POLITECNICO DI MILANO

OpenMP part 2

Parallel Computing

Serena Curzel
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
serena.curzel@polimi.it

Thread cancellation

- Until OpenMP 3.1, parallel execution cannot be aborted
  - Parallel regions must always run to completion
  - Alternatively, the region does not start at all
- OpenMP 4.0 introduces cancellation
  - Best effort: does not guarantee to trigger termination immediately
- Useful for divide-and-conquer algorithms (e.g., stop searching when you find the element) and to handle errors

Thread cancellation


```
#pragma omp cancel <construct-type>
```

- Cancel current thread

```
#pragma omp cancellation point
```

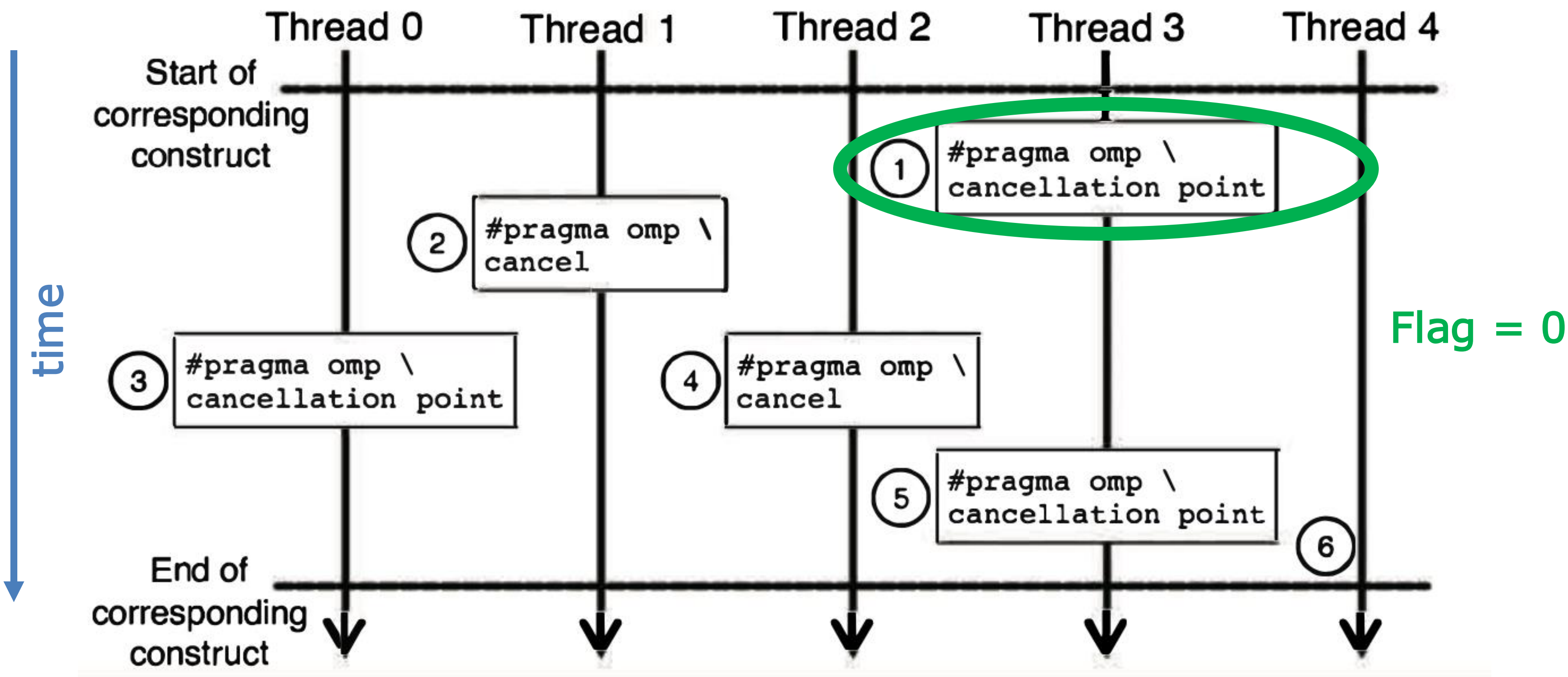
- Check if any thread requested cancellation

- A thread that wishes to terminate the execution must have `cancel` in its execution path
- The `cancel` directive raises a flag
- When a thread encounters `cancellation point`, it checks the cancellation flag

Thread cancellation

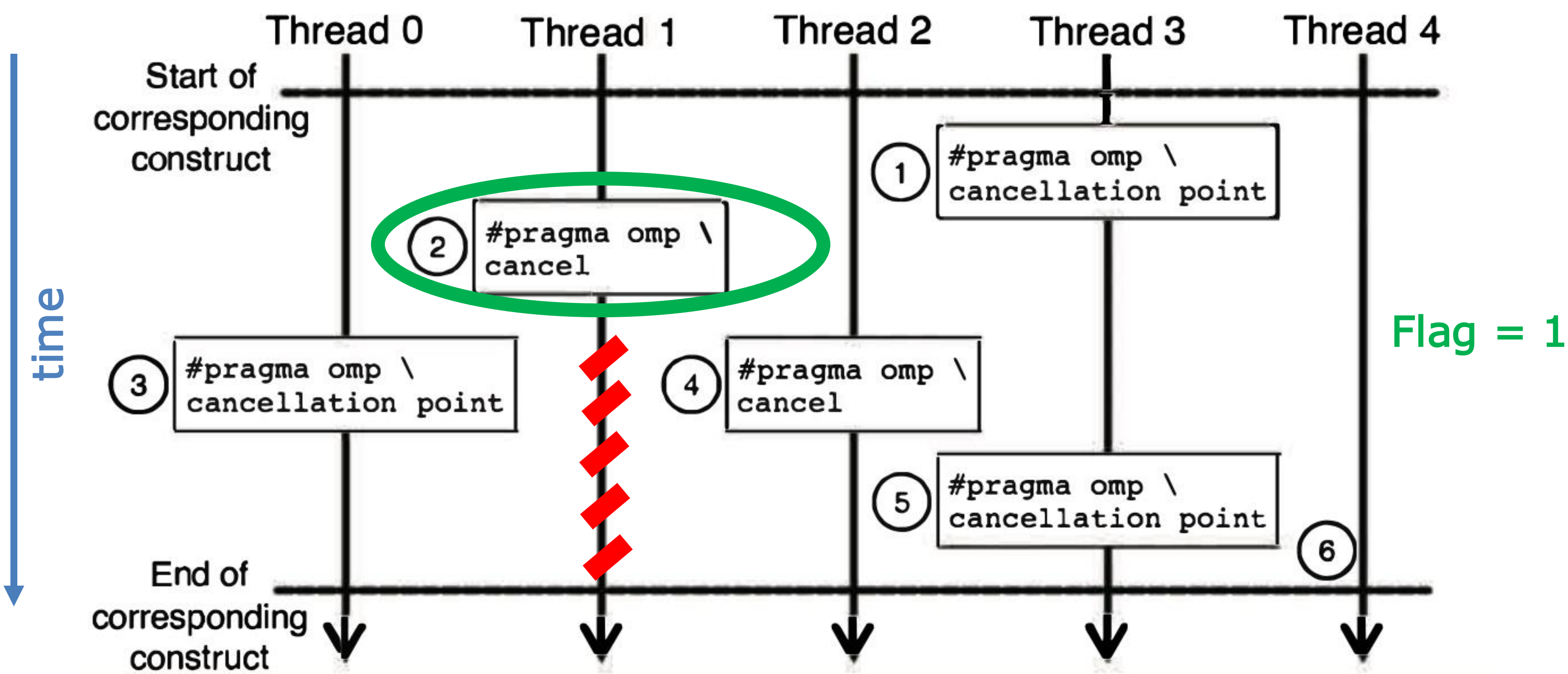
- The status of cancellation is also checked
  - At another `cancel` region
  - At a barrier (implicit or explicit)
- The user is responsible of setting cancellation points to ensure timely cancellation
- There is an overhead in checking for cancellation, so `OMP_CANCELLATION` is disabled by default

