UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

Prof. Marco Bertini

Shared memory: OpenMP C/C++ directives

shared / private variables

- Common clauses shared by several directives address how variables are shared/made private within threads:

- `private(list of variables)` - a new version of the original variable with the same type and size is created in the memory of each thread belonging to the parallel region

- `shared(list of variables)` - threads of the team access and modify the same original variable in the shared memory.

- `default(shared | none)` - used to specify whether variables in a parallel region are shared or private by default. The first option causes all variables referenced in the construct to be shared except the private variables which are specified explicitly, while none requires each variable in the construct to be specified explicitly as shared or private.

shared / private variables

- Common clauses shared by several directives address how variables are shared/made private within threads:

`private` variables **are not initialised**, i.e. they start with random values like any other local automatic variable (and they are often implemented using automatic variables on the stack of each thread).

- `default(shared | none)` - used to specify whether variables in a parallel region are shared or private by default. The first option causes all variables referenced in the construct to be shared except the private variables which are specified explicitly, while none requires each variable in the construct to be specified explicitly as shared or private.

shared / private variables

- Common clauses shared by several directives address how variables are shared/made private within threads:
- `private(list of variables)` - a new version of the original variable with the same type and size is created in the memory of each thread belonging to the parallel region
- `shared(list of variables)` - threads of the team access and modify the same original variable in the shared memory.
- `default(shared | none)` - used to specify whether variables in a parallel region are shared or private by default. The first option causes all variables referenced in the construct to be shared except the private variables which are specified explicitly, while none requires each variable in the construct to be specified explicitly as shared or private.

It's a good practice to explicitly decide the scope of the variables. Use `default(none)` to be forced to make the declaration !

- `private(list of variables)` - a new version of the original variable with the same type and size is created in the memory of each thread belonging to the parallel region

- `shared(list of variables)` - threads of the team access and modify the same original variable in the shared memory.

- `default(shared | none)` - used to specify whether variables in a parallel region are shared or private by default. The first option causes all variables referenced in the construct to be shared except the private variables which are specified explicitly, while none requires each variable in the construct to be specified explicitly as shared or private.

private variables in/out

- Use `firstprivate` to declare private variables that are initialized with the main thread's value of the variables

- Use `lastprivate` to declare private variables whose values are copied back out to main thread's variables by the thread that executes the last iteration of a parallel for loop, or the thread that executes the last parallel section

- These are special cases of private, and are useful to bring values in and out from the parallel section of code.

reduction

- A typical calculation which needs to be synchronized is a global reduction operation performed in parallel by the threads of a team.

- It is possible to use the `reduction` clause with `parallel` and `for` directives. The syntax is:

- `reduction (op: list)` with op ∈`{+, -, *, &, ˆ, |, &&, ||}`

- For each of the variables in list, a private copy is created for each thread of the team. The private copies are initialized to the neutral element of the operation op (e.g. 0 for +, 1 for *) and can be updated by the owning thread.
  At the end of the region for which the reduction clause is specified, the local values of the reduction variables are combined according to the operator op and the result of the reduction is written into the original shared variable.

reduction

- A typical calculation which needs to be synchronized is a global reduction operation performed in parallel by the threads of a team.

- It is possible to use the `reduction` clause with `parallel` and `for` directives. The syntax is:

- `reduction (op: list)` with op ∈{`+, -, *, &, ^, |, &&, ||`}

- For each of the variables in list, a private copy is created for each thread of the team. The private copies are initialized to the neutral element of the operation op (e.g. 0 for +, 1 for *) and can be updated by the owning thread.
  At the end of the region for which the reduction clause is specified, the local values of the reduction variables are combined according to the operator op and the result of the reduction is written into the original shared variable.

simpler than critical access to update the shared variable

`parallel` region

-
  ```
  #pragma omp parallel [clause [clause] ... ]
  {
    // structured block ...
  }
  ```

- A team of threads is created to execute the parallel region in parallel.
  Each thread of the team is assigned a unique thread number, (master=0 … n-1).
  The parallel construct creates the team but **does not distribute the work** of the parallel region among the threads of the team.

  - Use `for` or `sections` to distribute the work (*work sharing*)

`parallel` region clauses

- Define the scope of variables in the clauses with the clauses seen previously

- Set the number of threads with `num_threads(integer_expression)`

- Decide the parallel execution based on `if` clause that evaluates an expression returning non-zero as true or zero as false (serial execution)

`parallel` region nesting

- A nesting of parallel regions by calling a parallel construct within a parallel region is possible. However, the default execution mode assigns only one thread to the team of the inner parallel region.
- `int omp_get_nested()` to know if nesting is active
- `void omp_set_nested(int nested)` to set nesting (with `nested != 0`)

for loop

- The loop construct causes a distribution of the iterates of a parallel loop:

```
#pragma omp for [clauses [ ] ...]
```

- It is restricted to loops which are parallel loops, in which the iterates of the loop are independent of each other and for which the total number of iterates is **known in advance**.

- The index variable should not be changed within the loop (as lower and upper bounds) and is considered as private variable of the thread executing the corresponding iterate.

