MULTICORE PROGRAMMING WITH OPENMP

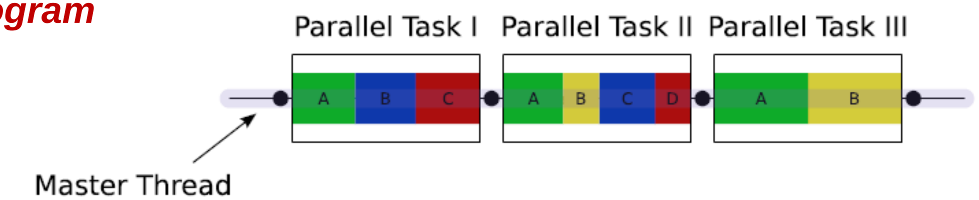
Ing. Andrea Marongiu
a.marongiu@iis.ee.ethz.ch

Programming model: OpenMP

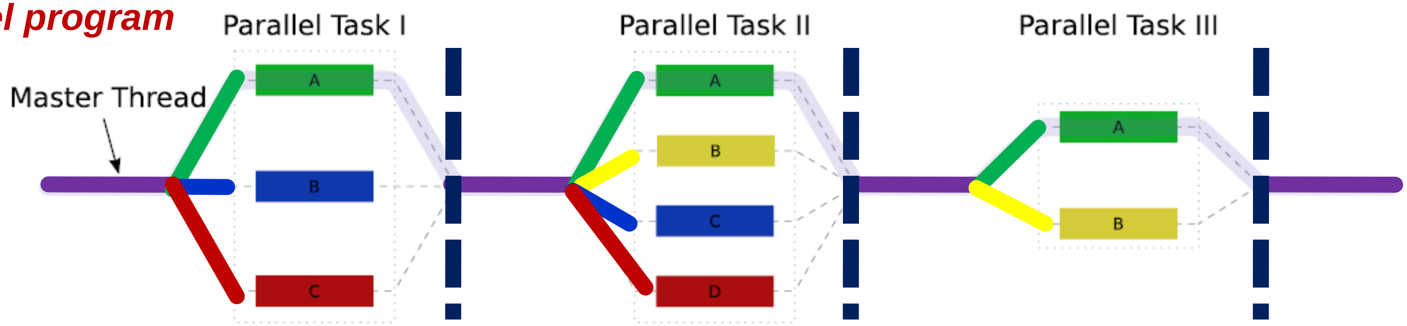
- De-facto standard for the shared memory programming model
- A collection of compiler directives, library routines and environment variables
- Easy to specify parallel execution within a serial code
- Requires special support in the compiler
- Generates calls to threading libraries (e.g. pthreads)
-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

Components of OpenMP

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!”);
  }
}
```

```
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

```
int main()
{
  int id;
  int a = 5;
#pragma omp parallel
  {
    id = omp_get_thread_num();
    if (id == 0)
      printf (“Master: a = %d.”, a);
    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

```
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

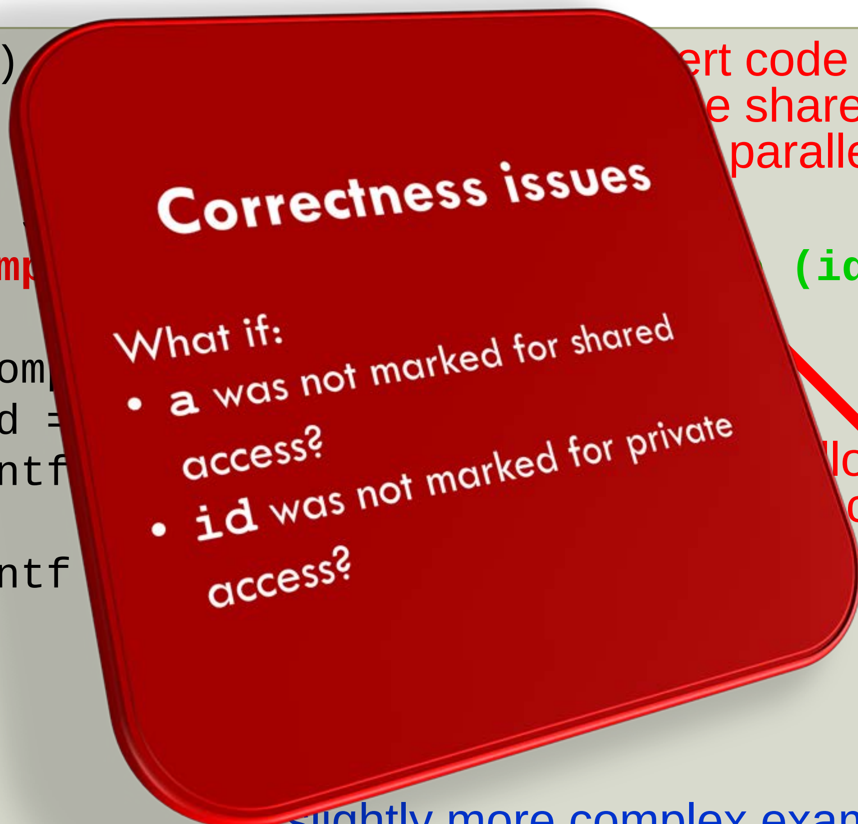
Data scope attributes

```
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

slightly more complex example



More data sharing clauses

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

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

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

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

```
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;
}
```

#pragma omp for

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

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

```
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;
}
```

#pragma omp for

..since lower and upper boundaries (**LB**, **UB**) are computed locally

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

```
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

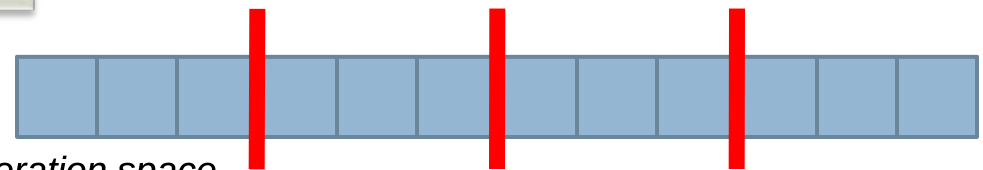
Es. 12 iterations (N), 4 threads (Nthr)

```
#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*

	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?

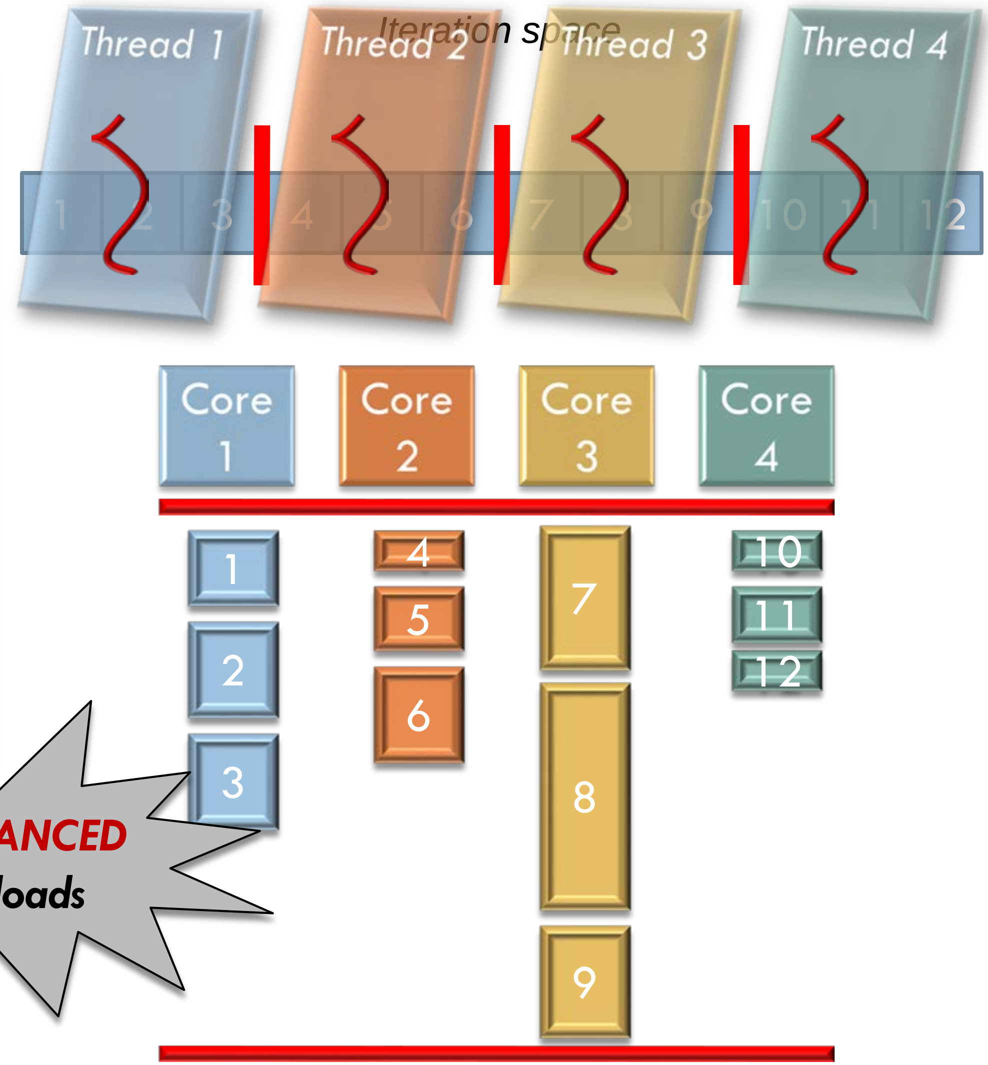
		0	1	2	3
LOWER BOU...		0	3	6	9
UPPER BOUND	...{[C * (TID + 1)], N}	3	6	9	12

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

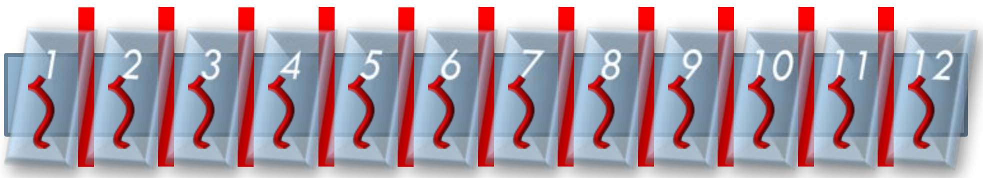
UNBALANCED workloads



The `schedule` clause

Dynamic Loop Partitioning

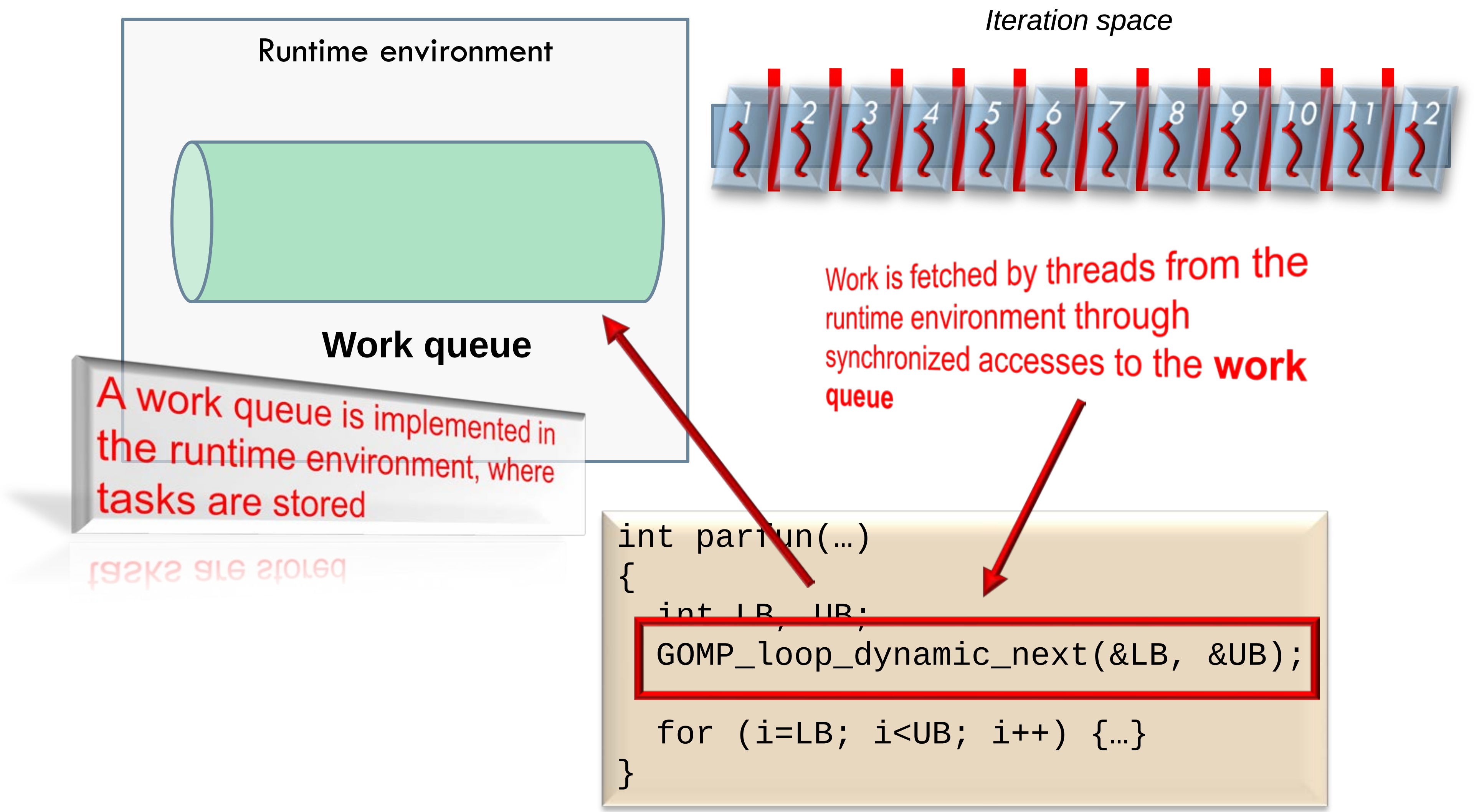
Iteration space



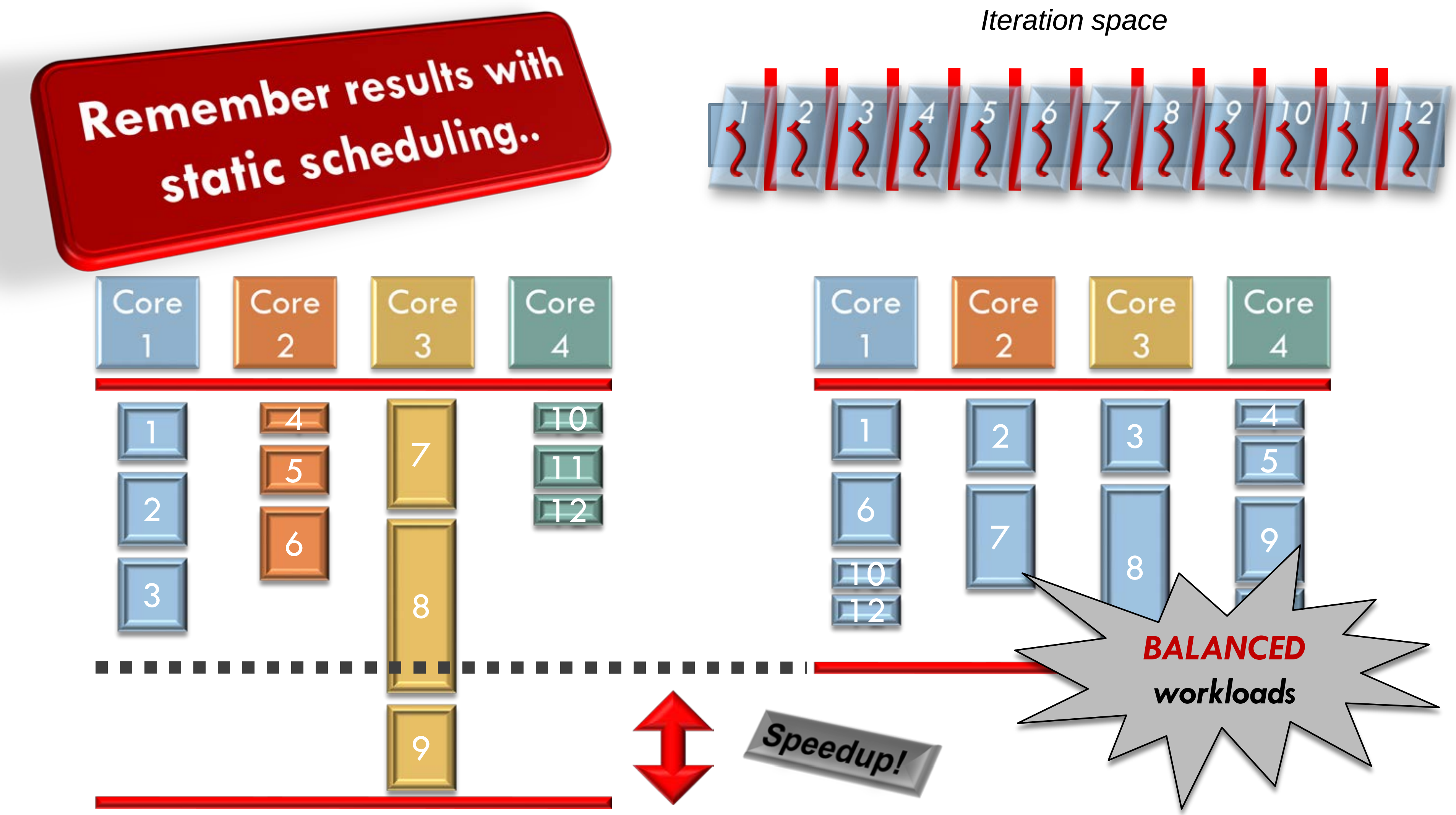

