Introduction to Parallel & Distributed Computing

CnC Tutorial

Lecture 7, Spring 2014

Instructor: 罗国杰

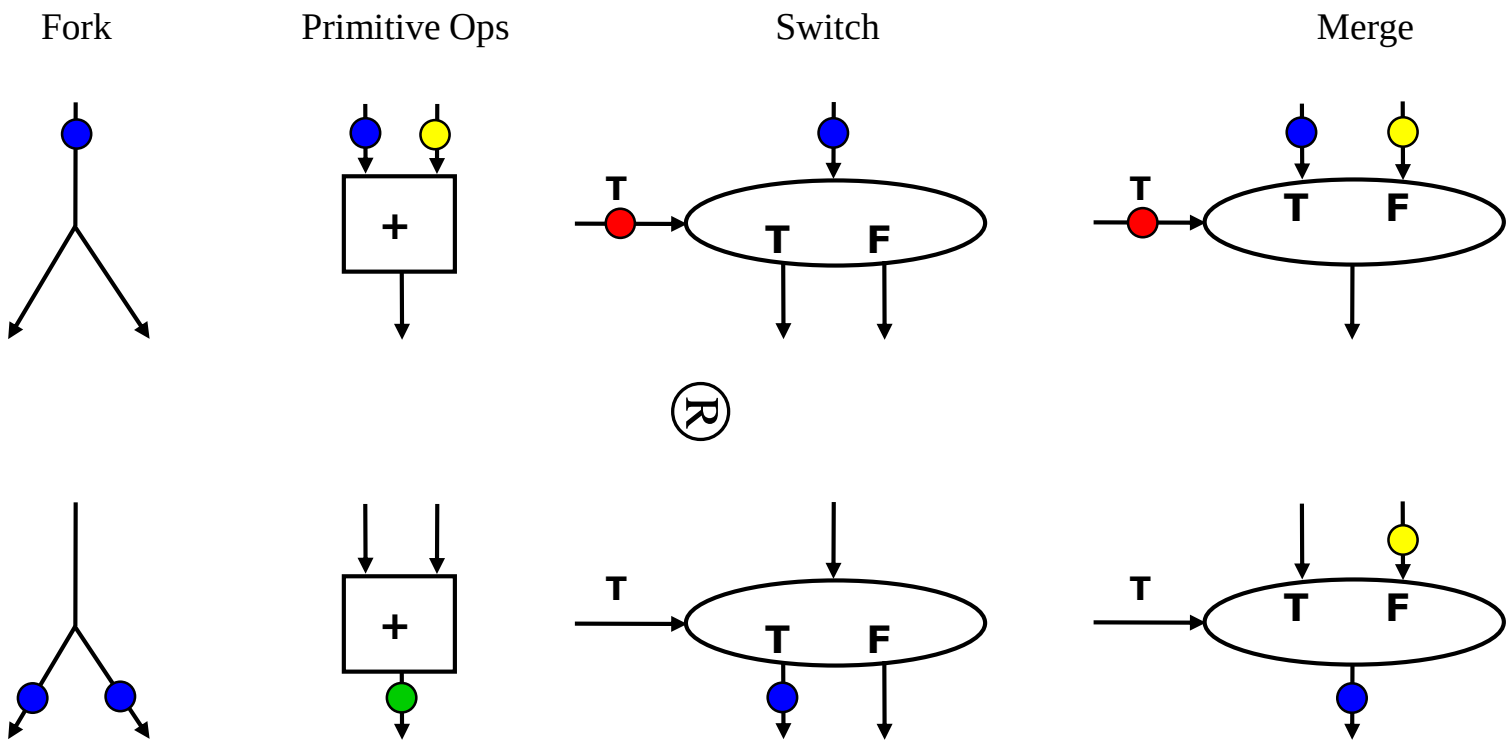
gluo@pku.edu.cn

Outline

- **Dataflow computing and Data-driven Tasks**
- **Concurrent Collections (CnC) Macro-Dataflow Programming Model**

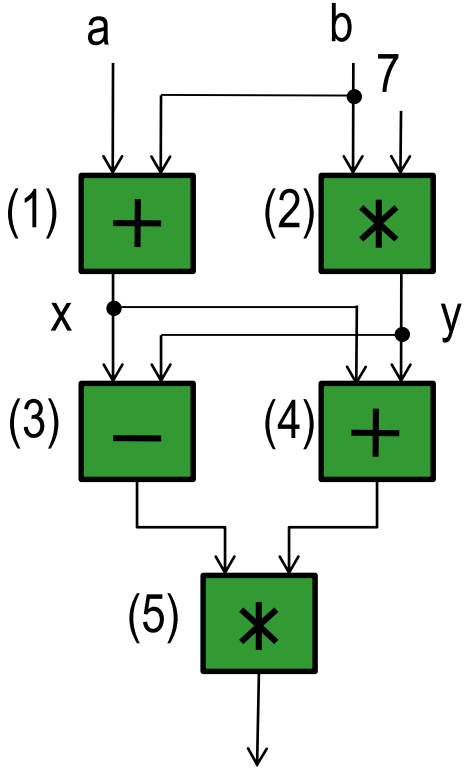
Dataflow Computing

- **Original idea: replace machine instructions by a small set of dataflow operators**



Example instruction sequence and its dataflow graph

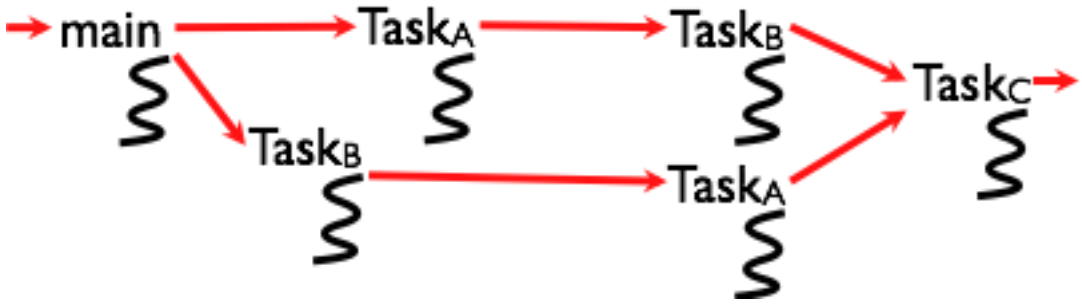
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



An operator executes when all its input values are present; copies of the result value are distributed to the destination operators.

No separate control flow

Macro-Dataflow Programming



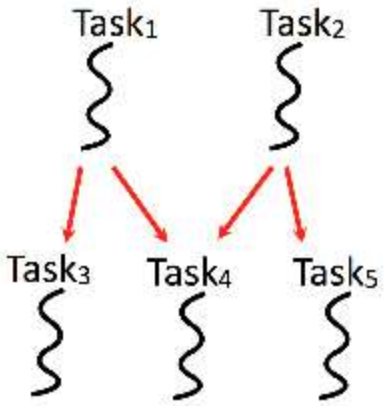
Communication via single-assignment variable

- "Macro-dataflow"
  - extension of dataflow model from instruction-level to task-level operations
- General idea:
  - build an arbitrary task graph, but restrict all inter-task communications to single-assignment variables
- Static dataflow ==> graph fixed when program execution starts
- Dynamic dataflow ==> graph can grow dynamically
- Semantic guarantees: race-freedom, determinism
- Deadlocks are possible due to unavailable inputs (but they are deterministic)

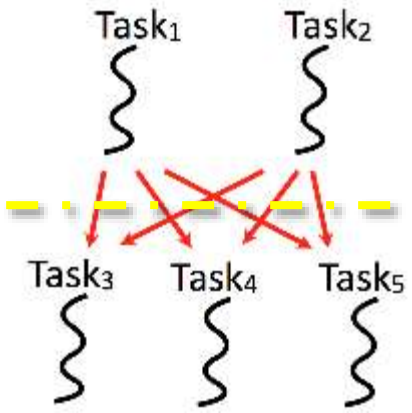
Implementing macro-dataflow in OpenMP can reduce parallelism

```
1. #pragma omp parallel
2. #pragma omp single
3. {
4.    #pragma omp task
5.    left = put(leftWriter()); // Task1
6.    #pragma omp task
7.    right = put(rightWriter()); // Task2
8. }
9. #pragma omp parallel
10.#pragma omp single
11.{
12.    #pragma omp task
13.   leftReader(left); // Task3
14.   #pragma omp task
15.   rightReader(right); // Task5
16.   #pragma omp task
17.   bothReader(left, right); // Task4
18.}
```

What we want:



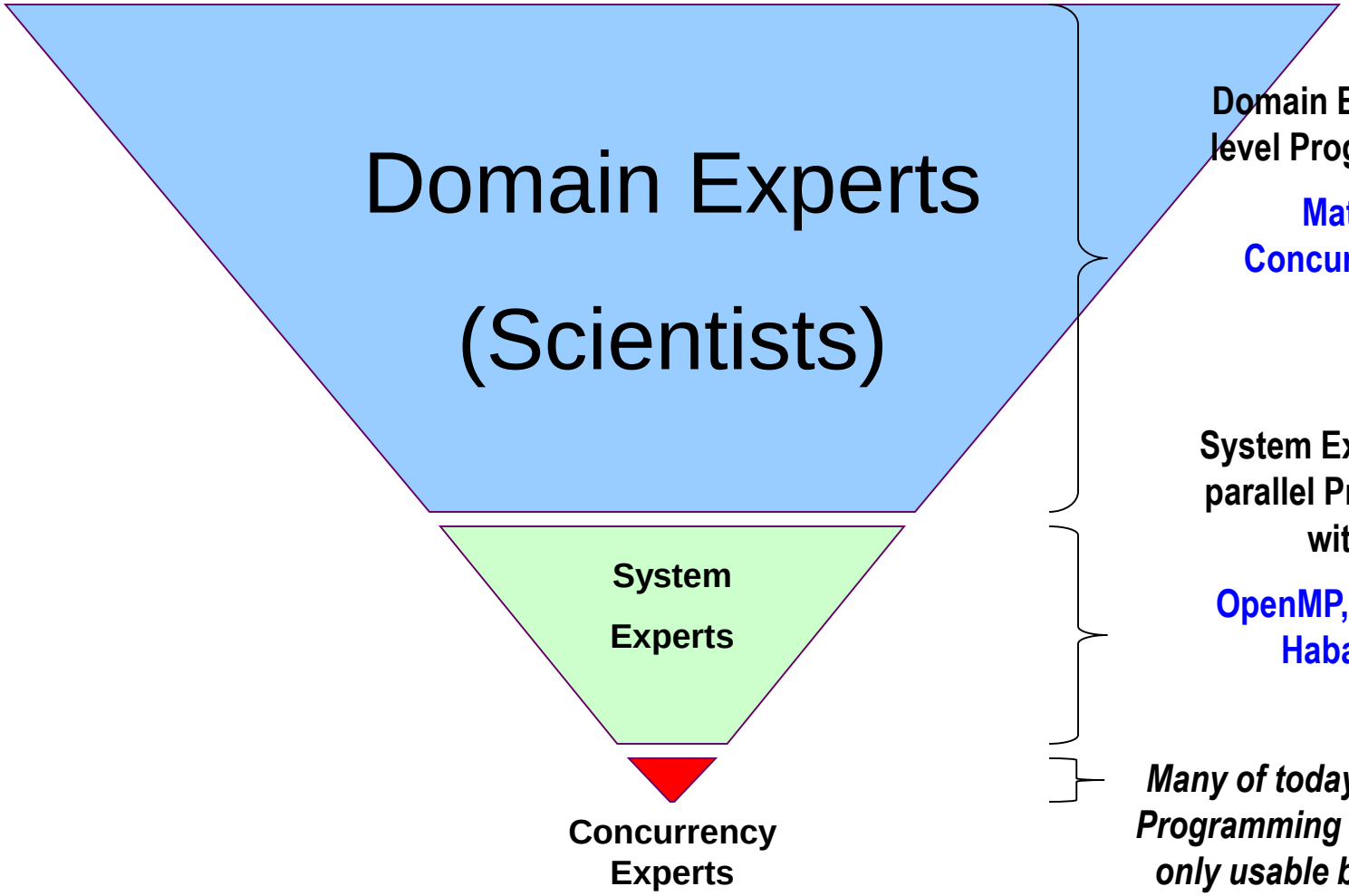
What we get:



Outline

- **Dataflow computing and Data-driven Tasks**
- **Concurrent Collections (CnC) Macro-Dataflow Programming Model**

Parallel Software Challenge & Inverted Pyramid of Parallel Programming Skills and Software Development



Domain Experts need high level Programming Models

Matlab, Python, Concurrent Collections (CnC), …

System Experts need portable parallel Programming Models with safety nets

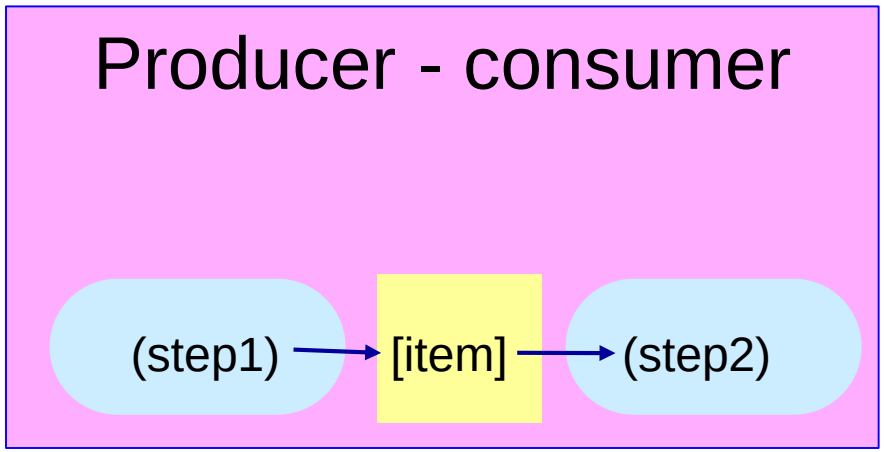
OpenMP, Cilk, Habanero-C, Habanero-Java, …

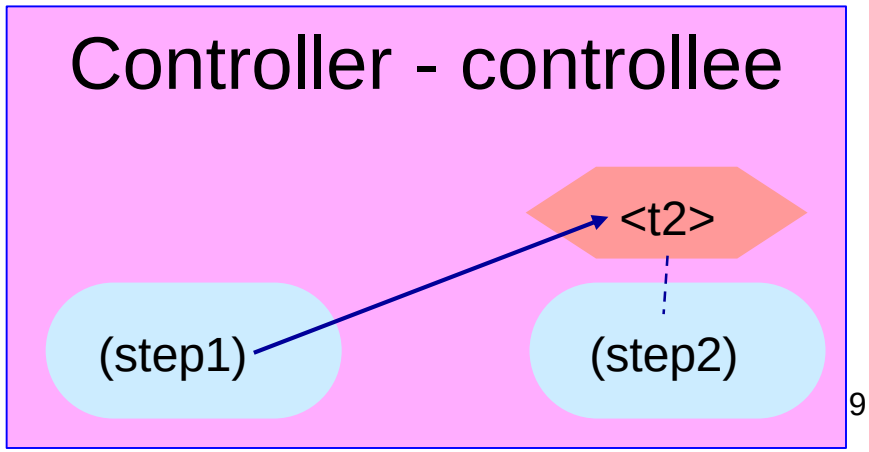
Many of today's Parallel Programming Models are only usable by experts

Pthreads, CUDA, OpenCL, MPI, ….