for loop

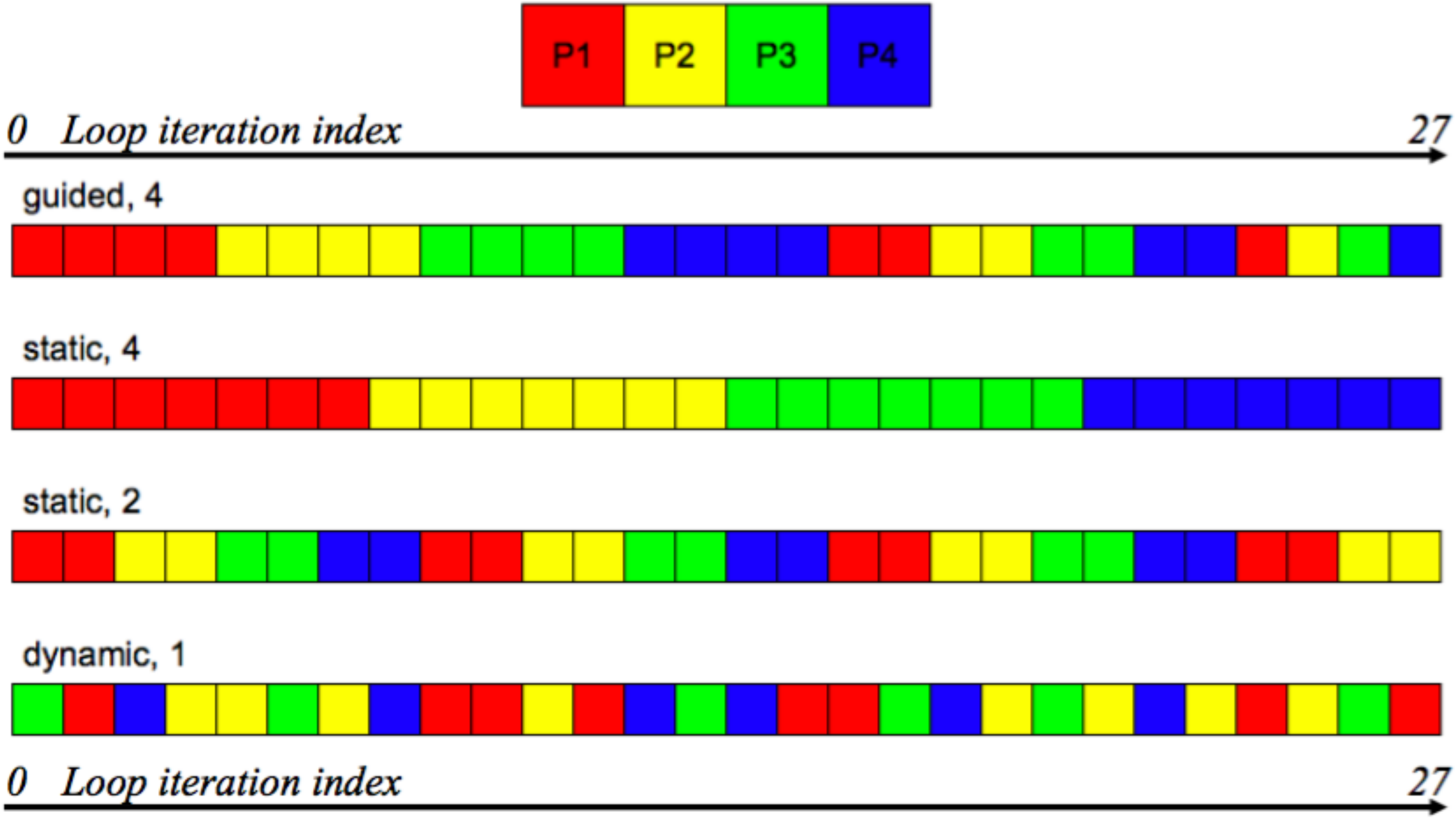
-
  ```
  for (name = lower_bound; name op upper_bound;
  incr_expr) {
   // loop iterate ...
  }
  ```

- Acceptable for loops must:
  - have `lower_bound` and `upper_bound` are integer expressions
  - have op in {`<, <=, >, >=`}
  - have `incr_expr` in the form: `++name, name++, --name, name--, name += incr, name -= incr, name = name + incr, name = incr + name, name = name - incr`, with `incr` that does not change within the loop
  - must not contain `break` instruction