Thread cancellation



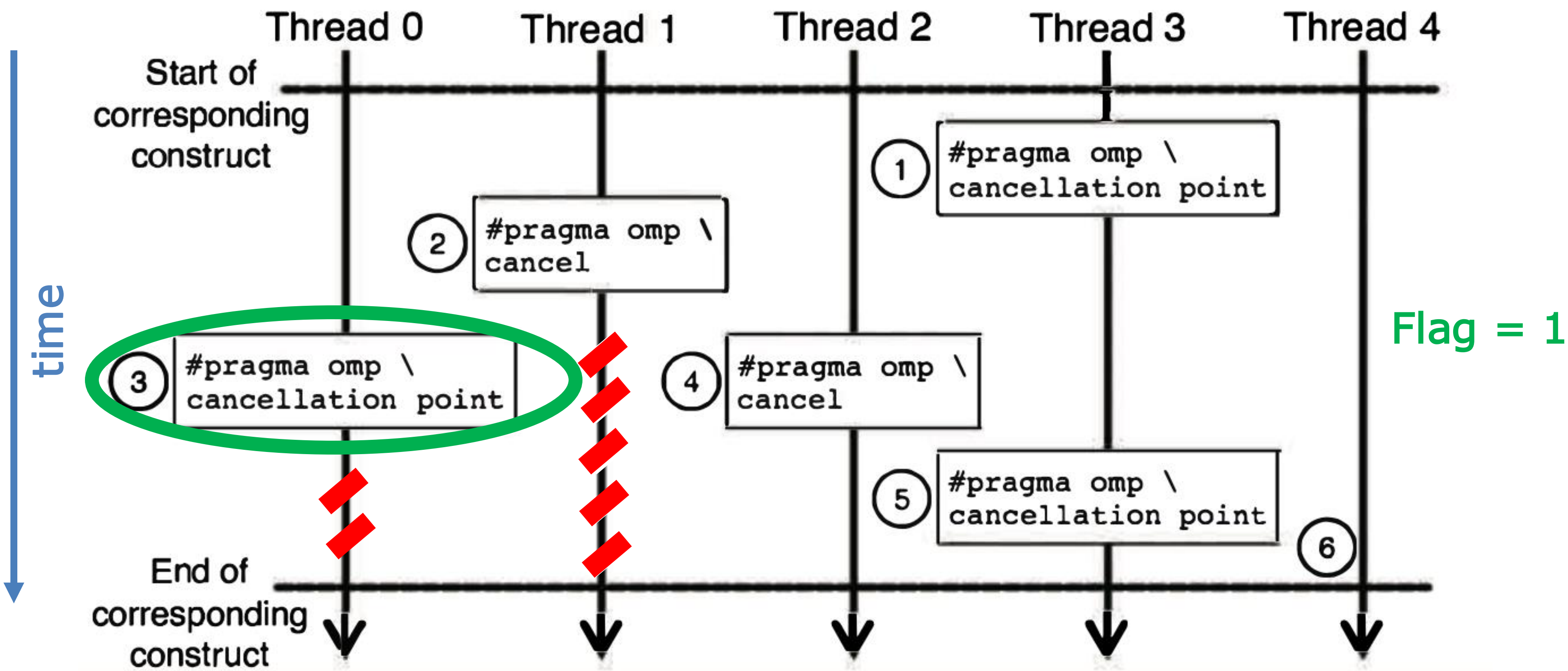
- Thread 3 encounters a cancellation point, nothing happens

Thread cancellation



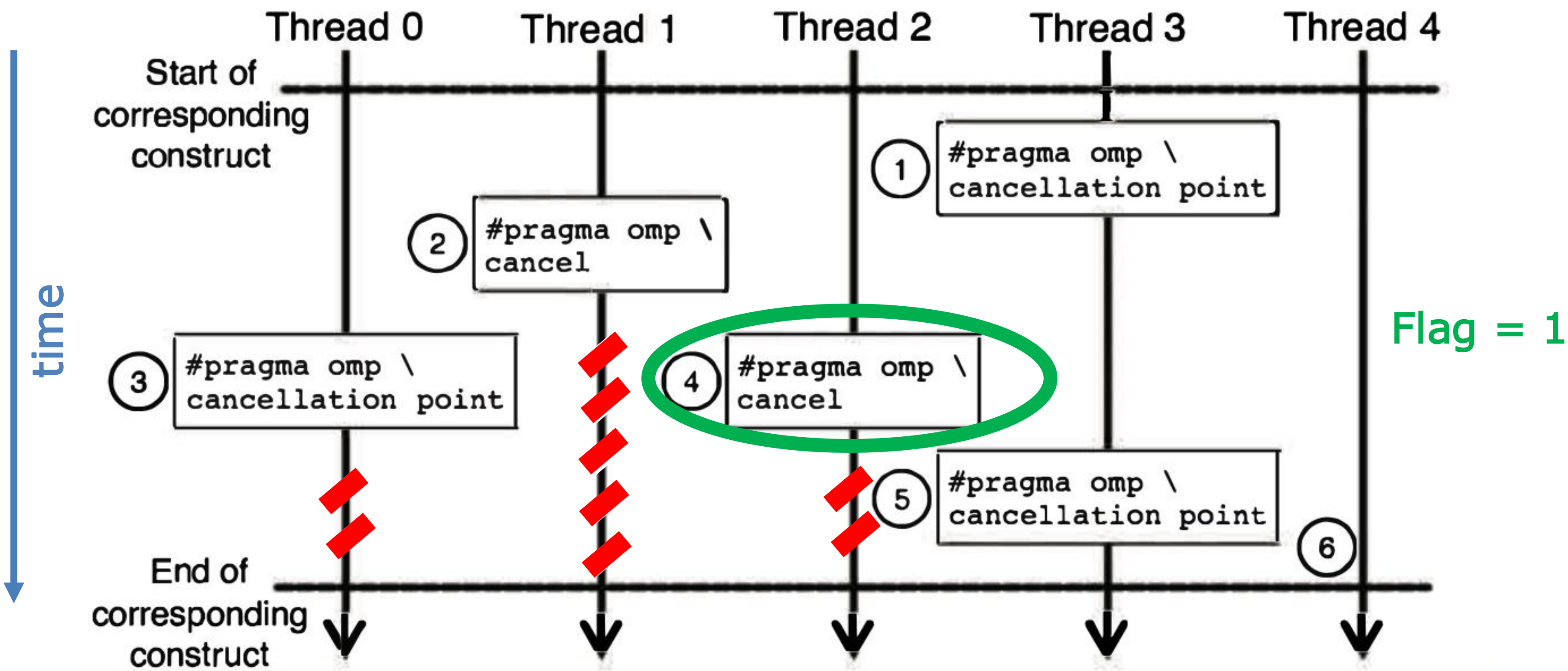
- Thread 1 requests cancellation and waits at the next synchronization point

