Introduction to Parallel & Distributed Computing

Lecture 4 Decomposition Techniques and OpenMP Task

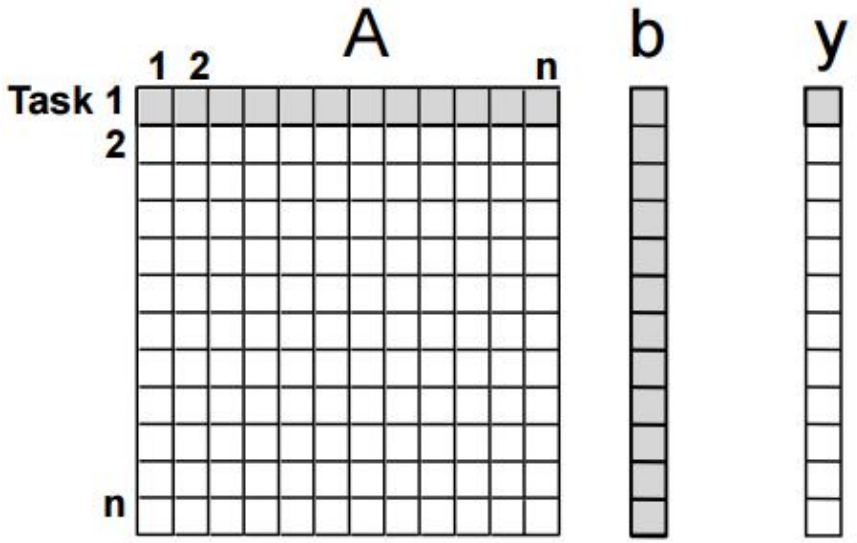
Spring 2024

Instructor: 罗国杰
gluo@pku.edu.cn

Outline

- Decomposition techniques
  - data decomposition
  - recursive decomposition?
- OpenMP task
- Decomposition techniques (cont.)
  - recursive decomposition
  - exploratory decomposition
  - speculative decomposition

Example: Dense Matrix-Vector Product

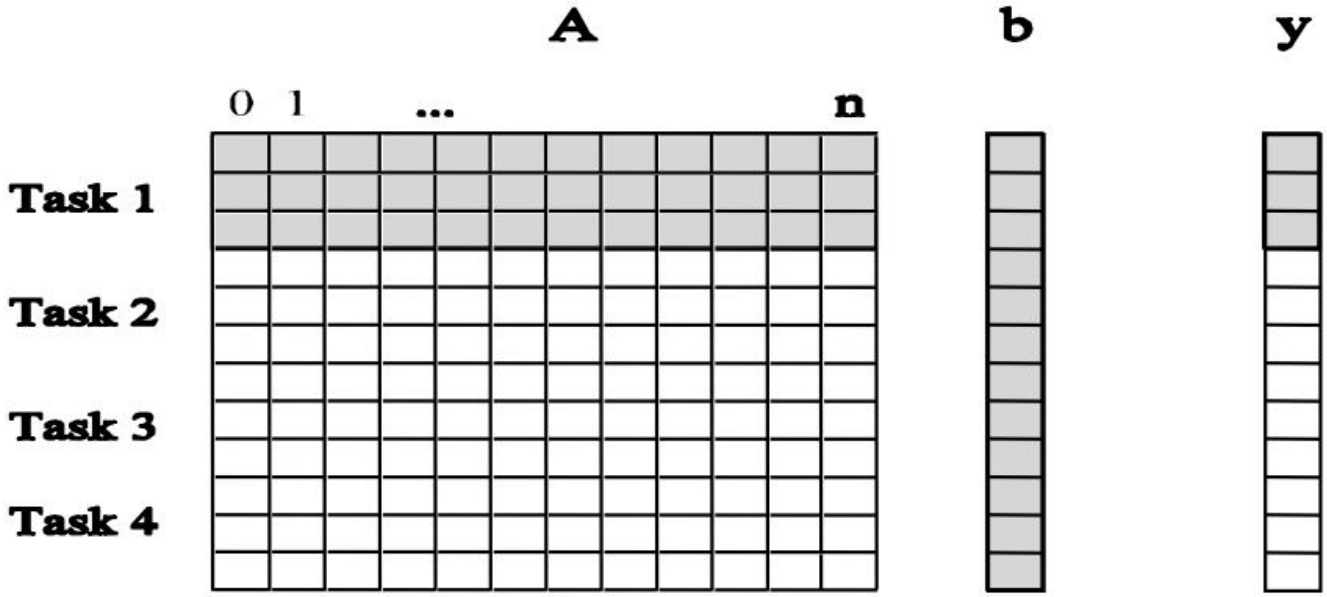


- Computing each element of output vector y is independent
- Easy to decompose dense matrix-vector product into tasks
  - one per element in y
- Observations
  - task size is uniform
  - no control dependences between tasks
  - tasks share b

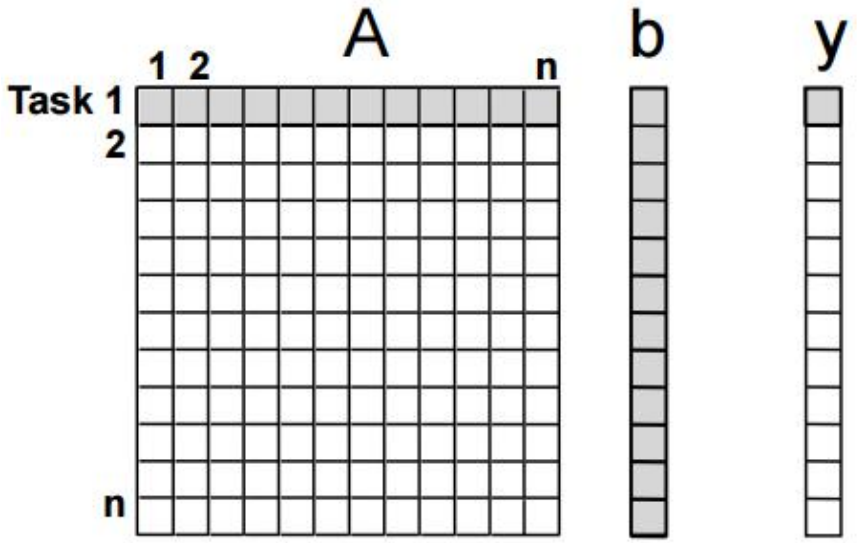
Source: COMP 422 at Rice Univ., Spring 2016.

Granularity of Decompositions

- Granularity = task size
  - depends on the number of tasks
- Fine-grain = large number of tasks
- Coarse-grain = small number of tasks
- Granularity examples for dense matrix-vector multiply
  - fine-grain: each task represents an individual element in y
  - coarser-grain: each task computes 3 elements in y



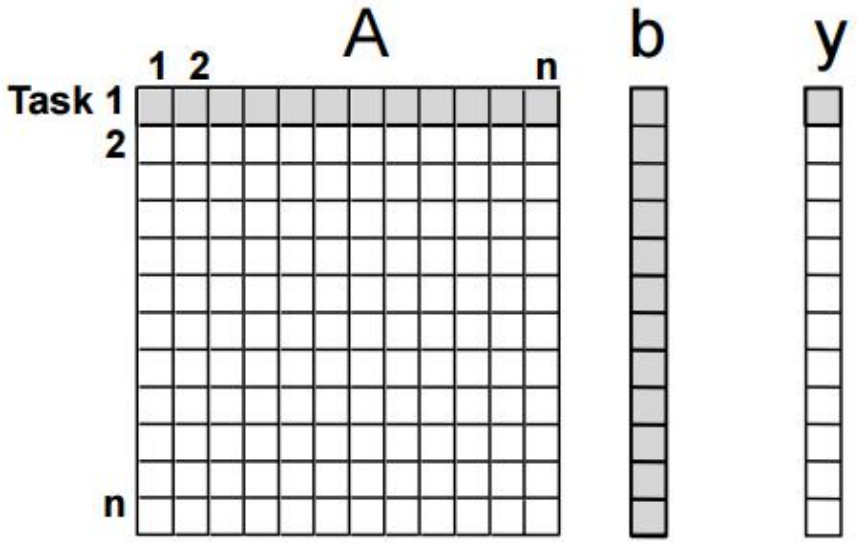
Example: Dense Matrix-Vector



- Computing each element of output vector y is independent
- Easy to decompose dense matrix-vector product into tasks
  - one per element in y
- Observations
  - task size is uniform
  - no control dependences between tasks
  - tasks share b

Question: Is n the maximum number of tasks possible?

Critical Path Length



Questions:

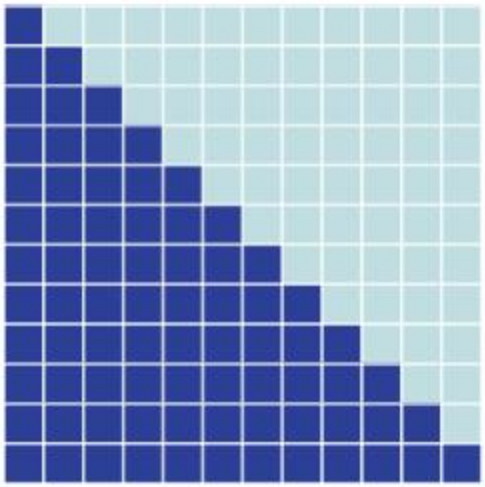
- What does a task dependency graph look like for DMVP?
- What is the shortest parallel execution time for the graph?
- How many processors are needed to achieve the minimum time?
- What is the maximum degree of concurrency?
- What is the average parallelism?