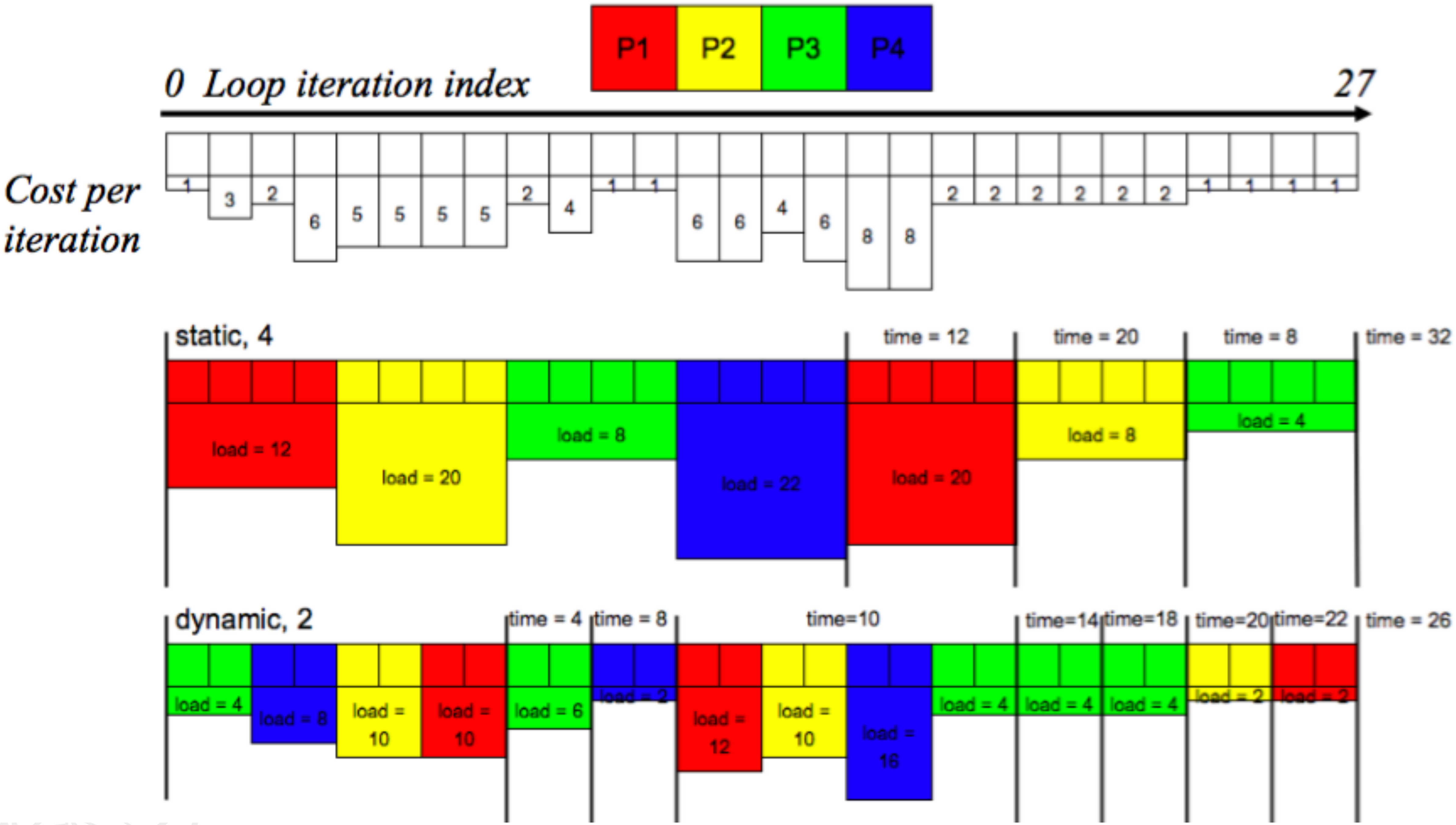
`for` loop scheduling

- Distribution of iterates to threads is done by a scheduling strategy, specified with schedule clause:
- `schedule(static, block_size)`: static distribution of iterates of blocks of size `block_size` in a round-robin fashion. If block_size is not given, blocks of almost equal size are formed and assigned to the threads.
- `schedule(dynamic, block_size)` specifies a dynamic distribution of blocks to threads. A new block of size `block_size` is assigned to a thread as soon as the thread has finished the computation of the previously assigned block. If `block_size` is not provided, it is set to 1.
- `schedule(guided, block_size)` specifies a dynamic scheduling of blocks with decreasing size.
- `schedule(auto)` delegates the scheduling decision to the compiler and/or runtime system.
- `schedule(runtime)` uses the value of environment variable OMP_SCHEDULE
- If no schedule is provided the a default implementation-specific strategy is used

`for` loop scheduling



for loop scheduling



Simplified syntax

- If a parallel region contains only one for os sections then instead of:

-
  ```
  #pragma omp parallel
  {
  #pragma omp for
     for (…) {
     }
  }
  ```

- it is possible to use:

-
  ```
  #pragma omp parallel for
  for (…) {
  }
  ```

Combining sequential parallel for

```
#pragma omp parallel for
    for (...) {

        // Work-sharing loop 1
    }

#pragma omp parallel for
    for (...) {

        // Work-sharing loop 2
    }

.........

#pragma omp parallel for

    for (...) {

        // Work-sharing loop N
    }
```

```
#pragma omp parallel

    {

        #pragma omp for  // Work-sharing loop 1

        { ...... }

        #pragma omp for  // Work-sharing loop 2

        { ...... }

.........

         #pragma omp for  // Work-sharing loop N

         { ...... }

 }
```

Bad: parallel overhead and implied barriers

Better: The cost of the parallel region is amortized over the various work-sharing loops.

Parallel in inner loops

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        #pragma omp parallel for
        for (k=0; k<n; k++)
            { .........}
```

Bad: n^2 overheads of parallel

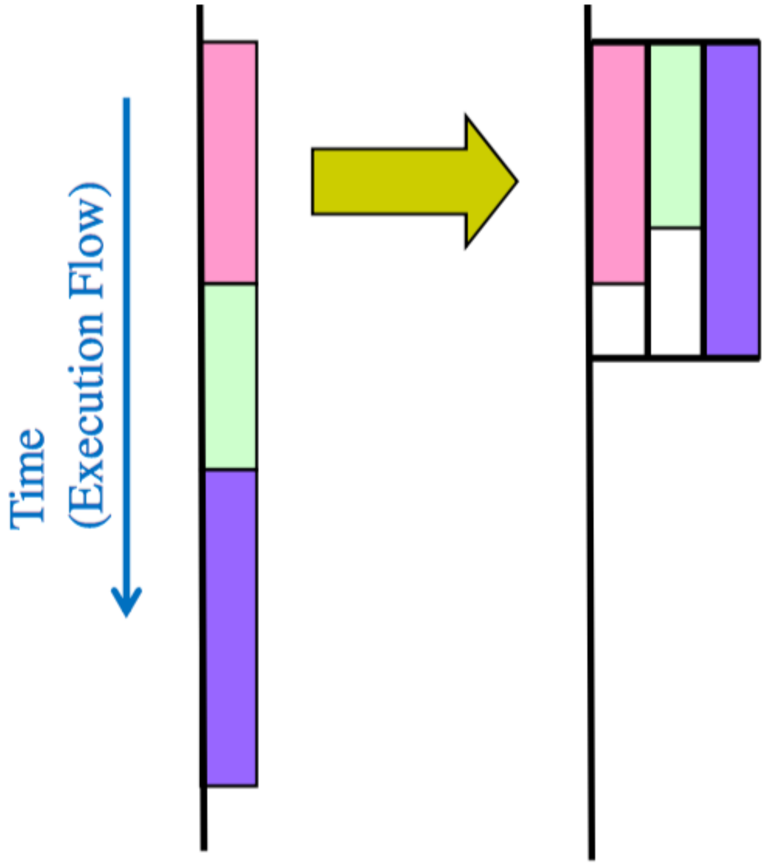
```
#pragma omp parallel
   for (i=0; i<n; i++)
       for (j=0; j<n; j++)
           #pragma omp for
           for (k=0; k<n; k++)
               { .........}
```

Better: 1 overhead of parallel

sections

- The goal is to execute concurrently independent sections of code.

```
#pragma omp parallel sections
{
#pragma omp section
        phase1();
#pragma omp section
        phase2();
#pragma omp section
        phase3();
}
```



`sections`

- At the end of execution of the assigned section, the threads synchronize, **unless** the `nowait` clause is used.

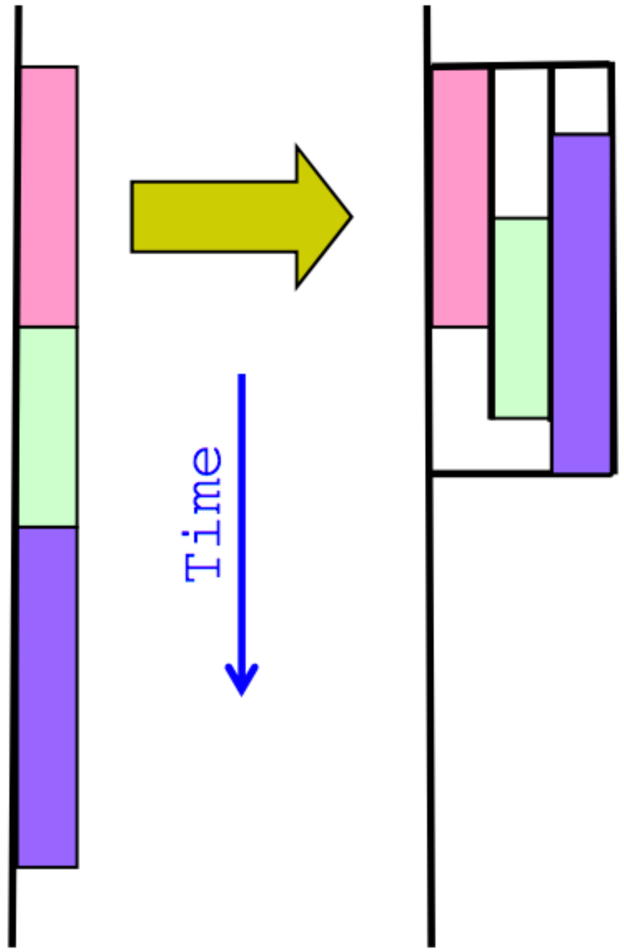
- It is illegal to branch in and out of `section` blocks.

for vs. sections

- The for directive splits the iterations of a loop across the different threads.
- The sections directive assigns each thread to explicitly identified tasks.

task

- Tasks are independent units of work. A thread is assigned to perform a task: `#pragma omp task`
- Tasks might be executed immediately or might be deferred
  - The runtime system decides which of the above
- Allows parallelization of irregular problems, e.g.:
  - Unbounded loops
  - Recursive algorithms (e.g. lists)
  - Producer/consumer



`task` completion

- A task stops at:
  - thread barriers, explicit or implicit. Tasks created in a parallel section will stop at its implicit barrier or at `#pragma omp barrier`
  - task barriers: `#pragma omp taskwait`

task completion