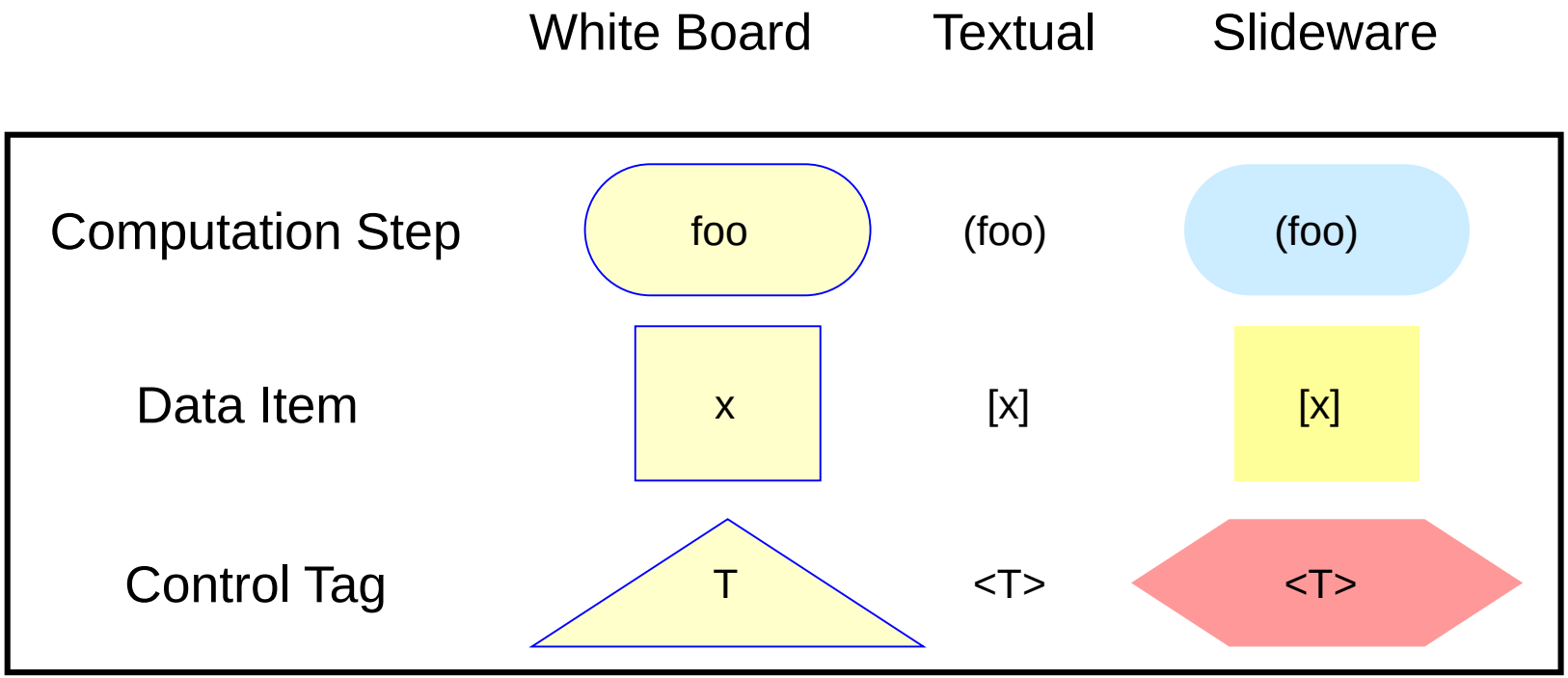
The problem for Domain Experts

- **Most serial languages over-constrain orderings**
  - **Require arbitrary serialization**
  - **Allow for overwriting of data**
  - **The decision of *if* and *when* to execute are bound together**
  - **This makes parallel programming hard**
- **Concurrent Collections (CnC) Approach: introduce a *coordination language* that specifies only the semantic ordering constraints**
  - **Producer-consumer ordering constraints (data dependence)**
  - **Controller-controllee ordering (control dependence)**





Notation


White Board
Textual
Slideware
Computation Step
foo
(foo)
(foo)
Data Item
x
[x]
[x]
Control Tag
T
<T>
<T>

Producer-Consumer Relationship in CnC: Steps and Data Items

Key can be any value (numeric, string, …) that supports equality comparison

Item can be any immutable data structure

Two **get**'s with the same key must return identical items

Single assignment rule

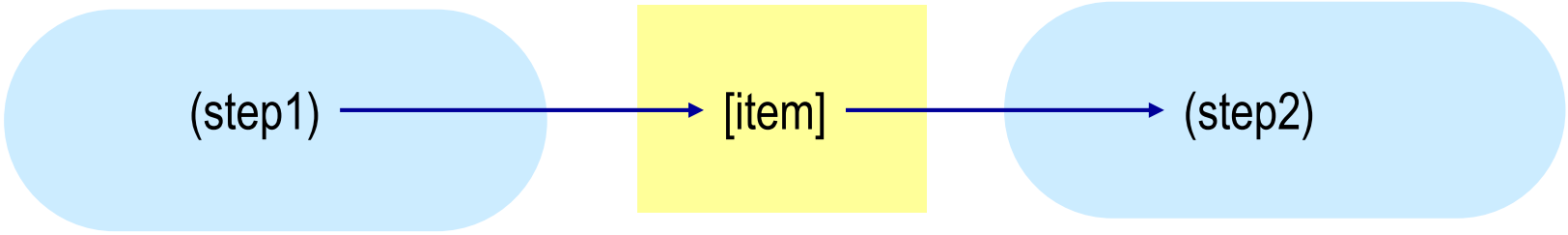
At most one **put** permitted with a given key value; an exception is thrown if a second **put** is attempted with the same key value

Blocking **get**'s

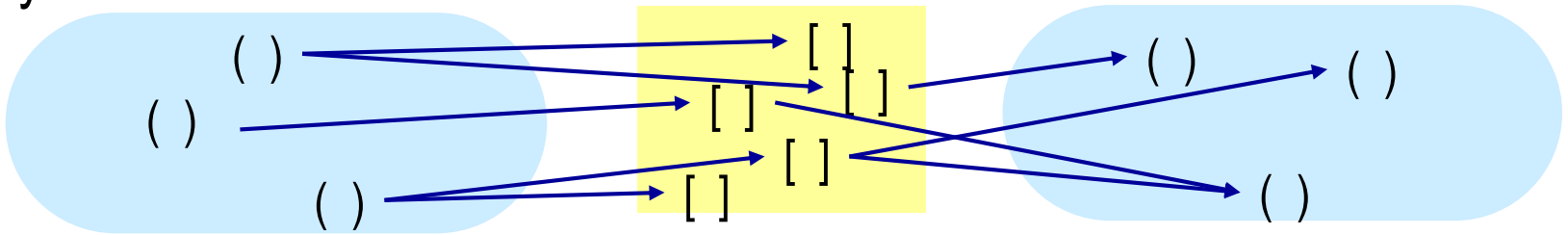
A **get** operation blocks if no item is present with the given key, and is unblocked when a matching **put** is performed

Static vs. Dynamic Instances of Steps and Items

Static

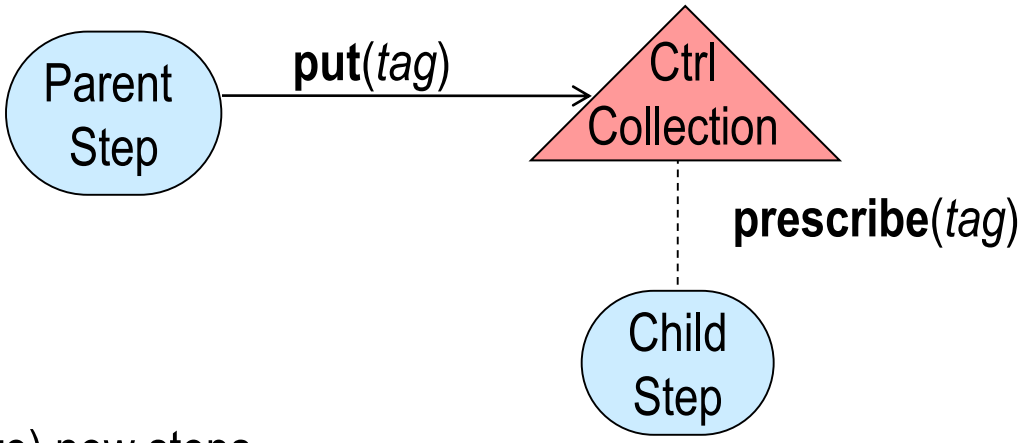


Dynamic



A step instance may produce multiple item instances
A step instance may consume multiple item instances
Dynamic single assignment: each item instance is produced once.

Creating new steps in CnC: Steps and Control Tags



Control collection

Its role is to prescribe (create) new steps

Like task creation in OpenMP or Habanero-C

Tag can be any key value (numeric, string, …) that supports equality comparison

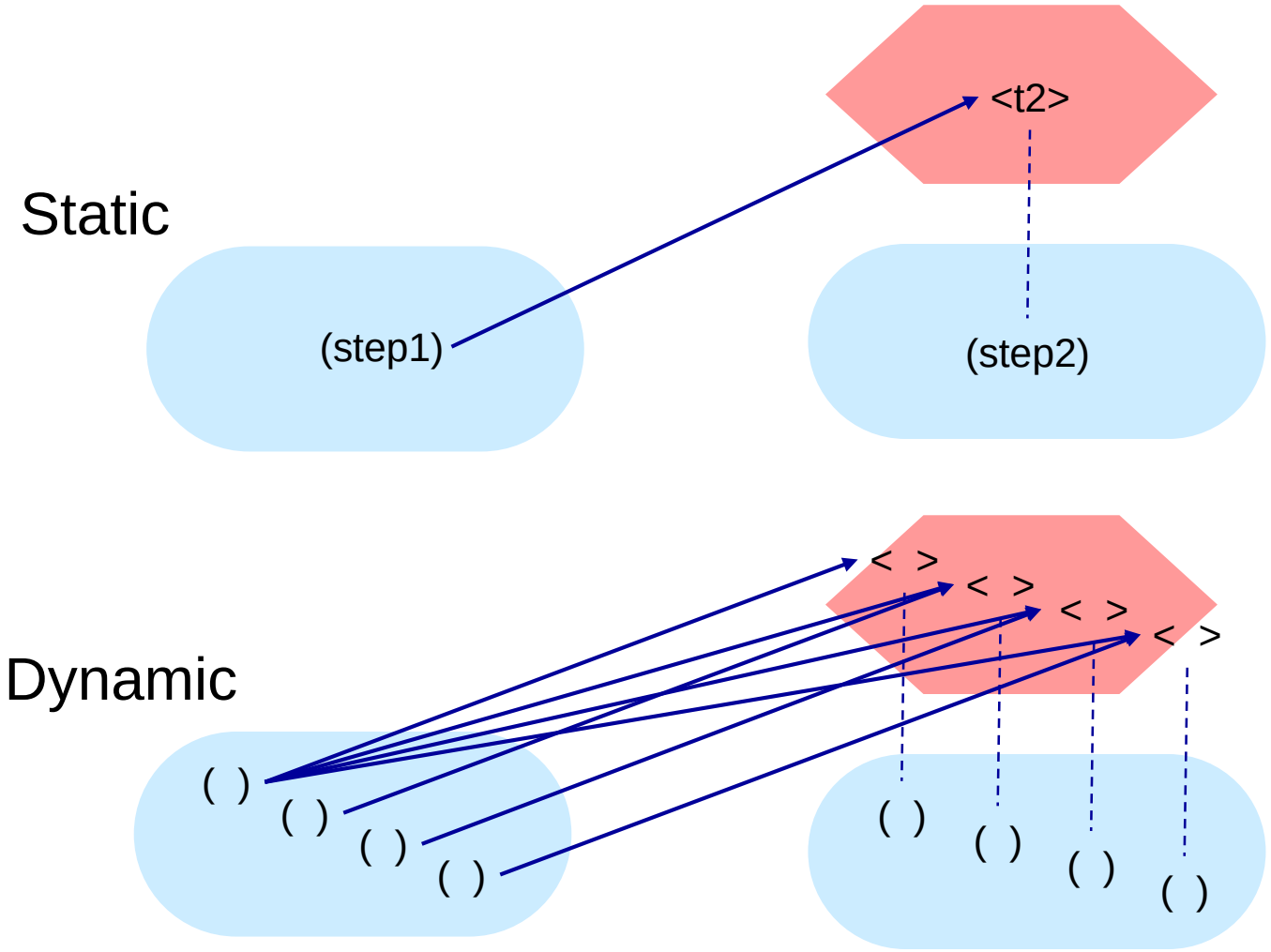
Single assignment rule

At most one **put** permitted with a given tag value; an exception is thrown if a second **put** is attempted with the same tag value

Step prescription

Runtime system guarantees that **prescribe** operation is performed eventually on child step for each tag in tag collection

Static vs. Dynamic Instances of Steps and Control Tags



Tag, Item, Step

- **Constraints among collections (the block diagram)**
  - **control/data flow among compute steps**
- **Constraints among instances (runtime)**
  - **dependency between data instances and step instances during runtime (static or dynamic)**

	Collection	Instance
Control	tag_collection<Tag, Tuner>	Tag instance
Data	item_collection<Tag, Item, Tuner>	Item instance
Compute	step_collection<UserStep, Tuner>	UserStep instance

CnC is a Coordination Language: paired with a computation language

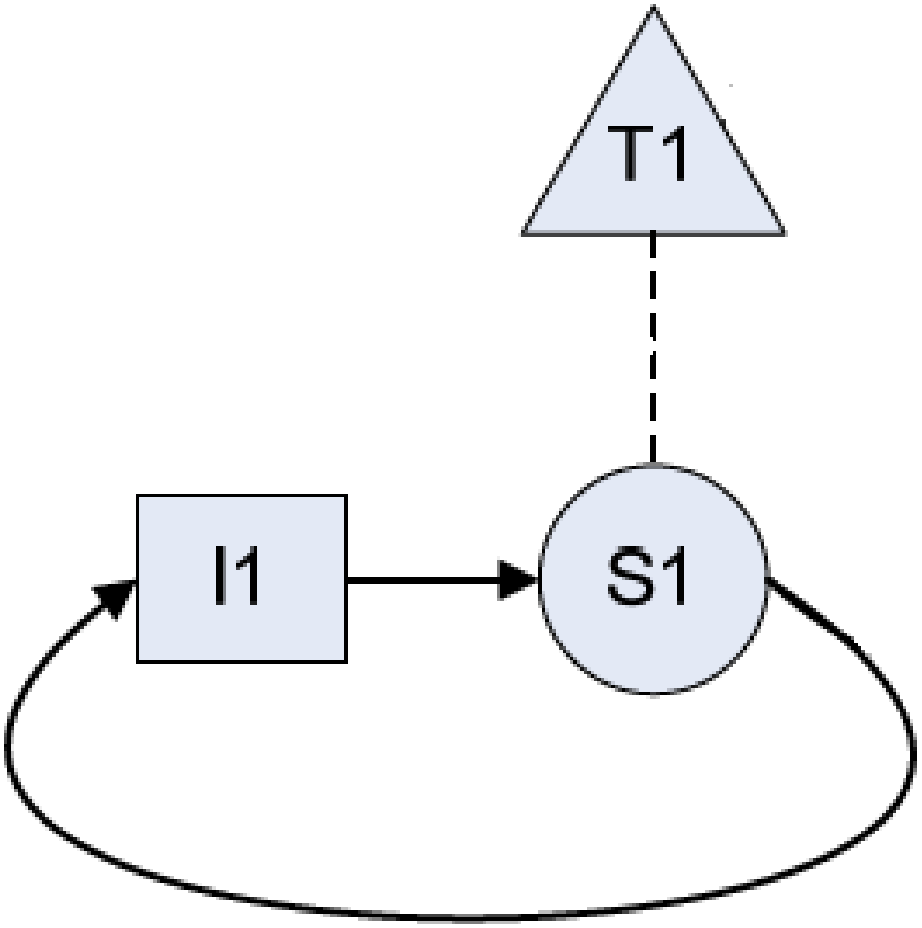
- **Existing**
  - C++ (Intel)
  - Java (Rice)
  - Haskell (Indiana)
  - Python (Rice)
  - Scala (Rice)
  - APL (Indiana)
- **In the works**
  - Fortran (Rice)
  - Chapel (UIUC)
- **Next up**
  - Matlab (Rice)

Rice = Vivek Sarkar's group
Indiana = Ryan Newton's group
UIUC = David Padua's group

Possible Expressions of the Model

- **Graphical interface**
- **Textual representation**
  - **Rice version**
  - **Pervious Intel versions**
- **API**
  - **Current Intel release**

Example: Fibonacci



Example: Fibonacci

```
struct fib_context : public CnC::context< fib_context >
{
   // the step collection for the instances of the compute-kernel
   CnC::step_collection< fib_step >       m_steps;
   // item collection holding the fib number(s)
   CnC::item_collection< int, fib_type >  m_fibs;
   // tag collection to control steps
   CnC::tag_collection< int >             m_tags;

   // constructor
   fib_context();
};
```

Example: Fibonacci

```
fib_context::fib_context()
        : CnC::context< fib_context >(),
          // pass context to collection constructors
          m_steps( *this ),
          m_fibs( *this ),
          m_tags( *this )
{
   // prescribe compute steps with this (context) as argument
   m_tags.prescribes( m_steps, *this );
   // step consumes m_fibs
   m_steps.consumes( m_fibs );
   // step also produces m_fibs
   m_steps.produces( m_fibs );
}
```

Example: Fibonacci

```
struct fib_step {
   // declaration of execute method goes here
   int execute( const int & tag, fib_context & c ) const;
};

int fib_step::execute( const int & tag, fib_context & ctxt ) const {
   switch( tag ) {
      case 0 : ctxt.m_fibs.put( tag, 0 ); break;
      case 1 : ctxt.m_fibs.put( tag, 1 ); break;
      default :
         // get previous 2 results
         fib_type f_1; ctxt.m_fibs.get( tag - 1, f_1 );
         fib_type f_2; ctxt.m_fibs.get( tag - 2, f_2 );
         // put our result
         ctxt.m_fibs.put( tag, f_1 + f_2 );
   }
   return CnC::CNC_Success;
}
```

Example: Fibonacci

```
int main( int argc, char* argv[] ) {
    int n = 42;
    // eval command line args
    if( argc < 2 ) {
        std::cerr << "usage: " << argv[0] << " n\nUsing default value " << n << std::endl;
    } else n = atol( argv[1] );

    // create context
    fib_context ctxt;
    // put tags to initiate evaluation
    for( int i = 0; i <= n; ++i ) ctxt.m_tags.put( i );
    // wait for completion
    ctxt.wait();
    // get result
    fib_type res2;
    ctxt.m_fibs.get( n, res2 );

    // print result
    std::cout << "fib (" << n << "): " << res2 << std::endl;
    return 0;
}
```