Thread cancellation



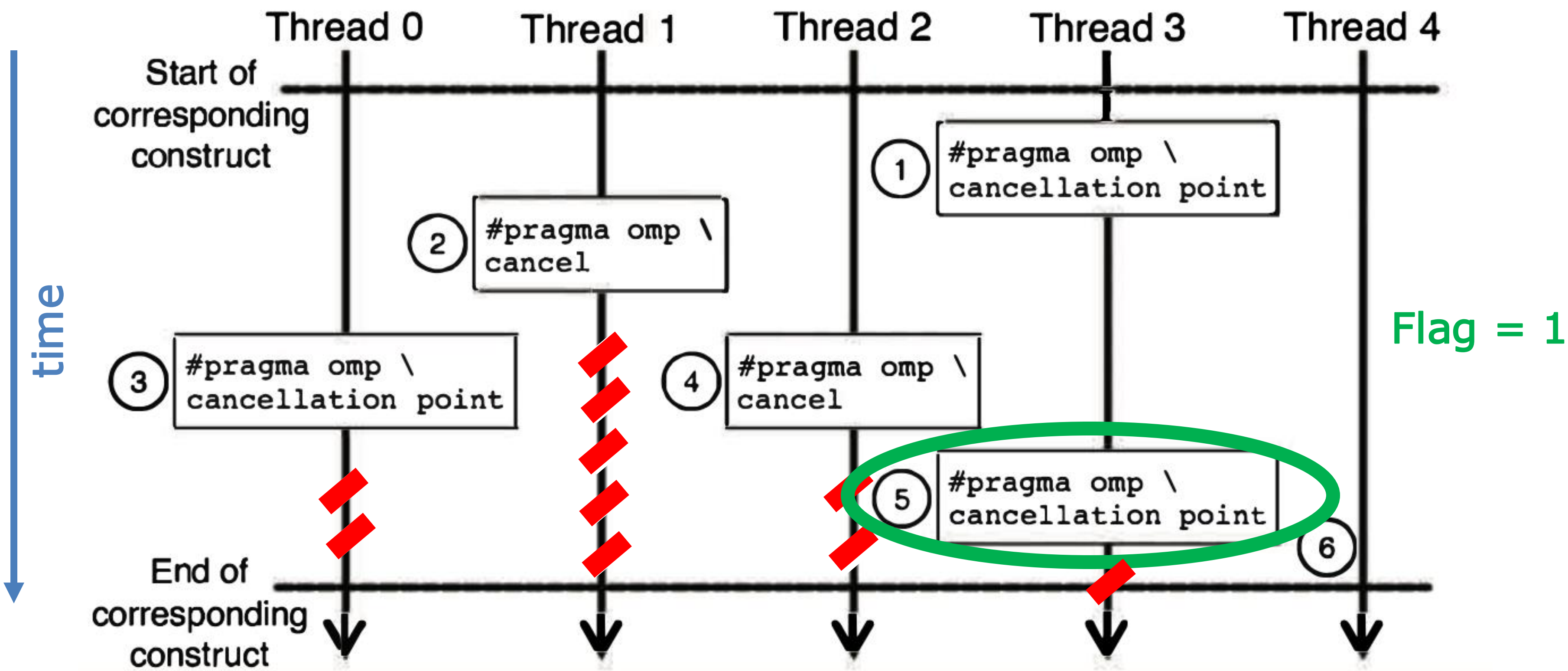
- Thread 0 checks for cancellation and terminates

Thread cancellation



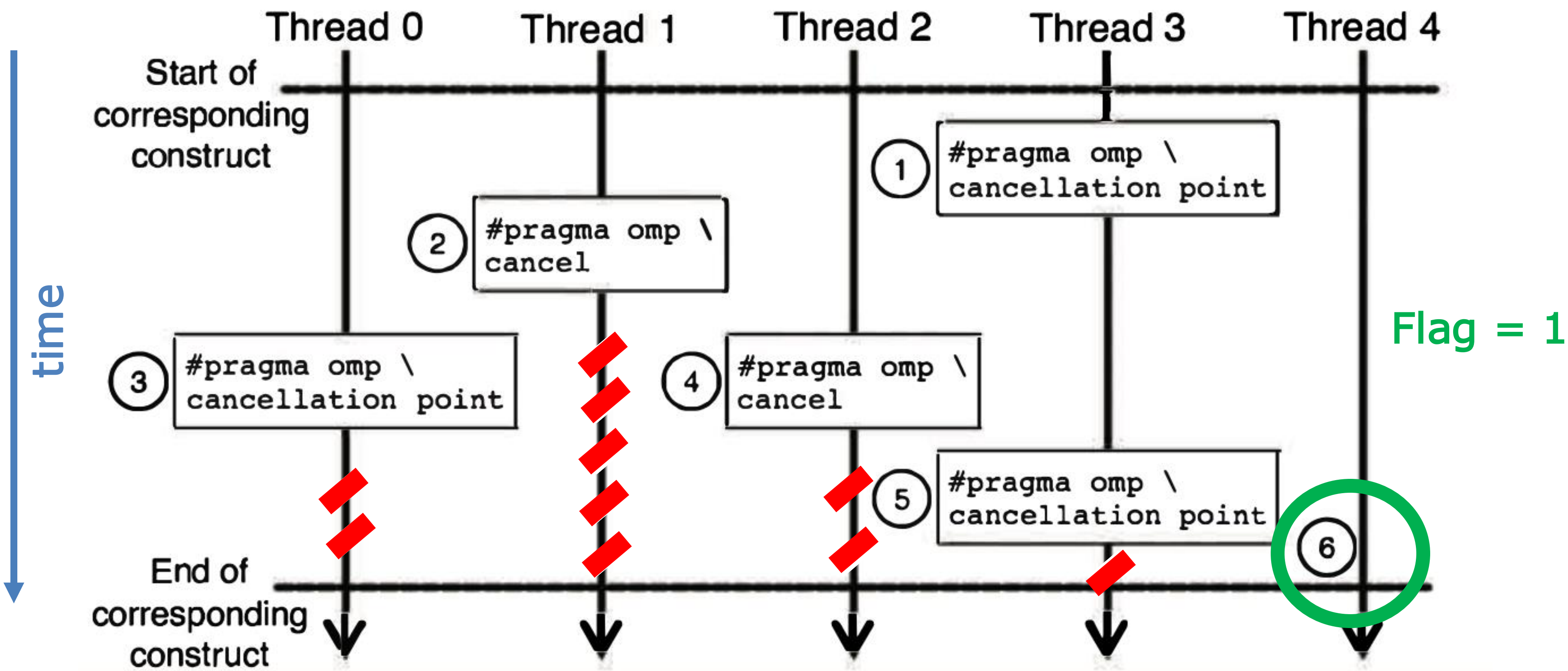
- A `cancel` directive first checks for the cancellation flag, thread 2 terminates