Exercise: Dense Matrix-Vector

- Try OpenMP with different granularities
- Which loop to paralleize?
- Related omp pragmas
  - parallel, for
- Related omp clauses
  - schedule

Loop Scheduling Example

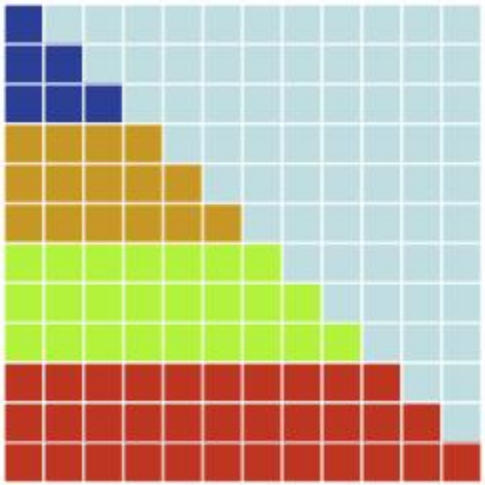
```
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = …;
```



Intel, “Introduction to Parallel Programming,” 2009

Loop Scheduling Example

```
#pragma omp parallel for
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = …;
```



Typically, the iterations are divided by the number of threads and assigned as chunks to a thread

Intel, “Introduction to Parallel Programming,” 2009

Loop Scheduling

- **Loop schedule**
  - **How loop iterations are assigned to threads**
  - **Static schedule**
    - **Iterations assigned to threads before execution of loop**
  - **Dynamic schedule**
    - **Iterations assigned to threads during execution of loop**
- **The OpenMP *schedule* clause affects how loop iterations are mapped onto threads**

Intel, "Introduction to Parallel Programming," 2009

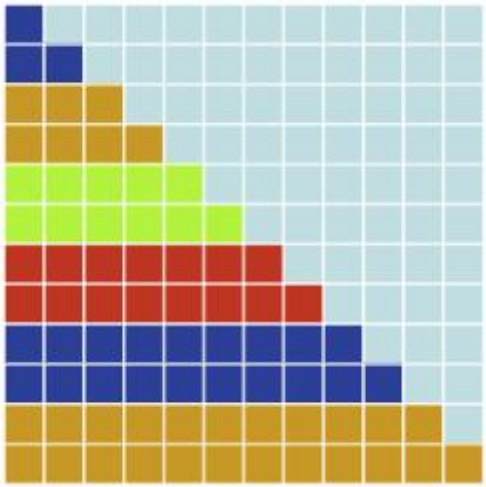
The *schedule* Clause

- *schedule (static [, chunk])*
  - **Blocks of iterations of size "chunk" to threads**
  - **Round robin distribution**
  - **Low overhead, may cause load imbalance**
- **Best used for predictable and similar work per iteration**

Intel, "Introduction to Parallel Programming," 2009

Loop Scheduling Example

```
#pragma omp parallel for schedule(static, 2)
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = …;
```



Intel, “Introduction to Parallel Programming,” 2009

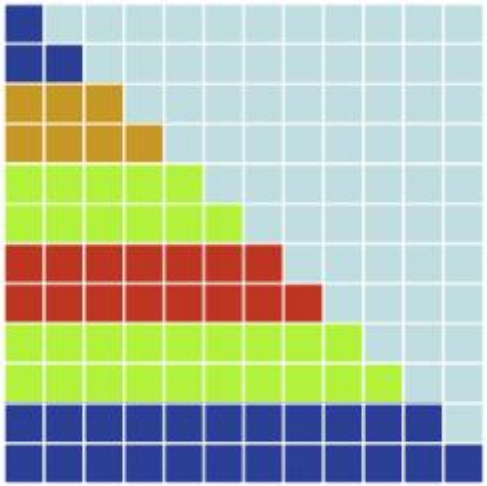
The *schedule* Clause

- *schedule (dynamic [, chunk])*
  - **Threads grab "chunk" iterations**
  - **When done with iterations, thread requests next set**
  - **Higher threading overhead, can reduce load imbalance**
- **Best used for unpredictable or highly variable work**

Intel, "Introduction to Parallel Programming," 2009

Loop Scheduling Example

```
#pragma omp parallel for schedule(dynamic, 2)
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = …;
```



Intel, “Introduction to Parallel Programming,” 2009

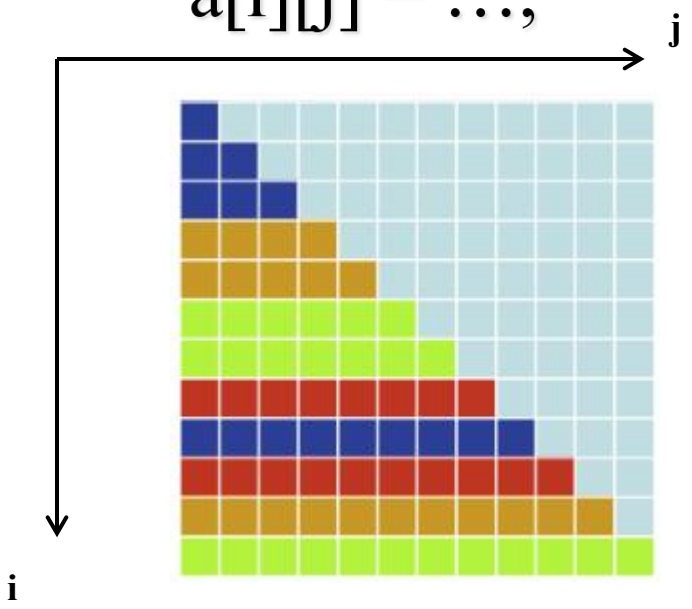
The schedule Clause

- *schedule (guided [, chunk])*
  - **Dynamic schedule starting with large block**
  - **Size of the blocks shrink; no smaller than "chunk"**
  - **The initial block is proportional to**
    - **number_of_chunks / number_of_threads**
  - **Subsequent blocks are proportional to**
    - **number_of_chunks_remaining / number_of_threads**
- **Best used as a special case of dynamic to reduce scheduling overhead when the computation gets progressively more time consuming**

Intel, "Introduction to Parallel Programming," 2009

Loop Scheduling Example

```
#pragma omp parallel for schedule(guided)
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = …;
```



Intel, “Introduction to Parallel Programming,” 2009

Decomposition Techniques

How should one decompose a task into various subtasks?

- **No single universal recipe**
- **In practice, a variety of techniques are used including**
  - **data decomposition**
  - **recursive decomposition**
  - **exploratory decomposition**
  - **speculative decomposition**

Source: COMP 422 at Rice Univ., Spring 2016.

Data Decomposition

- **Steps**
  1. identify the data on which computations are performed
  2. partition the data across various tasks
     - partitioning induces a decomposition of the problem
- **Data can be partitioned in various ways**
  - appropriate partitioning is critical to parallel performance
- **Decomposition based on**
  - input data
  - output data
  - input + output data
  - intermediate data

Source: COMP 422 at Rice Univ., Spring 2016.

Decomposition Based on Input Data

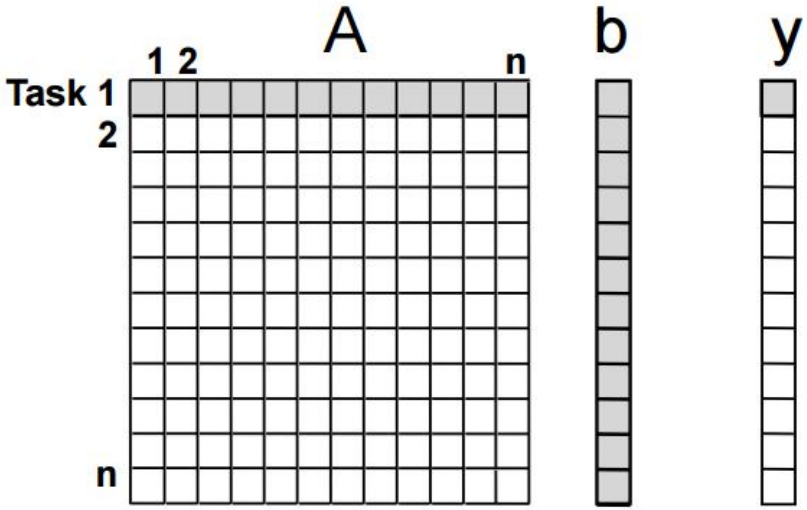
- Applicable if each output is computed as a function of the input
- May be the only natural decomposition if output is unknown
  - examples
    - finding the minimum in a set or other reductions
    - sorting a vector
- Associate a task with each input data partition
  - task performs computation on its part of the data
  - subsequent processing combines partial results from earlier tasks

Source: COMP 422 at Rice Univ., Spring 2016.