```
#pragma omp for schedule(dynamic)
  {
    for (i=0; i<12; i++)
      {
        int start = rand();
        int count = 0;

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

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

A thread is generated for every single iteration

The `schedule` clause
Dynamic Loop Partitioning



The `schedule` clause
Dynamic Loop Partitioning



Parallelization granularity

Iteration chunking

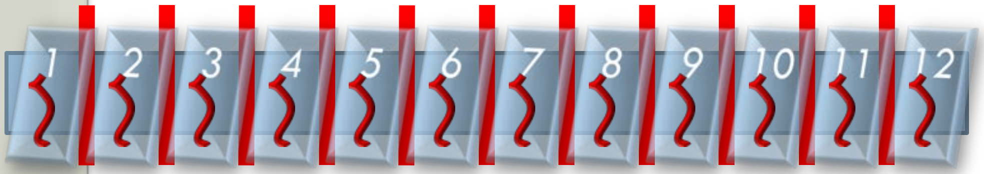
- Fine-grain Parallelism
  - Best opportunities for load balancing, but..
  - Small amounts of computational work between parallelism computation stages
  - Low computation to parallelization ratio ➔ High parallelization overhead
- Coarse-grain Parallelism
  - Harder to load balance efficiently, but..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

The `schedule` clause

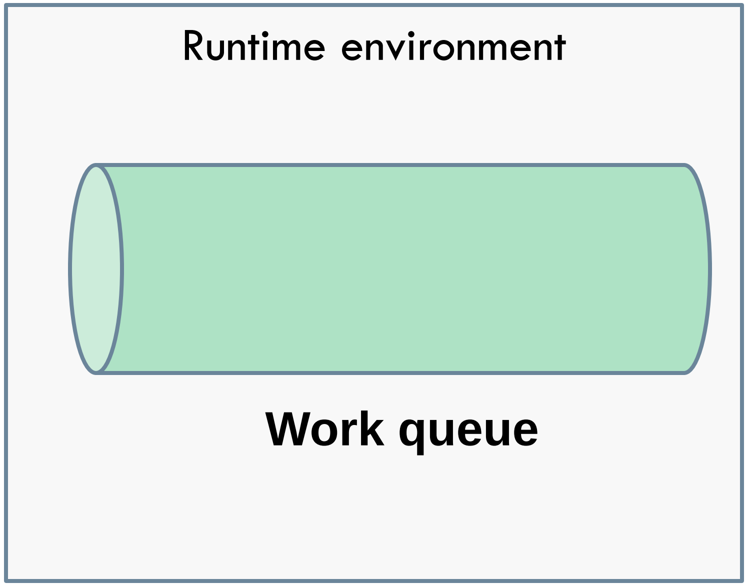
Dynamic Loop Partitioning

Iteration space

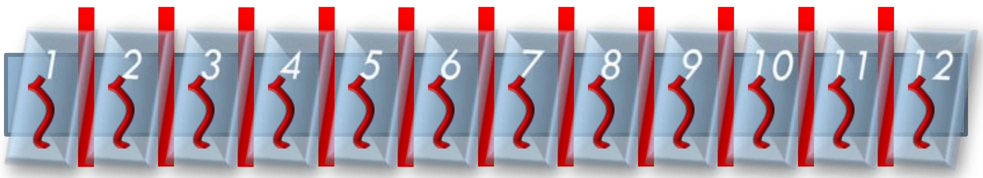
```
#pragma omp for schedule(dynamic, 1)
  {
    for (i=0; i<12; i++)
      {
        int start = rand();
        int count = 0;

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

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



The `schedule` clause
Dynamic Loop Partitioning



Iteration space



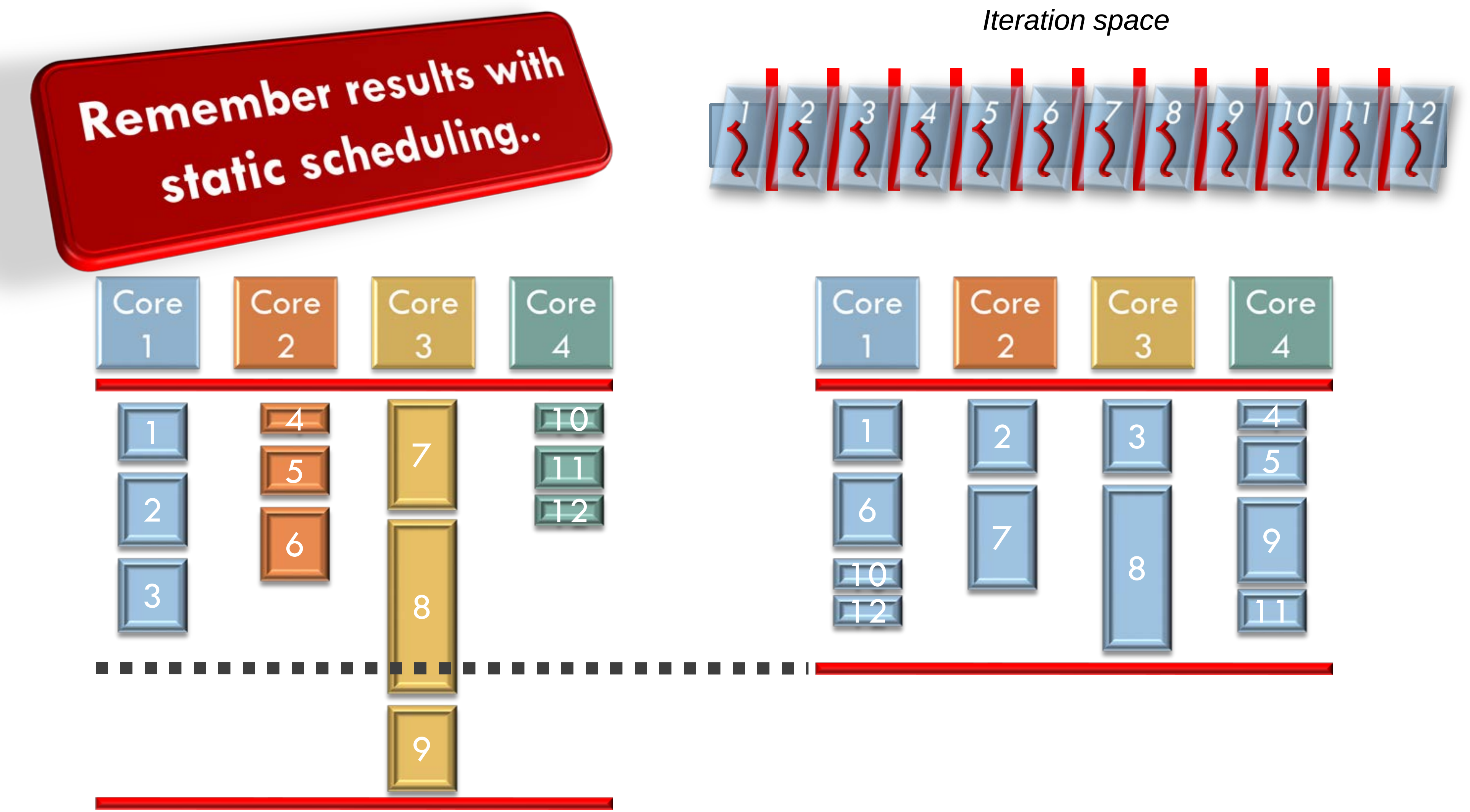
Runtime scheduling primitive is invoked at every iteration

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}
}
```