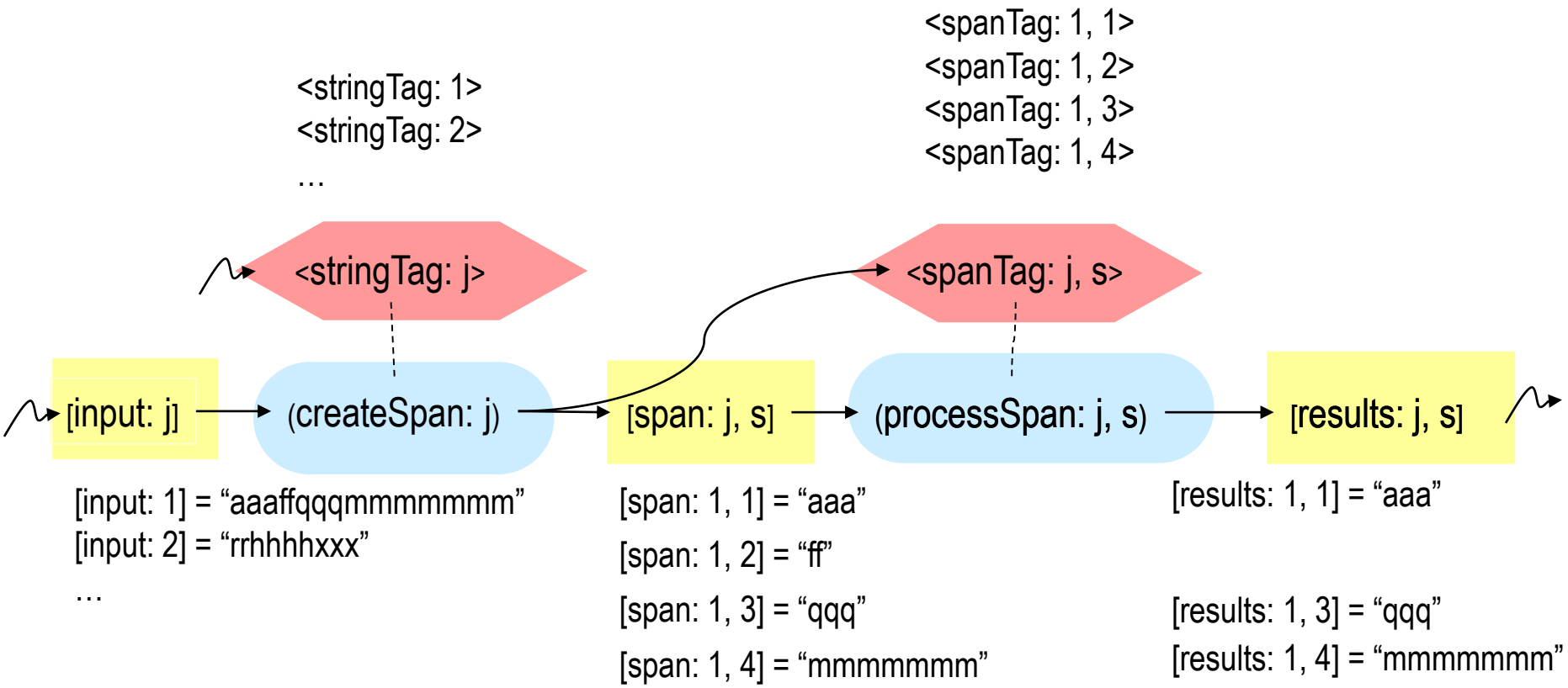
Example Program Specification

Break up an input string
- sequences of repeated single characters

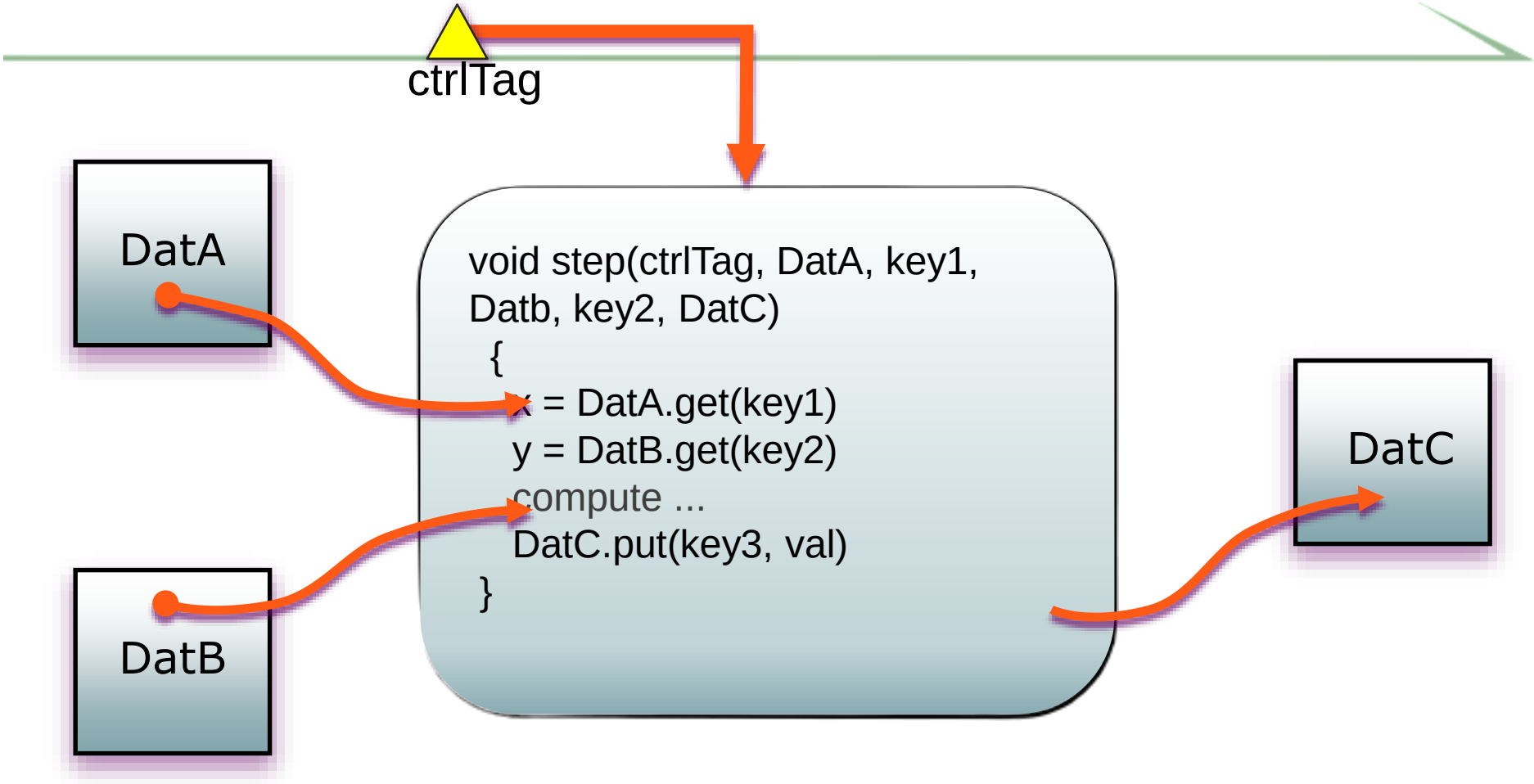
Filter allowing only
- sequences of odd length

Input string	Sequences of repeated characters	Filtered sequences
“aaaffqqqmmmmmmmm”	“aaa” “ff” “qqq” “mmmmmmm”	“aaa” “qqq” “mmmmmmm”

CnC Implementation of Example Program

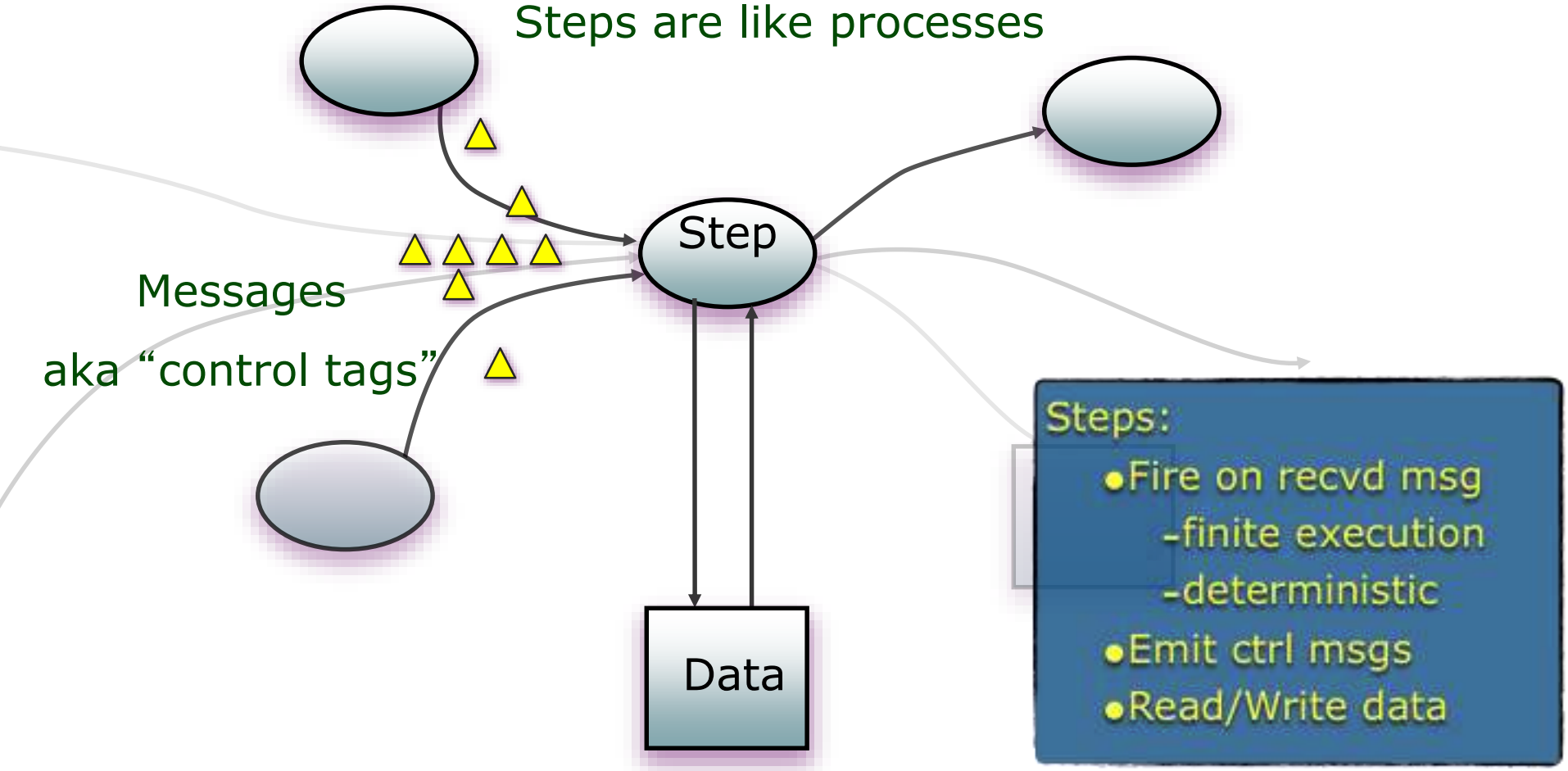


A CnC Step is Stateless



Relate to Kahn Process Networks by replacing FIFO's by associative key-value maps?

Viewing the CnC Execution Model as a Process Net



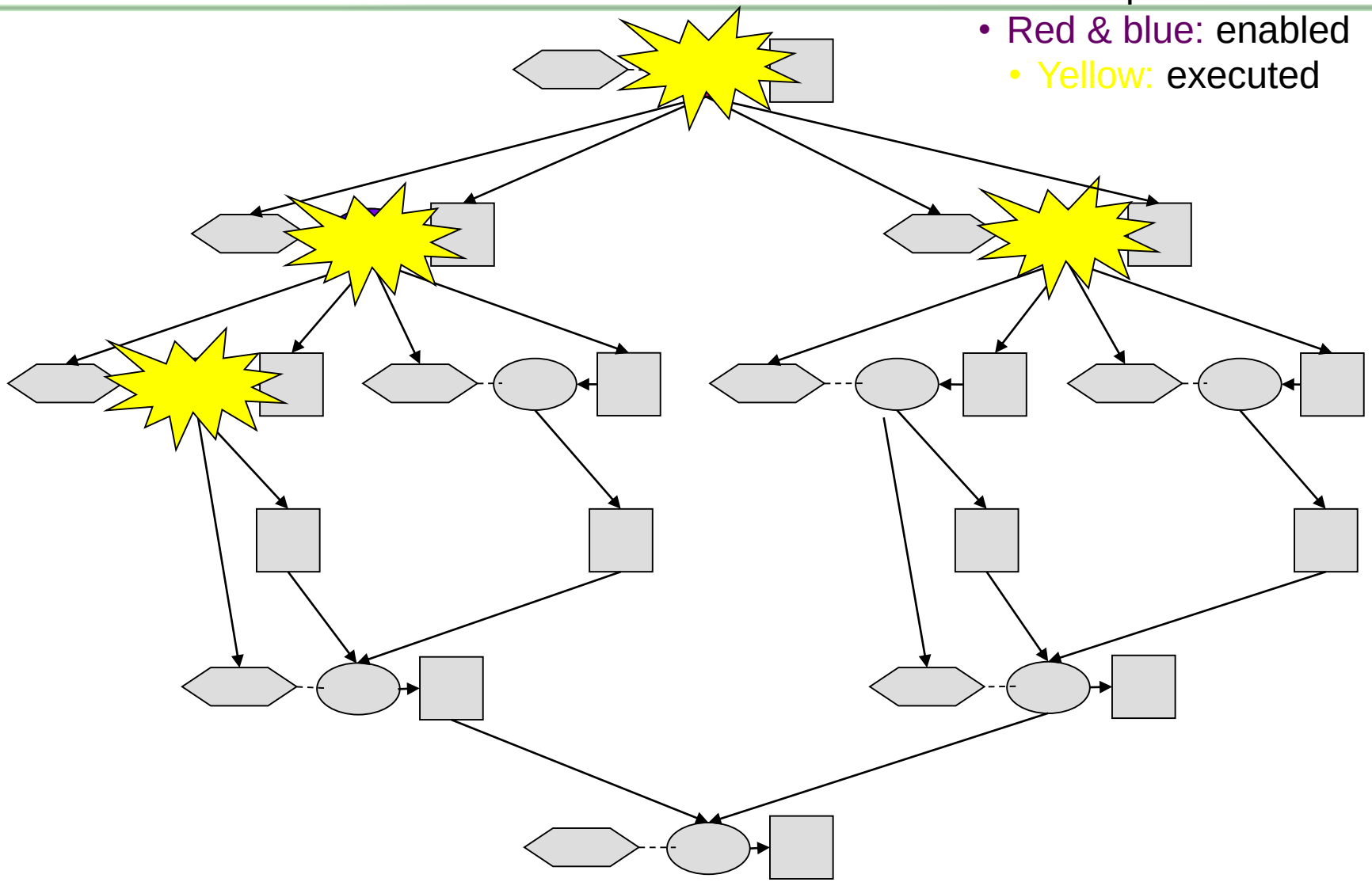
Key extension: replace local state by single assignment key/value store

Domain Expert's view of Concurrent Collections

- No thinking about parallelism
  - Only domain knowledge
- No overwriting
  - Single assignment collections
    - Can be extended with fetch-and-op & reduce operations
- No arbitrary serialization
  - only constraints on ordering via tagged puts and gets
- Result is:
  - Deterministic
  - Race-free
  - Fault-tolerant

CnC Program Execution

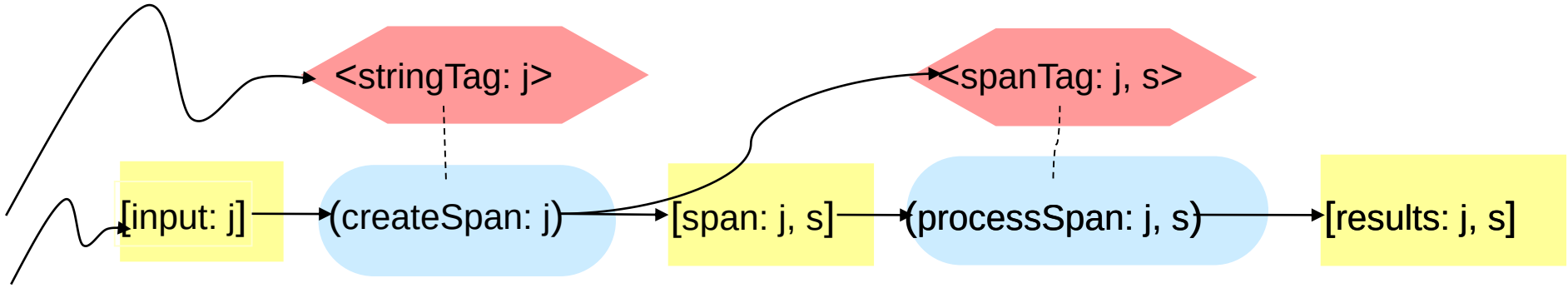
- Red: prescribed
- Blue: inputs available
- Red & blue: enabled
- Yellow: executed



Interaction of a CnC Program with its Environment: Initial State

The initial state σ_i of a CnC program is obtained by starting with σ_0, and then executing an *Environmental Input* computation, EI

- **The only CnC operations that can be performed by EI are *individual put* operations on item and tag collections**