```
#pragma omp parallel
{
#pragma omp task // multiple foo tasks
  foo();
#pragma omp barrier // explicit barrier
#pragma omp single
  {
#pragma omp task // one bar task
    bar();
  } // implicit barrier
}
```

task motivations

-
  ```
  while(my_pointer) {
     (void) do_independent_work(my_pointer);
     my_pointer = my_pointer->next ;
  } // End of while loop
  ```

- Hard to do without tasking: first count number of iterations, then convert while loop to for loop

- A solution is to use the `single` construct : one thread generates the tasks. All other threads execute the tasks as they become available.

task motivations

```
my_pointer = listhead;

#pragma omp parallel
{
#pragma omp single
  {
    while(my_pointer) {
#pragma omp task firstprivate(my_pointer)
      {
         (void) do_independent_work (my_pointer);
      }
      my_pointer = my_pointer->next ;
    } // end while
  } // end single region
} // end parallel region
```

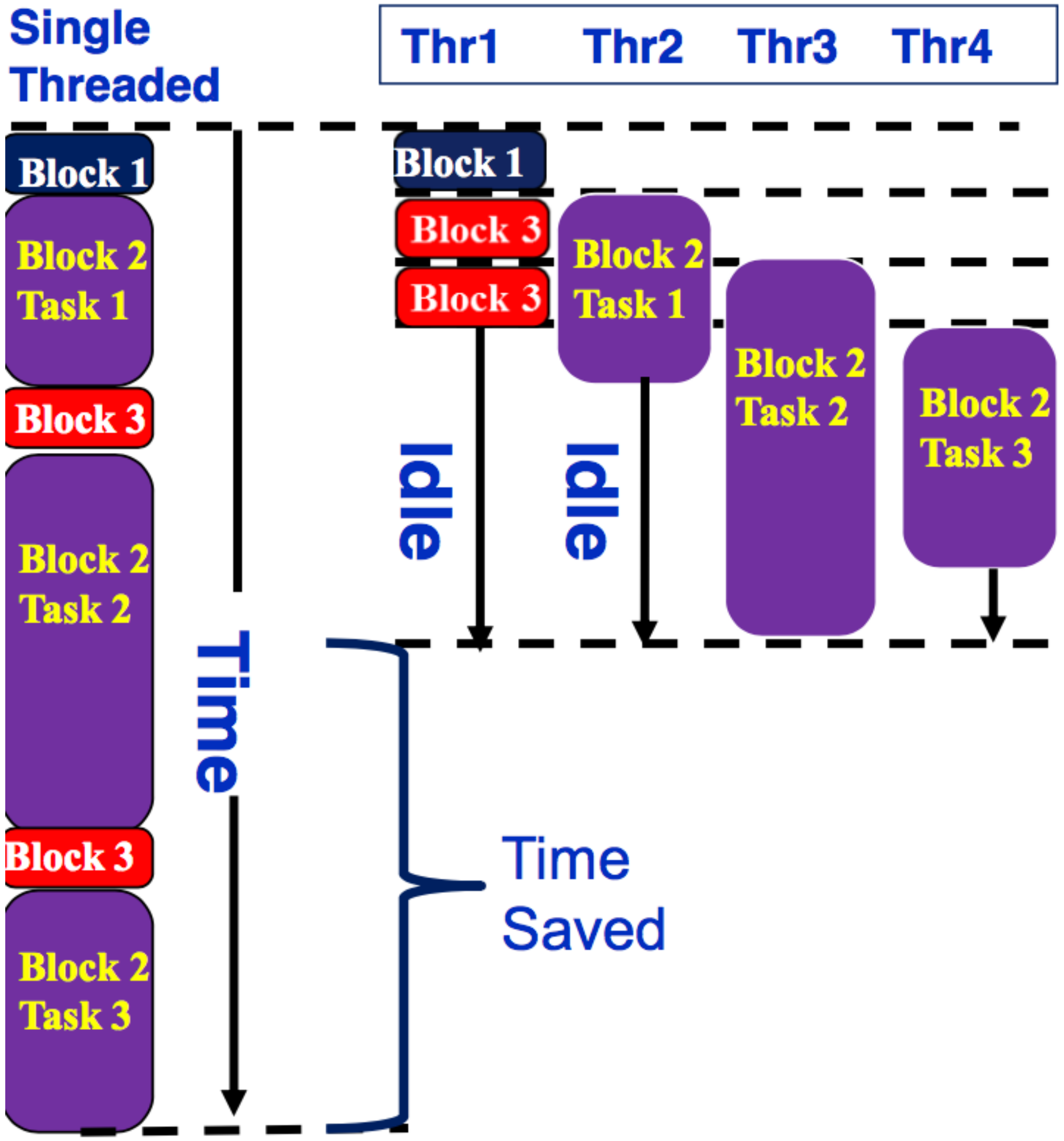
task motivations

```
my_pointer = listhead;