Thread cancellation



- This time, thread 3 terminates

Thread cancellation



- Thread 4 never encounters any cancellation points and it finishes execution normally

Tasking in OpenMP

- Tasks are useful to parallelize algorithm with *irregular* and *runtime-dependent* execution flow
- Simplest example: `while` loop
- Queuing system that dynamically assigns work to threads
- Simplest definition: *an OpenMP task is a block of code contained in a parallel region that can be executed simultaneously with other tasks in the same region*
- Tasks are not guaranteed to be executed where they are defined in the source code

Linked list traversal in OpenMP

- Without tasks: → highly inefficient!

```
while (p != NULL)
{
        p = p->next;
    count++;
}

p = head;
for(i=0; i<count; i++)
{
        parr[i] = p;
    p = p->next;
}

#pragma omp parallel for
for(i=0; i<count; i++)
        processwork(parr[i]);
```

1. Measure how long is the list

2. Put the content into an array

Most of the work is done here!

3. Process array elements in parallel

Linked list traversal in OpenMP


```
#pragma omp parallel
{
    #pragma omp master
    printf("Threads:      %d\n", omp_get_num_threads());
    #pragma omp single
    {
         p=head;
         while (p)
         {
              #pragma omp task firstprivate(p)
              {
                   processwork(p);
                   printf("I am thread %d\n", omp_get_thread_num());
              }
         p = p->next;
         }
    }
}
```

- How many threads create the tasks?
- Which thread executes which tasks?
- Why do we need firstprivate?

Printing a sentence


```
#pragma omp parallel
{
  #pragma omp single
  {
    printf("A ");
    #pragma omp task
      {printf("race ");} // Task #1
    #pragma omp task
      {printf("car ");}  // Task #2
    printf("is fun to watch.\n");
  } // End of single region
} // End of parallel region
```

Text printed
A is fun to watch. race car
A is fun to watch. car race
A race is fun to watch. car
A car is fun to watch. race
A race car is fun to watch.
A car race is fun to watch.

Threads start picking up work from the queue when they get here!

Printing a sentence


```
#pragma omp parallel
{
  #pragma omp single
  {
    printf("A ");
    #pragma omp task
      {printf("race ");} // Task #1
    #pragma omp task
      {printf("car ");}  // Task #2
    #pragma omp taskwait
      printf("is fun to watch.\n");
  } // End of single region
} // End of parallel region
```

- Correction: use the `taskwait` construct to force the pending child tasks to complete before resuming execution

- Note that you could use `master` instead of `single`, but there is no barrier on exit!
- Same concern if `nowait` is used

Task synchronization

- When a thread encounters a `task` construct, the task is created but not immediately executed
- Tasks are guaranteed to be completed
  - At a barrier (implicit or explicit)
  - At task synchronization points

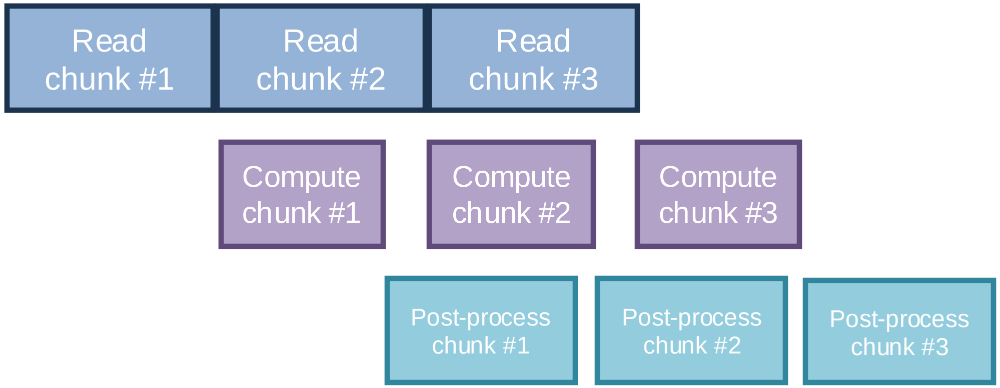
```
#pragma omp taskwait
```

```
#pragma omp taskgroup
```

- `taskwait` forces completion of child tasks only, `taskgroup` synchronizes also their descendants

Task dependences

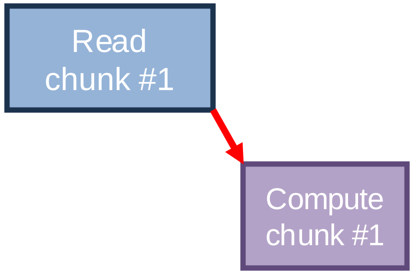
- Tasks can be used to overlap computation and I/O (pipeline parallelism)



- After start-up, the three activities can be run at the same time on different chunks of data
- The pipeline can be implemented through `sections`, but communication and load imbalance are a problem

Task dependences

- By creating tasks instead of sections, each thread can execute any task as long as its inputs are ready
- The `depend` clause indicates a dependence between tasks




```
#pragma omp task depend(out: variable)
{read;}
#pragma omp task depend(in: variable)
{compute;}
```

- The `priority` clause can be used to hint that more important tasks should be executed more frequently

Task dependences


```
#pragma omp parallel default(none) \
                  shared(fp_read, n_io_chunks, n_work_chunks) \
                  shared(a, b, c) \
                  shared(status_read, status_processing) \
                  shared(status_postprocessing)
{
  #pragma omp single nowait
  {
    for (int64_t i=0; i<n_io_chunks; i++) {

                         /* Create tasks */

    } // End of for-loop
  } // End of single region
} // End of parallel region
```