Decomposition Based on Output Data

- If each element of the output can be computed independently
- Partition the output data across tasks
- Have each task perform the computation for its outputs

Example:
dense matrix-vector
multiply



Source: COMP 422 at Rice Univ., Spring 2016.

Output Data Decomposition: Example

- **Matrix multiplication: C = A x B**
- **Computation of C can be partitioned into four tasks**

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

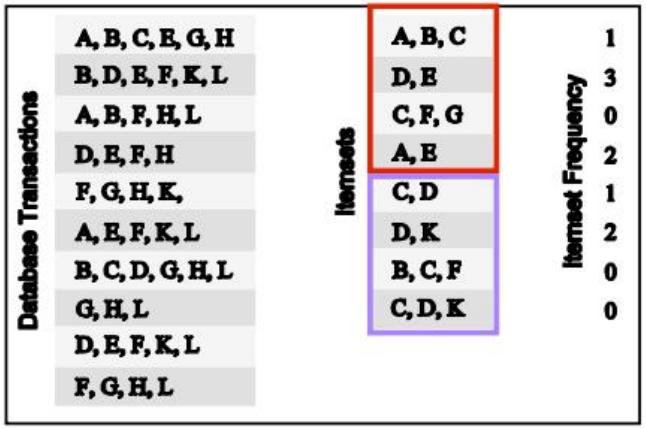
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

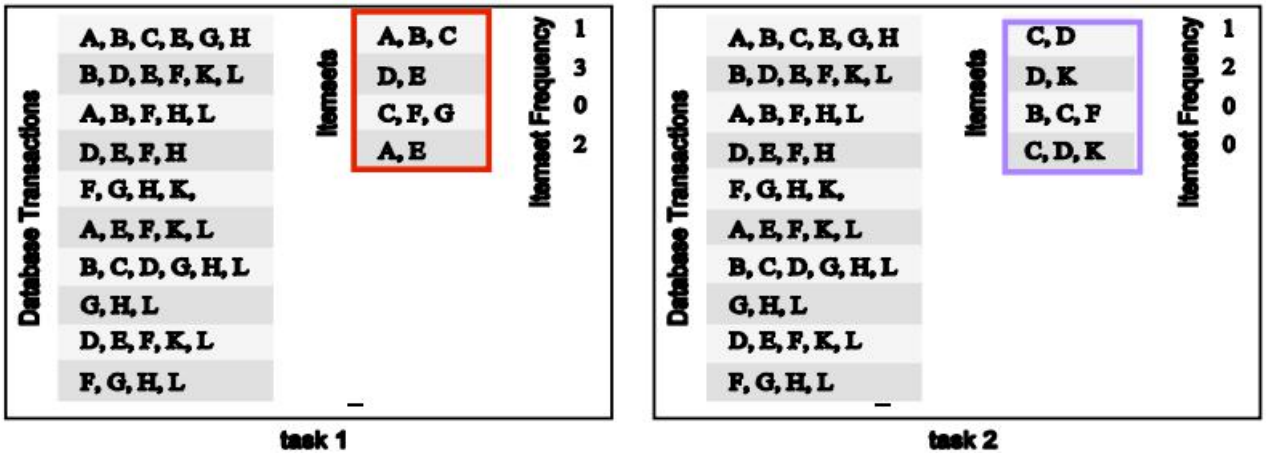
Source: COMP 422 at Rice Univ., Spring 2016.

Example: Decomposition Based on Output Data

- Count the frequency of item sets in database transactions



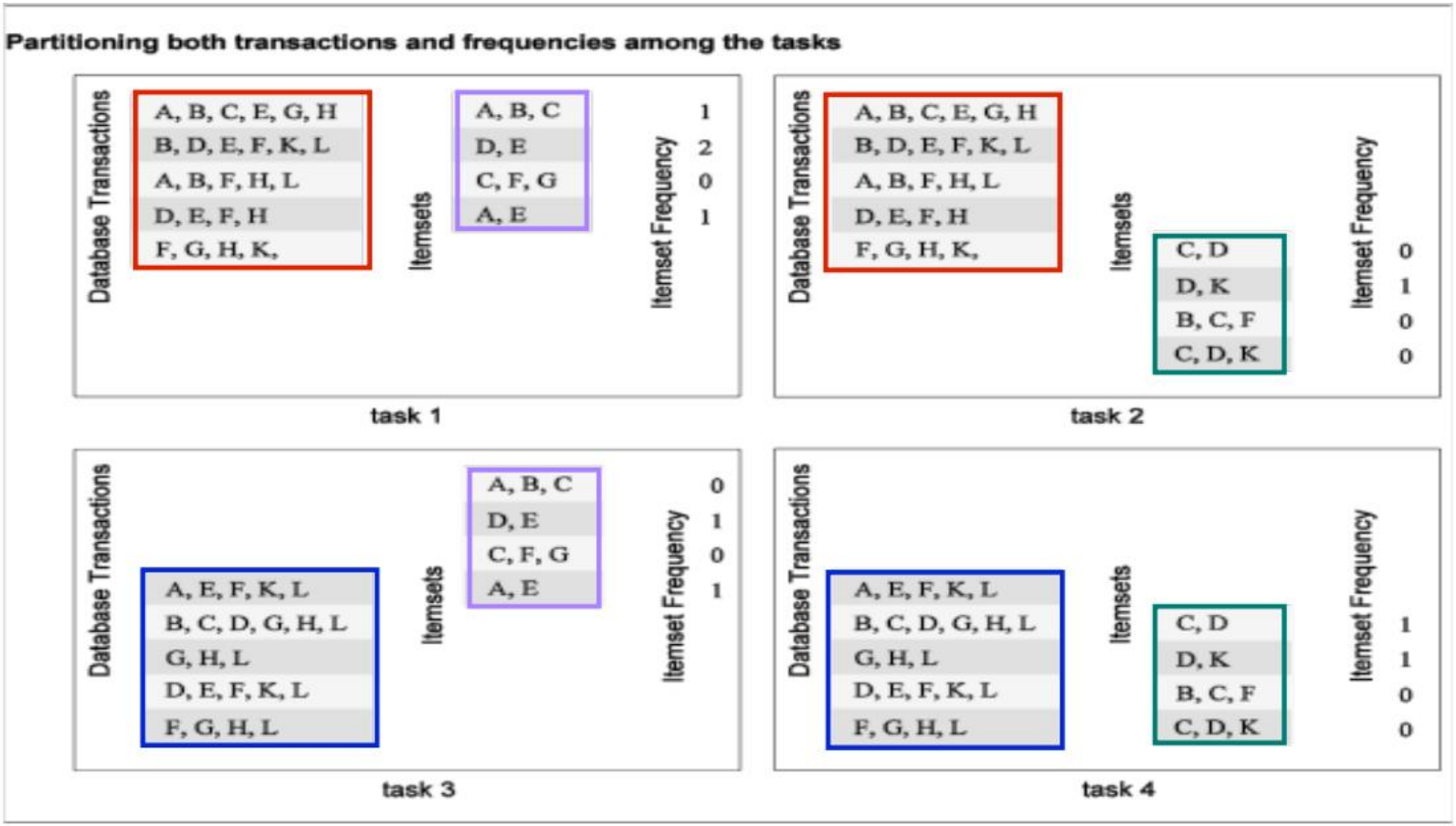
- Partition computation by partitioning the item sets to count
  - each task computes total count for each of its item sets



  - append total counts for item subsets to produce result

Partitioning Input and Output Data

- Partition on both input and output for more concurrency
- Example: item set counting



Source: COMP 422 at Rice Univ., Spring 2016.

Intermediate Data Partitioning

- If computation is a sequence of transforms
  - (from input data to output data)
- Can decompose based on data for intermediate stages

Source: COMP 422 at Rice Univ., Spring 2016.

Example: Intermediate Data Partitioning

- Dense Matrix Multiply
- Decomposition of intermediate data: yields 8 + 4 tasks

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,1} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,1} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 01: $D_{1,1,1} = A_{1,1} B_{1,1}$ **Task 02:** $D_{2,1,1} = A_{1,2} B_{2,1}$

Task 03: $D_{1,1,2} = A_{1,1} B_{1,2}$ **Task 04:** $D_{2,1,2} = A_{1,2} B_{2,2}$

Task 05: $D_{1,2,1} = A_{2,1} B_{1,1}$ **Task 06:** $D_{2,2,1} = A_{2,2} B_{2,1}$

Task 07: $D_{1,2,2} = A_{2,1} B_{1,2}$ **Task 08:** $D_{2,2,2} = A_{2,2} B_{2,2}$

Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$ **Task 10:** $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

Task 11: $C_{2.1} = D_{1.2.1} + D_{2.2.1}$ **Task 12:** $C_{2..2} = D_{1.2.2} + D_{2.2.2}$