#pragma omp parallel
{
#pragma omp single nowait // eliminate a barrier
  {
    while(my_pointer) {
#pragma omp task firstprivate(my_pointer)
      {
         (void) do_independent_work (my_pointer);
      }
      my_pointer = my_pointer->next ;
    } // end while
  } // end single block - no implied barrier (nowait)
} // end parallel block - implied barrier
```

task motivations

```
#pragma omp parallel
{
#pragma omp single
  { //block 1
    node * p = head;
    while (p) { // block 2
#pragma omp task
      process(p);
      p = p->next; //block 3
    } // end while
  } // end single block
} // end parallel block
```



for and task

-
  ```
  #pragma omp parallel
  {
  #pragma omp for private(p)
    for ( int i =0; i <numlists ; i++) {
      p = listheads[ i ] ;
      while( p ) {
  #pragma omp task
        process(p)
        p = next( p ) ;
      } // end while
    } // end for
  } // end parallel
  ```

- Example – parallel pointer chasing on multiple lists using tasks (nested parallelism)

OpenMP memory model and synchronization

OpenMP memory model

- In the shared memory model of OpenMP all threads share an address space ... but what they actually see at a given point in time may be complicated: a variable residing in shared memory may be in the cache of several CPUs/cores.

- A memory model is defined in terms of:
  - Coherence: Behavior of the memory system when a single address is accessed by multiple threads.
  - Consistency: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads.

OpenMP memory model

- In the shared memory model of OpenMP all threads share an address space ... but what they actually see at a given point in time may be complicated: a variable residing in shared memory may be in the cache of several CPUs/cores.

At a given point in time, the "private view" seen by a thread may be different from the view in shared memory.

In fact, there are several re-orderings from the original source code:
- Compiler re-orders program order to the code order
- Machine re-orders code order to the memory commit order

Consistency

- Sequential Consistency:
  - In a multi-processor, ops (R, W, S) are sequentially consistent if:
    - They remain in program order for each processor.
    - They are seen to be in the same overall order by each of the other processors.
  - Program order = code order = commit order
- Relaxed consistency:
  - Remove some of the ordering constraints for memory ops (R, W, S).

OpenMP consistency

- OpenMP has a relaxed consistency:
  - S ops must be in sequential order across threads.
  - Can not reorder S ops with R or W ops on the same addresses on the same thread
  - The S operation provided by OpenMP is `flush`

OpenMP consistency

Relaxed consistency means that memory updates made by one CPU may not be immediately visible to another CPU

- Data can be in registers
- Data can be in cache
  (cache coherence protocol is slow or non-existent)

Therefore, the updated value of a shared variable that was set by a thread may not be available to another

The `flush` construct flushes shared variables from local storage (registers, cache) to shared memory

- The S operation provided by OpenMP is `flush`

OpenMP consistency

Relaxed consistency means that memory updates made by one CPU may not be immediately visible to another CPU

- Data can be in registers
- Data can be in cache
  (cache coherence protocol is slow or non-existent)

Therefore, the updated value of a shared variable that was set by a thread may not be available to another

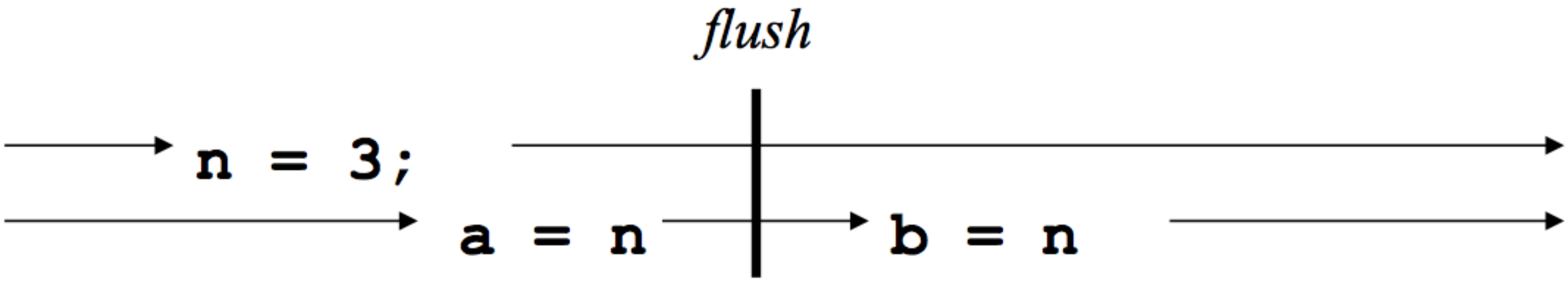
The `flush` construct flushes shared variables from local storage (registers, cache) to shared memory