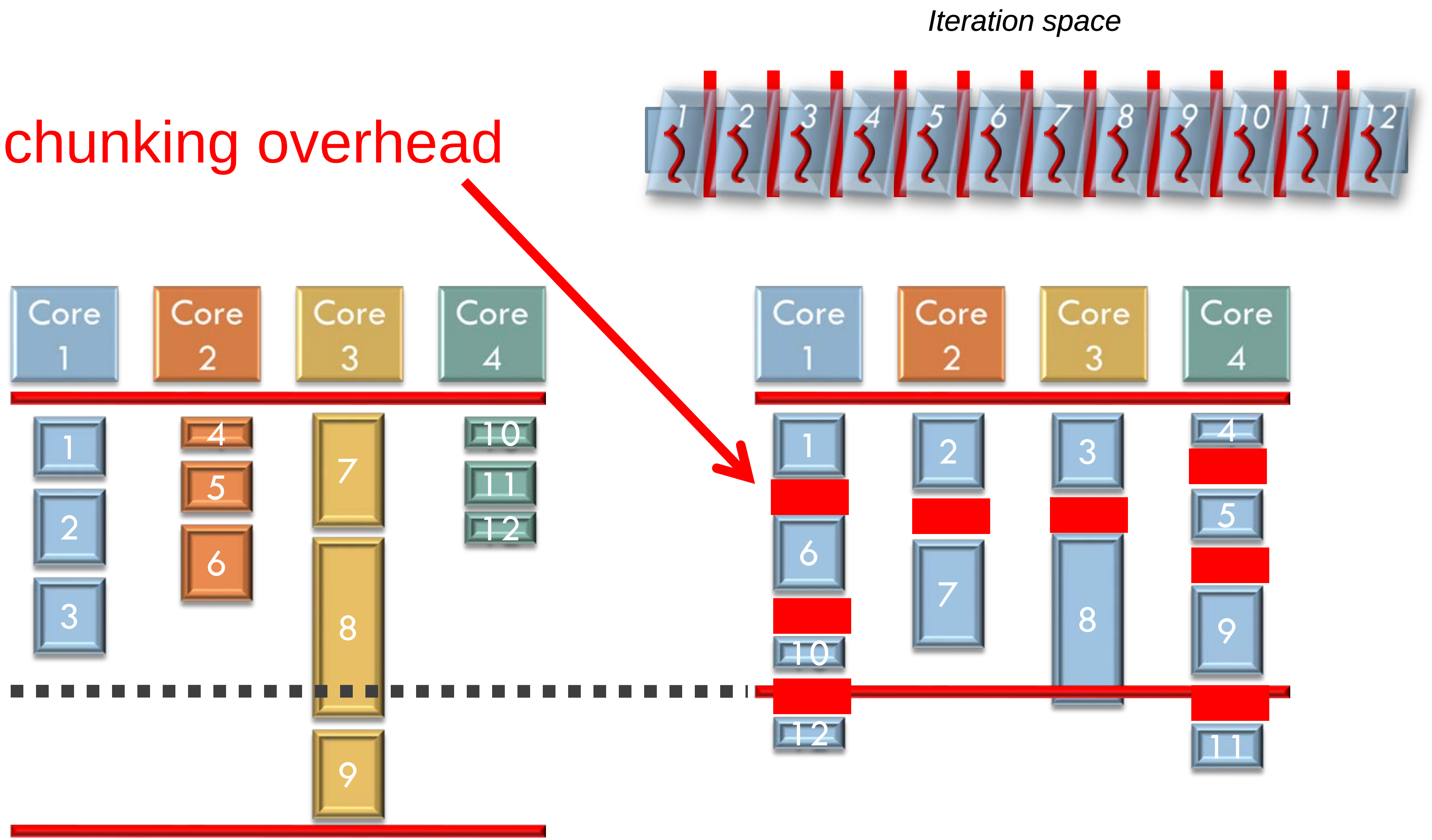
The `schedule` clause

Dynamic Loop Partitioning



The `schedule` clause

Dynamic Loop Partitioning

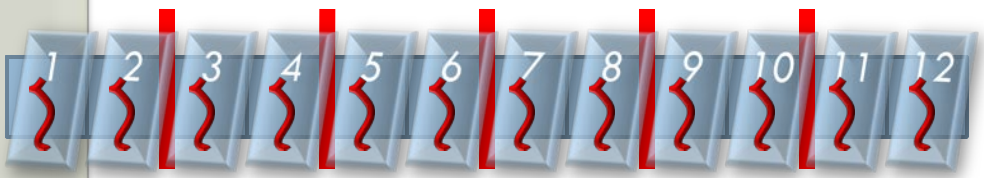


The `schedule` clause

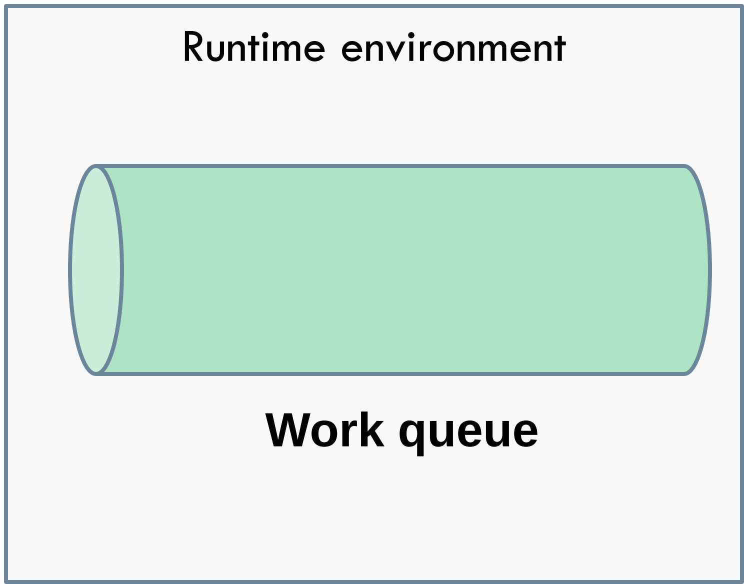
Dynamic Loop Partitioning

Iteration space

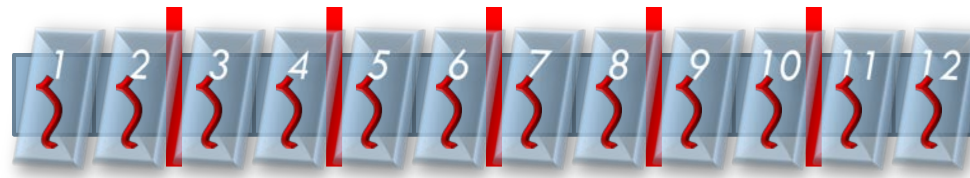
```
#pragma omp for schedule(dynamic, 2)
  {
    for (i=0; i<12; i++)
      {
        int start = rand();
        int count = 0;

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

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



The `schedule` clause
Dynamic Loop Partitioning



Iteration space

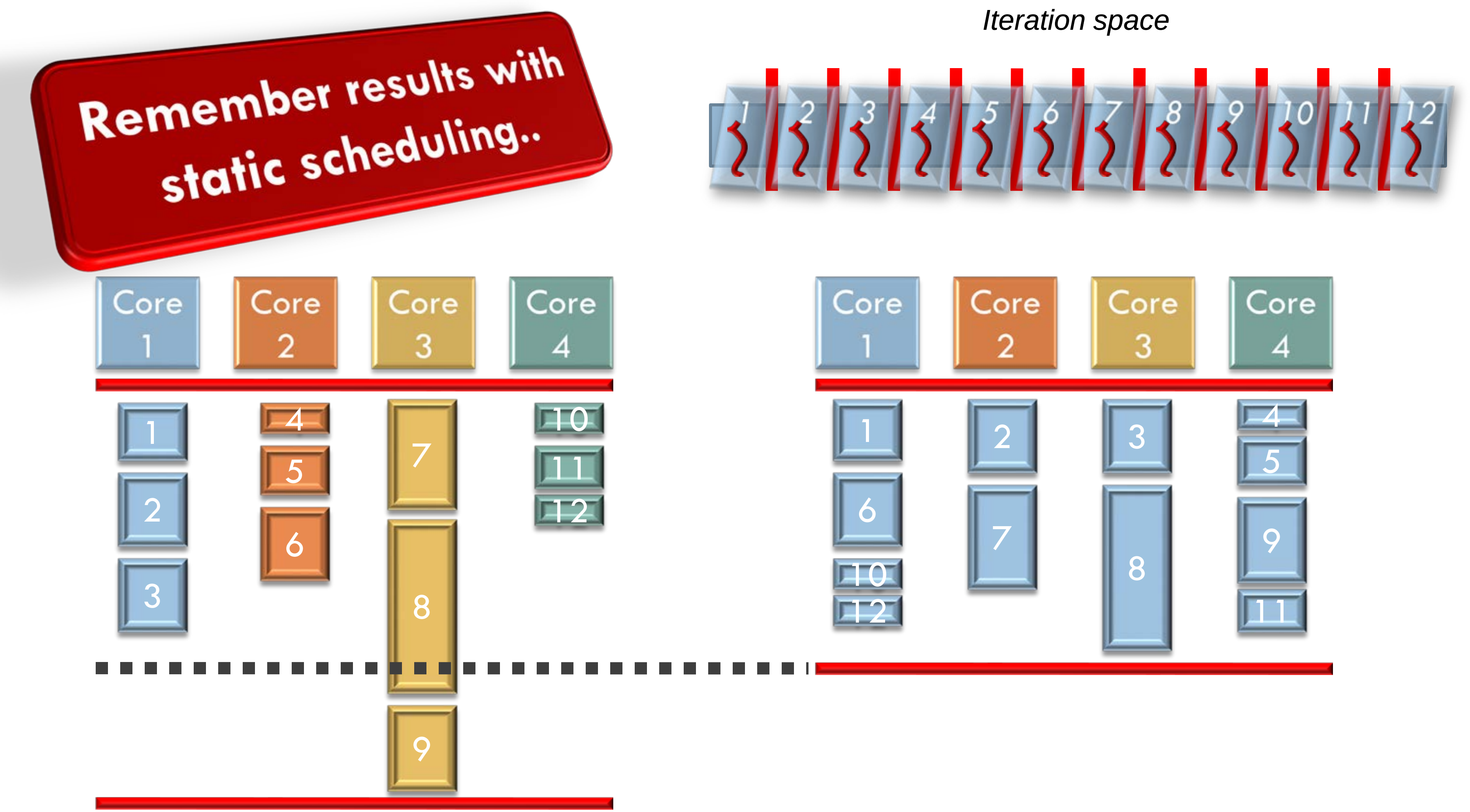


Runtime scheduling primitive is invoked every two iterations (half the times of previous case)

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}
}
```