Intermediate Data Partitioning: Example

Tasks: dense matrix multiply decomposition of intermediate data

Task 01: $D_{1,1,1} = A_{1,1}\, B_{1,1}$

Task 02: $D_{2,1,1} = A_{1,2}\, B_{2,1}$

Task 03: $D_{1,1,2} = A_{1,1}\, B_{1,2}$

Task 04: $D_{2,1,2} = A_{1,2}\, B_{2,2}$

Task 05: $D_{1,2,1} = A_{2,1}\, B_{1,1}$

Task 06: $D_{2,2,1} = A_{2,2}\, B_{2,1}$

Task 07: $D_{1,2,2} = A_{2,1}\, B_{1,2}$

Task 08: $D_{2,2,2} = A_{2,2}\, B_{2,2}$

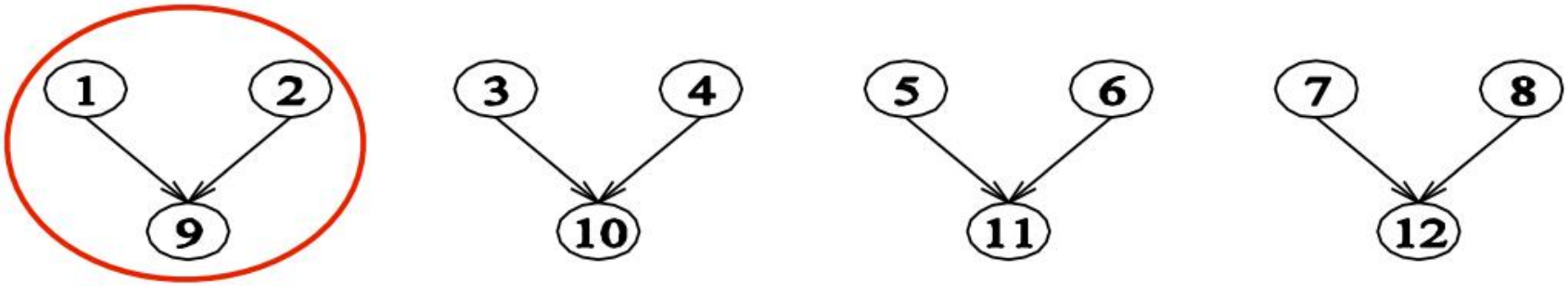
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$

Task 12: $C_{2,,2} = D_{1,2,2} + D_{2,2,2}$

Task dependency graph



Owner Computes Rule

- Each datum is assigned to a thread
- Each thread computes values associated with its data
- Implications
  - input data decomposition
    - all computations using an input datum are performed by its thread
  - output data decomposition
    - an output is computed by the thread assigned to the output data

Source: COMP 422 at Rice Univ., Spring 2016.

Recursive Decomposition

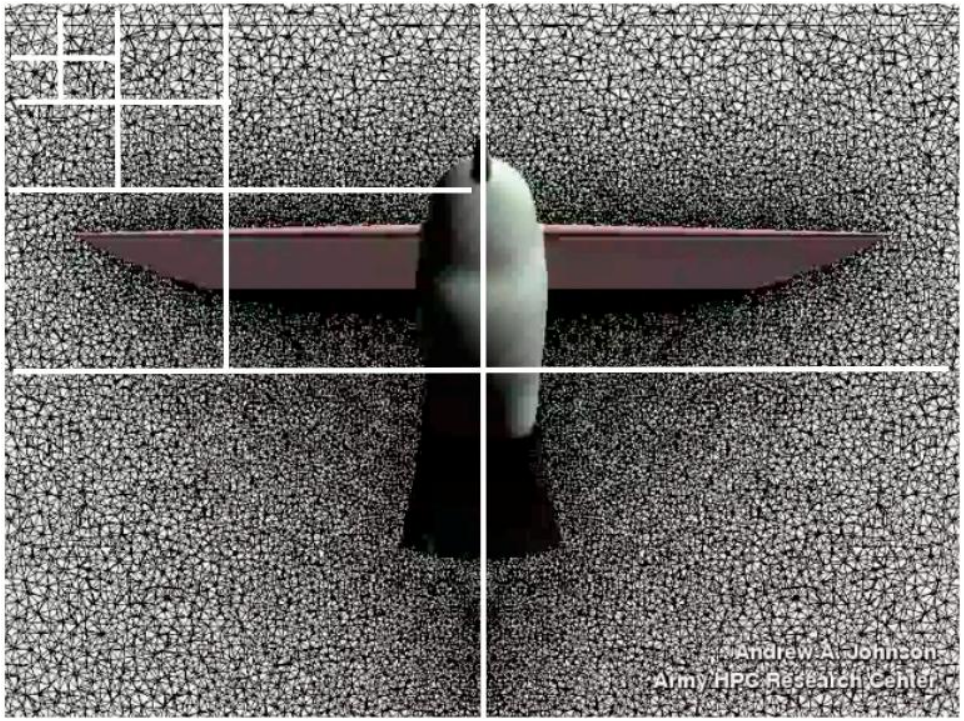
Suitable for problems solvable using divide-and-conquer 分治

Steps

1. decompose a problem into a set of sub-problems

2. recursively decompose each sub-problem

3. stop decomposition when minimum desired granularity reached



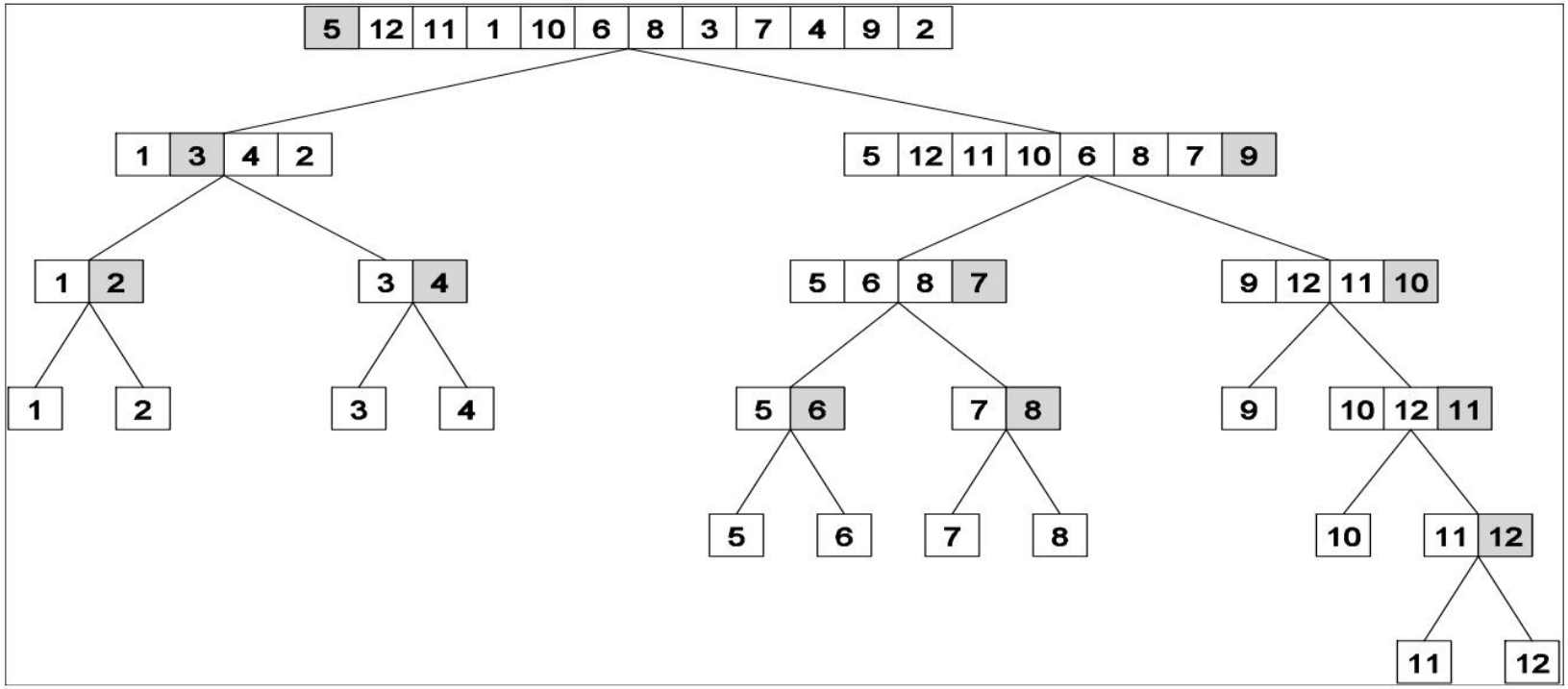
Recursive Decomposition for Quicksort

Sort a vector v:

1. Select a pivot

2. Partition v around pivot into vleft and vright

3. In parallel, sort vleft and sort vright

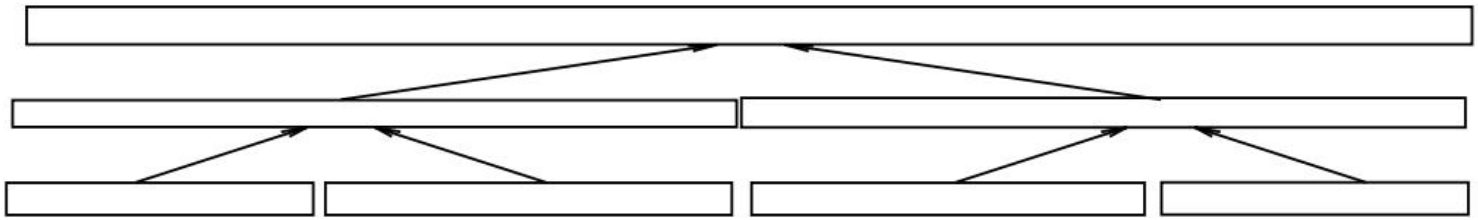


Source: COMP 422 at Rice Univ., Spring 2016.