```
double A;
A = compute();
#pragma omp flush(A); // flush to memory to make sure other
                      // threads can pick up the right value
```

Implicit `flush`

- An OpenMP `flush` is automatically performed at:
  - Entry and exit of `parallel` and `critical` and `atomic` (only variable being atomically updated)
    - unless `nowait` is specified
  - Exit of `for`
  - Exit of `sections`
  - Exit of `single`
  - Barriers
  - When setting/unsetting/testing locks after acquisition

- the `flush` operation **does not actually synchronize** different threads. It just ensures that a thread’s values are made consistent with main memory.



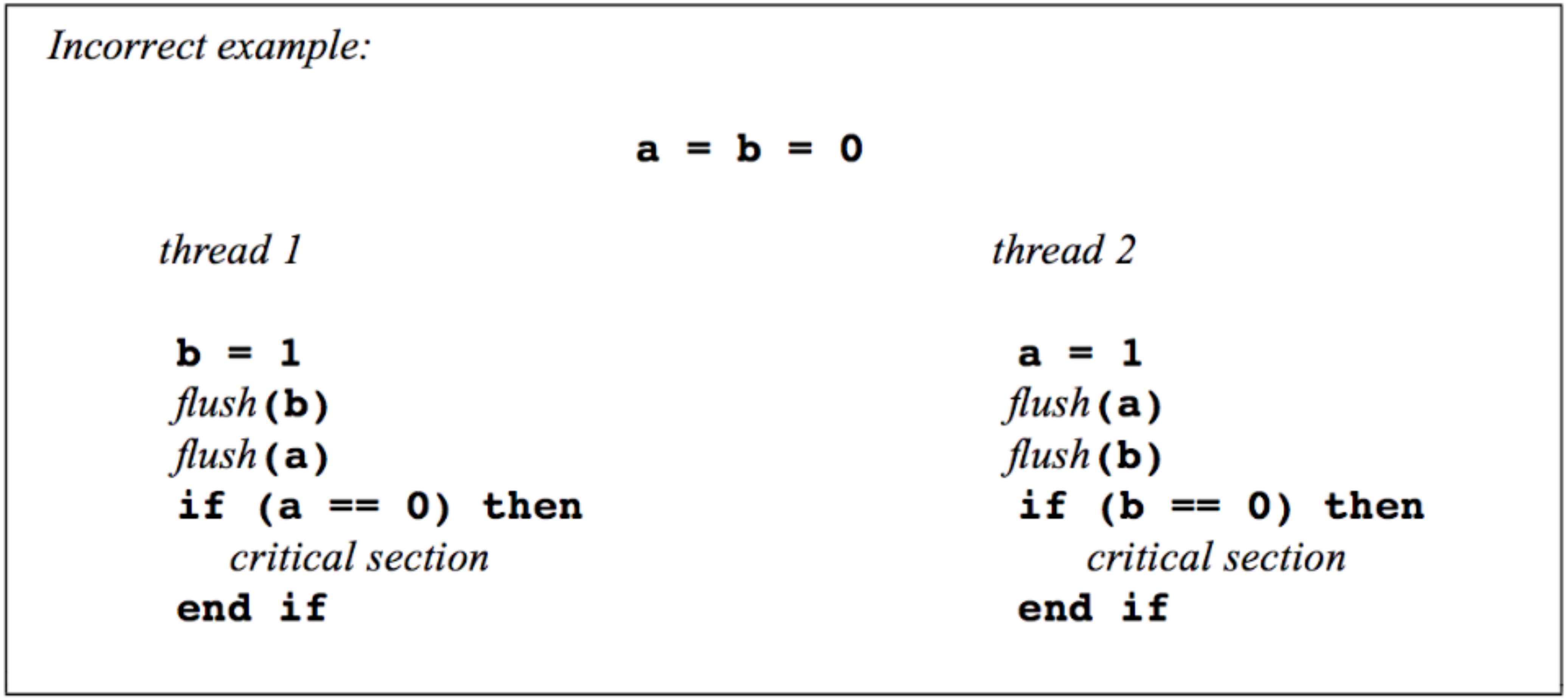
- b = 3, but there is no guarantee that a will be 3

flush

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the “flush set”.
- The `flush` set is:
  - “all thread visible variables” for a `flush` construct without an argument list.
  - a list of variables when the `flush(list)` construct is used.
- The action of `flush` is to guarantee that:
  - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
  - All R,W ops that overlap the flush set and occur after the flush don’t execute until after the flush.
  - Flushes with overlapping flush sets can not be reordered.
- Flush forces data to be updated in memory so other threads see the most recent value.

- A `flush` construct with a list applies the flush operation to the items in the list,and does not return until the operation is complete for all specified list items.
  - If a pointer is present in the list, the pointer itself is flushed, not the object to which the pointer refers
- A `flush` construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program is flushed.

-



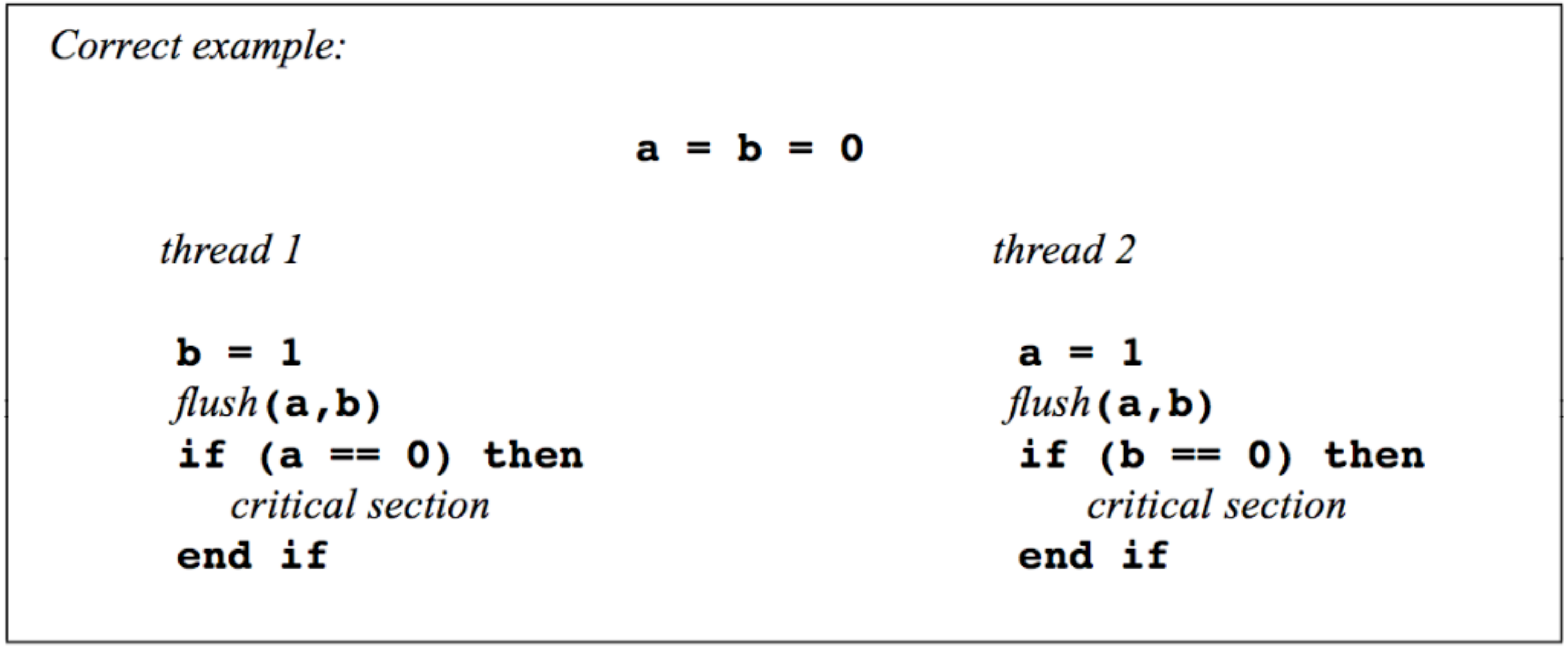
Incorrect example:

```
                        a = b = 0

   thread 1                               thread 2

    b = 1                                  a = 1
    flush(b)                               flush(a)
    flush(a)                               flush(b)
    if (a == 0) then                       if (b == 0) then
       critical section                       critical section
    end if                                 end if
```

-



Correct example:

```
                        a = b = 0

   thread 1                               thread 2

    b = 1                                  a = 1
    flush(a,b)                             flush(a,b)
    if (a == 0) then                       if (b == 0) then
       critical section                       critical section
    end if                                 end if
```

Atomics

- If a variable used as flag is flushed, does the flush assure that the flag value is cleanly communicated ?
- No: if the flag variable straddles word boundaries or is a data type that consists of multiple words, it is possible for the read to load a partial result.
- We need the ability to manage updates to memory locations atomically.

atomic

- `# pragma omp atomic [read | write | update | capture]`
- Atomic can protect loads

  -
    ```
    # pragma omp atomic read
    v = x;
    ```
- Atomic can protect stores

  -
    ```
    # pragma omp atomic write
    x = expr;
    ```
- Atomic can protect updates to a storage location (this is the default behavior ... i.e. when you don’t provide a clause)

  -
    ```
    # pragma omp atomic update
    x++; or ++x; or x--; or –x;
    or x binop= expr; or x = x binop expr;
    where binop ∈ {+, -, *, /, &, ˆ, |, <<, >>}
    ```

atomic

- Atomic can protect the assignment of a value (its capture) **and** an associated update operation:
  - `# pragma omp atomic capture`
    `statement or structured block`
- Where the statement is one of the following forms:
  - `v=x++; v=++x; v=x--; v= --x; v=x binop expr;`
- Where the structured block is one of the following forms:
  - `{v=x; x binop = expr;}` `{x binop = expr; v=x;}`
  - `{v=x; x = x binop expr;}` `{x = x binop expr; v=x;}`
  - `{v=x; x++;}` `{v=x; ++x}`
  - `{++x; v=x}` `{x++; v=x;}`
  - `{v=x; x--;}` `{v=x; --x;}`
  - `{--x; v=x;}` `{x--; v=x;}`

atomic and flush

Thread 1

```
#pragma omp flush(flag)
#pragma omp atomic write
  flag = 1;
#pragma omp flush(flag)
```

Thread 2

```
#pragma omp flush(flag)
#pragms omp atomic read
  flg_tmp= flag;
if (flg_tmp == 1)
  // do something…
```

critical

- The `critical` construct protects access to shared, modifiable data.

- The `critical` section allows only one thread to enter it at a given time.

-
  ```
  float dot_prod(float* a, float* b, int N)
  {
    float sum = 0.0;
  #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
  #pragma omp critical
      sum += a[i] * b[i];
    }
    return sum;
  }
  ```

critical

- The `critical` construct protects access to shared, modifiable data.
- The `critical` section allows only one thread to enter it at a given time.

-
  ```
  float RES;
  #pragma omp parallel
  {
  #pragma omp for
    for(int i=0; i<niters; i++){
      float B = big_job(i);
  #pragma omp critical (RES_lock)
      consum(B, RES);
    }
  }
  ```

atomic vs. critical

- atomic is a special case of a critical section
- atomic introduces less overhead then critical
    - the atomic construct does not enforce exclusive access to x with respect to a critical region specified by a critical construct.
      An advantage of the atomic construct over the critical construct is that parts of an array variable can also be specified as being atomically updated.
      The use of a critical construct would protect the entire array.
-
  ```
  #pragma omp parallel for shared(x, y, index, n)
      for (i = 0; i < n; i++) {
  #pragma omp atomic
          x[index[i]] += work1(i);
          y[i] += work2(i);
  }
  ```

locks vs. `critical`

- locks: performance and function similar to named `critical`; best used for structures
- E.g.: message queue data structure composed by:
  - list of messages
  - pointer to rear of queue
  - pointer to fron of queue
  - count of messages dequeued
  - `omp_lock_t` lock variable

Caveats

- Do not mix different types of mutual exclusion for a variable:

```
# pragma omp atomic          # pragma omp critical
x += f(y);                   x = g(x);
```

- the `critical` section can not block the `atomic` directive from accessing x

- Solution: use `critical` in both fragments or rewrite `critical` in a form suitable for `atomic`

Caveats

- Do not nest anonymous critical sections to avoid deadlocks:

```
#pragma omp critical
y = f(x);
...
double f(double x) {
#pragma omp critical
  z=g(x); /* z is shared */
  ...
}
```

- When a thread attempts to enter the second critical section, it will block forever. A thread blocked waiting to enter the second critical block will never leave the first, and it will stay blocked forever.

Caveats

- Do not nest anonymous critical sections to avoid deadlocks:

```
# pragma omp critical(one)
y = f(x);
...
double f(double x) {
#pragma omp critical(two)
  z=g(x); /* z is global */
  ...
}
```

- When a thread attempts to enter the second critical section, it will block forever. A thread blocked waiting to enter the second critical block will never leave the first, and it will stay blocked forever.

Caveats

- Remind Dijkstra: named critical sections acquired in wrong order may lead to deadlock:

Time	Thread u	Thread v
0	Enter crit. sect. *one*	Enter crit. sect. *two*
1	Attempt to enter *two*	Attempt to enter *one*
2	Block	Block

Unnecessary protection

- Any protection slows down the program's execution and it does not matter whether you use atomic operations, critical sections or locks. Therefore, you should not use memory protection when it is not necessary.

- A variable should not be protected from concurrent writing in the following cases:

  - If a variable is local for a thread (also, if the variable is `threadprivate`, `firstprivate`, `private` or `lastprivate`).

  - If the variable is accessed in a code fragment which is guaranteed to be executed by a single thread only (in a `master` or `single` section).

Reduce critical sections

- Critical sections always slow down a program's execution. Do not use critical sections where it is not necessary. For example:

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i ) {
  #pragma omp critical
  {
    if (arr[i] > max) max = arr[i];
  }
}
```

- can be rewritten as:

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i ) {
   #pragma omp flush(max)
   if (arr[i] > max) {
      #pragma omp critical
      {
       if (arr[i] > max) max = arr[i];
      }
   }
}
```

Producer / Consumer

Queue

- We need a queue data structure, e.g. following a FIFO policy in which a producer enqueues workloads at the rear of the queue and a consumer dequeues at the front.

```
struct queue_node_s {
   int src;
   int msg;
   struct queue_node_s* next_p;
};

struct queue_s{
   int enqueued;
   int dequeued;
   struct queue_node_s* front_p;
   struct queue_node_s* tail_p;
};
```

```
struct queue_s*
Allocate_queue(void);
void Free_queue(struct queue_s*
q_p);
void Print_queue(struct queue_s*
q_p);
void Enqueue(struct queue_s* q_p,
int src, int msg);
int Dequeue(struct queue_s* q_p,
int* src_p, int* msg_p);
int Search(struct queue_s* q_p, int
msg, int* src_p);
```

Message passing

- Let us consider a case in which any thread may communicate with other threads.