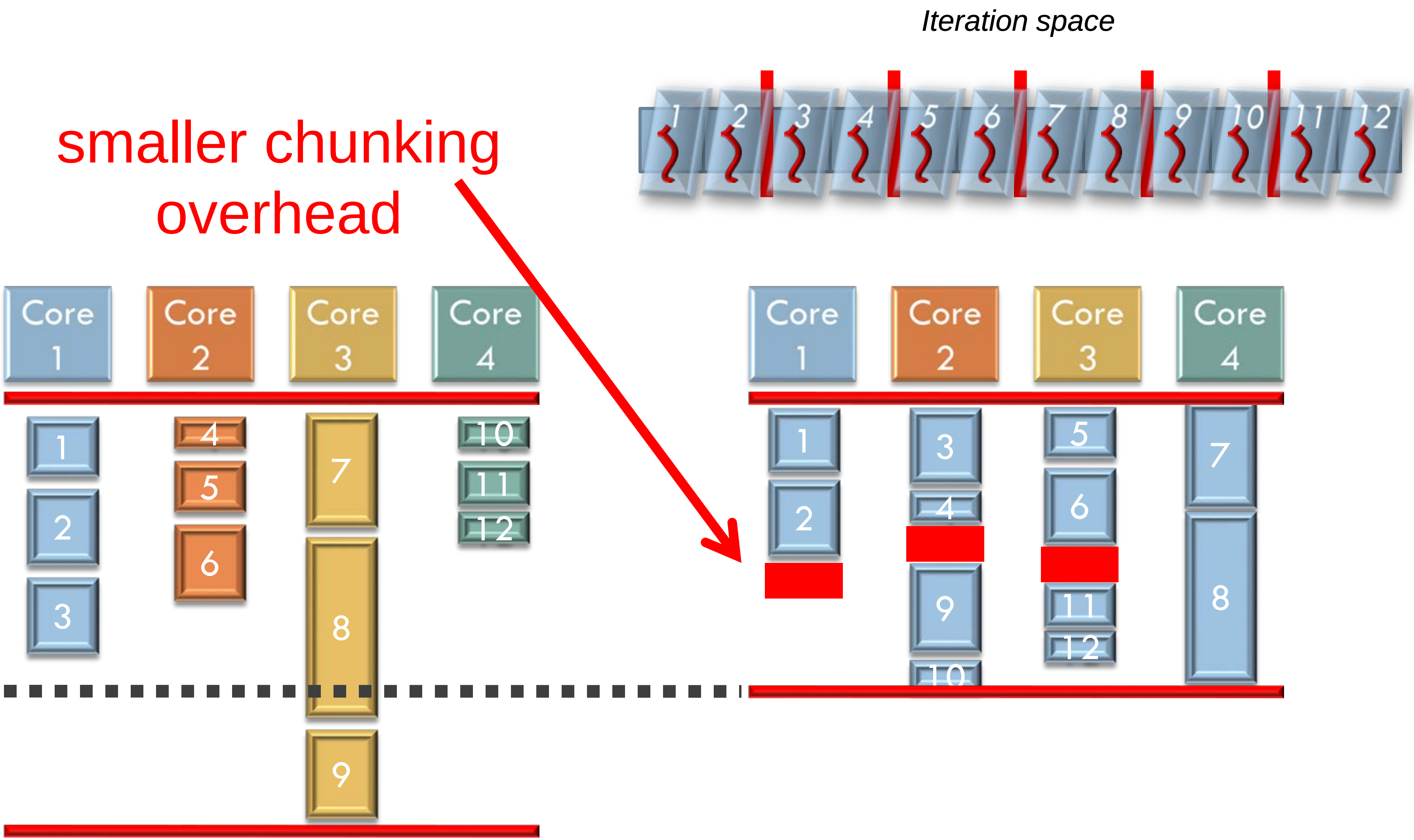
The `schedule` clause
Dynamic Loop Partitioning



The `schedule` clause

Dynamic Loop Partitioning

smaller chunking overhead



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)

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
```

SPMD VS MPMD

Recall..

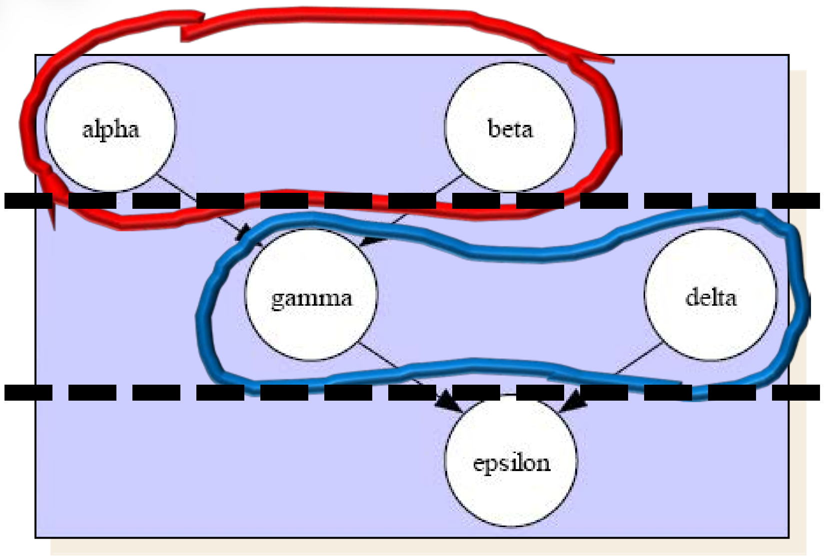
- SPMD (single program, multiple data)
  - Processors execute the same stream of instructions over different data
  - `#pragma omp for`
- MPMD (multiple program, multiple data)
  - Processors execute different streams of instructions over (possibly) different data
  - `#pragma omp sections`
  - `#pragma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

Task Parallelism Example

```
int main()
{


      v = alpha();

      w = beta ();



      y = delta ();

      x = gamma (v, w);

      z = epsilon (x, y));


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

FIRST SOLUTION

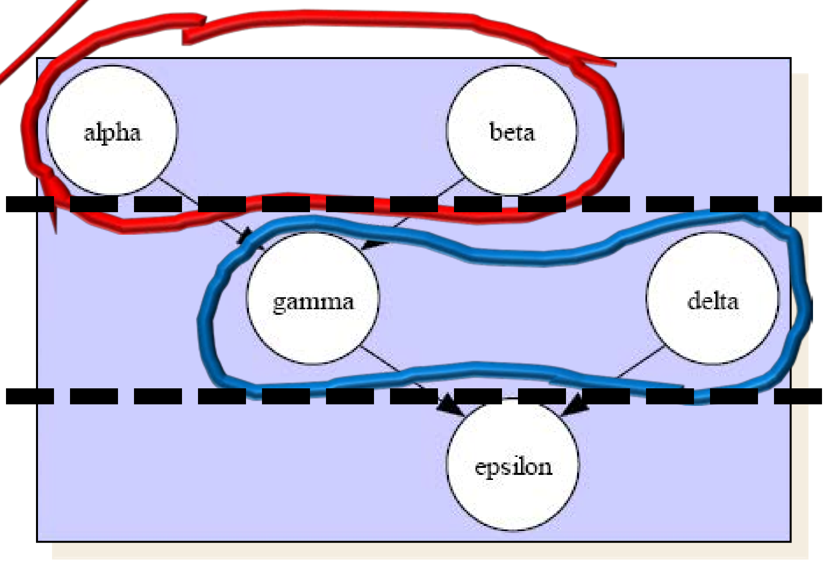
Identify independent nodes in the task graph, and outline parallel computation with the **sections** directive



Task Parallelism Example

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

      v = alpha();

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

      y = delta ();

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


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

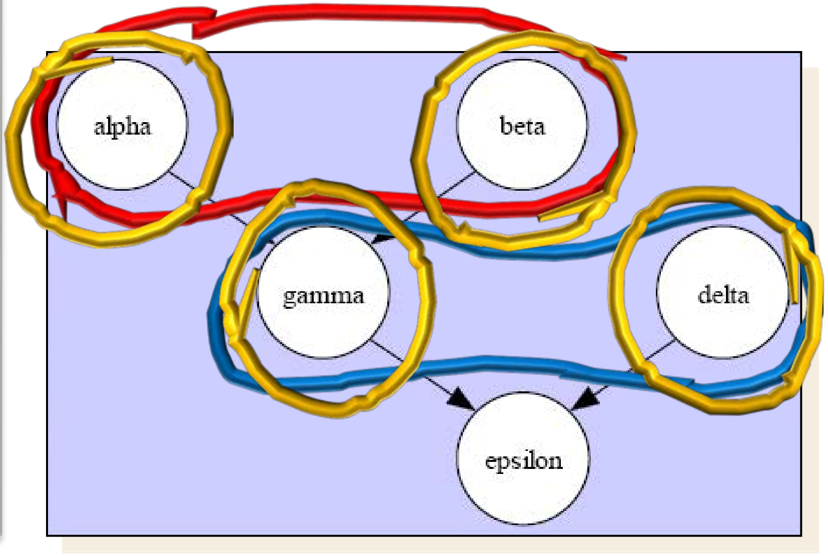
Barriers implied here!



Task Parallelism Example

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

      v = alpha();

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

      y = delta ();

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


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

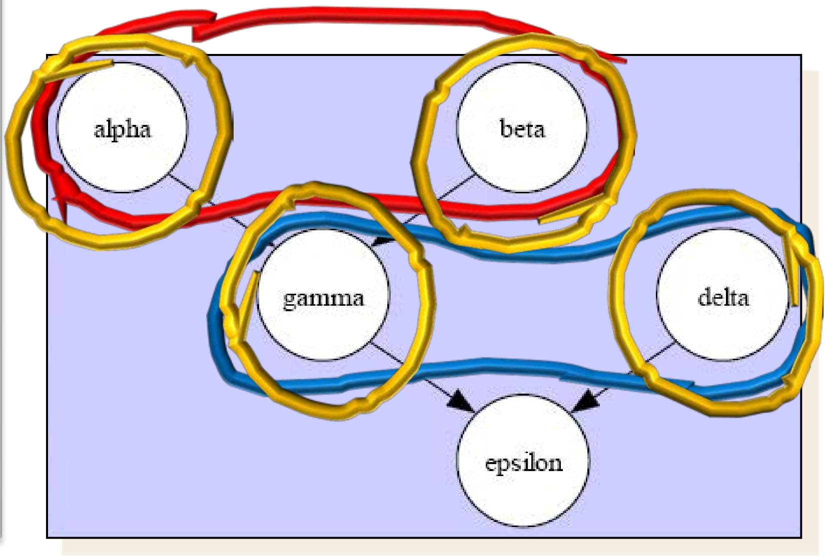
Each parallel task within a **sections** block identifies a **section**