Environmental Input computation, EI:

put([input: 1], “aaaffqqqmmmmmmmm”);

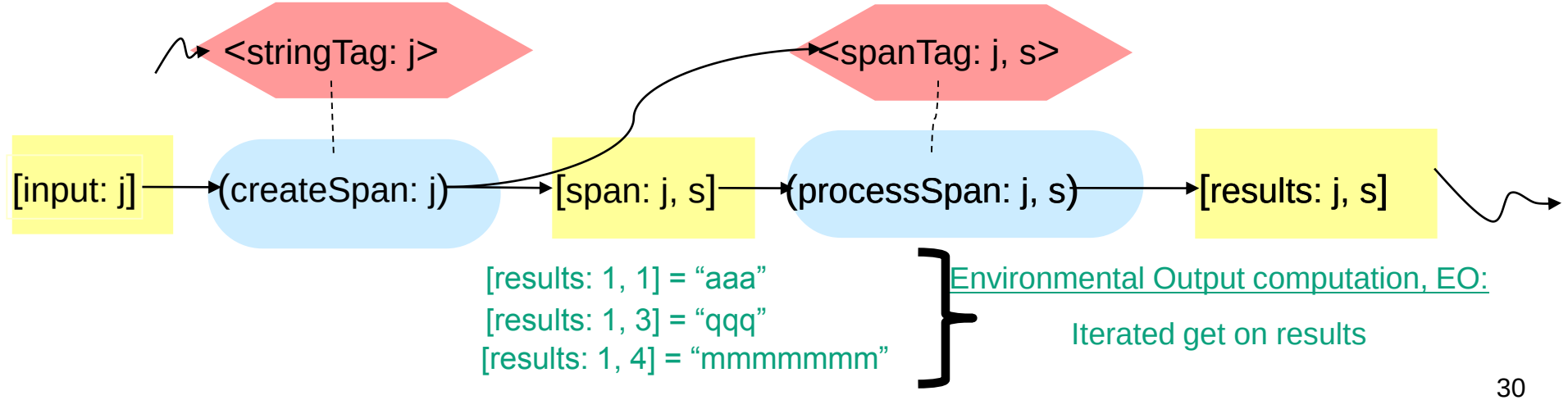
put([input: 2], “rrhhhhxxx”);

put(<stringTag: 1>);

put(<stringTag: 2>);

Interaction of a CnC Program with its Environment: Final State

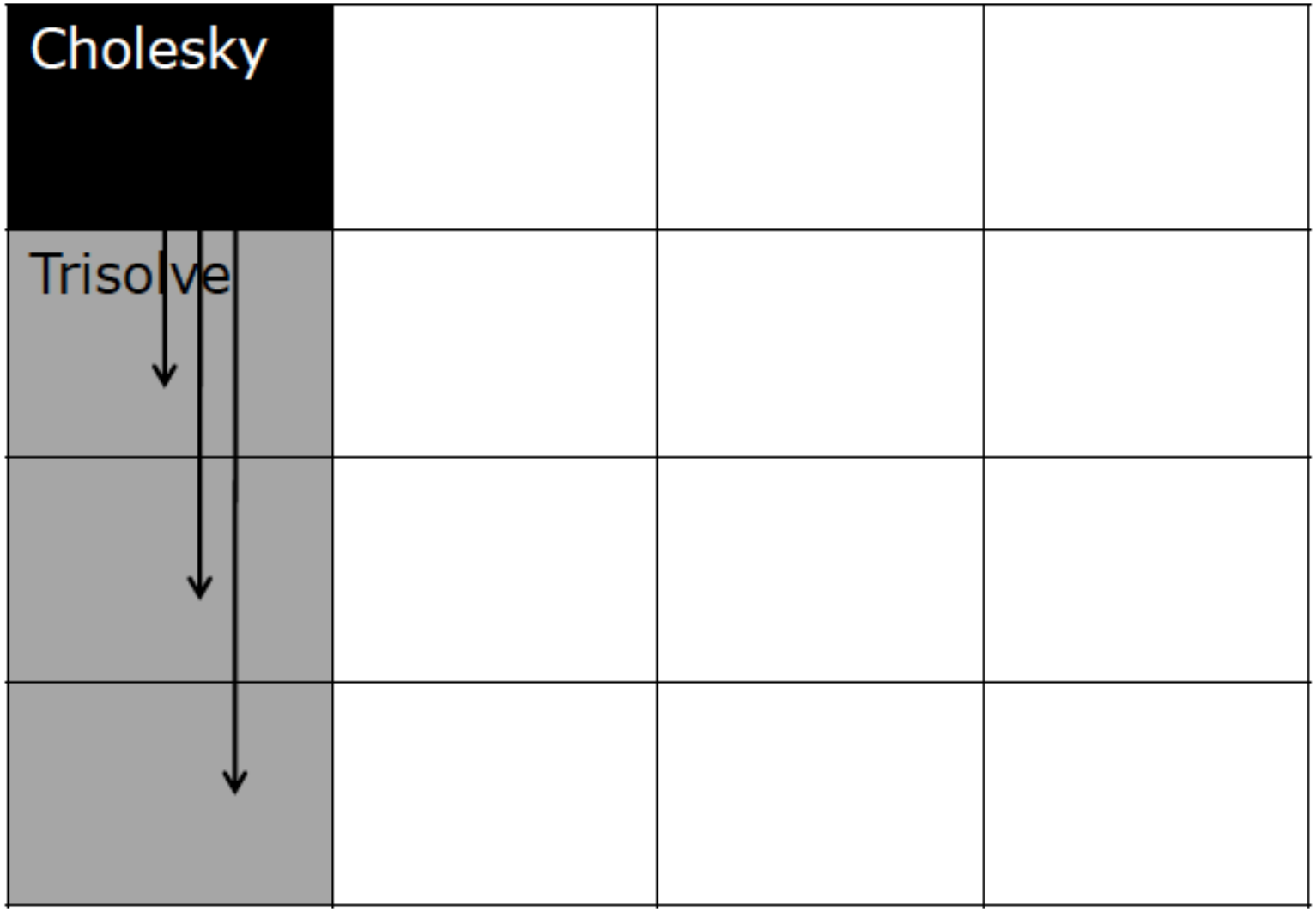
- The final state σ_f of a CnC program is examined by executing an Environmental Output computation EO that can perform the following special CnC operations:
  - *Nonblocking get operation on a specific item* [IC:T], which returns the value if [IC:T].Avail = true or an error code otherwise (instead of blocking)
  - *Iterated get operation on an item collection* IC, which returns all available items [IC:T] in IC with their values, [IC:T].Value
  - *Nonblocking get operation on a specific tag* <TC:T> that returns the value of <TC:T>.Avail
  - *Iterated get operation on a tag collection* TC, which returns the tags for all available items <TC:T> in TC



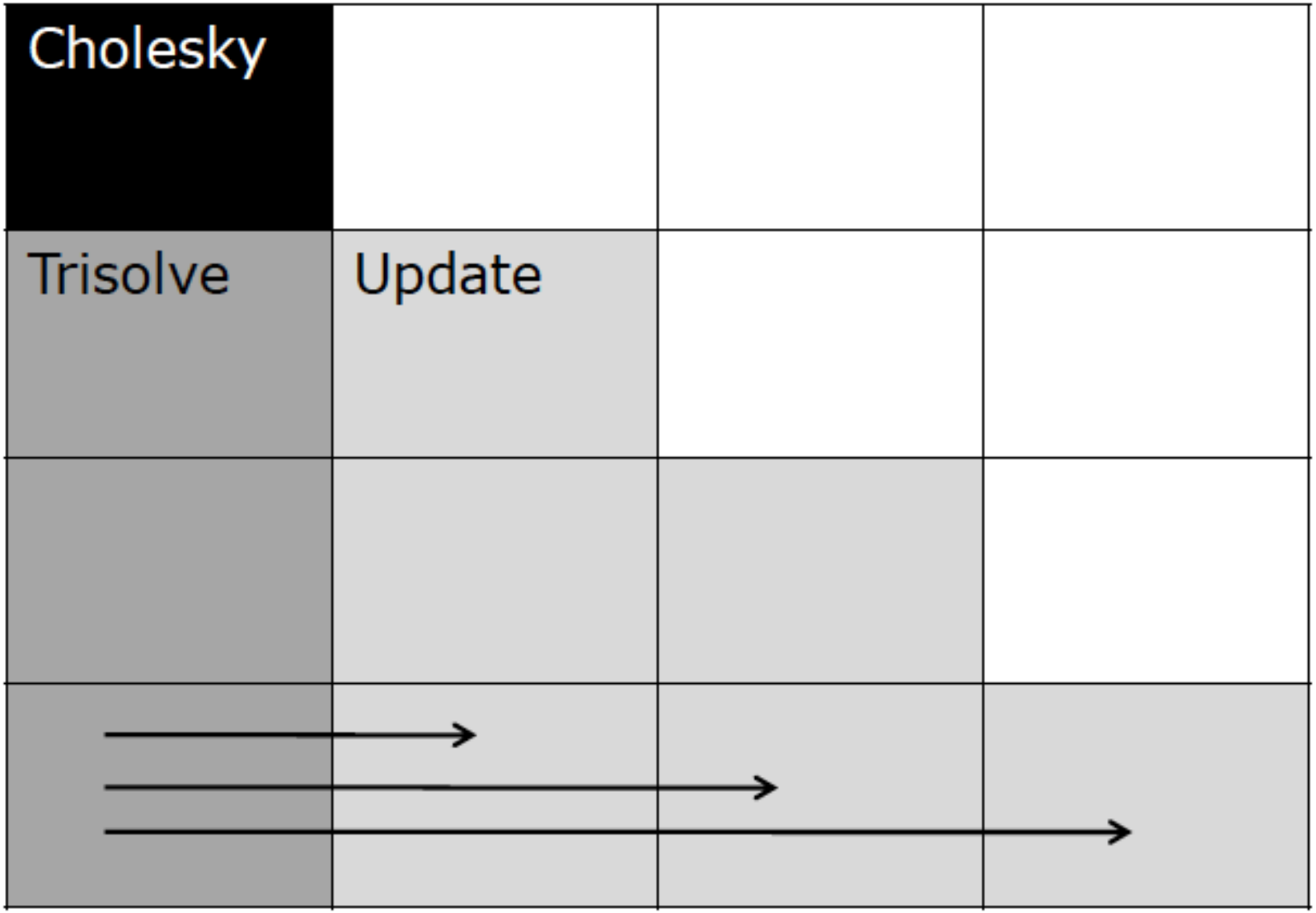
Guidelines to converting a sequential program into CnC

1. Identify step collections
   - Tiling/chunking is important to amortize overhead of scheduling
   - Ideally, each step/task should have one or more tile size parameters that can be used for performance tuning
2. Identify item collections for communication among steps
   - Chunking is important for data as well, to amortize overhead of put/get operations
3. Identify input/output relationships among steps and data
4. Identify spawn/prescribe relationships among steps
   - For simplicity, each step collection can have its own tag collection
5. Add tags and tag functions to steps and items