The parallel region is a single region creating tasks for the other threads

Task dependences


```
#pragma omp parallel default(none) \
                     shared(fp_read, n_io_chunks, n_work_chunks) \
                     shared(a, b, c) \
                     shared(status_read, status_processing) \
                     shared(status_postprocessing)
{
  #pragma omp single nowait
  {
    for (int64_t i=0; i<n_io_chunks; i++) {

            /* Create tasks */

    } // End of for-loop
  } // End of single region
} // End of parallel region
```

Data environment is not trivial when using tasks

Assumption: we are going to process `n_io_chunks` of data

Task synchronization point: barrier at the end of the `parallel` region

Task dependences


```
#pragma omp task depend(out: status_read[i]) \
                priority(20)
{
   (void) read_input(fp_read, i, a, b, &status_read[i]);
} // End of task reading in a chunk of data

#pragma omp task depend(in: status_read[i]) \
                 depend(out: status_processing[i]) \
                 priority(10)
{
   (void) compute_results(i, n_work_chunks, a, b, c,
                          &status_processing[i]);
} // End of task performing the computations

#pragma omp task depend(in: status_processing[i]) \
                 priority(5)
{
   (void) postprocess_results(i, n_work_chunks, c,
                              &status_postprocessing[i]);
} // End of task postprocessing the results
```

Read -> compute dependence

Compute -> postprocess dependence

Task dependences

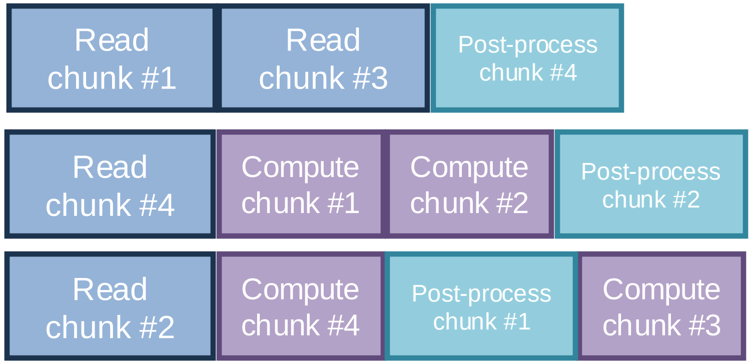

```
#pragma omp task depend(out: status_read[i]) \
              priority(20)
{
   (void) read_input(fp_read, i, a, b, &status_read[i]);
} // End of task reading in a chunk of data

#pragma omp task depend(in: status_read[i]) \
                 depend(out: status_processing[i]) \
                 priority(10)
{
   (void) compute_results(i, n_work_chunks, a, b, c,
                          &status_processing[i]);
} // End of task performing the computations

#pragma omp task depend(in: status_processing[i]) \
                 priority(5)
{
   (void) postprocess_results(i, n_work_chunks, c,
                              &status_postprocessing[i]);
} // End of task postprocessing the results
```

Before running the program, `OMP_MAX_TASK_PRIORITY` has to be set (default is 0)

Numbers are relative, the execution must not depend on them

Task dependences

- Possible execution order with 3 threads



- As long as dependences are respected, the loop iterations can be executed in an arbitrary order
- No explicit `flush` required (it is implied before and after every task)
- Task scheduling overhead might be a problem

Tasks and loops

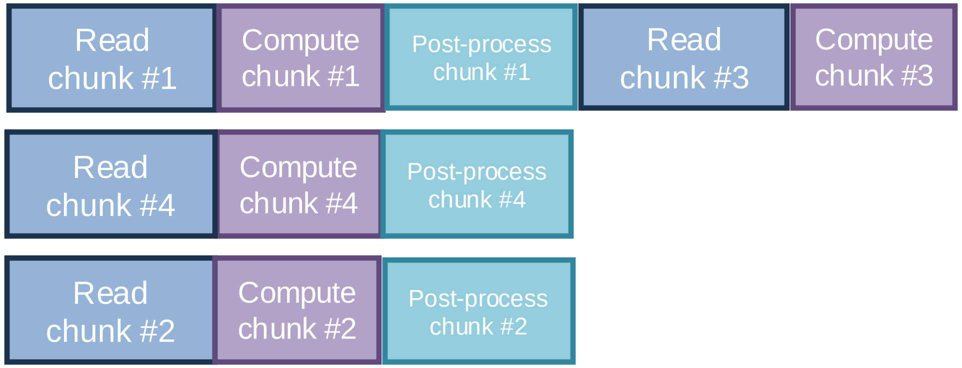
- Covenience construct to simplify using tasks when there is a loop in the code

```
#pragma omp taskloop num_tasks(n_io_chunks/10) grainsize(50)
for (int64_t i=0; i<n_io_chunks; i++) {
        (void) read_input(fp_read, i, a, b);
        (void) compute_results(i, n_work_chunks, a, b, c);
        (void) postprocess_results(i, n_work_chunks, c);
} // End of for-loop
```

- To keep the number of created tasks low, `num_tasks` sets the number of tasks that the runtime system can generate
- Each task gets assigned a number of iterations which is the minimum between `grainsize` and the total number of iterations

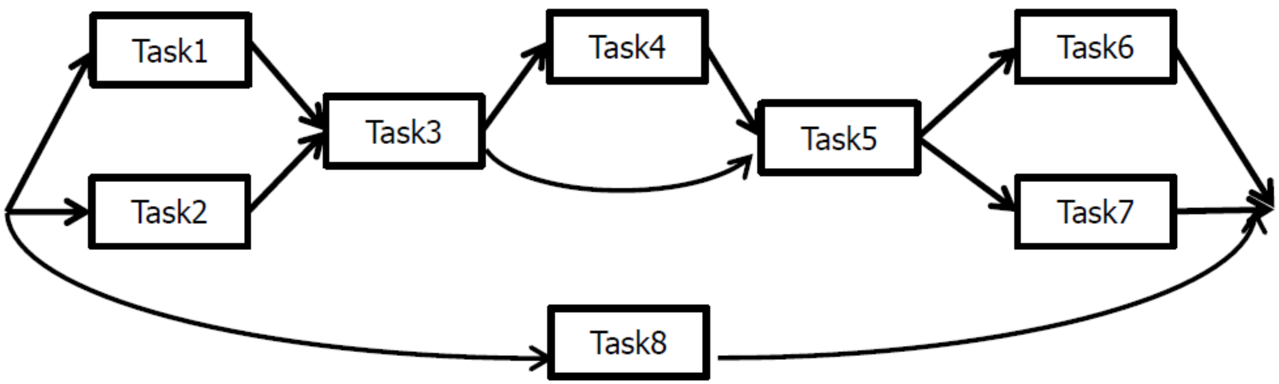
Tasks and loops

- Each iteration became a task
- Dependences are no longer needed



- Larger units of work are scheduled, lower overhead, but less flexibility to handle load imbalance