Task Parallelism Example

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

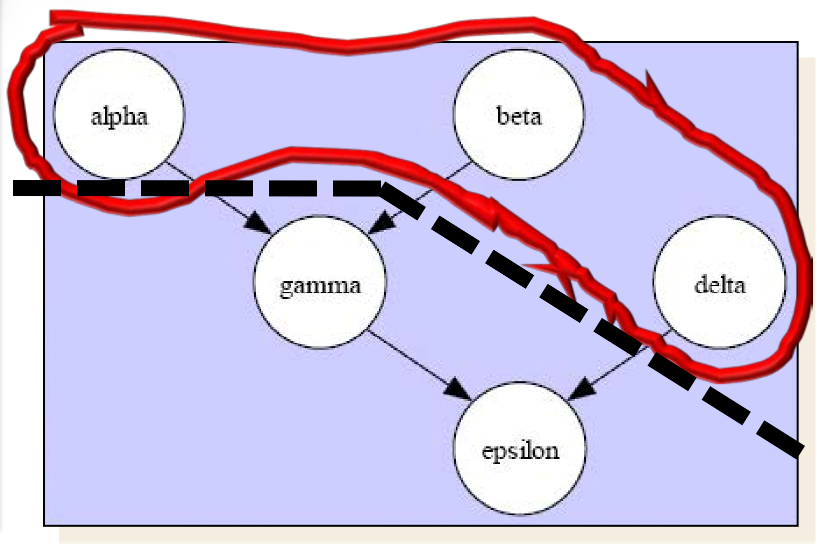


Task Parallelism Example

```
int main()
{

      v = alpha();

      w = beta ();

      y = delta ();


      x = gamma (v, w);

      z = epsilon (x, y));


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

SECOND SOLUTION

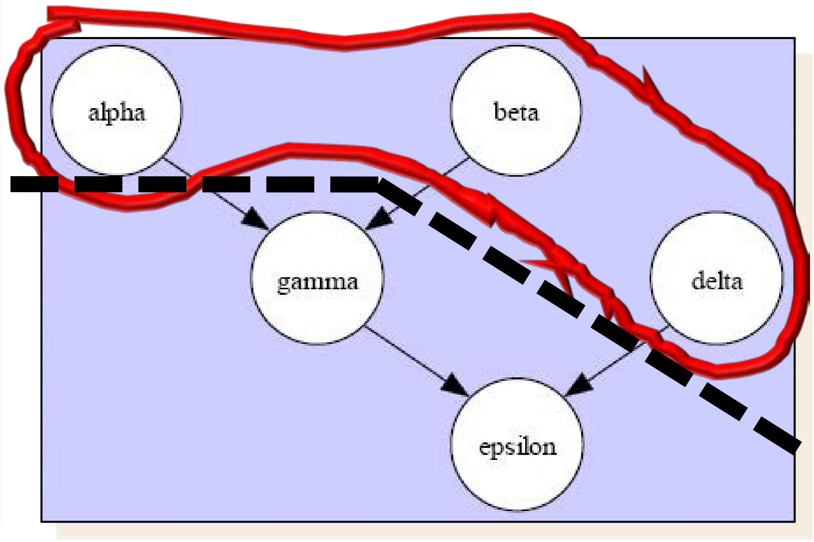
Identify independent nodes in the task graph, and outline parallel computation with the **sections** directive



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);
}
```

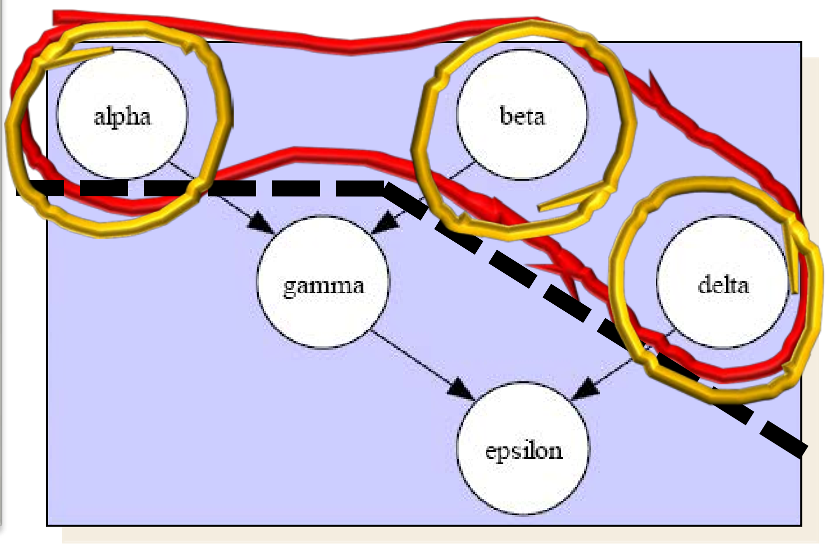
Barrier implied here!



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);
}
```

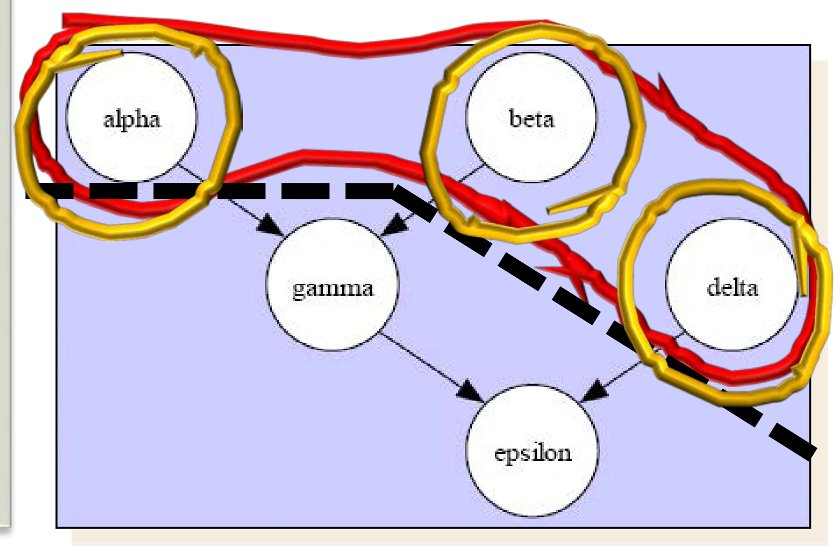
Each parallel task within a **sections** block identifies a **section**



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**



#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)

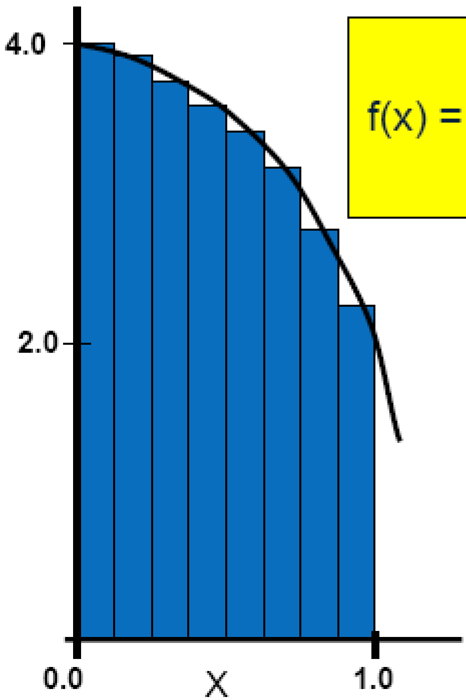
#pragma omp critical

- 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



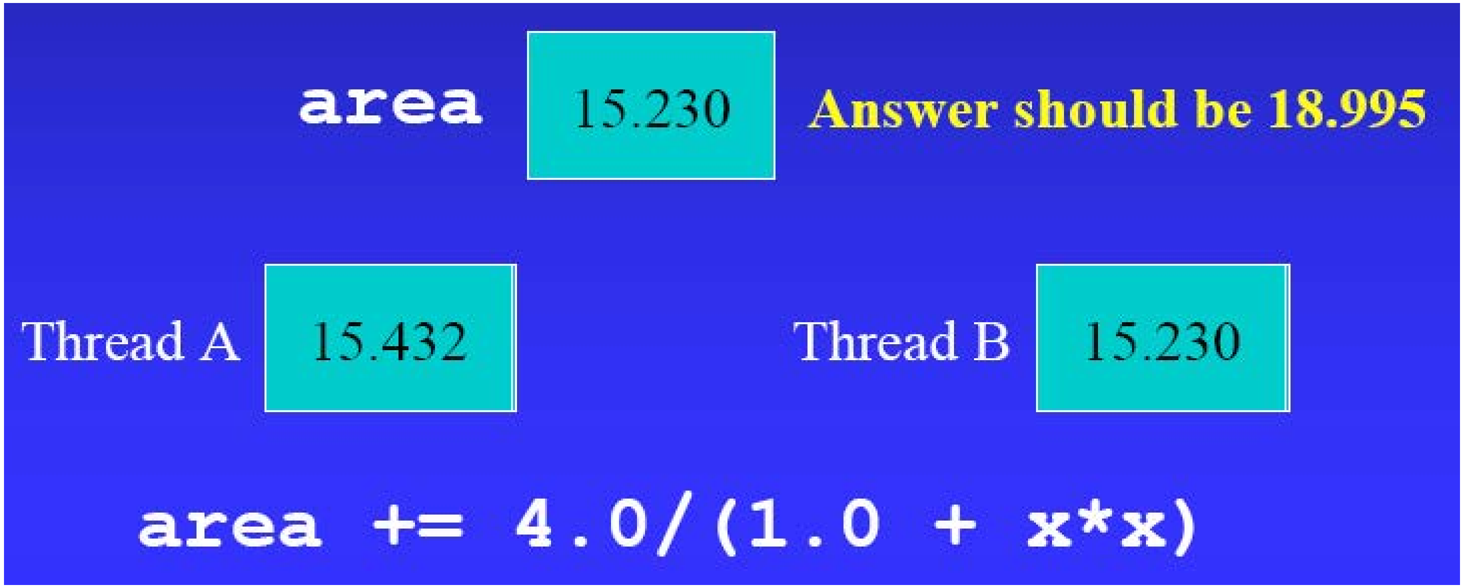
$$f(x) = \frac{4.0}{(1+x^2)}$$

```
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

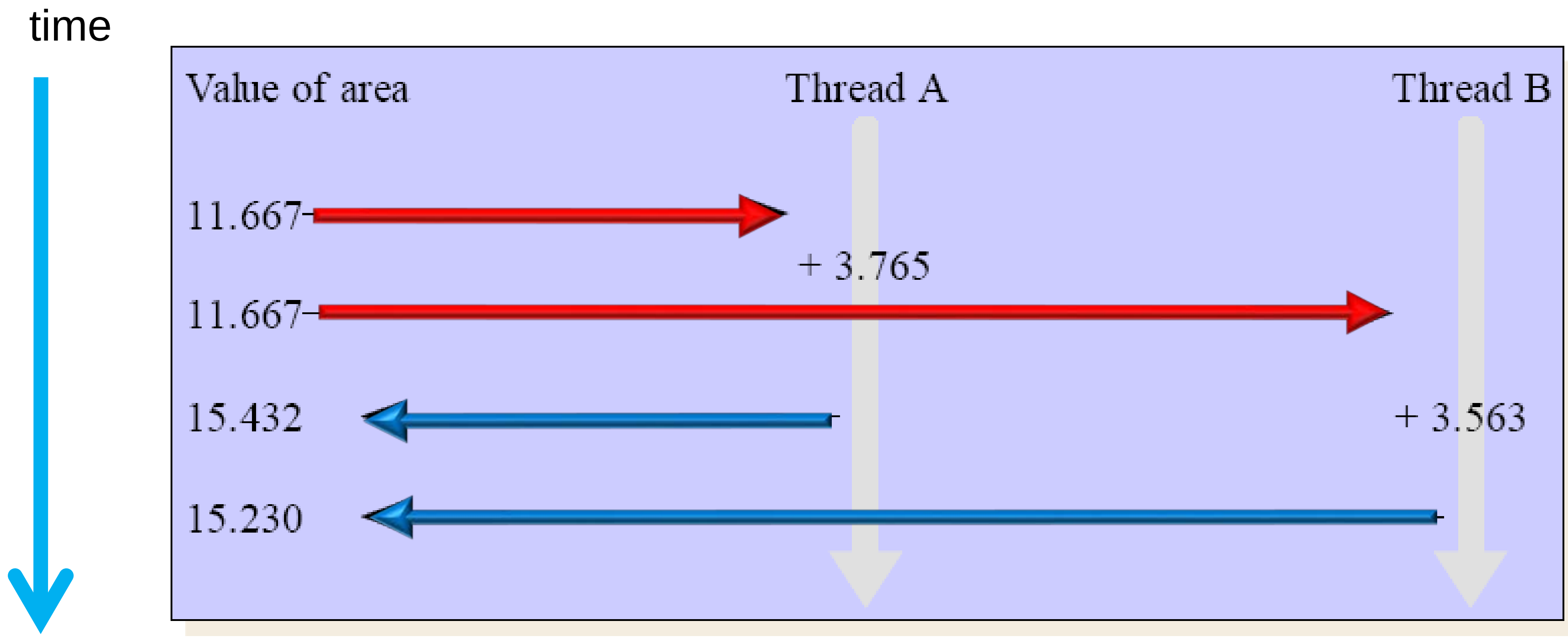
Race condition

- Ensure atomic updates of the shared variable **area** to avoid a race condition in which one process may "race ahead" of another and ignore changes