Example: Cholesky Decomposition



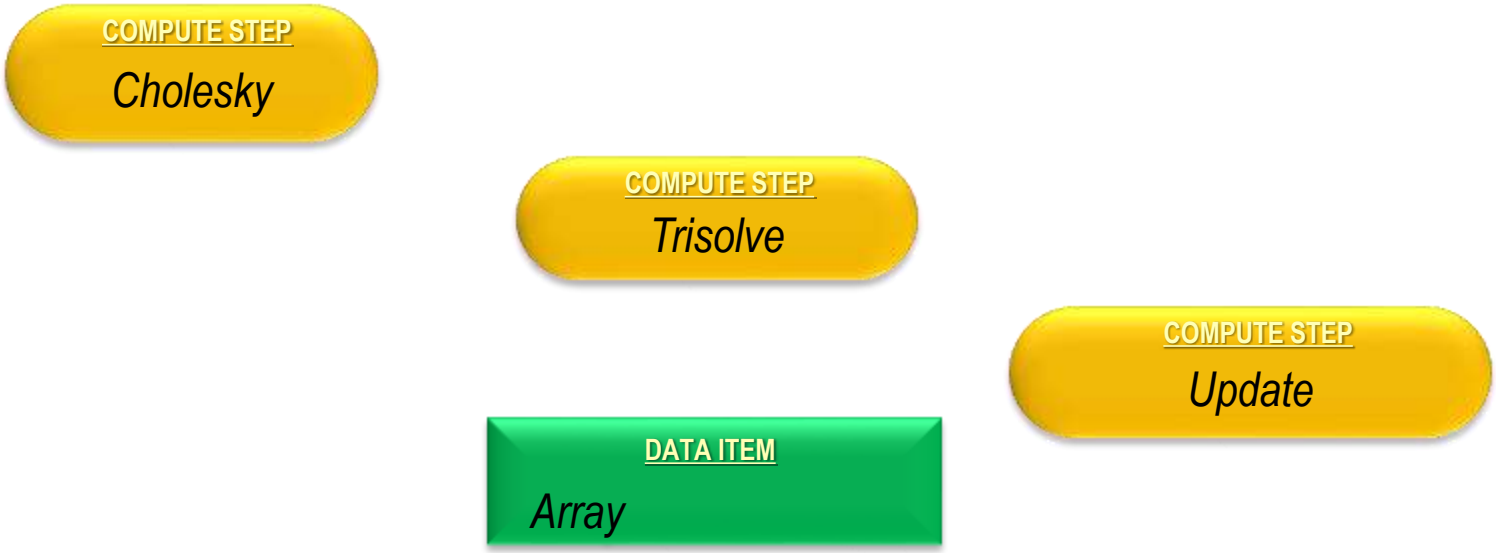
Example: Cholesky Decomposition



1: Identify Step Collections

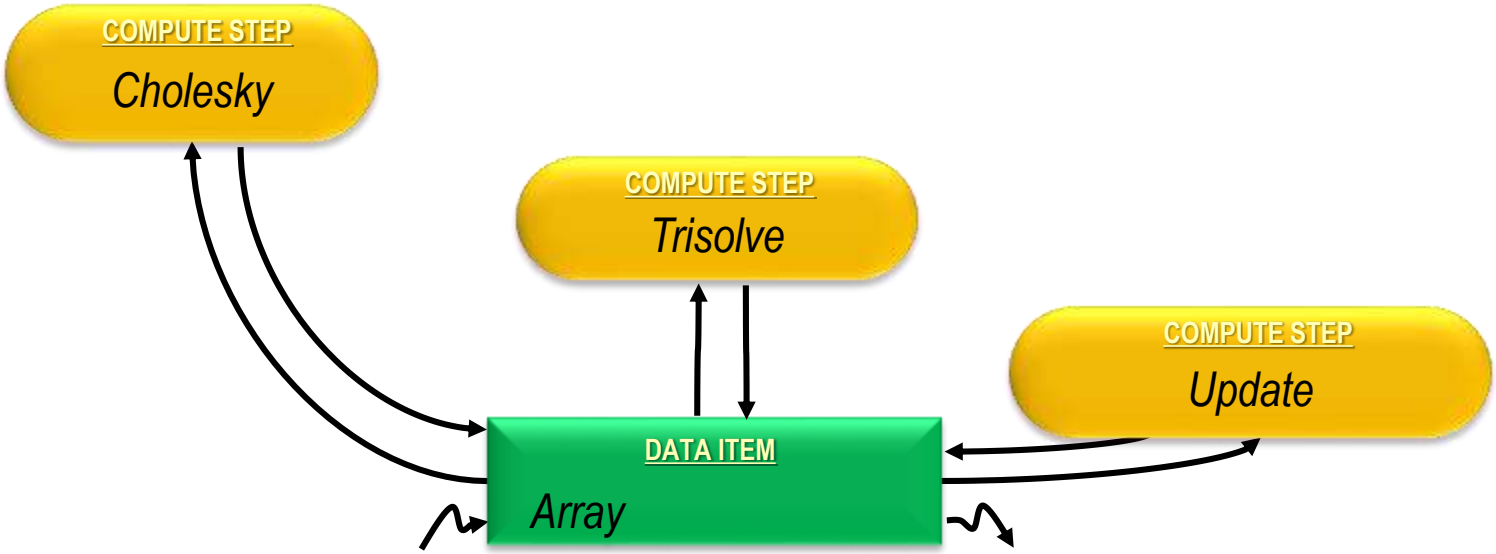
35

2: Identify Item Collections



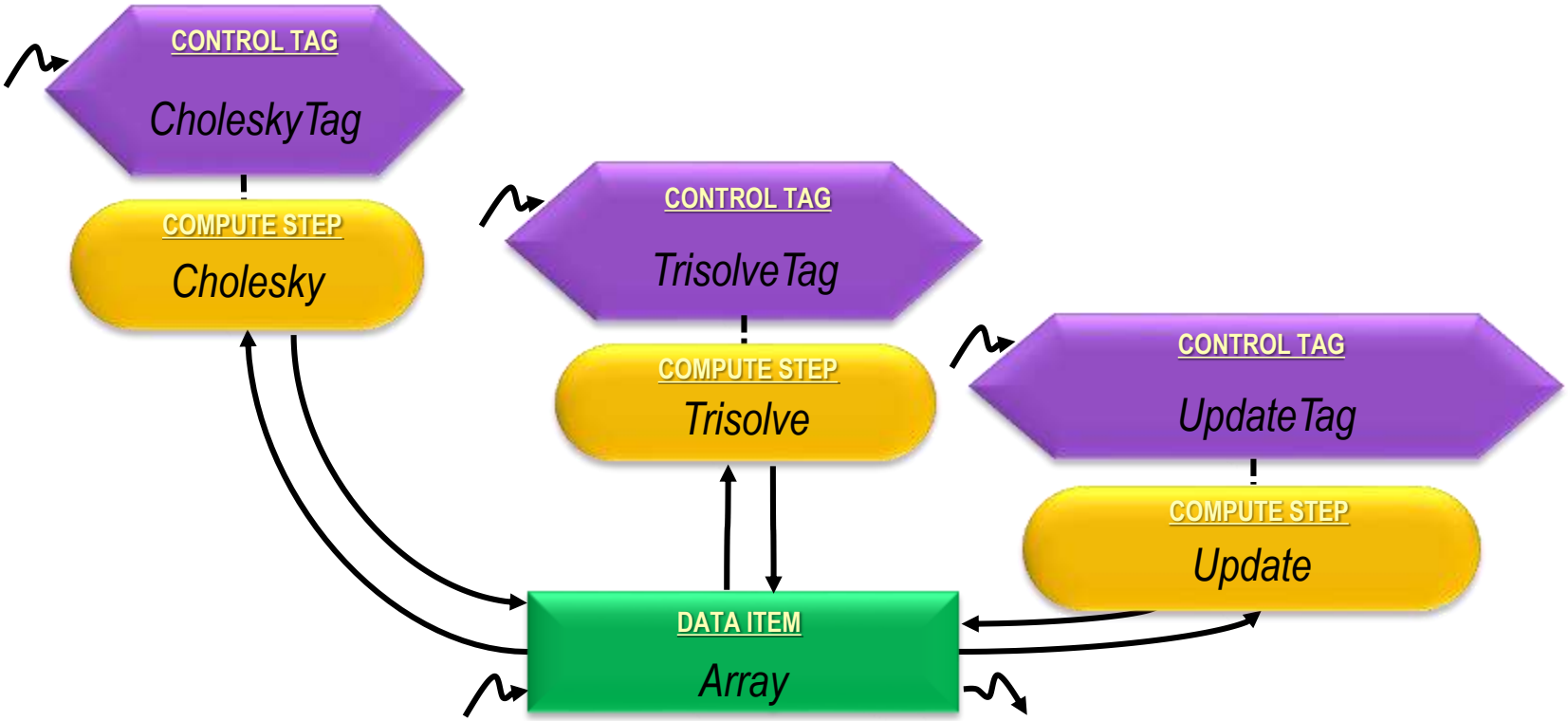
36

3: Identify Input-Output Relationships



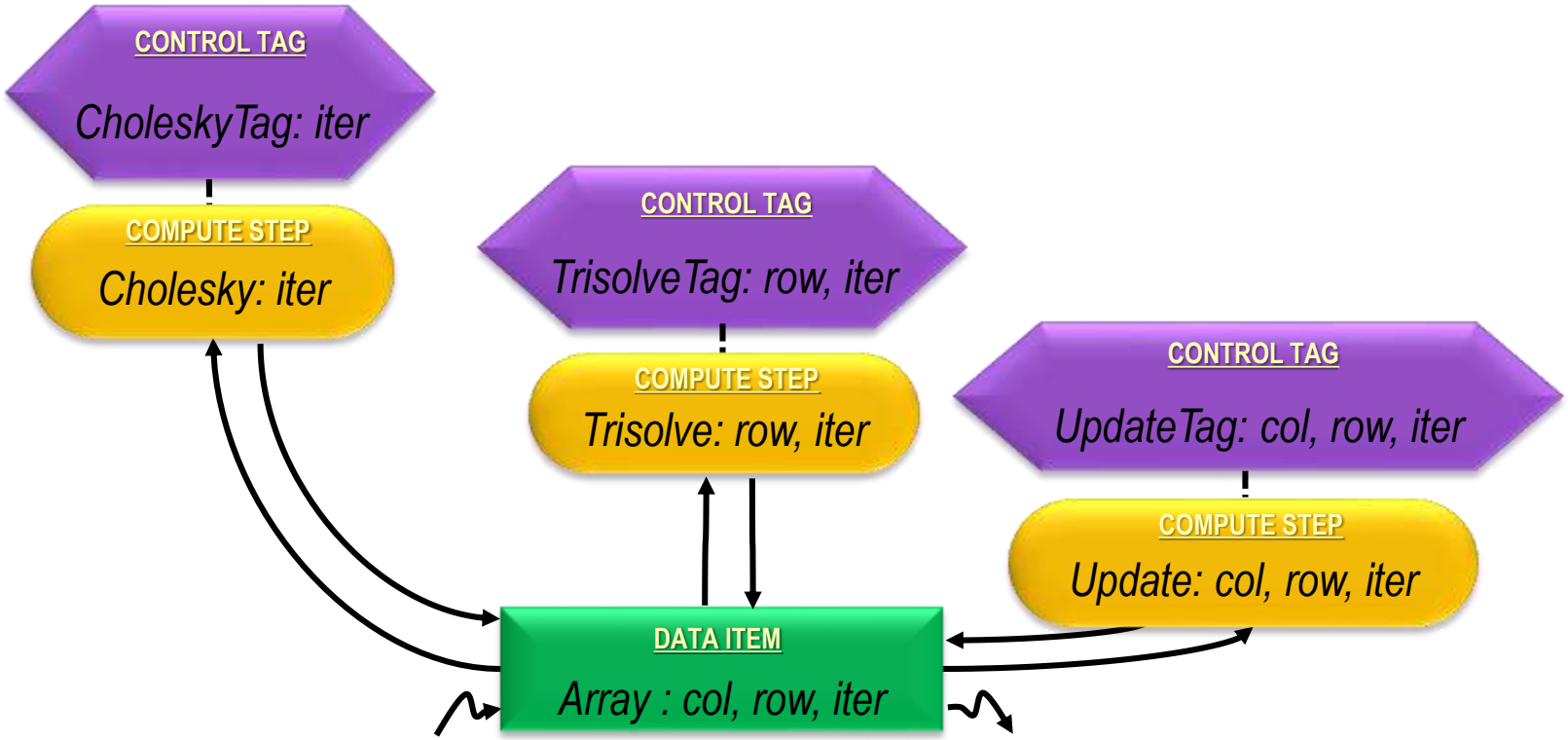
37

4: Identify spawn/control relationships



38

5: Add tags and tag functions



39

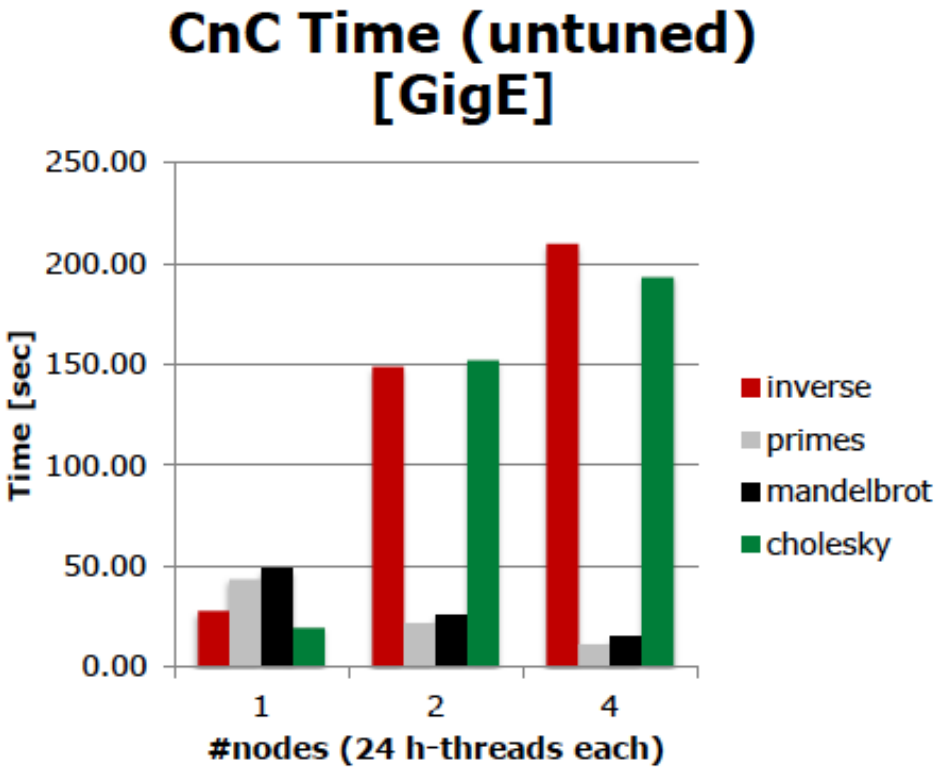
Distributed CnC

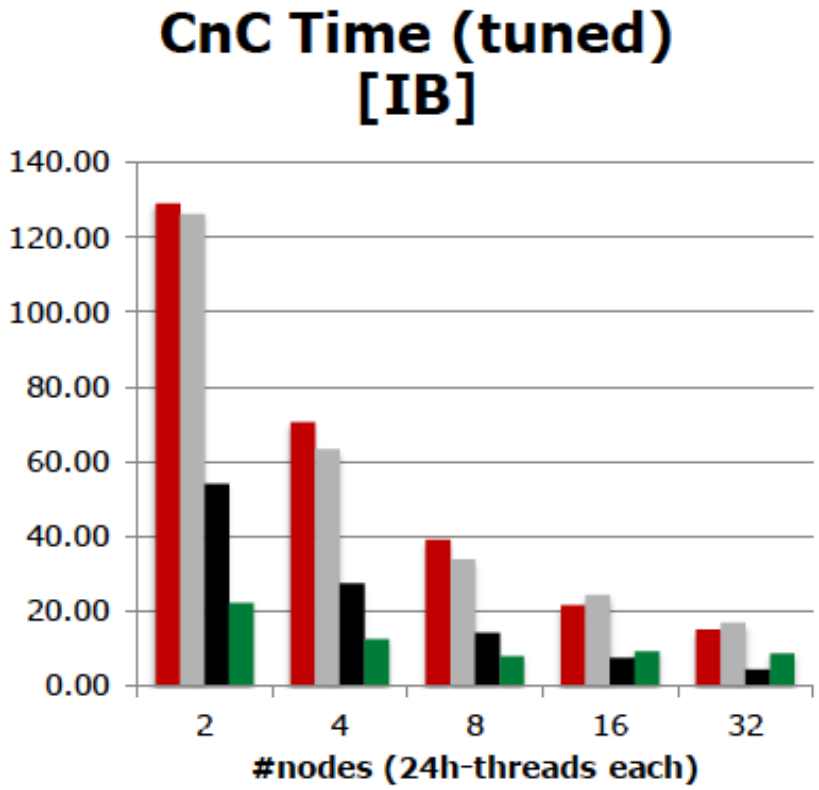
- CnC is a declarative and high level model
  - Abstracts from memory model
- Provides a unified programming model for shared and distributed memory
  - Same binary runs on both
  - Easy to switch between shared and distributed memory (no explicit message passing needed)
- CnC comes with a control methodology
  - Allows controlling work distribution
- CnC comes with a data methodology
  - Provides hooks for selective data distribution

Limitations of Distributed Computing Apply

- **Usual caveats for distributed memory apply**
  - **e.g., ratio between data-exchange and computation**
- **Different algorithm might be needed for distributed or shared memory**
  - **Programming methodology and framework stays the same in any source**
  - **Over a wide class of applications the algorithm stays the same**

Untuned Performance often Bad: Tuning





Summary

- CnC is a simple and powerful model for domain experts
- CnC implementations are available for homogeneous & heterogeneous multicore processors, as well as distributed systems
- The dataflow model guarantees data-race freedom and determinism
  - Good for domain experts, but can sometimes be limiting for system experts
  - Ideal solution: programming system that combines productivity benefits of CnC with flexibility of OpenMP, MPI, and OpenCL
    - ➔ Goal of Habanero project at Rice University (http://habanero.rice.edu)

Acknowledgments

- CnC slides from Kathleen Knobe, Intel
- Members of Habanero Group at Rice University
  - Habanero-C team
  - CnC team
- CS133 slides from Prof. Jason Cong
  - Spring 2013, UCLA

Backup: Task in OpenMP

- **Tasks – Main change for OpenMP 3.0**
- **Allows parallelization of irregular problems**
  - **Unbounded loops**
  - **Recursive algorithms**
  - **Producer/consumer**
- **Compilers**
  - **May not be supported on Visual C++ 2011**
  - **Supported by Intel C++ 10.1, gcc 4.4, and llvm-gcc 4.2**

Intel, "Introduction to Parallel Programming," 2009

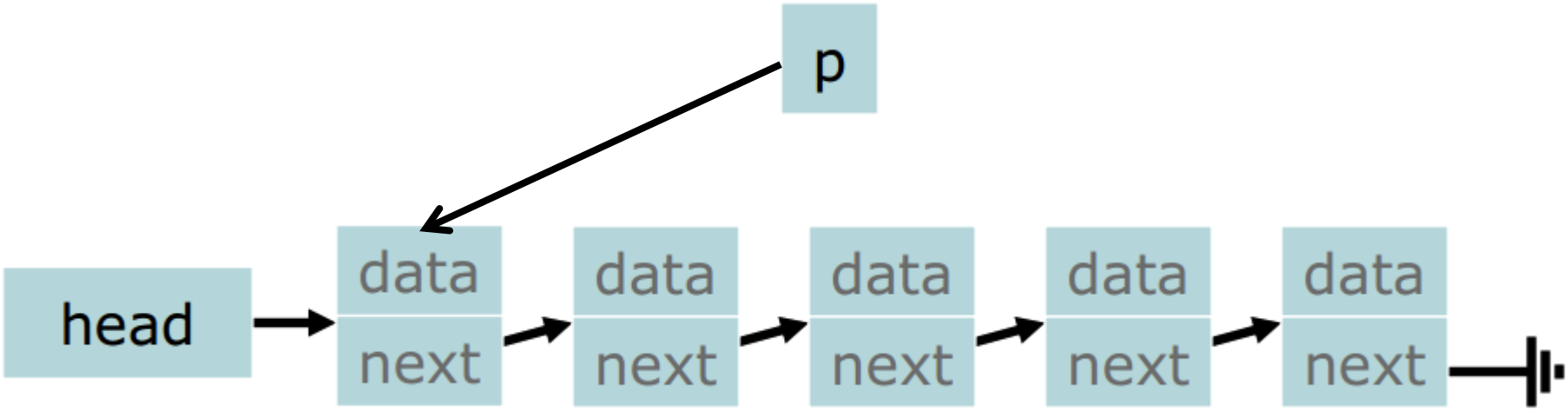
What are Tasks?

- **Tasks are independent units of work**
  - **Threads are assigned to perform the work of each task**
  - **Tasks may be deferred**
  - **Tasks may be executed immediately**
- **The runtime system decides which of the above**

Intel, “Introduction to Parallel Programming,” 2009

A Linked List Example

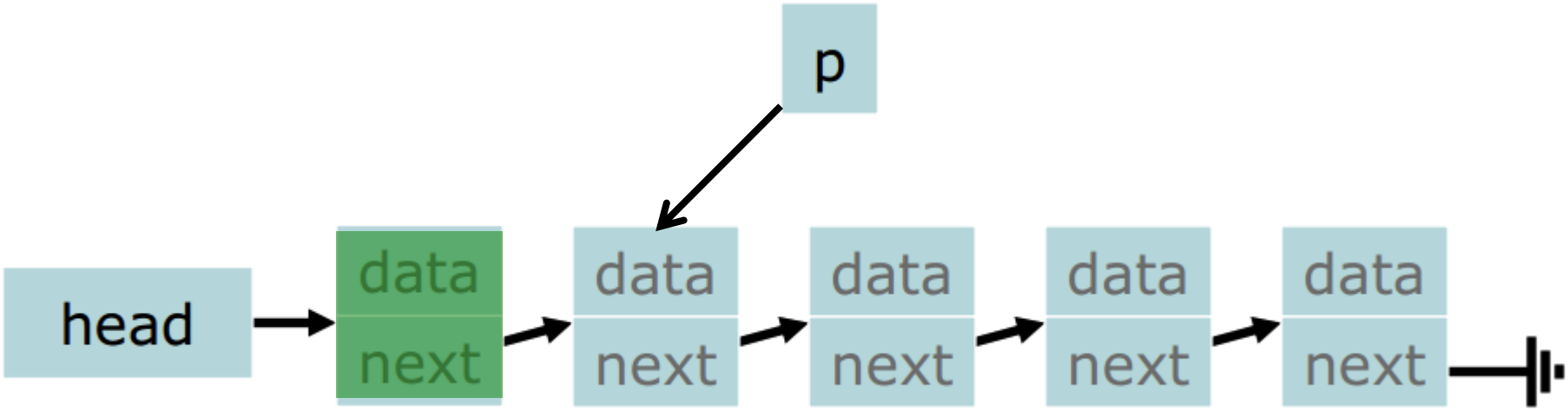
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, “Introduction to Parallel Programming,” 2009