Recursive Decomposition for Min

```
procedure SERIAL_MIN(A, n)
begin
   min = A[0];
   for i := 1 to n − 1 do
      if (A[i] < min) min := A[i];
   return min;
```

```
procedure RECURSIVE_MIN (A, n)
begin
   if ( n = 1 ) then
      min := A[0];
   else
      lmin := spawn RECURSIVE_MIN(&A[0], n/2 );
      rmin := spawn RECURSIVE_MIN(&A[n/2], n-n/2);
      if (lmin  < rmin) then
         min := lmin;
      else
         min := rmin;
    return min;
```



Applicable to other associative operations, e.g. sum, AND …

Source: COMP 422 at Rice Univ., Spring 2016.

Exercise: Recursive Decomposition

- Recursive decomposition using OpenMP
- How to use OpenMP to do so?
- Tasks: allows parallelization of irregular problems
  - Unbounded loops
  - Recursive algorithms
  - Producer/consumer

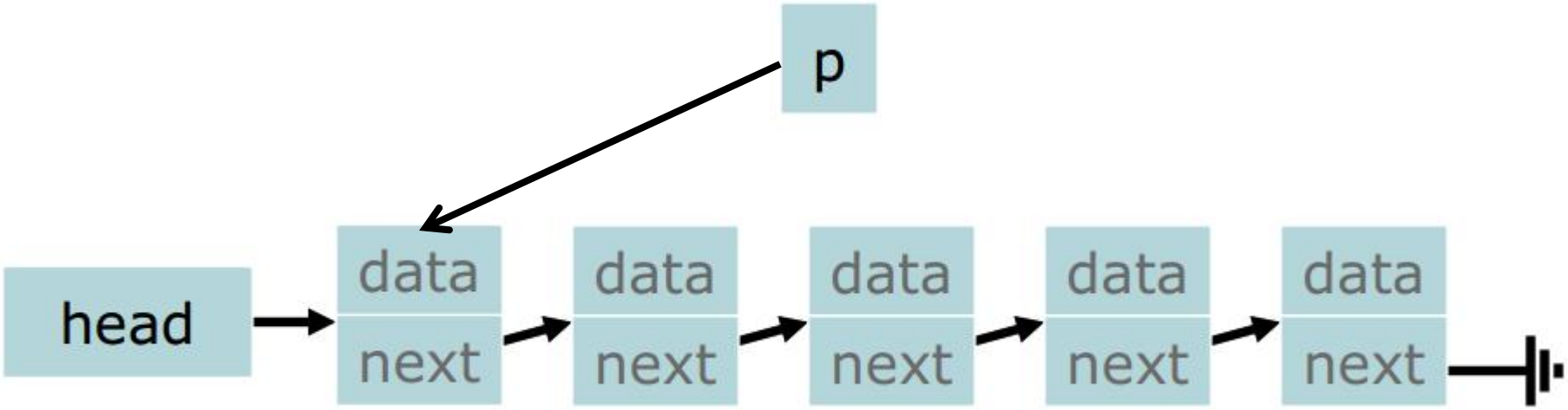
What are Tasks?

- 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

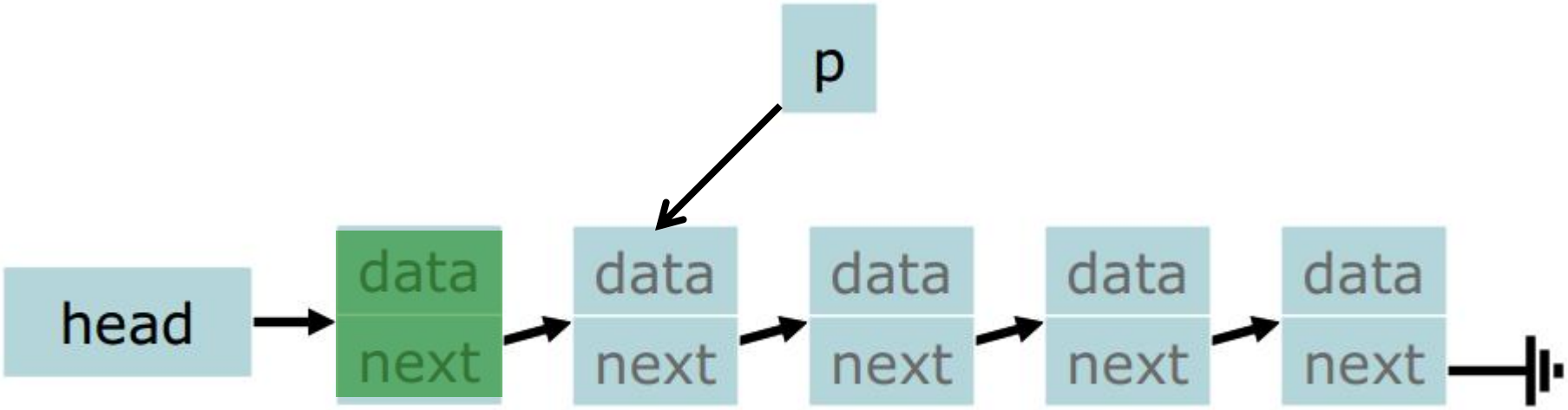
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, “Introduction to Parallel Programming,” 2009

A Linked List Example

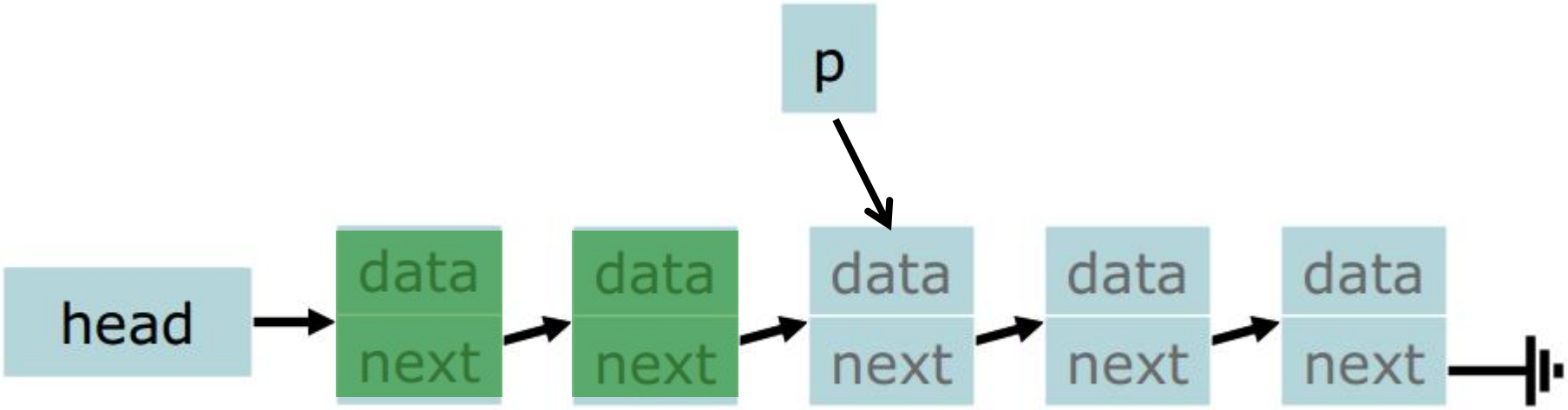
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, “Introduction to Parallel Programming,” 2009

A Linked List Example

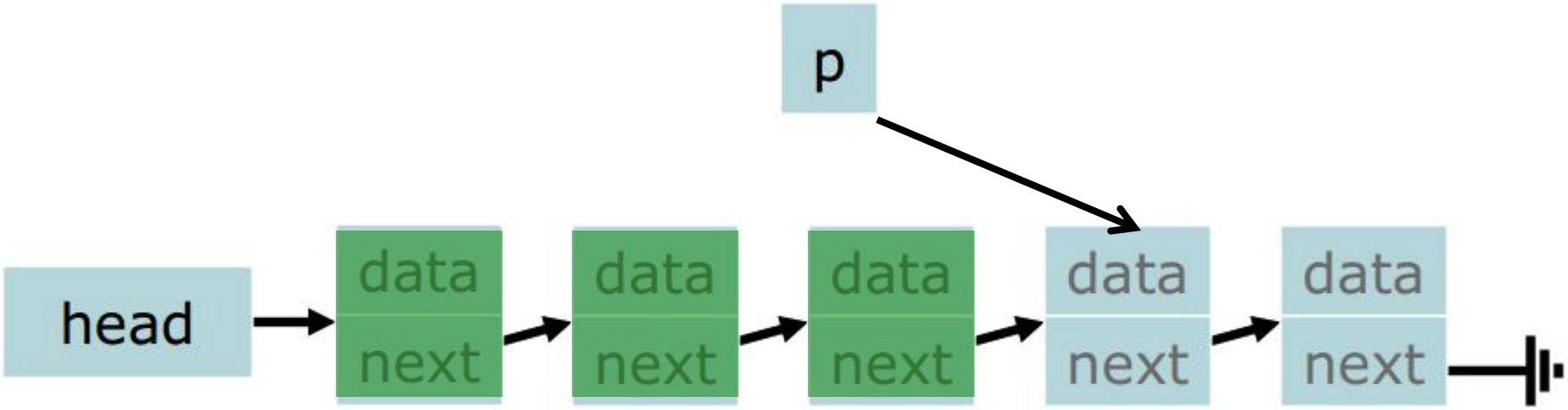
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, “Introduction to Parallel Programming,” 2009