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”*



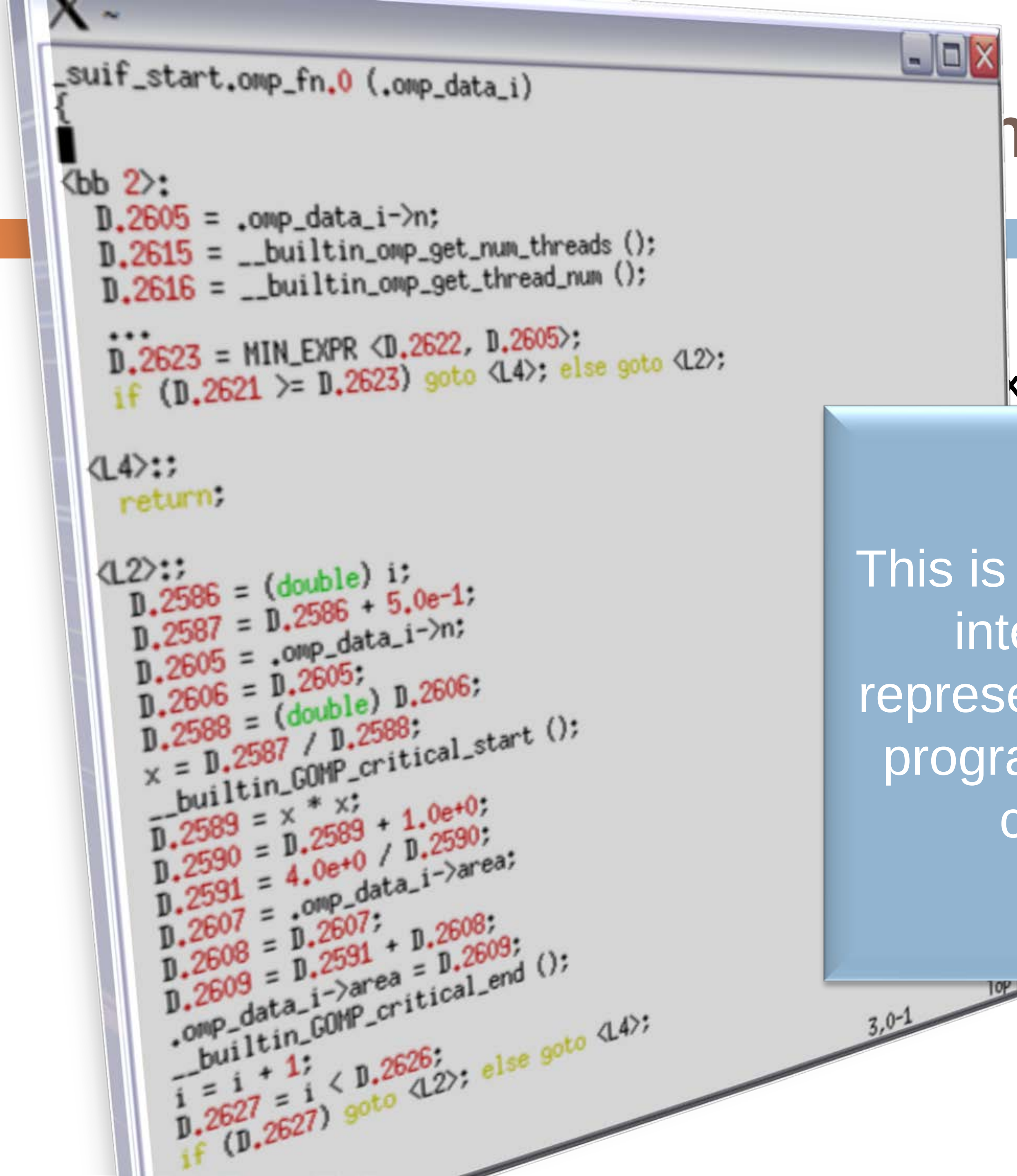
π-finding code example

```
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

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



mance!

xecution sequential

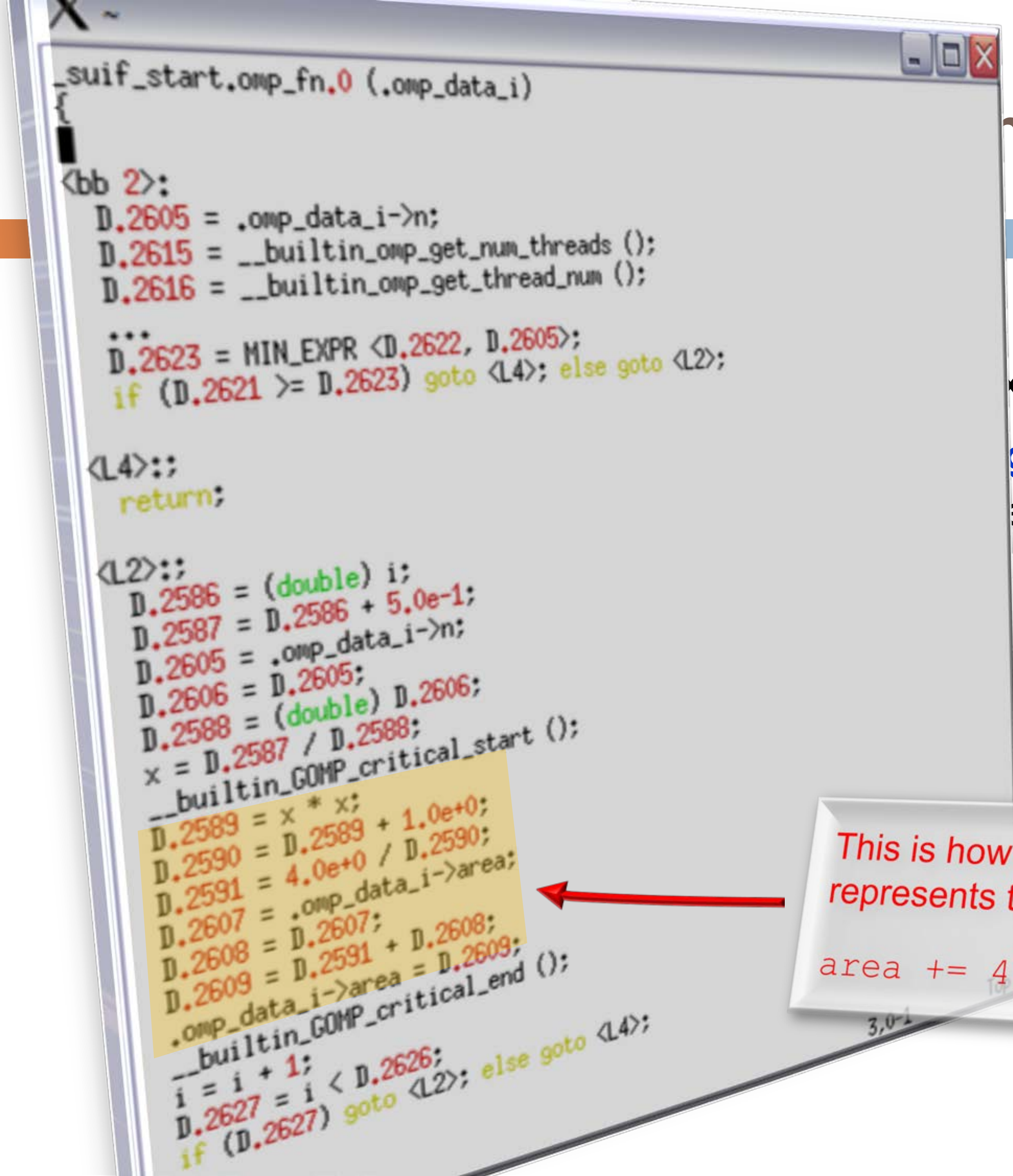
.e.

ne

This is a dump of the intermediate representation of the program within the compiler

mance!

- xecution sequential
- grained locking (i.e.
e)
- of area in the
/ more simpler
- l granularity of the

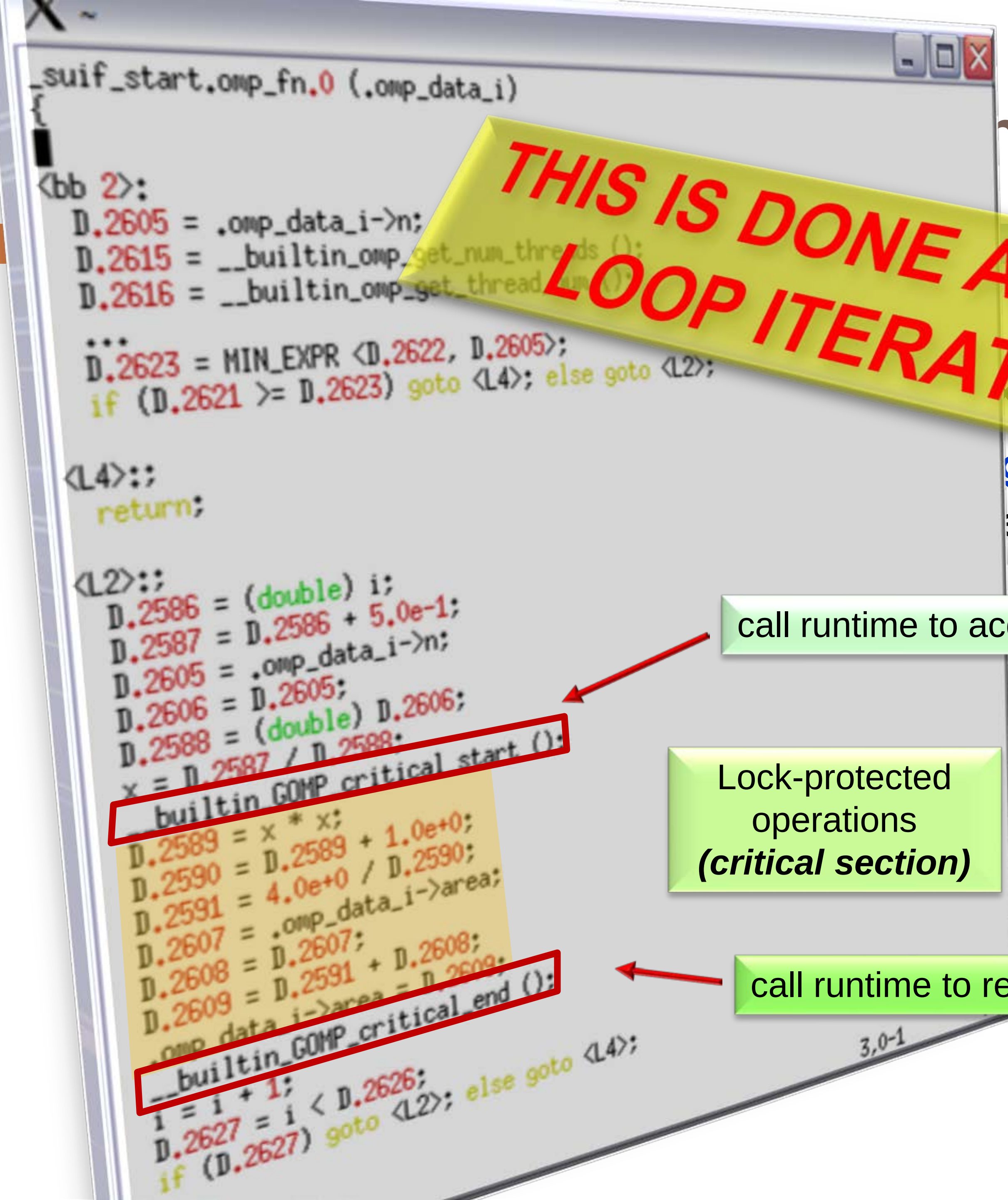


This is how the compiler represents the instruction

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

…mance!

- …xecution sequential
- …grained locking (i.e. …e)
- …of area in the … more simpler
- …l granularity of the



THIS IS DONE AT EVERY LOOP ITERATION!

call runtime to acquire lock

Lock-protected operations
(critical section)

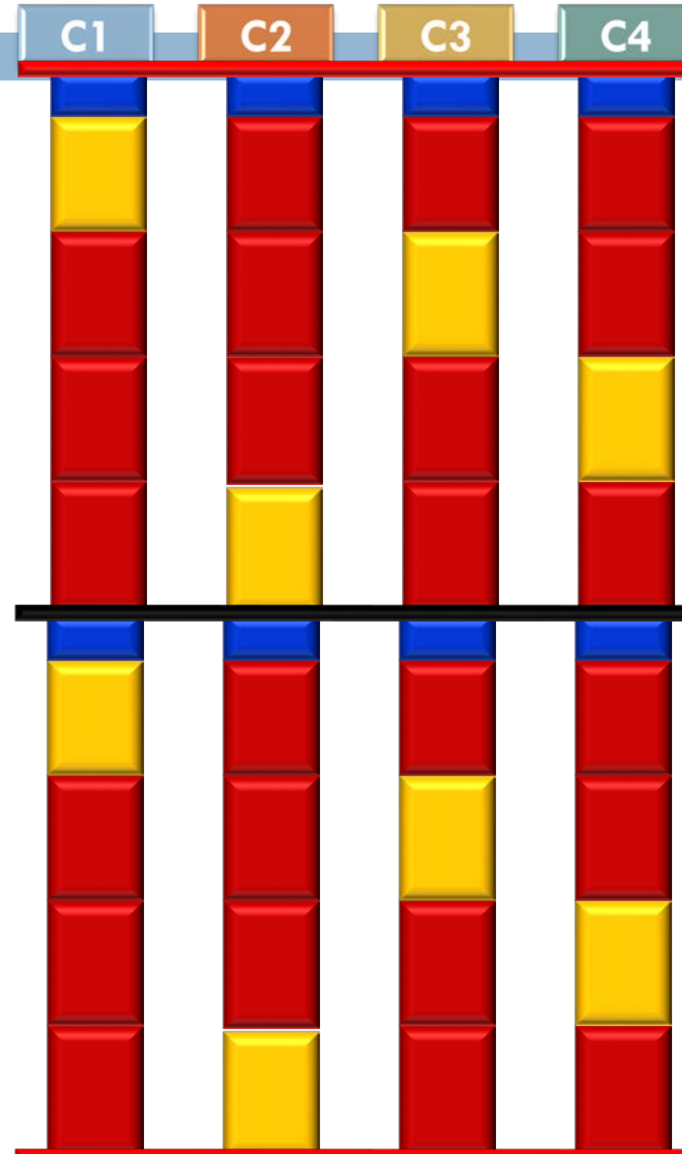
call runtime to release lock

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

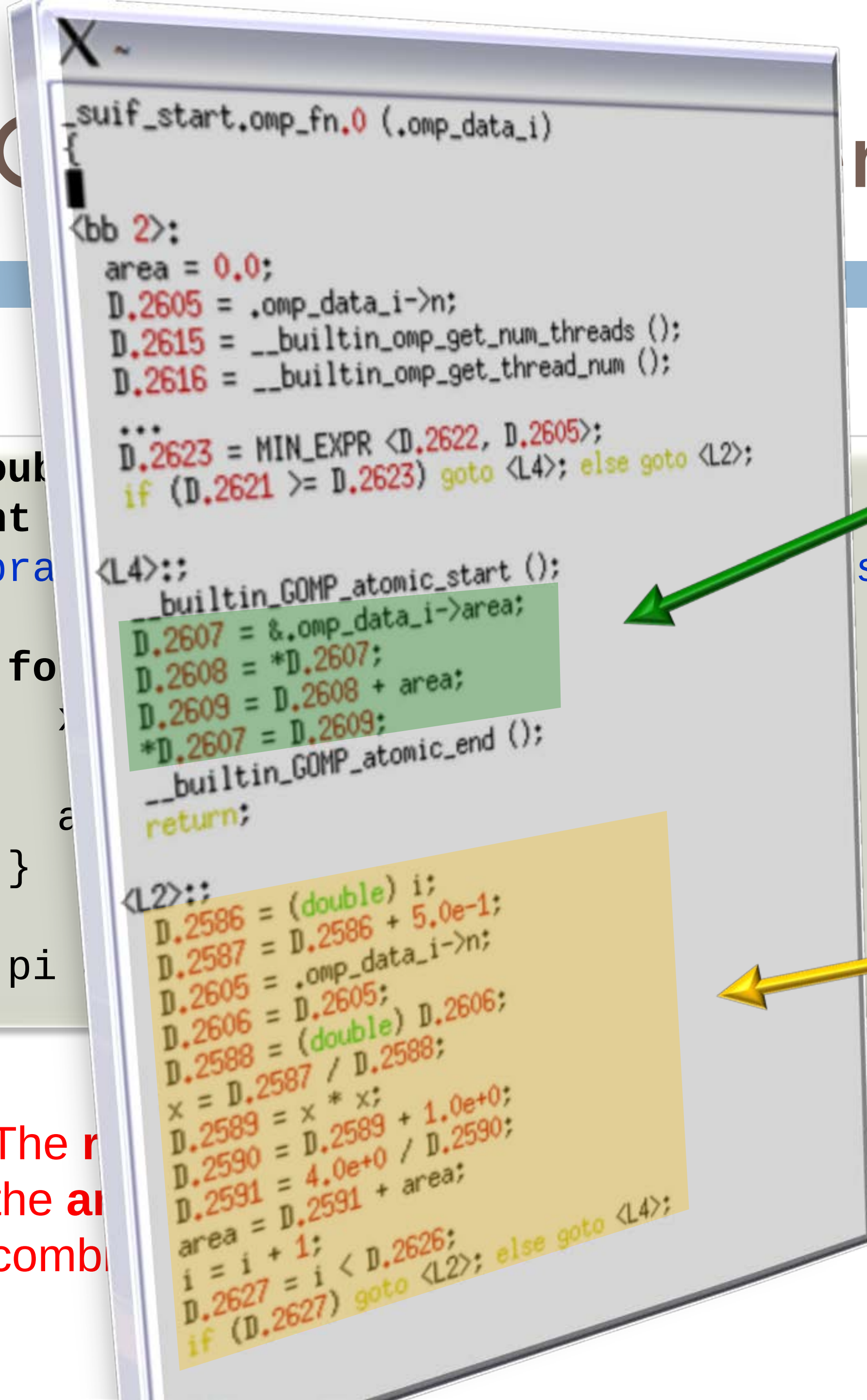
Correctness, not performance!