A Linked List Example

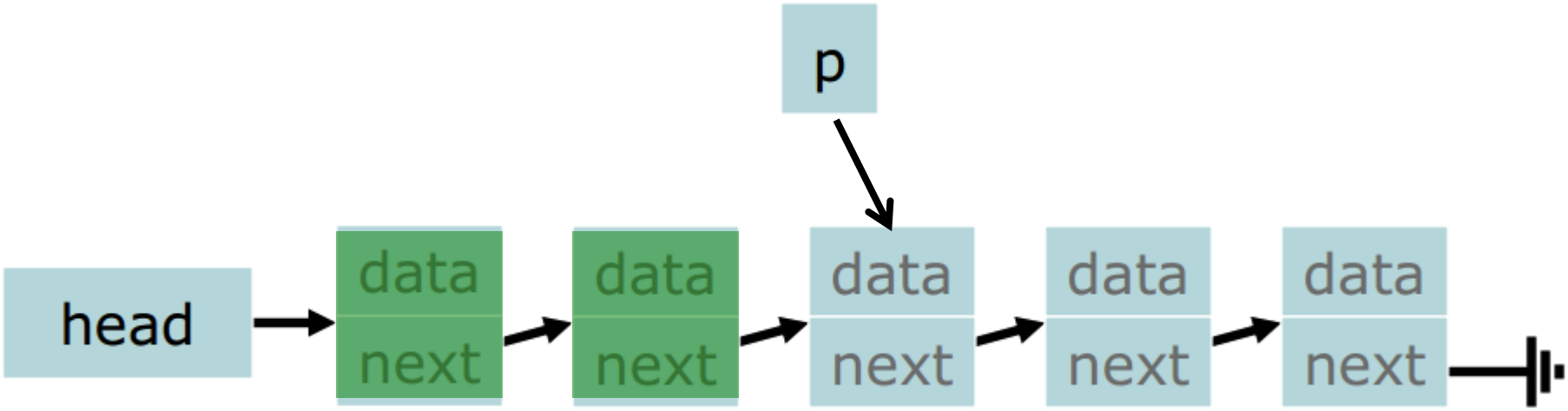
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, “Introduction to Parallel Programming,” 2009

A Linked List Example

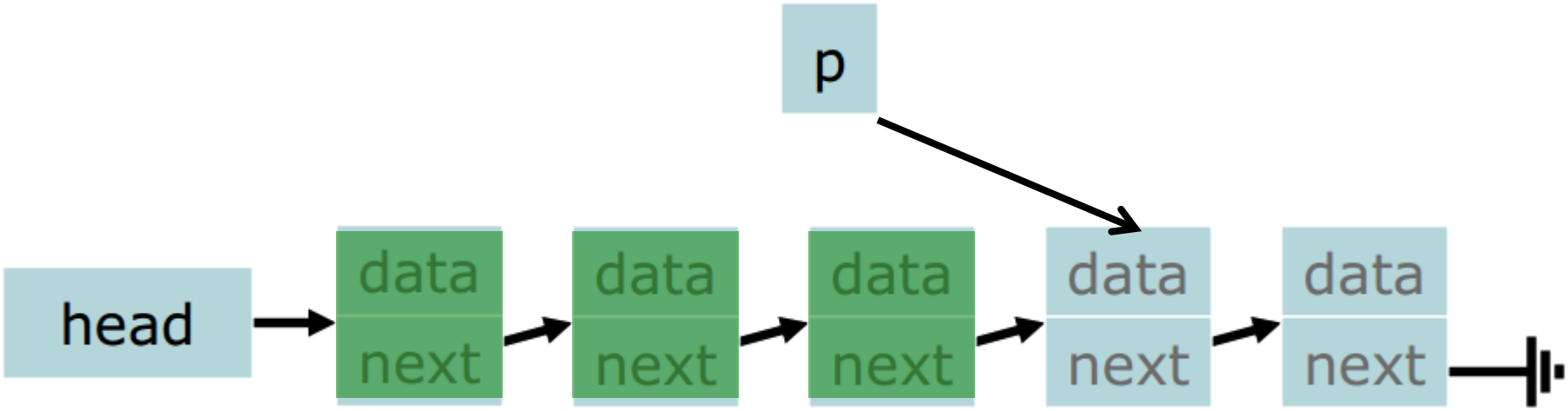
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

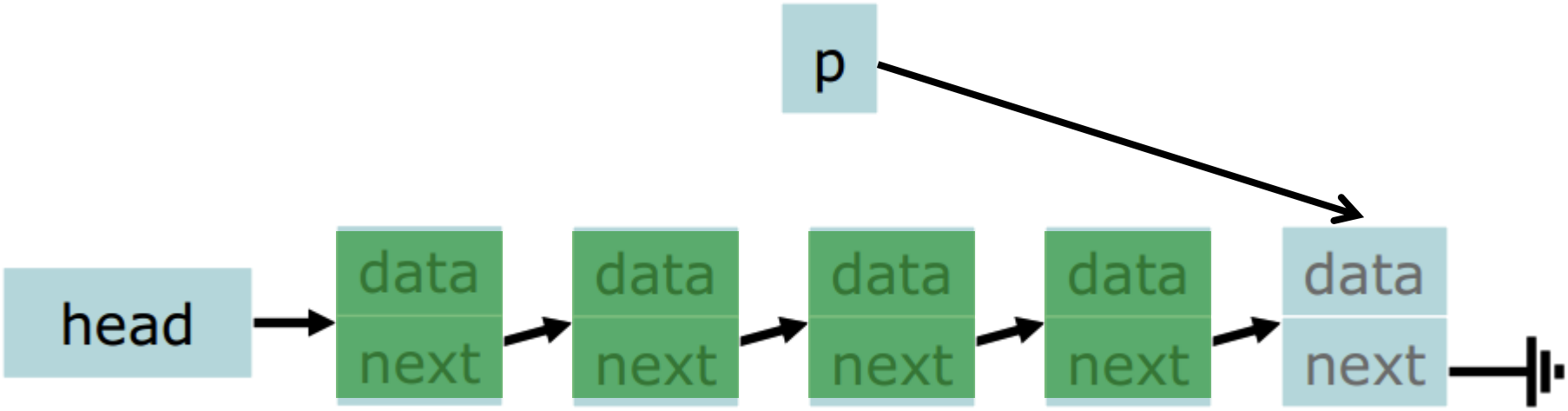
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

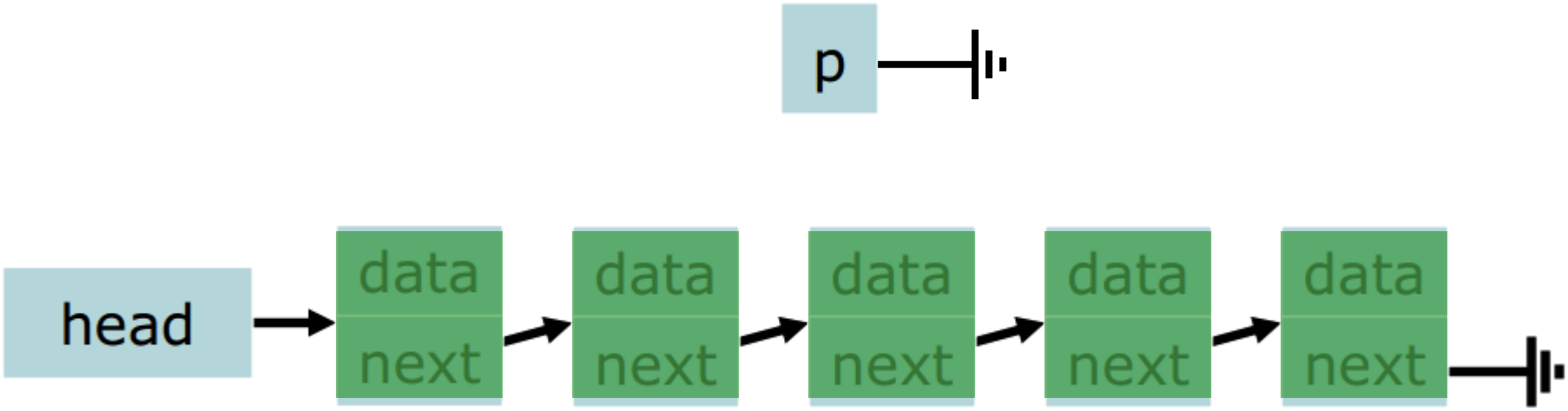
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, “Introduction to Parallel Programming,” 2009

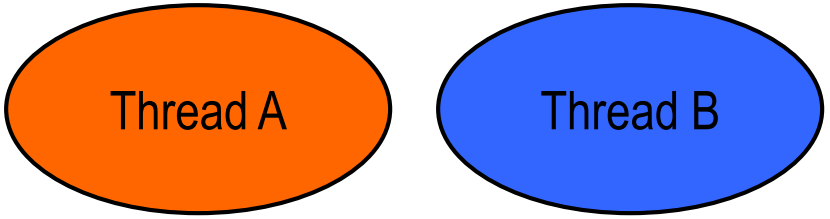
Task Construct – Explicit Task View

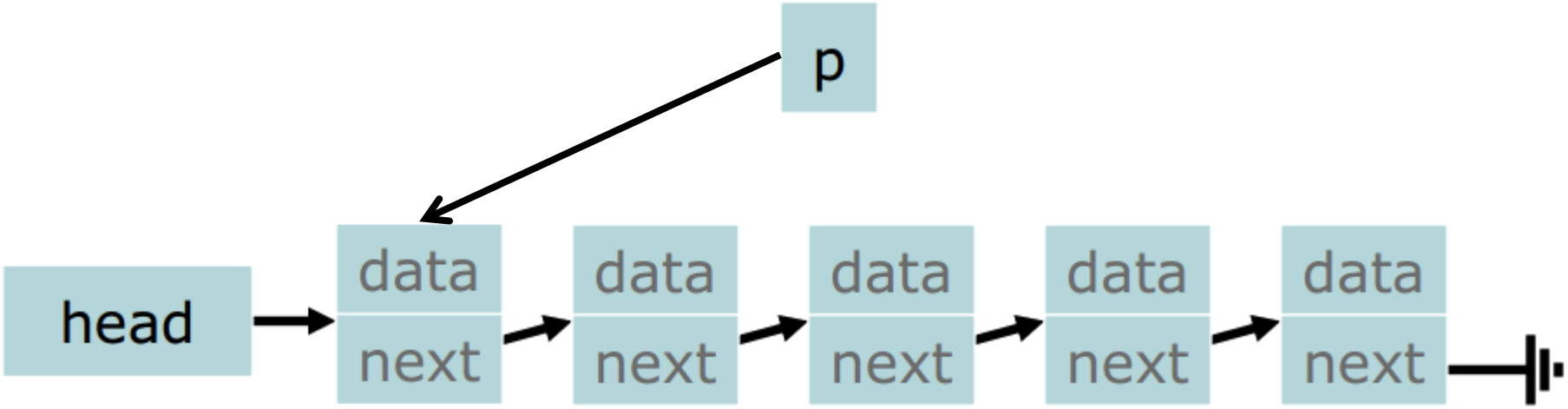
```
node* p = head;
#pragma omp parallel
// a team of threads is forked here
{
#pragma omp single
    // a single thread, T, executes the while loop
    while (p) {
#pragma omp task
        // each time T crosses the pragma, it generates a new task
        process(p);    // each task runs in a thread
        p = p->next;  // only T updates the p pointer
    } // all task complete at this implicit barrier of single
}
```