A Linked List Example

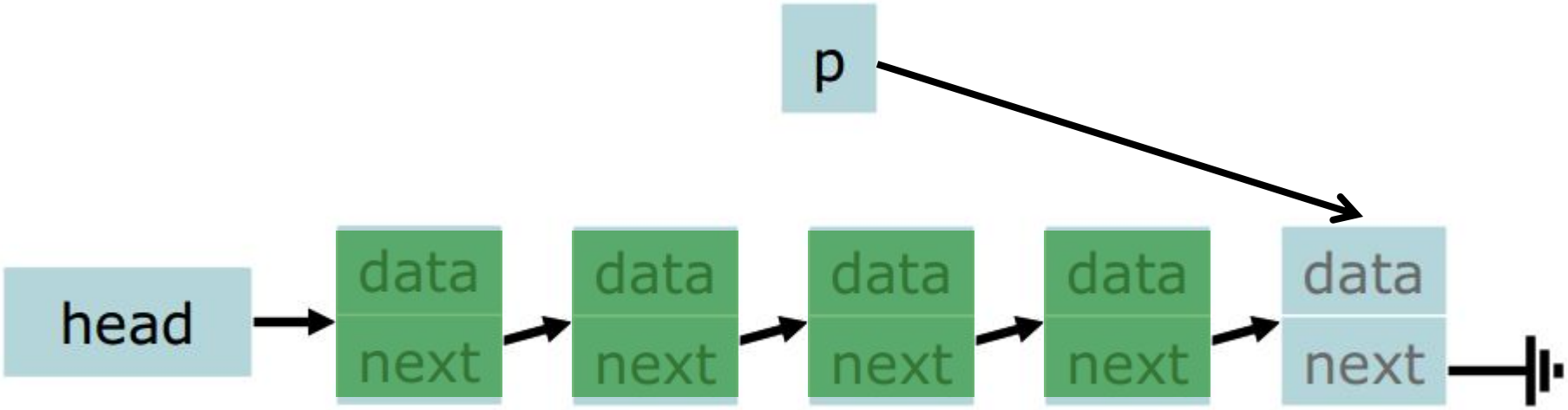
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



Intel, “Introduction to Parallel Programming,” 2009

A Linked List Example

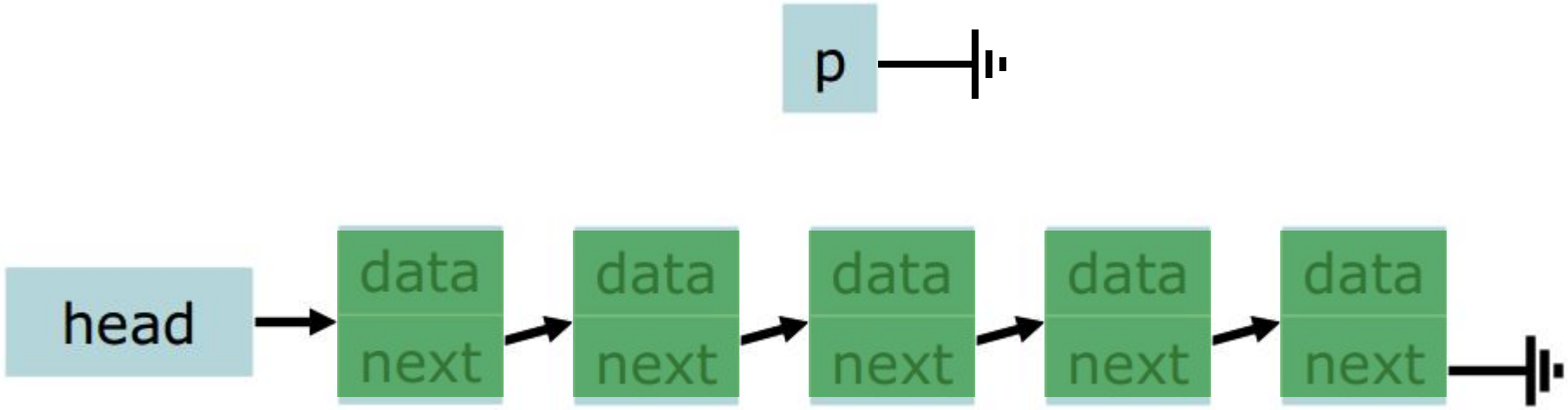
```
node* p = head;
while (p) {
    process(p);
    p = p->next;
}
```



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

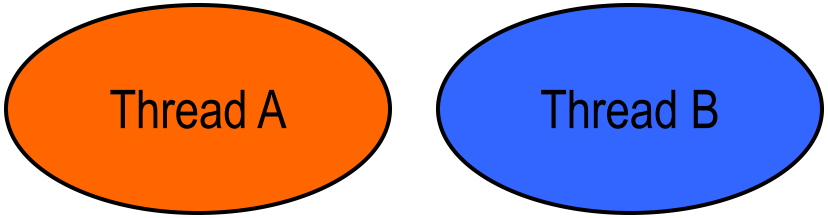
Task Construct – Explicit Task View

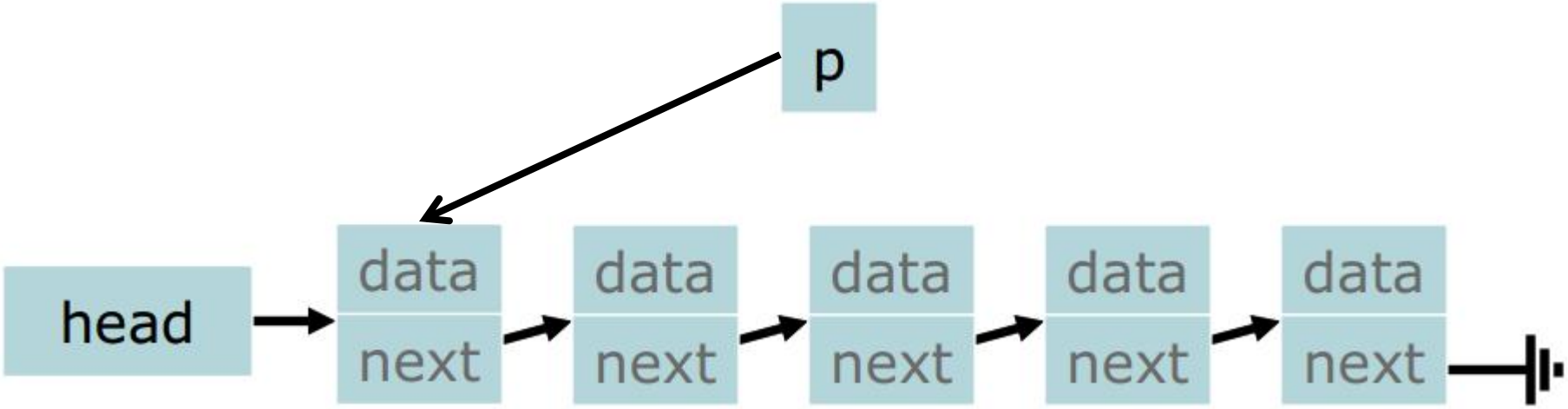
```
node* p = head;
#pragma omp parallel
// a team of threads is forked here
{
#pragma omp single
    // a single thread, T, executes the while loop
    while (p) {
#pragma omp task
        // each time T crosses the pragma, it generates a new task
        process(p);    // each task runs in a thread
        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; } }
```