```
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

G rformance!

```
doub
int
#pra                                  shared(area) reduction(+:area)
{
   fo
      x

      a
   }
}
   pi
```

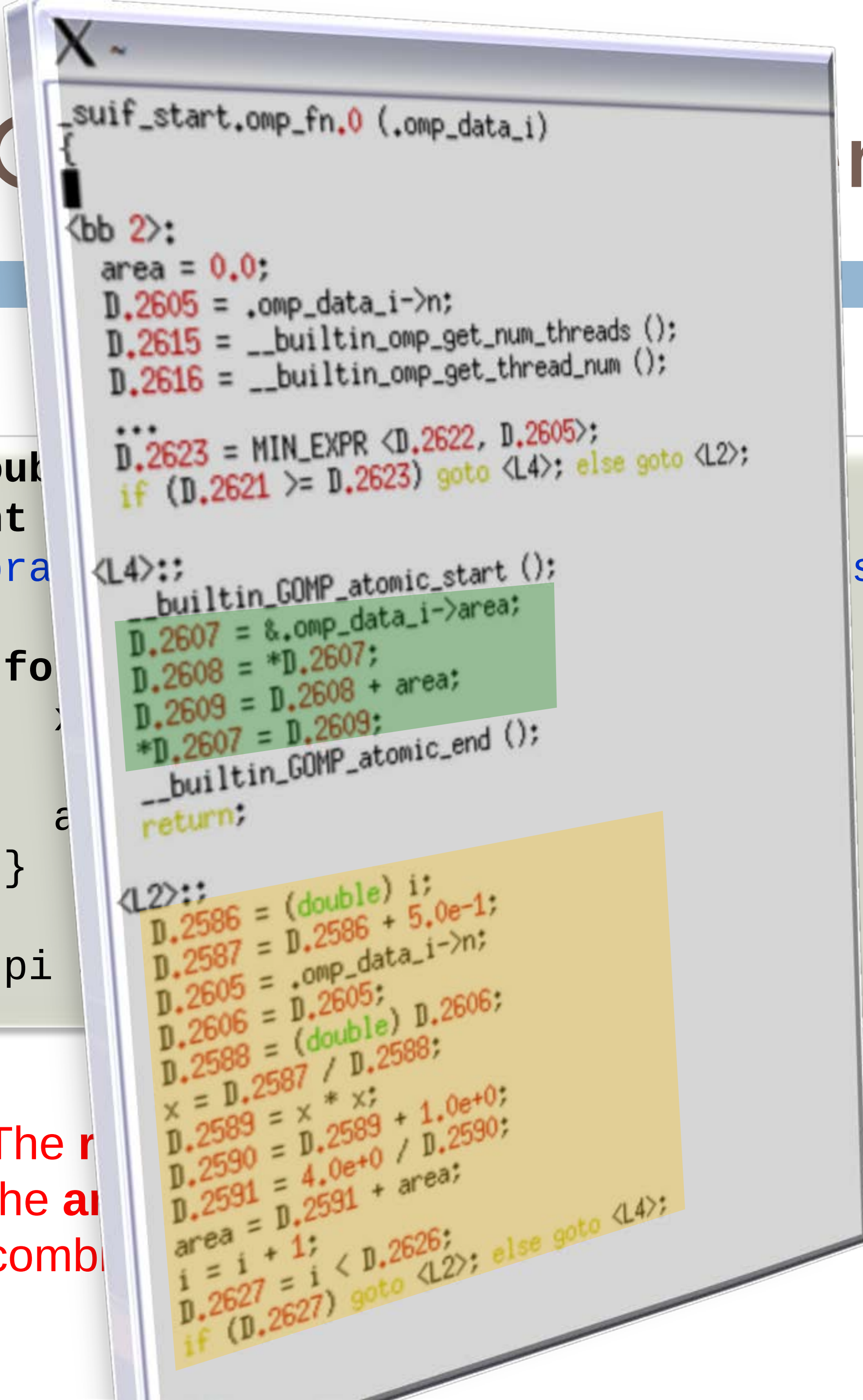


Shared variable is only updated at the end of the loop, when partial sums are computed

Each thread computes partial sums on private copies of the reduction variable

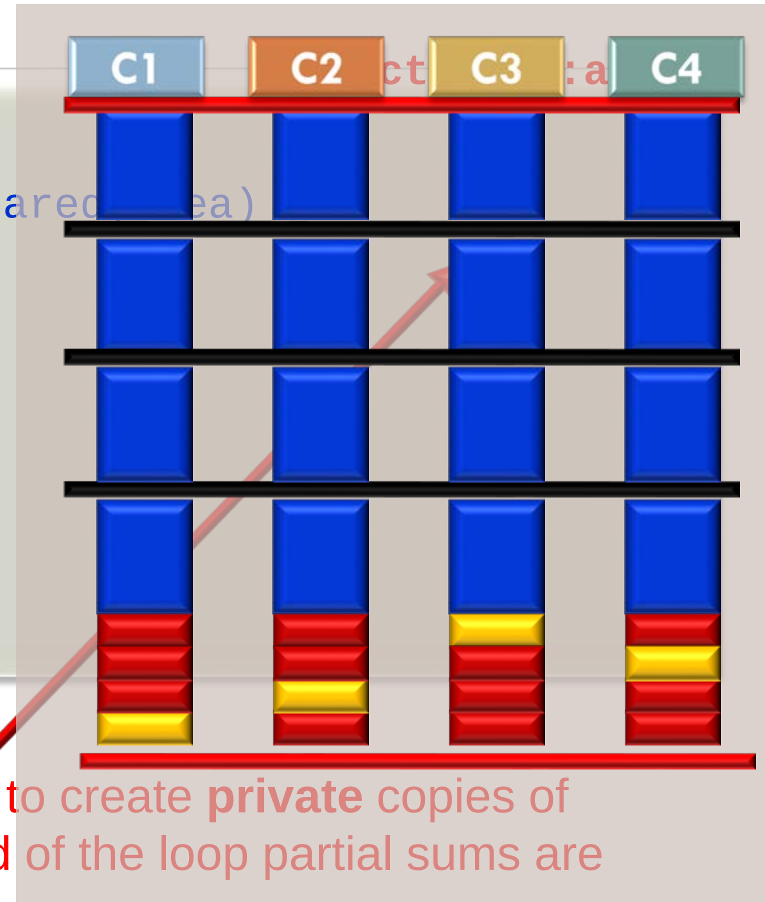
The r… er to create **private** copies of the ar… nd of the loop partial sums are combi…