- Each thread needs a queue. It enqueues messages to other threads and dequeues from its own queue to receive from other threads.

  - Let us suppose each thread first sends a message then check is any message has been received.

- When a thread is done sending messages, it receives messages until all the threads are done, at which point all the threads quit.

Message passing

```
for (sent msgs = 0; sent msgs < send max; sent msgs++) {
  Send_msg();
  Try_receive();
}
while (!Done())
  Try_receive();
```

Send_msg() and Try_receive() use the Enqueue/Dequeue of the queue.
Done() checks if the queue is empty and no other threads are still alive to send messages

- When a thread is done sending messages, it receives messages until all the threads are done, at which point all the threads quit.

Sending messages

- Even without looking at the details of the implementation of:

```
void Enqueue(struct queue_s* q_p, int src, int msg);
```

we may think that it is going to update the pointer to the last element, i.e. something that is critical…

- Send_msg() should protect the access to that function…

Sending messages

- Even without looking at the details of the implementation of:

```
mesg = random();
dest = random() % thread count;
# pragma omp critical
Enqueue(queue, dest, my rank, mesg);
```

- Send_msg() should protect the access to that function…

Receiving messages

- In this scenario only the owner of the queue dequeues a message, so the requirement for synchronization is different.

- If we dequeue one message at a time, and there are at least two messages in the queue, a call to Dequeue can't possibly conflict with any calls to Enqueue:

    - keep track of the size of the queue, so that we can avoid any synchronization as long as there are at least two messages.

    - Keep track of size using 2 variables:
      queue_size = enqueued − dequeued

    - dequeued is owned by the receiving thread, while enqueued may be changed by a different thread, but we have small delays only if we get an error when computing size 0 or 1 instead of actual values of 1 or 2.

Receiving messages

It’s a tradeoff: we renounce to certain synchronization costs for some uncertain possible delays: waiting when erroneously computing a size of 0 instead of 1, or synchronizing when computing size of 1 instead of 2.
`Try_receive()` can be implemented as:

```
queue_size = enqueued – dequeued;
if (queue size == 0)
  return;
else if (queue_size == 1)
#pragma omp critical
  Dequeue(queue, &src, &mesg);
else
  Dequeue(queue, &src, &mesg);
Print message(src, mesg);
```

- dequeued is owned by the receiving thread, while enqueued may be changed by a different thread, but we have small delays only if we get an error when computing size 0 or 1 instead of actual values of 1 or 2.

Termination detection

- We need to know if the queue is empty because nobody will ever produce a new message. Use a counter to know how many active threads are still there…

- Function Done() can be implemented as:

```
queue_size = enqueued – dequeued;
if (queue_size == 0 && done_sending ==
thread_count)
  return TRUE;
else
  return FALSE;
```

Termination detection

When a producer says that it has finished it must update `done_sending`
The update must be `critical` or `atomic`:

```
#pragma omp atomic
done_sending++;
```

- Function Done() can be implemented as:

```
queue_size = enqueued – dequeued;
if (queue_size == 0 && done_sending ==
thread_count)
  return TRUE;
else
  return FALSE;
```

Startup

- The master thread, will allocate an array of message queues, one for each thread.
  This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.

- We can start the threads using a parallel directive, and each thread can allocate storage for its individual queue.

  - But we must check that all the queues have been built before starting any threads, to a void writing to a queue not yet available.

Startup

- The master thread, will allocate an array of message queues, one for each thread.
  This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.

We can not rely on implicit barrier. Add a:

```
# pragma omp barrier
```

When a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier. After all the threads have reached the barrier, all the threads in the team can proceed.

  - But we must check that all the queues have been built before starting any threads, to a void writing to a queue not yet available.

Using locks

- Using unnamed critical sections adds unnecessary synchronization costs in the previous example, where Enqueue and Dequeue block themselves, even when different threads are sending messages to other different threads…

- we can not rely on named `critical` sections since they have to be defined at compile time… we do not know which thread will try to communicate with another thread.

- Solution: add locks to the data structure.

Using locks

```
struct queue_node_s {
   int src;
   int mesg;
   struct queue_node_s* next_p;
};

struct queue_s{
   omp_lock_t lock;
   int enqueued;
   int dequeued;
   struct queue_node_s* front_p;
   struct queue_node_s* tail_p;
};
```

- Solution: add locks to the data structure.

Sending message

-
  ```
  # pragma omp critical
  /* q_p = msg queues[dest] */
  Enqueue(q_p, my rank, mesg);
  ```

- can be replaced with

-
  ```
  /* q p = msg queues[dest] */
  omp_set_lock(&q_p->lock);
  Enqueue(q_p, my rank, mesg);
  omp_unset_lock(&q_p->lock);
  ```

Receiving message

-
  ```
  # pragma omp critical
  /* q_p = msg queues[my rank] */
  Dequeue(q_p, &src, &mesg);
  ```

- can be replaced with

-
  ```
  /* q_p = msg queues[my rank] */
  omp_set_lock(&q_p->lock);
  Dequeue(q_p, &src, &mesg);
  omp_unset_lock(&q_p->lock);
  ```

Creation/destruction of lock

- Add initialization of the lock to the function that initializes an empty queue.
- Destruction of the lock can be done by the thread that owns the queue before it frees the queue.

Credits

- These slides report material from:
  - Prof. Robert van Engelen (Florida State University)
  - Prof. Dan Negrut (UW-Madison)
  - Prof. Robert M. Keller (Harvey Mudd College)
  - Prof. Vivek Sarkar (Rice University)
  - Tim Mattson (Intel Corp.)
  - Mary Thomas (SDSU)

Books

- The Art of Concurrency, Clay Breshears, O’Reilly - Chapt. 5
- Principles of Parallel Programming, Calvin Lyn and Lawrence Snyder, Pearson - Chapt. 6
- Parallel Programming for Multicore and Cluster Systems, Thomas Dauber and Gudula Rünger, Springer - Chapt. 6
- An introduction to parallel programming, Peter S. Pacheco, Morgan Kaufman - Chapt. 5