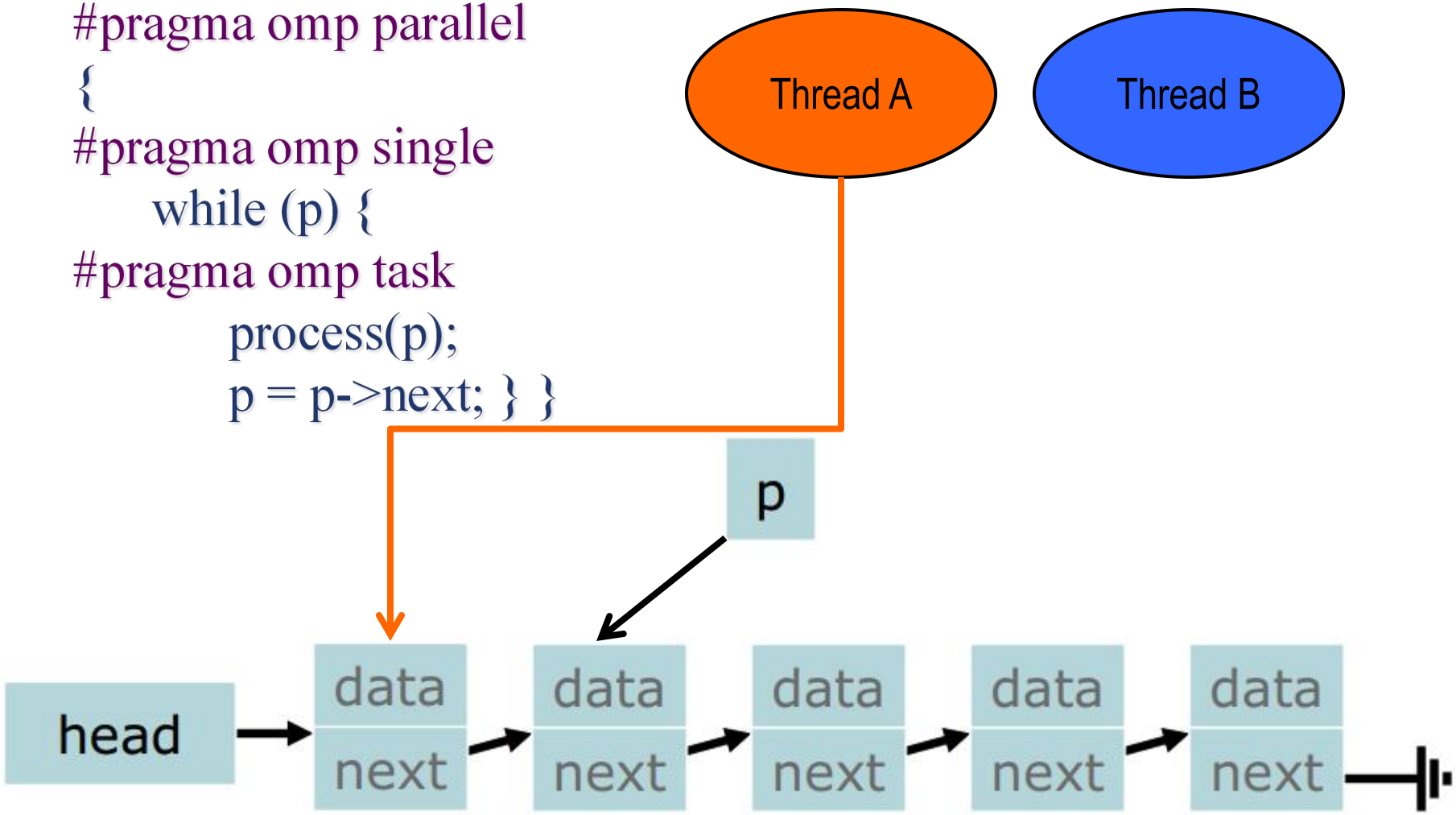




Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

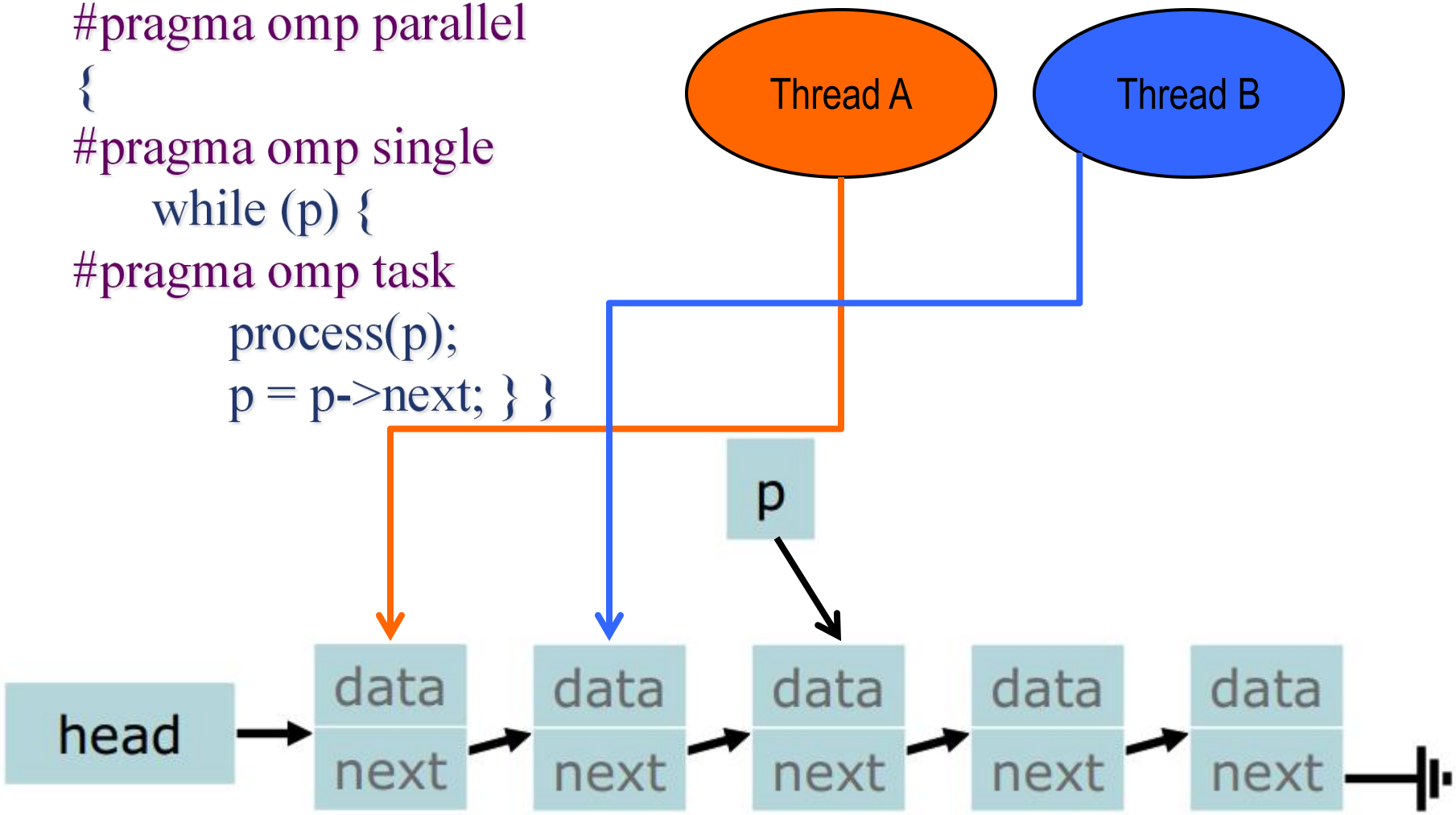
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

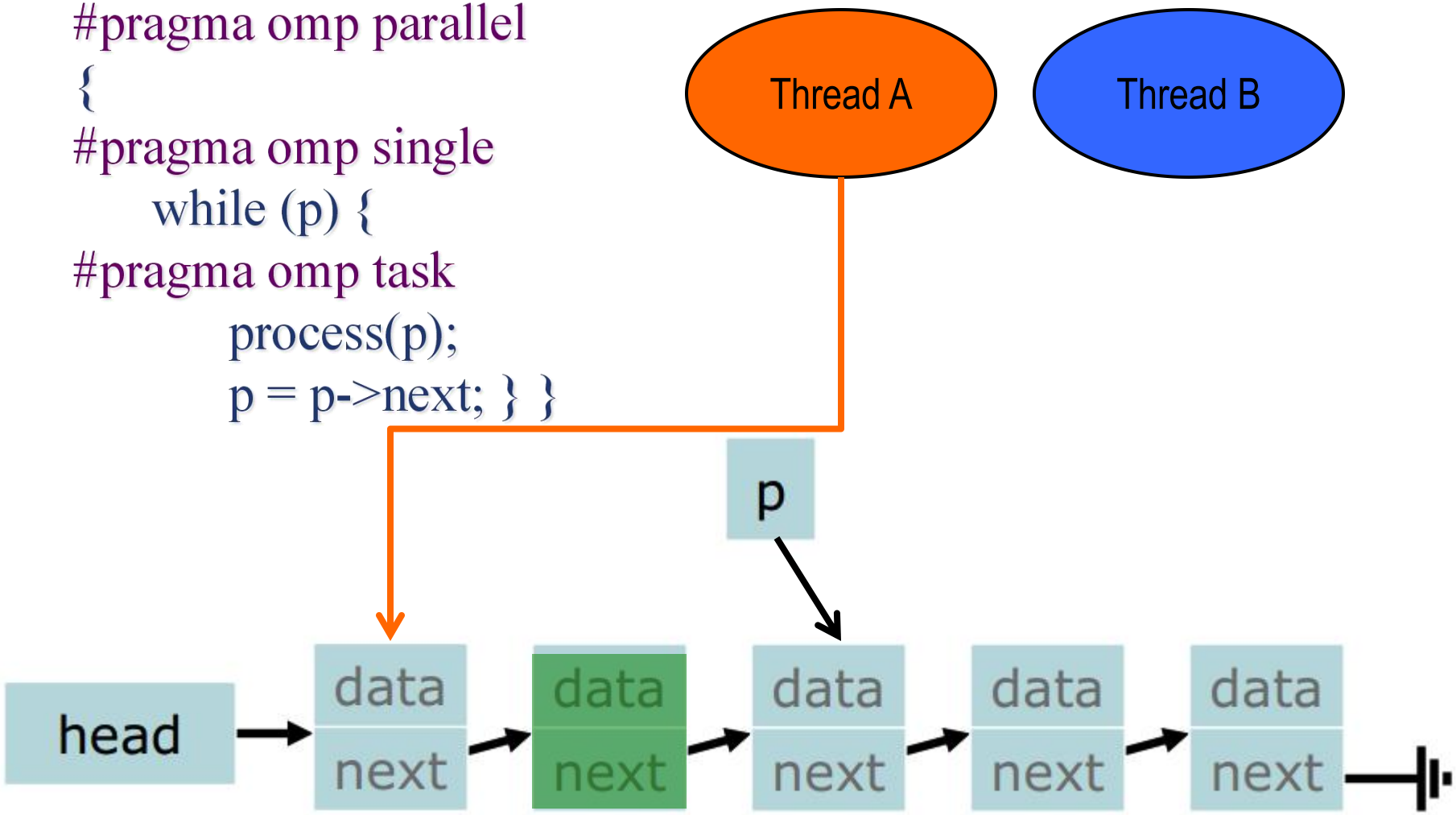
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