C … rformance!



```
_suif_start.omp_fn.0 (.omp_data_i)
{

<bb 2>:
  area = 0.0;
  D.2605 = .omp_data_i->n;
  D.2615 = __builtin_omp_get_num_threads ();
  D.2616 = __builtin_omp_get_thread_num ();
  ...
  D.2623 = MIN_EXPR <D.2622, D.2605>;
  if (D.2621 >= D.2623) goto <L4>; else goto <L2>;

<L4>:;
  __builtin_GOMP_atomic_start ();
  D.2607 = &.omp_data_i->area;
  D.2608 = *D.2607;
  D.2609 = D.2608 + area;
  *D.2607 = D.2609;
  __builtin_GOMP_atomic_end ();
  return;

<L2>:;
  D.2586 = (double) i;
  D.2587 = D.2586 + 5.0e-1;
  D.2605 = .omp_data_i->n;
  D.2606 = D.2605;
  D.2588 = (double) D.2606;
  x = D.2587 / D.2588;
  D.2589 = x * x;
  D.2590 = D.2589 + 1.0e+0;
  D.2591 = 4.0e+0 / D.2590;
  area = D.2591 + area;
  i = i + 1;
  D.2627 = i < D.2626;
  if (D.2627) goto <L2>; else goto <L4>;
```

```
doub
int
#pra                                   shared(area)
{
  fo
    )

    a
  }
}
  pi
```



The r… er to create **private** copies of the ar… nd of the loop partial sums are combi…

More worksharing constructs

The `master` directive

- The **`master`** construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel
{
  do_many_things();
  #pragma omp master
  { exchange_boundaries(); }

  #pragma omp barrier
  do_many_other_things();
}
```

The `single` directive

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a **nowait** clause).

```
#pragma omp parallel
{
   do_many_things();
   #pragma omp single
   { exchange_boundaries(); }

   #pragma omp barrier
   do_many_other_things();
}
```

Recap: TASK parallelism in OpenMP 2.5
The `sections` directive

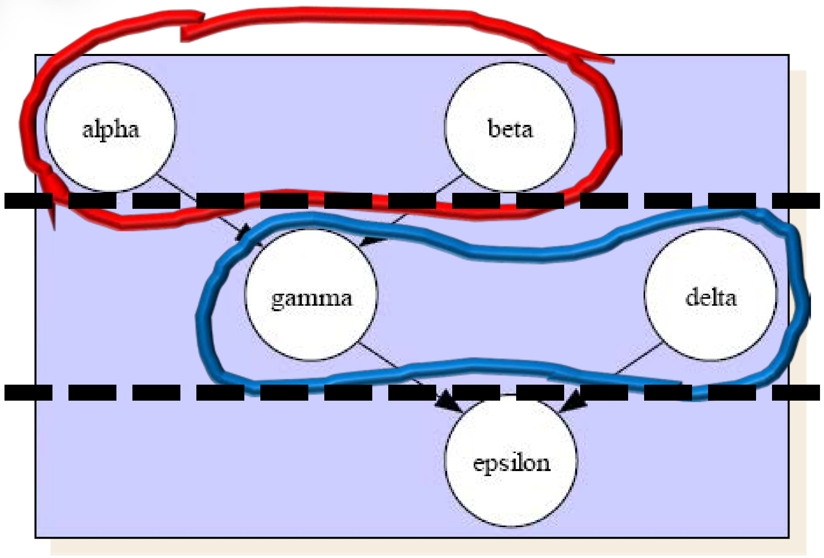
- The **for** pragma allows to exploit data parallelism in loops
- OpenMP 2.5 also provides a directive to exploit task parallelism

```
#pragma omp sections
```

Task Parallelism Example

```c
int main()
{

    v = alpha();

    w = beta ();


    y = delta ();

    x = gamma (v, w);

    z = epsilon (x, y));


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

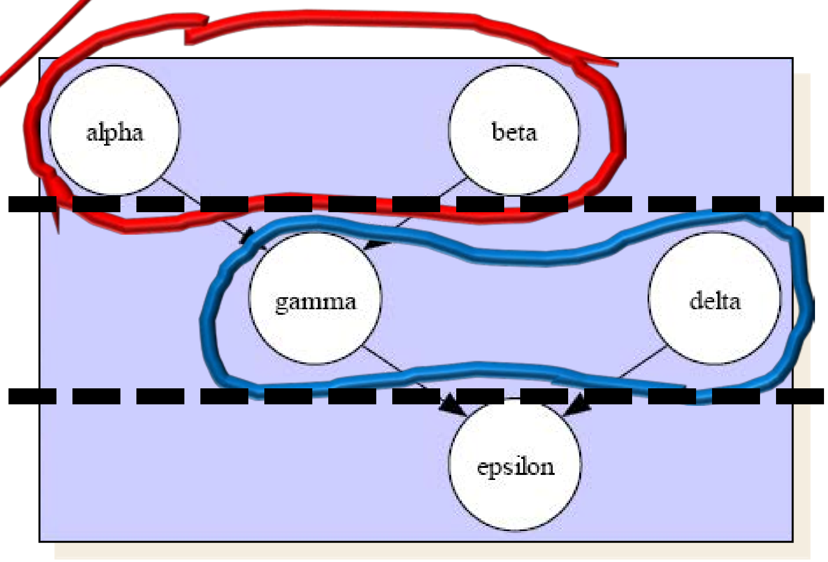
Identify independent nodes in the task graph, and outline parallel computation with the **sections** directive



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);
}
```

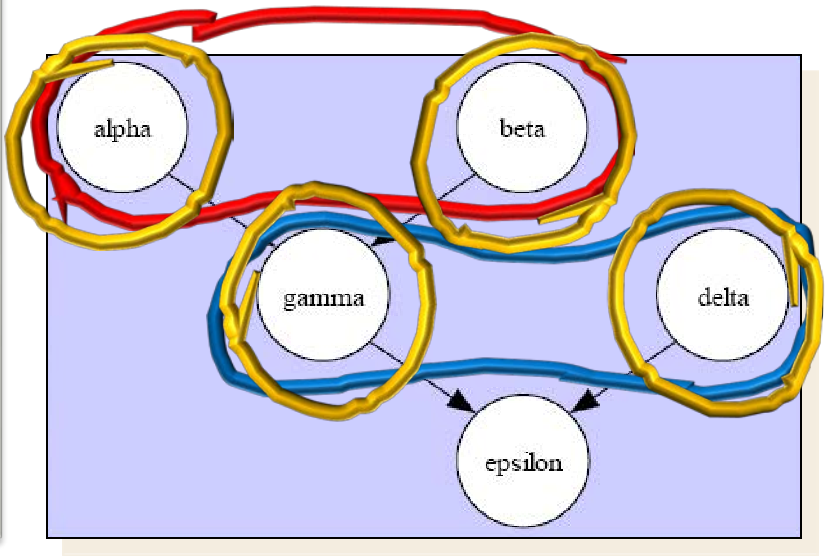
Barriers implied here!



Task Parallelism Example

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

      v = alpha();

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

      y = delta ();

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


  printf (“%f\n”, z);
}
```

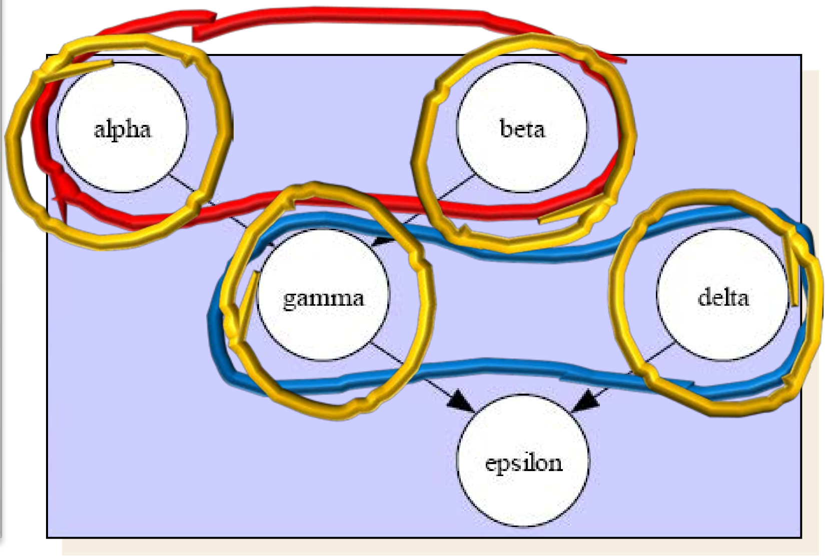
Each parallel task within a **sections** block identifies a **section**



Task Parallelism Example

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

- 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?

Task parallelism

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

OpenMP v2.5

- **Awkward!**
- **Poor performance**
- **Not composable**

EXAMPLE

```
void traverse_list (List l)
{
  Element e ;
#pragma omp parallel private ( e )
  for ( e = e→first; e; e = e→next )
#pragma omp single nowait
    process ( e ) ;
}
```

Task parallelism

- Why?
  - Example: **tree traversal**

EXAMPLE

```
void traverse_tree (Tree *tree)
{
#pragma omp parallel sections
{
#pragma omp section
  if ( tree→left )
    traverse_tree ( tree→left );
#pragma omp section
  if ( tree→right)
    traverse_tree ( tree→right);
}
  process (tree);
}
```

OpenMP v2.5

- **Too many parallel regions**
  - **Extra overheads**
  - **Extra synchronizations**
  - **Not always well supported**

Task parallelism

- Better solution for those problems
- Main addition to OpenMP 3.0a
- Allows to parallelize irregular problems
  - unbounded loops
  - recursive algorithms
  - producer/consumer schemes
  - ...

Ayguadé et al., The Design of OpenMP Tasks, IEEE TPDS March 2009

Task parallelism

- The OpenMP tasking model
  - Creating tasks
  - Data scoping
  - Syncronizing tasks
  - Execution model

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

List traversal with tasks

EXAMPLE

```
int a ;
void foo ( ) {
int b , c ;
#pragma omp parallel shared(c)
{
  int d ;
#pragma omp task
  {
    int e ;
    a = shared
    b = firstprivate
    c = shared
    d = firstprivate
    e = private
} } }
```

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 execution model

- Task are executed by a thread of the **team** that generated it
  - Can be executed **immediately** by the same thread that creates it
- Parallel regions in 3.0 create tasks!
  - One **implicit** task is created for each thread
    - So all task-concepts have sense inside the parallel region
- Threads can **suspend** the execution of a task and **start/resume** another

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 <u>parallelize task creation among threads</u>

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 ) ;
}
```

Multiple traversals

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

Task scheduling

How it works?

- Tasks are **tied** by default
  - Tied tasks are executed always by the same thread
  - Tied tasks have scheduling restrictions
    - Deterministic scheduling points (creation, synchronization, ...)
    - Another constraint to avoid deadlock problems
  - Tied tasks may run into performance problems
- Programmer can use **untied** clause to lift all restrictions
  - **Note**: Mix very **carefully** with threadprivate, critical and thread-ids

And last..

The IF clause

- If the expression of a `if` clause is **false**
  - The encountering task is suspended
  - The new task is **executed immediately**
    - with its own data environment
    - different task with respect to synchronization
  - The parent task resumes when the task finishes
  - Allows implementations to **optimize** task creation

```
#pragma omp task if (...)
    process ( e ) ;
```

EXERCISE!