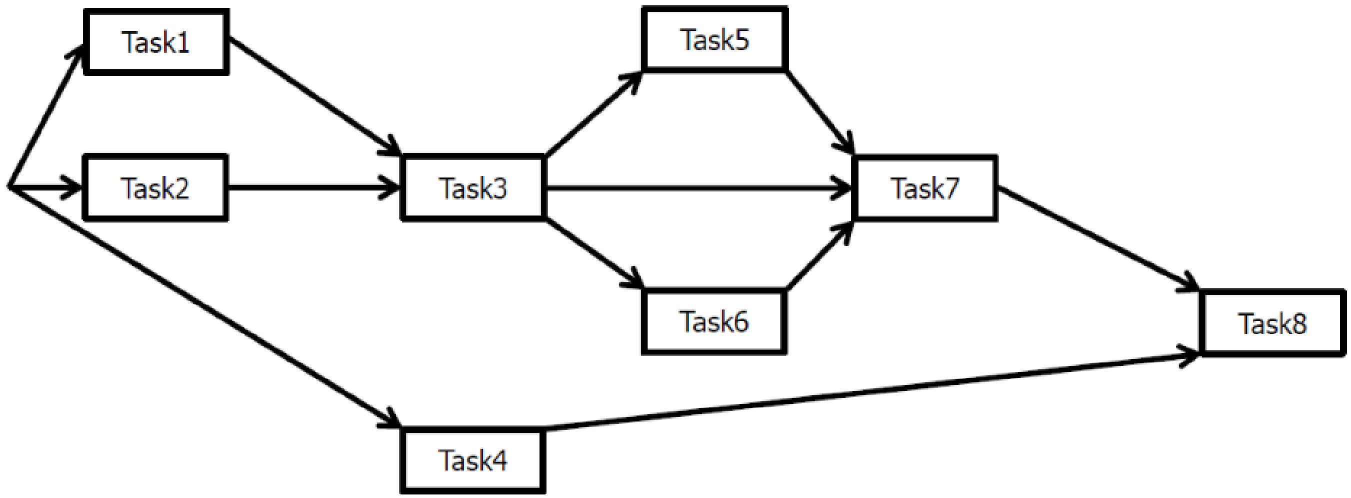
Task Graph Exercises/4



W(T1)=100
W(T2)=100
W(T3)=75
W(T4)=50
W(T5)=75
W(T6)=100
W(T7)=100
W(T8)=200

- Calculate work and span.
- Write an OpenMP implementation reflecting the structure of the task graph.
- How many threads are needed to achieve the maximum theoretical parallelism?

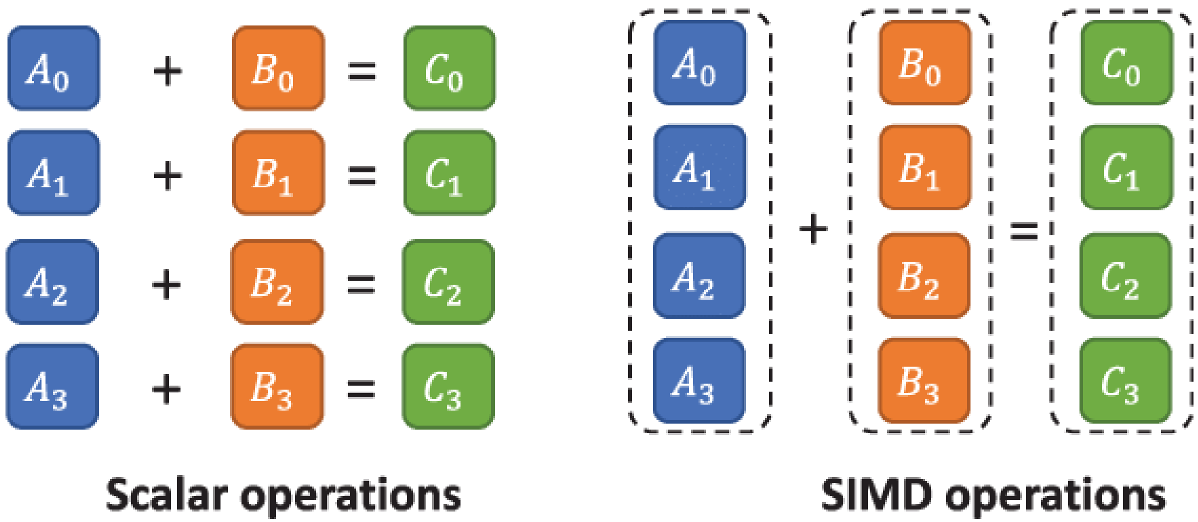
Task Graph Exercises/5



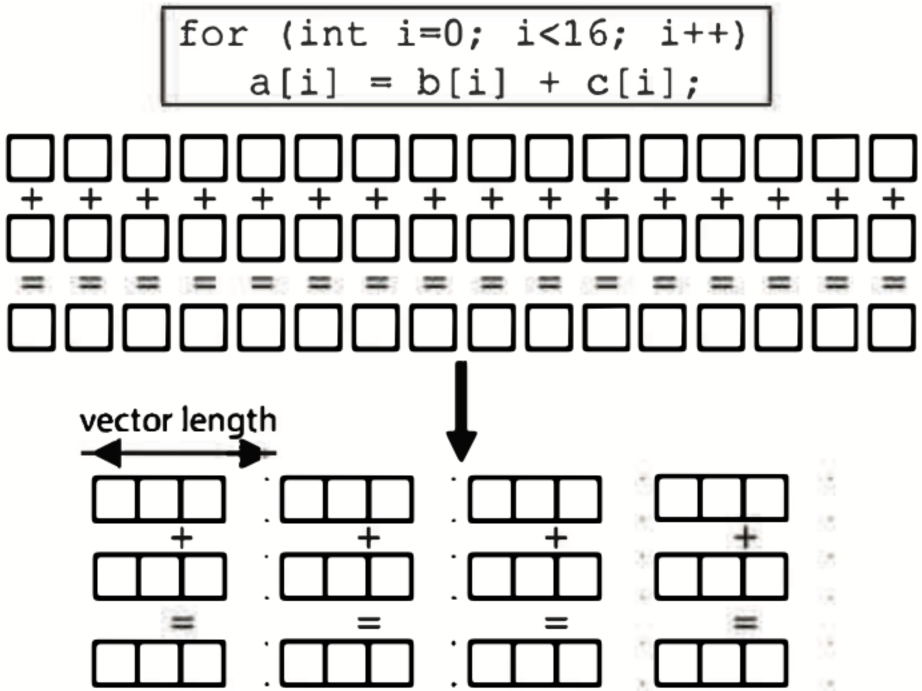
W(T1)=50
W(T2)=50
W(T3)=50
W(T4)=75
W(T5)=75
W(T6)=100
W(T7)=100
W(T8)=200

- Calculate work and span.
- Write an OpenMP implementation reflecting the structure of the task graph.
- How many threads are active during the execution of Task 3? How many during Task 5?