Intel, “Introduction to Parallel Programming,” 2009

A Linked List Example

```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```

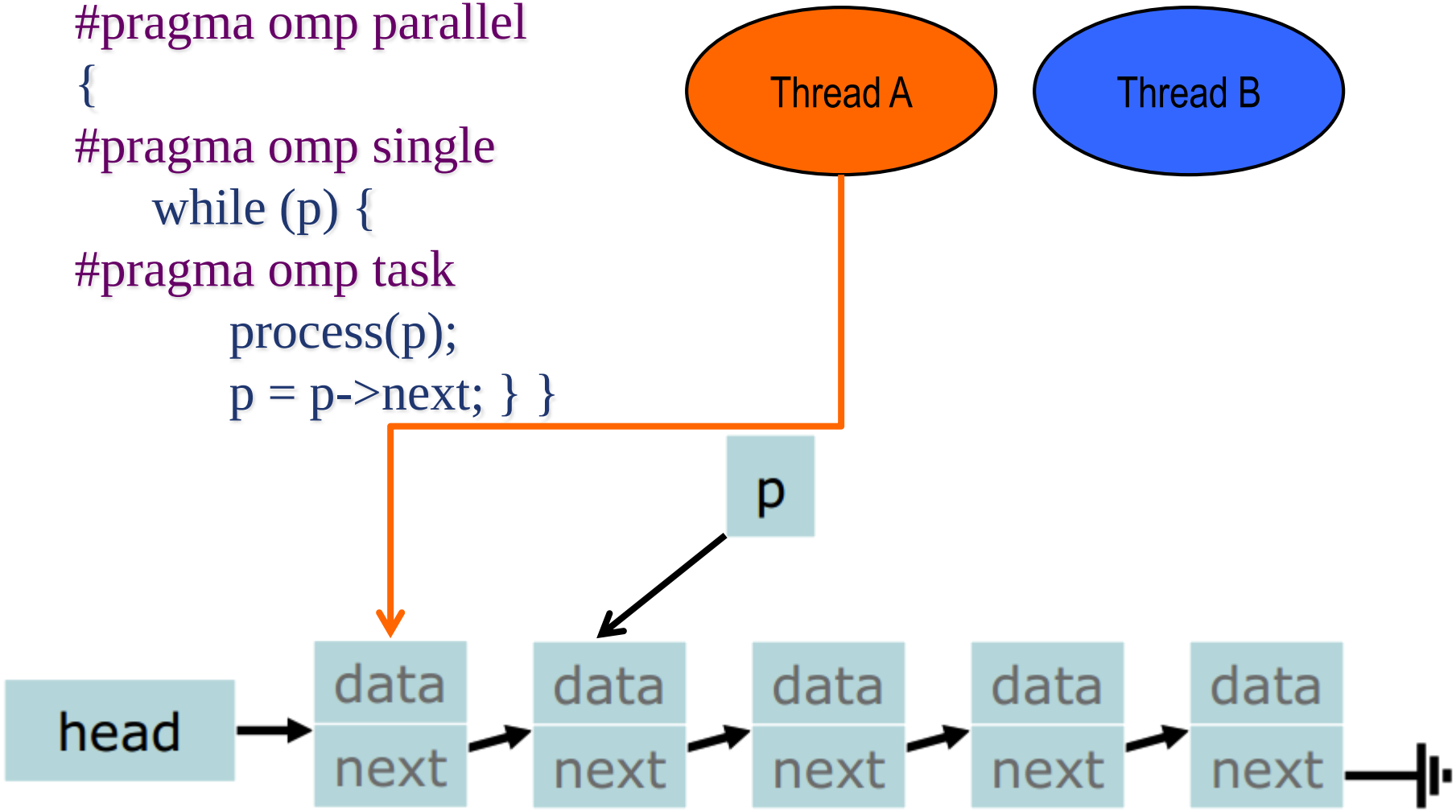




Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

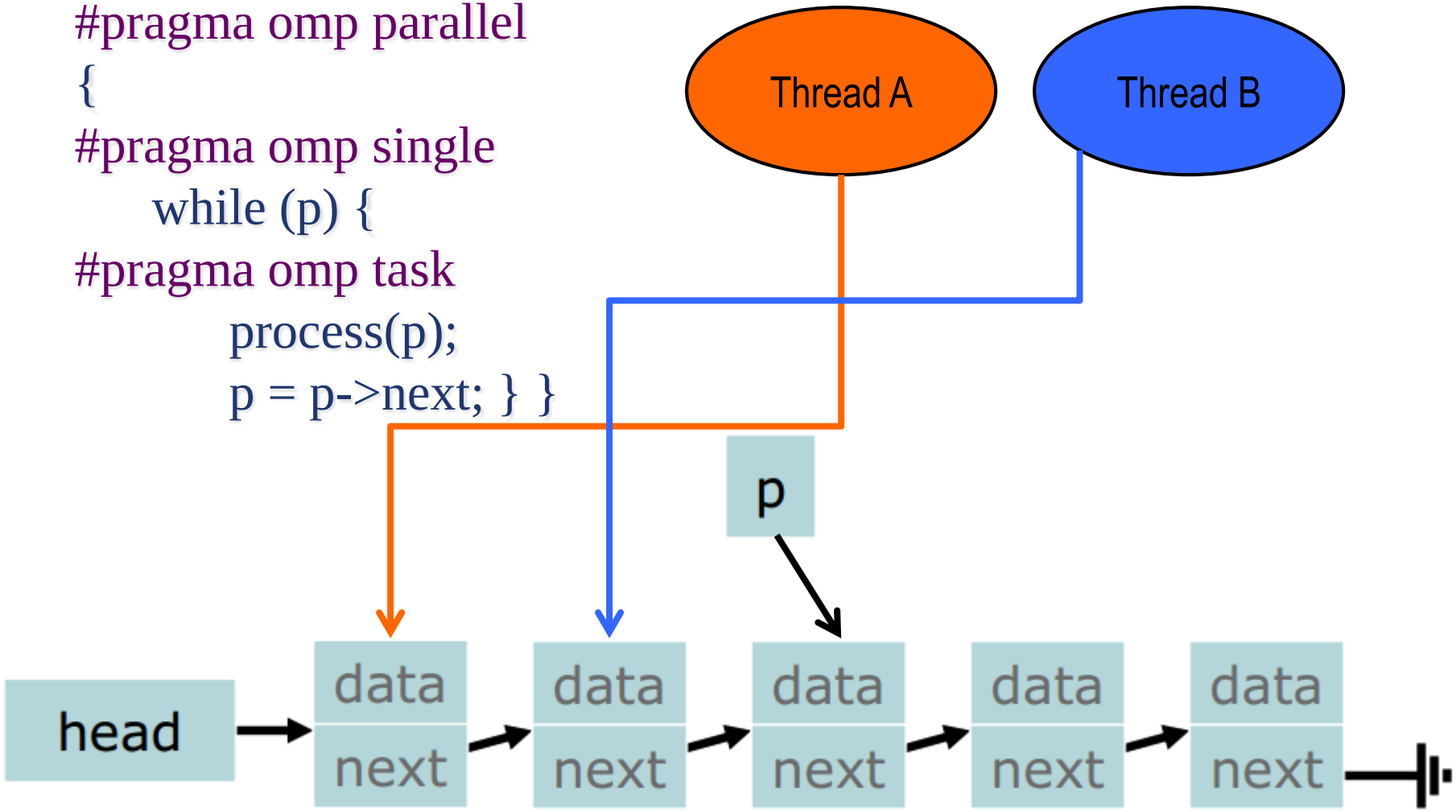
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

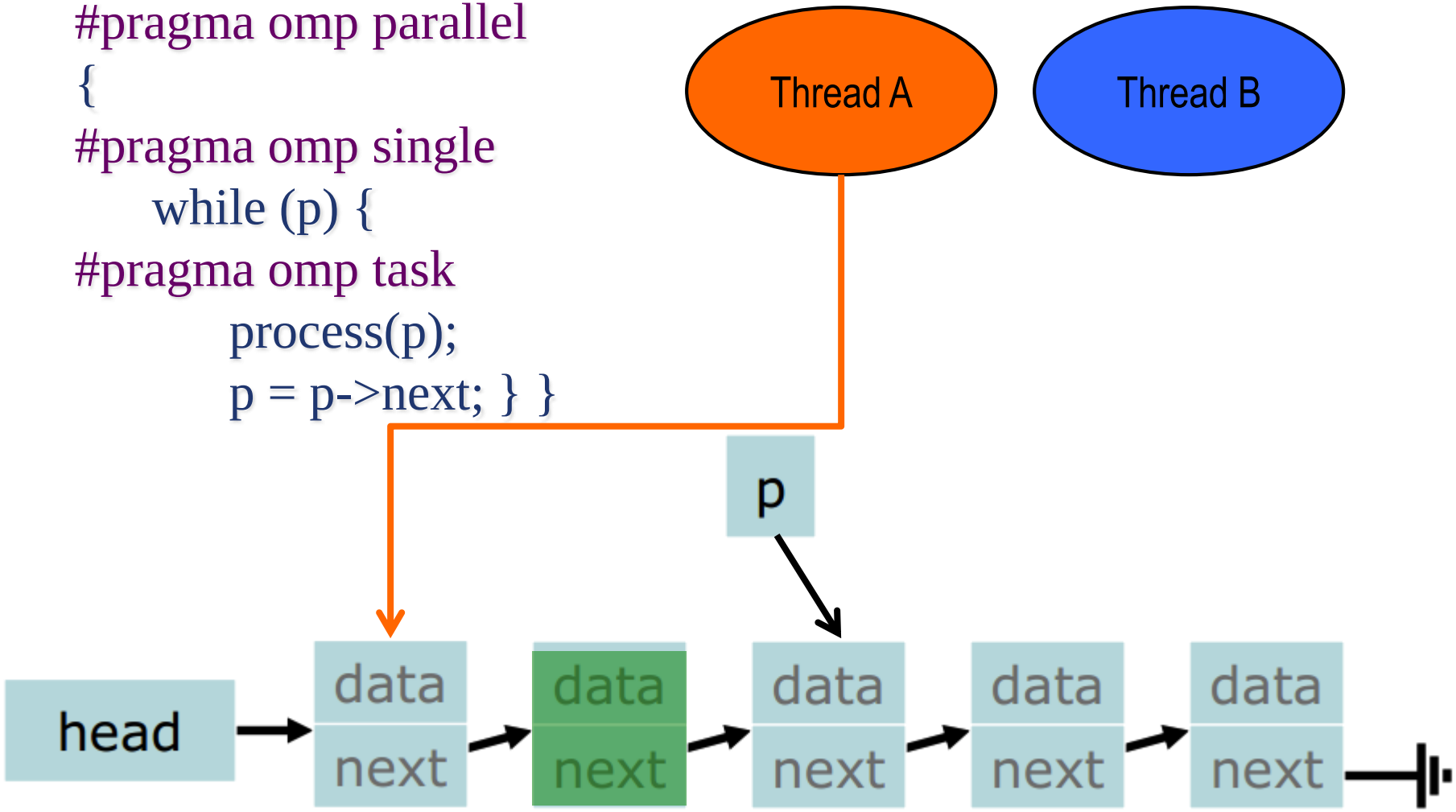
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

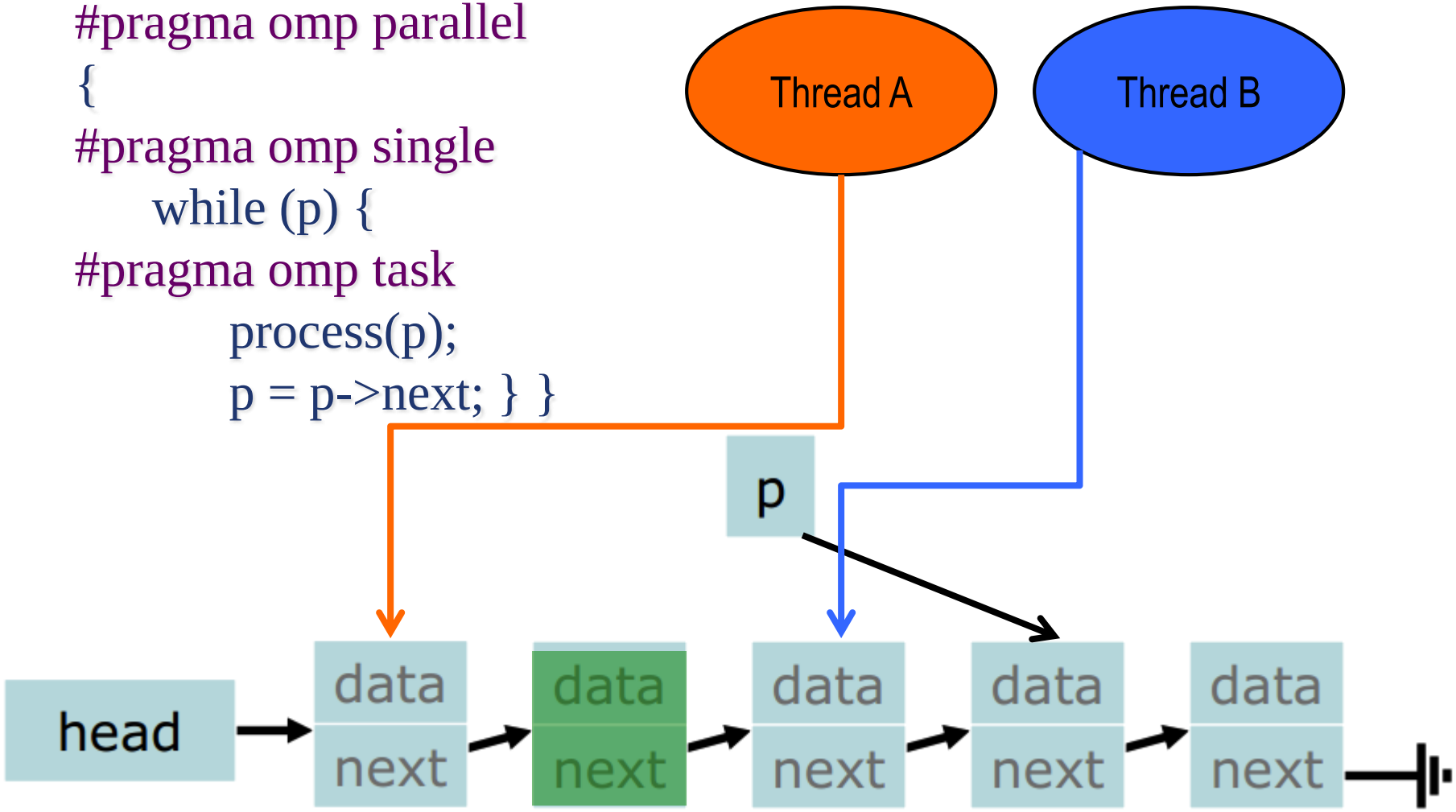
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

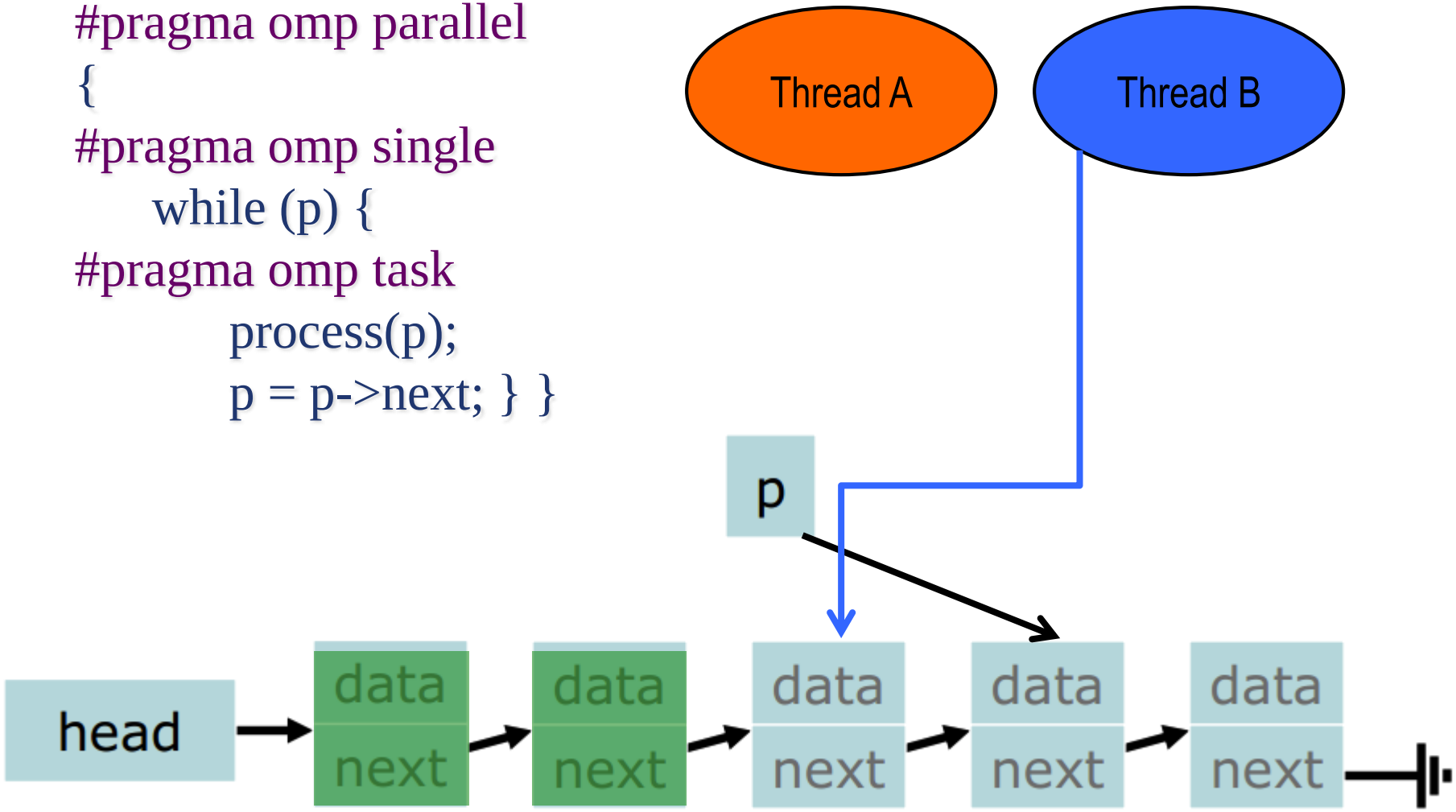
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

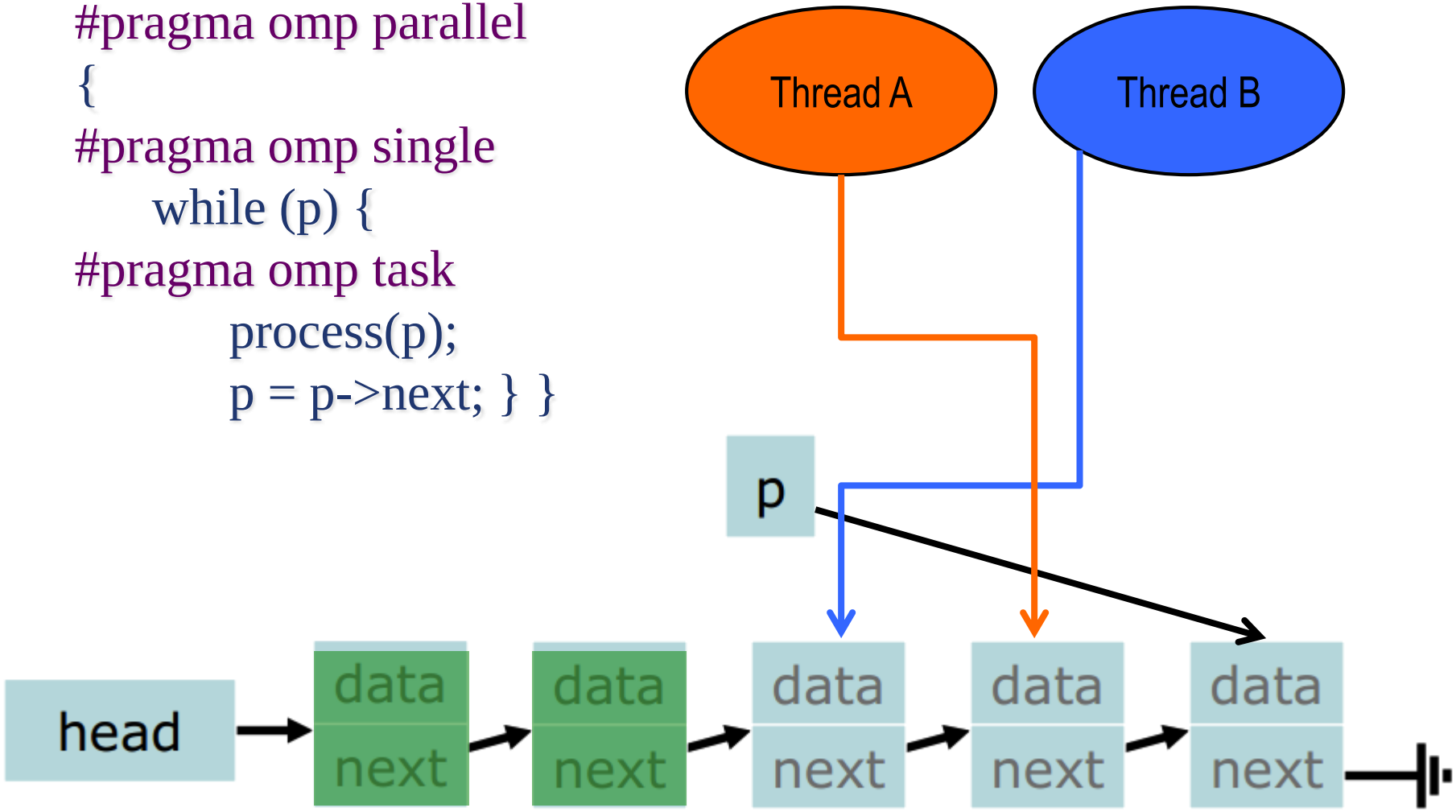
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

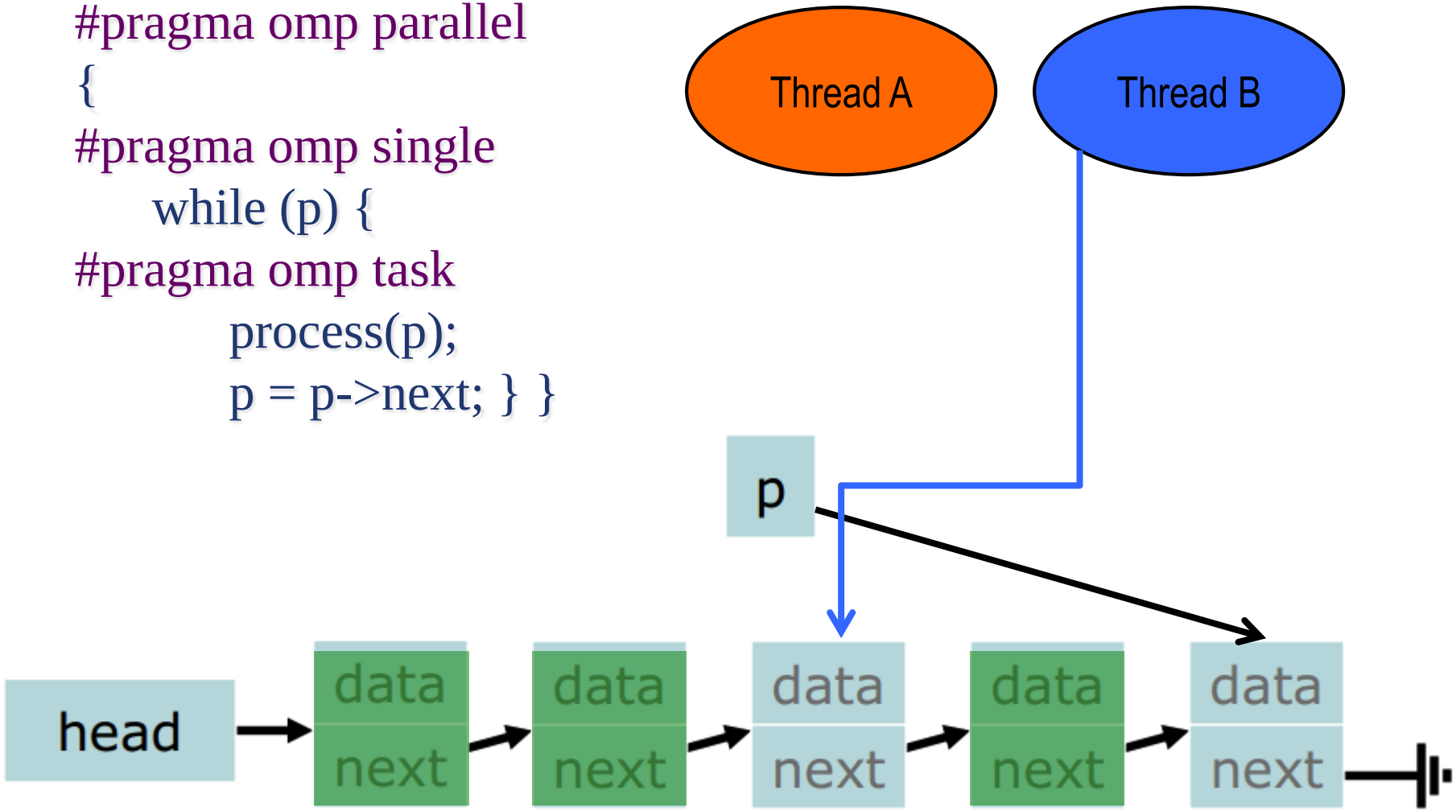
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

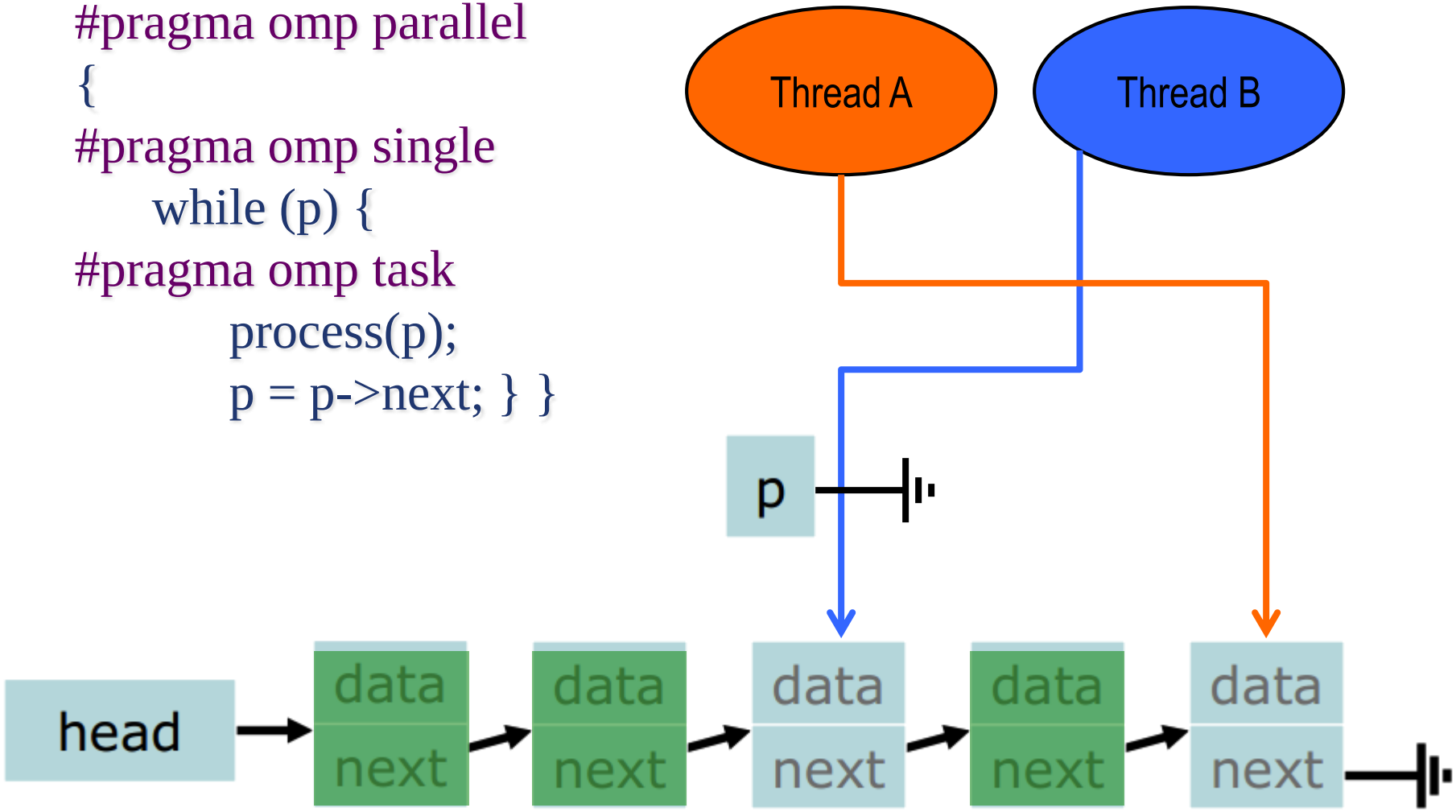
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

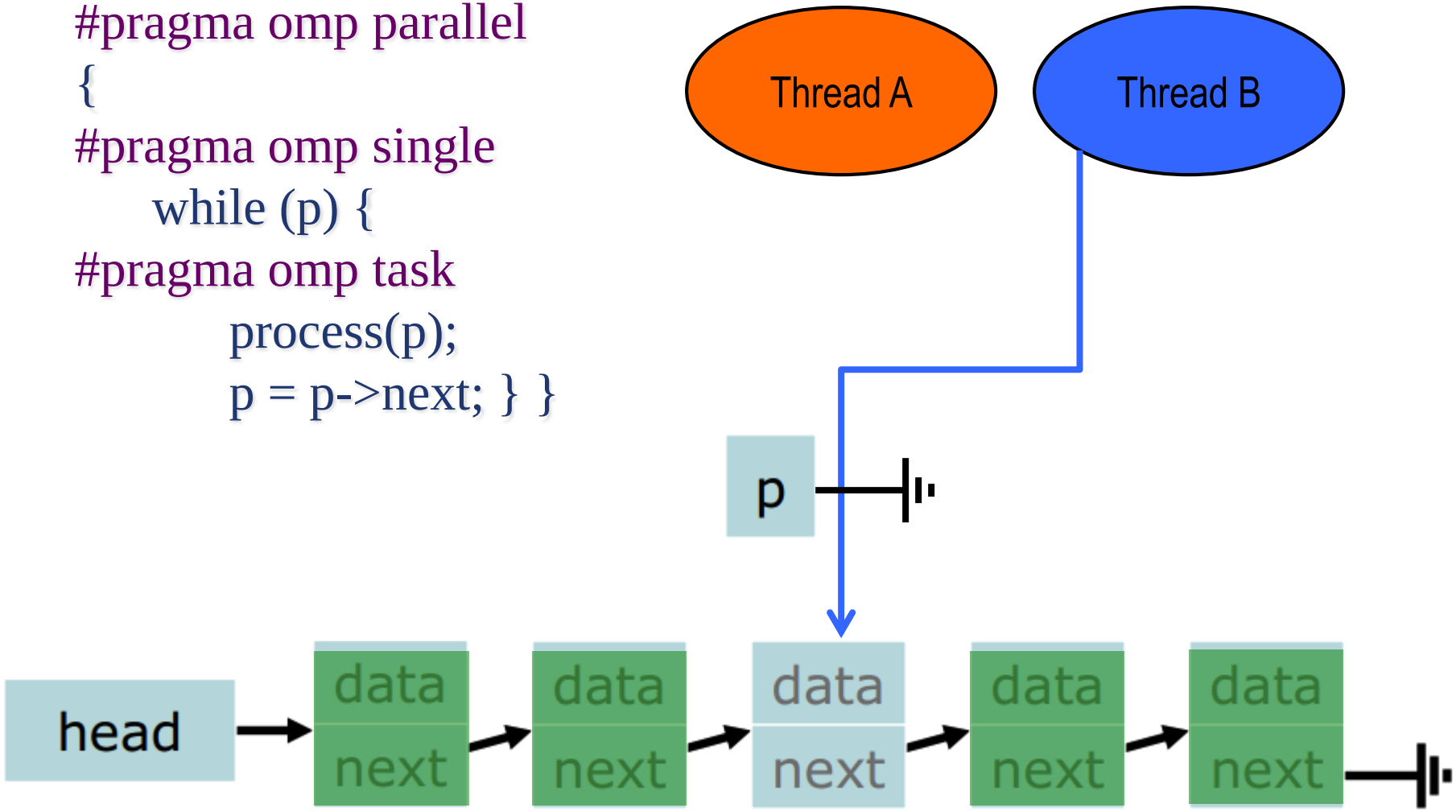
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

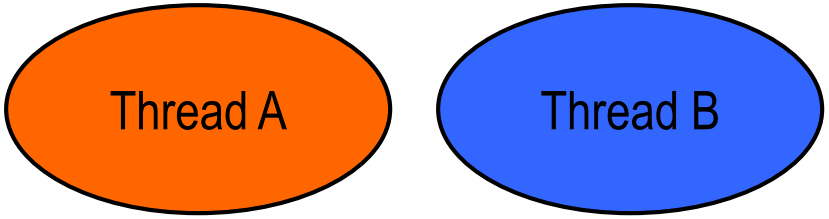
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```

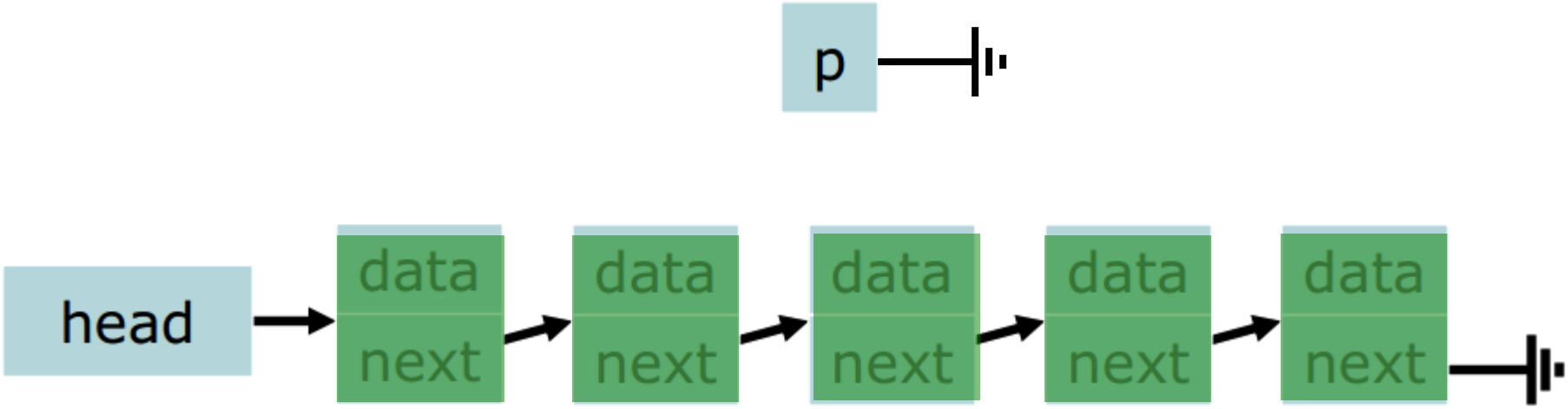


Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```





Intel, "Introduction to Parallel Programming," 2009

When are Tasks Guaranteed to be Complete?

- **Tasks are guaranteed to be complete:**
