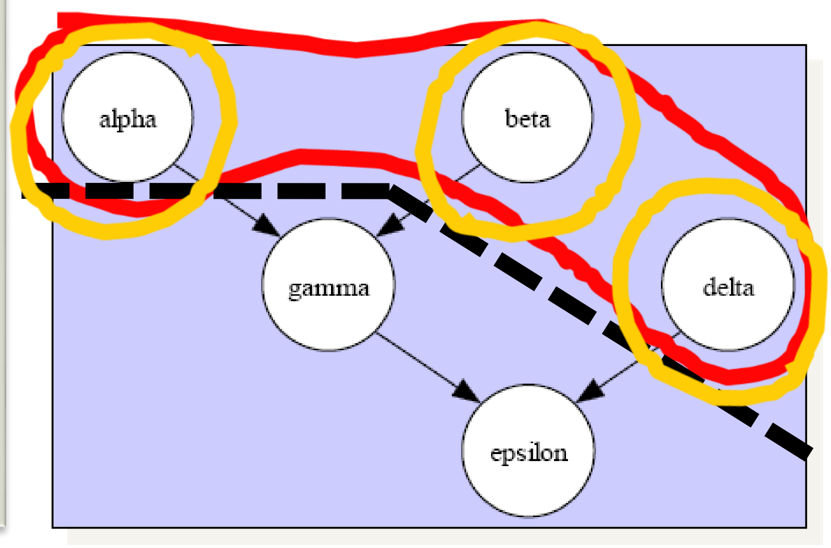
Identify independent nodes in the task graph, and outline parallel computation with the **sections** directive



Task Parallelism Example

```
int main()
{
 #pragma omp parallel sections {

      v = alpha();

      w = beta ();

      y = delta ();
}


      x = gamma (v, w);

      z = epsilon (x, y));


  printf ("%f\n", z);
}
```

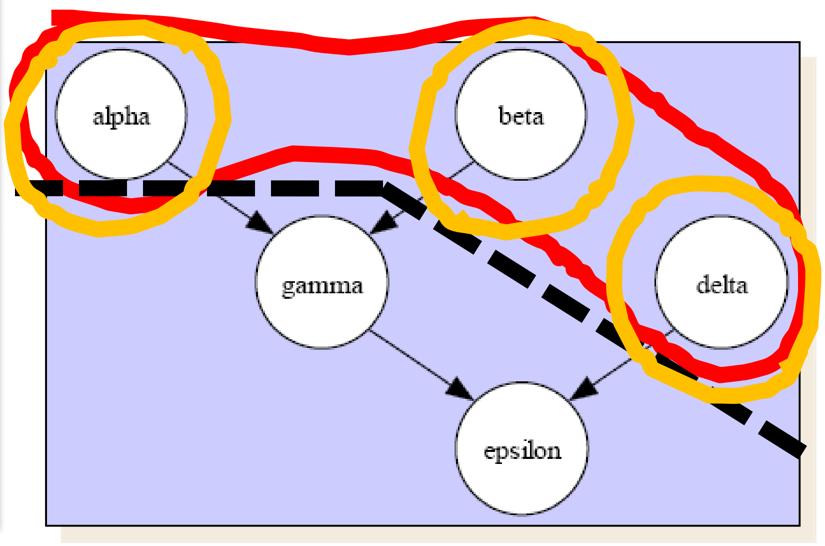
Each parallel task within a **sections** block identifies a **section**



Task Parallelism Example

```
int main()
{
 #pragma omp parallel sections {
   #pragma omp section
      v = alpha();
   #pragma omp section
      w = beta ();
   #pragma omp section
      y = delta ();
}


      x = gamma (v, w);

      z = epsilon (x, y));


  printf ("%f\n", z);
}
```

Each parallel task within a **sections** block identifies a **section**



Task parallelism

- The `sections` directive allows a very limited form of task parallelism

- All tasks must be statically outlined in the code
  - What if a functional loop (`while`) body is identified as a task?
    - **Unrolling**? Not feasible for high iteration count
  - What if recursion is used?

What is an OpenMP task?

- Tasks are work units which execution **may** be deferred
  - they can also be executed immediately!
- Tasks are composed of:
  - code to execute
  - data environment
    - Initialized at creation time
  - internal control variables (ICVs)

Task directive

#pragma omp task [clauses]

structured block

- Each encountering thread creates a task
  - Packages code and data environment
- Highly composable. Can be nested
  - inside parallel regions
  - inside other tasks
  - inside worksharing constructs (for, sections)

List traversal with tasks

- Why?
  - Example: **list traversal**

EXAMPLE

```
void traverse_list (List l)
{
  Element e ;

  for ( e = e→first; e; e = e→next )
#pragma omp task
    process ( e ) ;
}
```

What is the scope of e?

Task data scoping

- Data scoping clauses
  - shared(list)
  - private(list)
  - firstprivate(list)
    - data is captured at creation
  - default(shared | none)

Task data scoping

when there are no clauses..

- If no clause
  - Implicit rules apply
    - e.g., global variables are shared
- Otherwise...
  - firstprivate
  - shared attribute is lexically inherited

Tip

default(none) is your friend

Use it if you do not see it clear

List traversal with tasks

EXAMPLE

```
void traverse_list (List l)
{
  Element e ;

  for ( e = e→first; e; e = e→next )
#pragma omp task
    process ( e ) ;
}
```

e is **firstprivate**

List traversal with tasks

EXAMPLE

```
void traverse_list (List l)
{
  Element e ;

  for ( e = e→first; e; e = e→next )
#pragma omp task
    process ( e ) ;
}
```

how we can guarantee here that the traversal is finished?

Task synchronization

- Barriers (implicit or explicit)
  - All tasks created by any thread of the current team are guaranteed to be completed at barrier exit
- Task barrier

  #pragma omp taskwait
  - Encountering task suspends until **child** tasks complete
    - Only direct **childs**, not descendants!

List traversal with tasks

EXAMPLE

```
void traverse_list (List l)
{
  Element e ;

  for ( e = e→first; e; e = e→next )
#pragma omp task
    process ( e ) ;

#pragma omp taskwait
}
```

All tasks guaranteed to be completed here

Task parallelism

- Why?
  - Example: **list traversal**

CAREFUL!

- **Multiple traversal of the same list**

EXAMPLE

```
List l;

#pragma omp parallel

traverse_list (l);
```

```
void traverse_list (List l)
{
  Element e ;

  for ( e = e→first; e; e = e→next )
#pragma omp task
    process ( e ) ;
}
```

Task parallelism

- Why?
  - Example: **list traversal**

EXAMPLE

```
List l;

#pragma omp parallel
#pragma omp single
traverse_list (l);
```

Single traversal

- One thread enters single and creates all tasks
- All the team cooperates executing them

```
void traverse_list (List l)
{
  Element e ;

  for ( e = e→first; e; e = e→next )
#pragma omp task
    process ( e ) ;
}
```

Task parallelism

- In case **`task`** is within a regular **counted loop** an alternative is to parallelize task creation among threads

Multiple traversals

- Multiple threads create tasks
- All the team cooperates executing them

EXAMPLE

```
/* A DIFFERENT EXAMPLE */

#pragma omp parallel

Myfunc ();
```

```
void Myfunc ()
{
  int i;
#pragma omp for
  for (i=LB; i<UB; i++)
#pragma omp task
    process ( i ) ;
}
```

EXERCISE!