- A SIMD processor exploits data parallelism by providing instructions that operate on blocks of data (vectors)
- SIMD provides data parallelism at the *instruction* level, it can be combined with other OpenMP constructs to achieve multi-level parallelism



SIMD Vectorization



32 loads
16 adds
16 stores

8 loads
4 adds
4 stores

- SIMD instructions use SIMD registers
- Width of the register → vector length
- Similar latency as scalar instructions

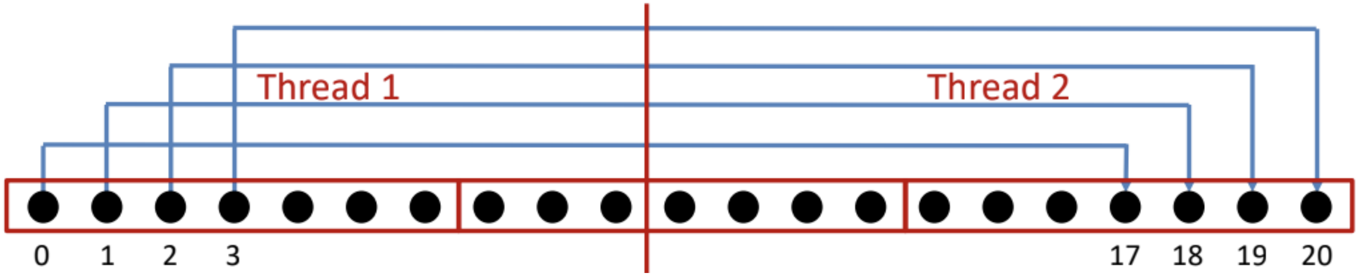
SIMD Vectorization

- Compilers deal with multiple issues to identify whether a loop can be vectorized through SIMD instructions
  - analysis of dependences across iterations
  - alias analysis of pointers
  - data layout/alignment issues
  - conditional execution
  - loop bounds not multiple of vector length
- Loop iterations at the beginning and end may not be vectorized (loop *peeling*, *tail*)

Loop-Carried dependencies

```
void lcd_ex(float* a, float* b, size_t n,
float c1, float c2) {
  for (int i = 0; i < n; i++) {
    a[i] = c1 * a[i + 17] + c2 * b[i];
  }
}
```

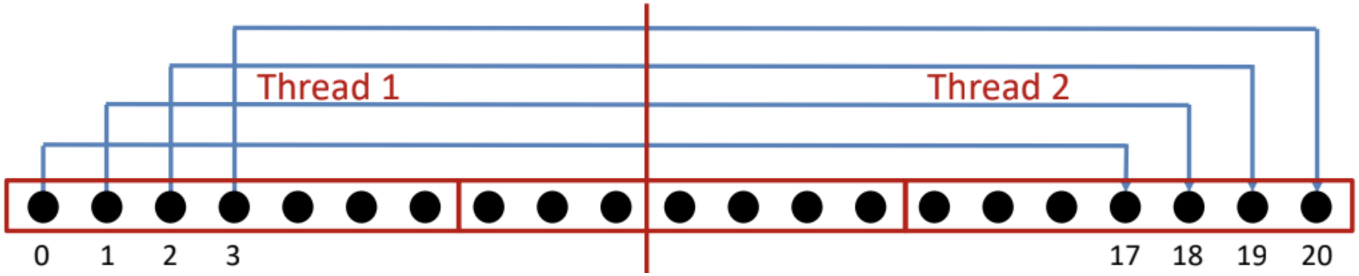
- Can this loop be parallelized?
  - Test: can you reverse the loop and still get correct results?



Loop-Carried dependencies


```
void lcd_ex(float* a, float* b, size_t n,
float c1, float c2) {
  for (int i = 0; i < n; i++) {
    a[i] = c1 * a[i + 17] + c2 * b[i];
  }
}
```

- Can this loop be vectorized?
  - Yes, *only* if vector length < distance length



OpenMP SIMD Vectorization

```
#pragma omp simd
        /* for loop */
```

- The loop is divided into chunks, all iterations are executed *by a single thread* with SIMD vector instructions
  - Chunks should fit a vector register for performance
  - Each iteration is executed by a SIMD *lane*
- The compiler will generate SIMD instructions, it is up to the user to ensure this maintains correct behavior

OpenMP SIMD Vectorization

- Data scope clauses (`private`, `firstprivate`, `reduction`, etc.) can be used in a `simd` directive
- A `collapse` clause can be used to fuse two perfectly nested loops (watch out for complexity!)
- The `simdlen(size)` clause suggests a preferred vector length
    - Maybe the code will work better with a specific vector length
    - The compiler is free to ignore it
    - It can hurt performance but the results remain correct

OpenMP SIMD Vectorization

- In the case of loop-carried dependencies, the vector length must be smaller than the smallest *dependence distance* in the loop
- The `safelen` clause sets an upper limit to the vector length that the compiler cannot exceed

```
void simd_loop_safelen(double *a, double *b, double *c, int n,
                       int offset)
{
  int i;
  #pragma omp simd safelen(16)
  for (i=offset; i<n; i++)
    a[i] = b[i-offset] + c[i];
}
```

OpenMP SIMD Vectorization

- Composite loop-SIMD work-sharing construct:

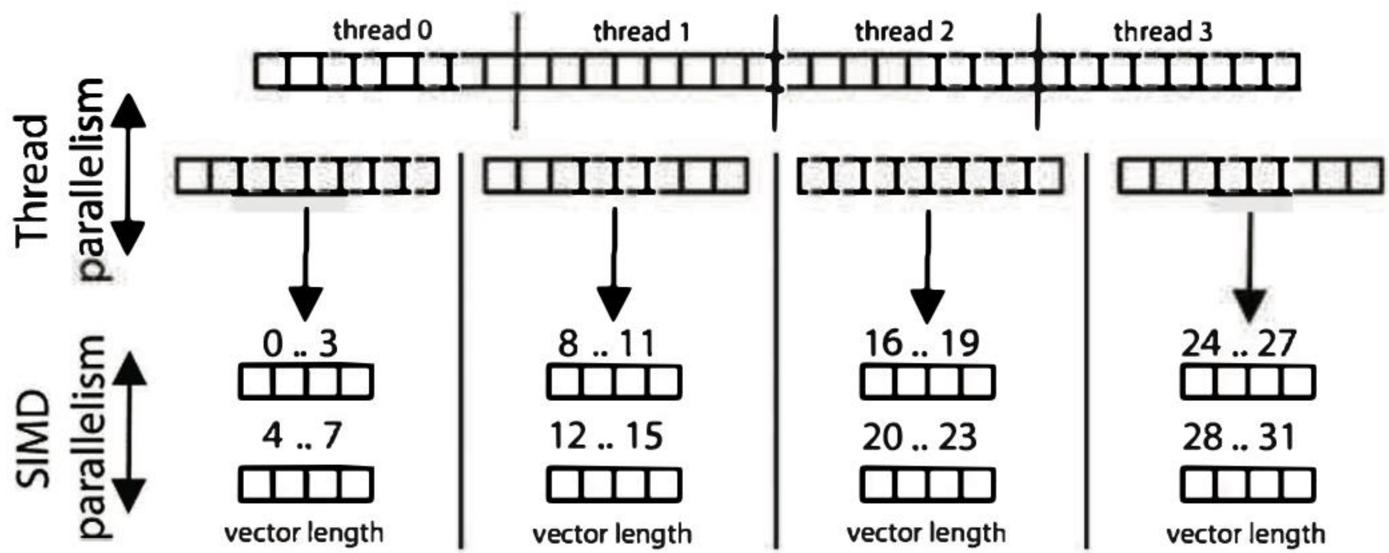
```
#pragma omp for simd
        /* for loop */
```