  - **At thread or task barriers**
  - **At the directive: #pragma omp barrier**
    - **all the threads in the team will be synchronized**
  - **At the directive: #pragma omp taskwait**
    - **only the child threads generated since the beginning of the current task will be synchronized**

Intel, "Introduction to Parallel Programming," 2009

Example: Naïve Fibonacci Calculation

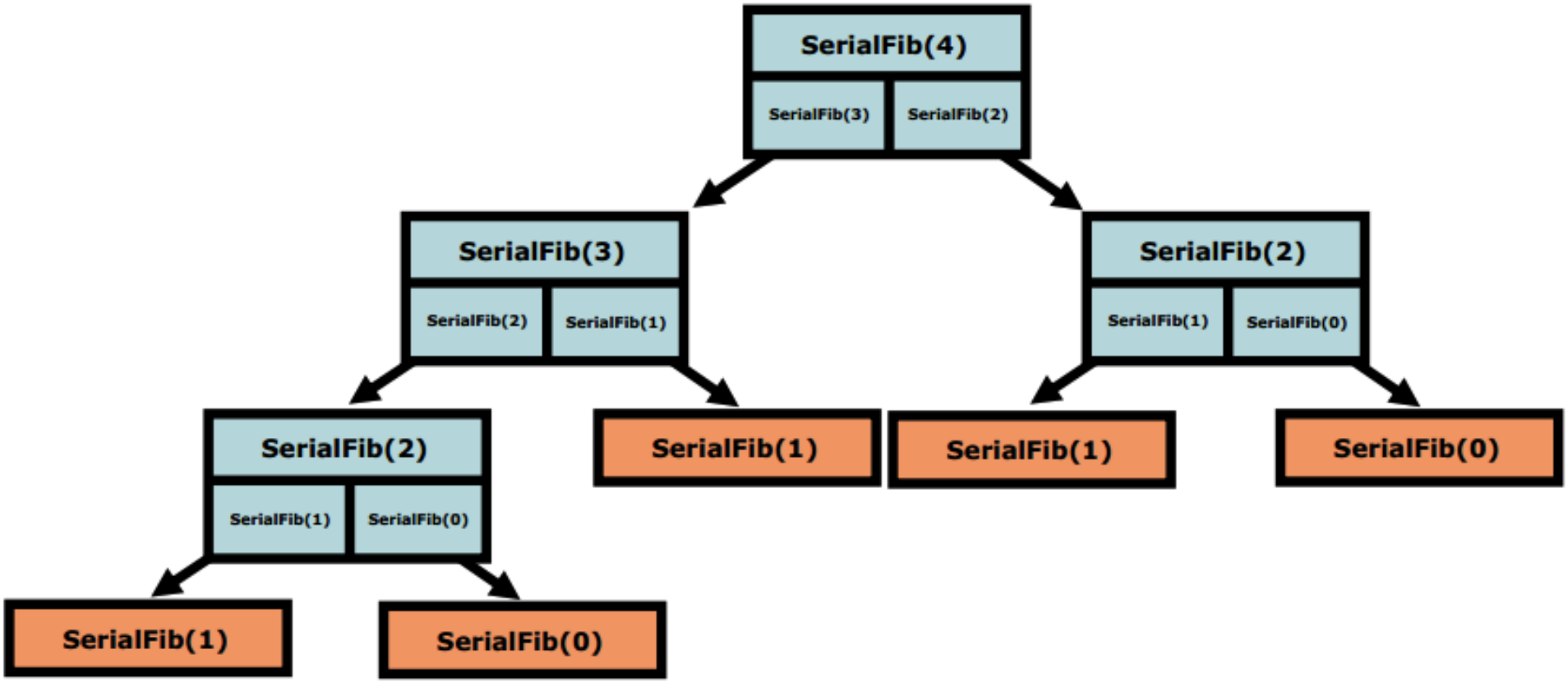
- **Recursion typically used to calculate Fibonacci number**
- **Widely used as toy benchmark**
  - **Easy to code**
  - **Has unbalanced task graph**

```
long SerialFib(long n) {
    if (n < 2) return n;
    else return SerialFib(n-1) + SerialFib(n-2);
}
```

Intel, "Introduction to Parallel Programming," 2009

Example: Naïve Fibonacci Calculation

- **We can envision Fibonacci computation as a task graph**



Intel, "Introduction to Parallel Programming," 2009

Fibonacci – Task Spawning Solution

```
long ParallelFib(long n) {
    long sum;
#pragma omp parallel
    {
#pragma omp single
        FibTask(n, &sum);
    }
    return sum;
}
```

```
void FibTask(
    long, n, long* sum) {
  if (n < CutOff)
    *sum = SerialFib(n);
  else {
    long x, y;
#pragma omp task
    FibTask(n-1, &x);
#pragma omp task
    FibTask(n-2, &y);
#pragma omp taskwait
    *sum = x + y;
  }
}
```

Intel, “Introduction to Parallel Programming,” 2009