- Portable implementation
- Number of threads and scheduling policy greatly affect performance
  - If the number of threads increases, work for each thread is smaller
  - Each thread should work with a chunk corresponding to the vector length

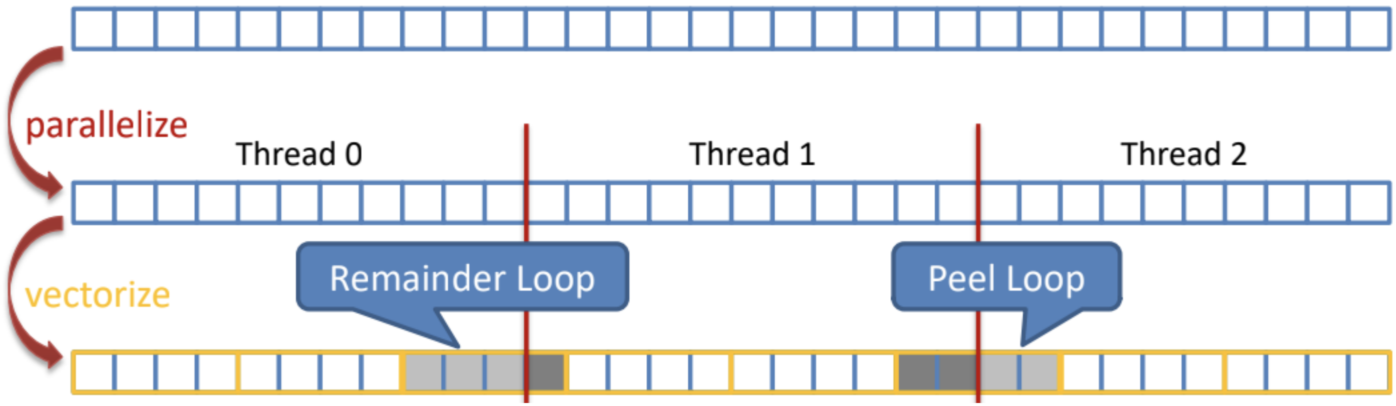
OpenMP SIMD Vectorization

- Distribute iterations among threads in a team, then each thread uses SIMD instructions

```
#pragma omp for simd
for (int i=0; i<32; i++)
    a[i] += 1;
```



OpenMP SIMD Vectorization



- To avoid performance degradation, the `simd:` modifier can be added to the scheduling directive

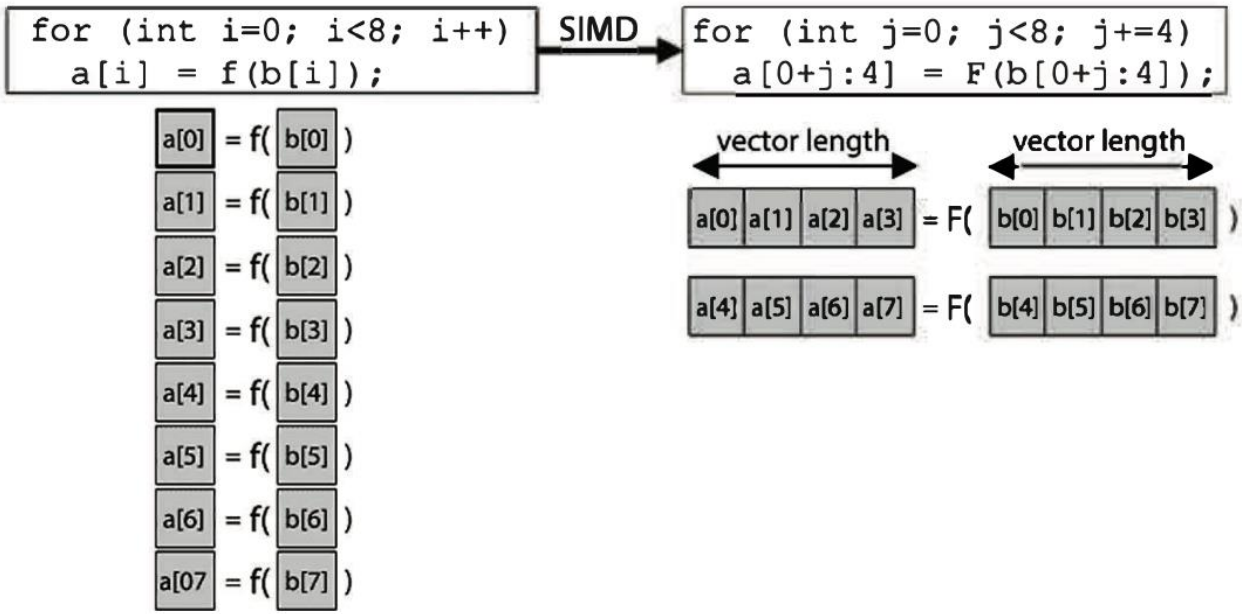
```
#pragma omp for simd schedule(simd:static, 5)
```

*chunk_size = ceil(req_size/simd_len) * simd_len*

SIMD Functions

- Declare a function to be compiled for calls within a SIMD loop

```
#pragma omp declare simd
        /* function definition */
```



SIMD Functions

- Multiple directives can be added to generate multiple compiled versions

value changes linearly
from one call to the other

same value
for all calls

```
#pragma omp declare simd linear(pixel) uniform(mask) inbranch
#pragma omp declare simd linear(pixel) notinbranch
#pragma omp declare simd
extern void compute_pixel(char *pixel, char mask);
```

is always/never called
within a branch

Credits & References

- *OpenMP Tutorial* by Blaise Barney, Lawrence Livermore National Laboratory https://hpc.llnl.gov/training/tutorials
- UC Berkeley CS267: *Applications of Parallel Computers* https://sites.google.com/lbl.gov/cs267-spr2020
- Video series from Tim Mattson (Intel) Introduction to OpenMP
- Ruud van der Pas, Eric Stotzer, and Christian Terboven, "Using OpenMP-The Next Step. Affinity, Accelerators, Tasking, and SIMD", MIT Press 2017