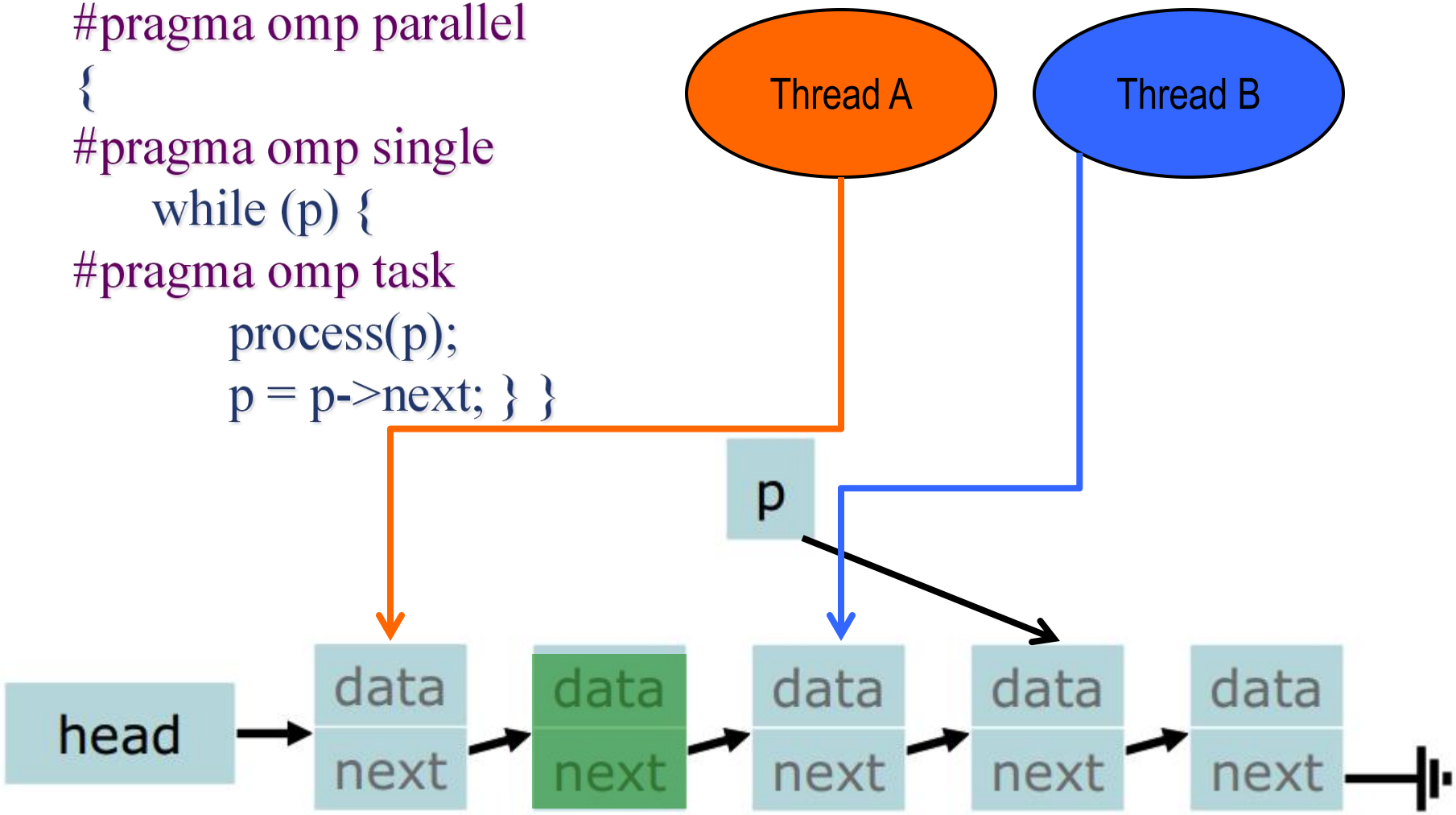
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

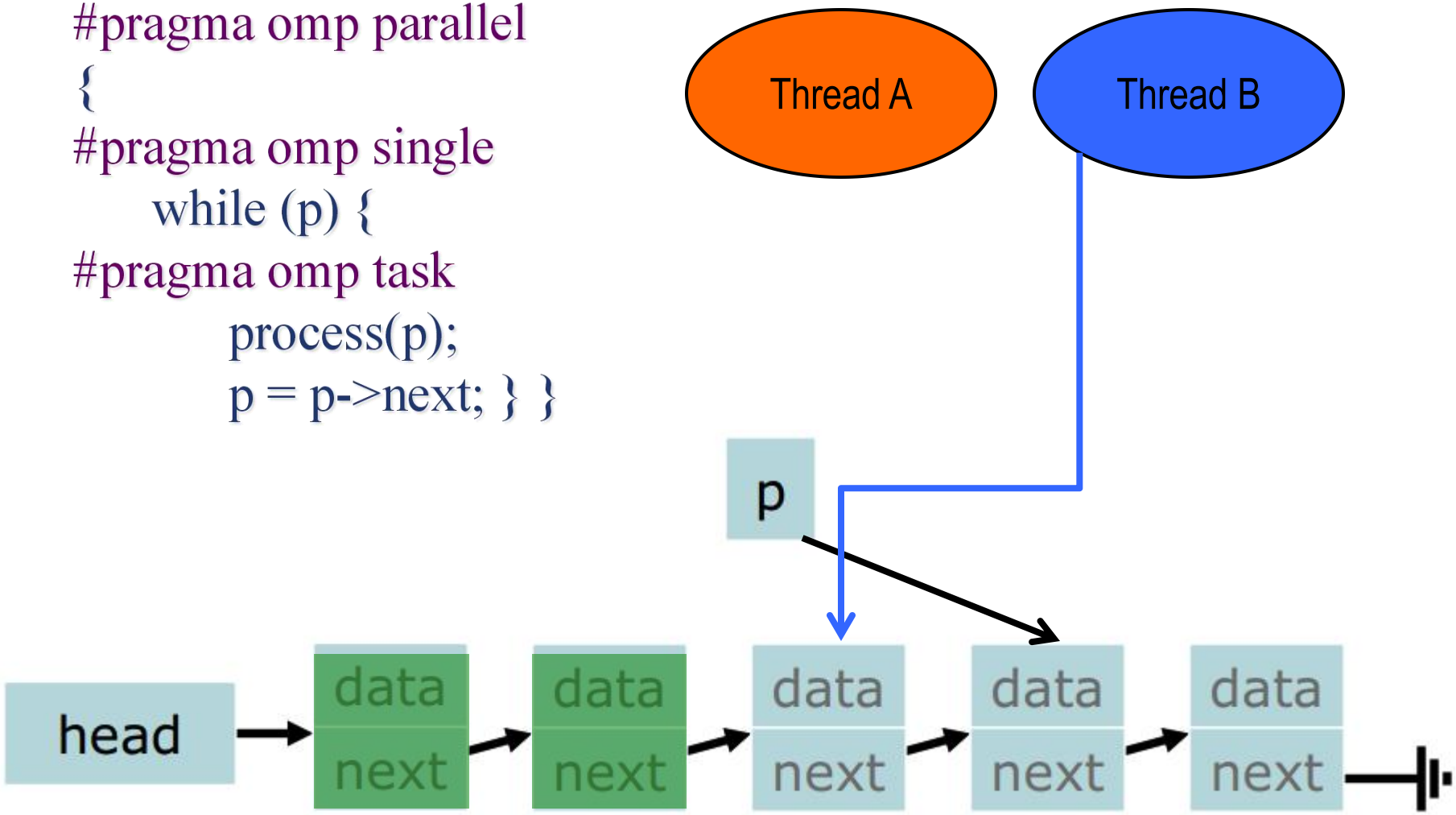
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

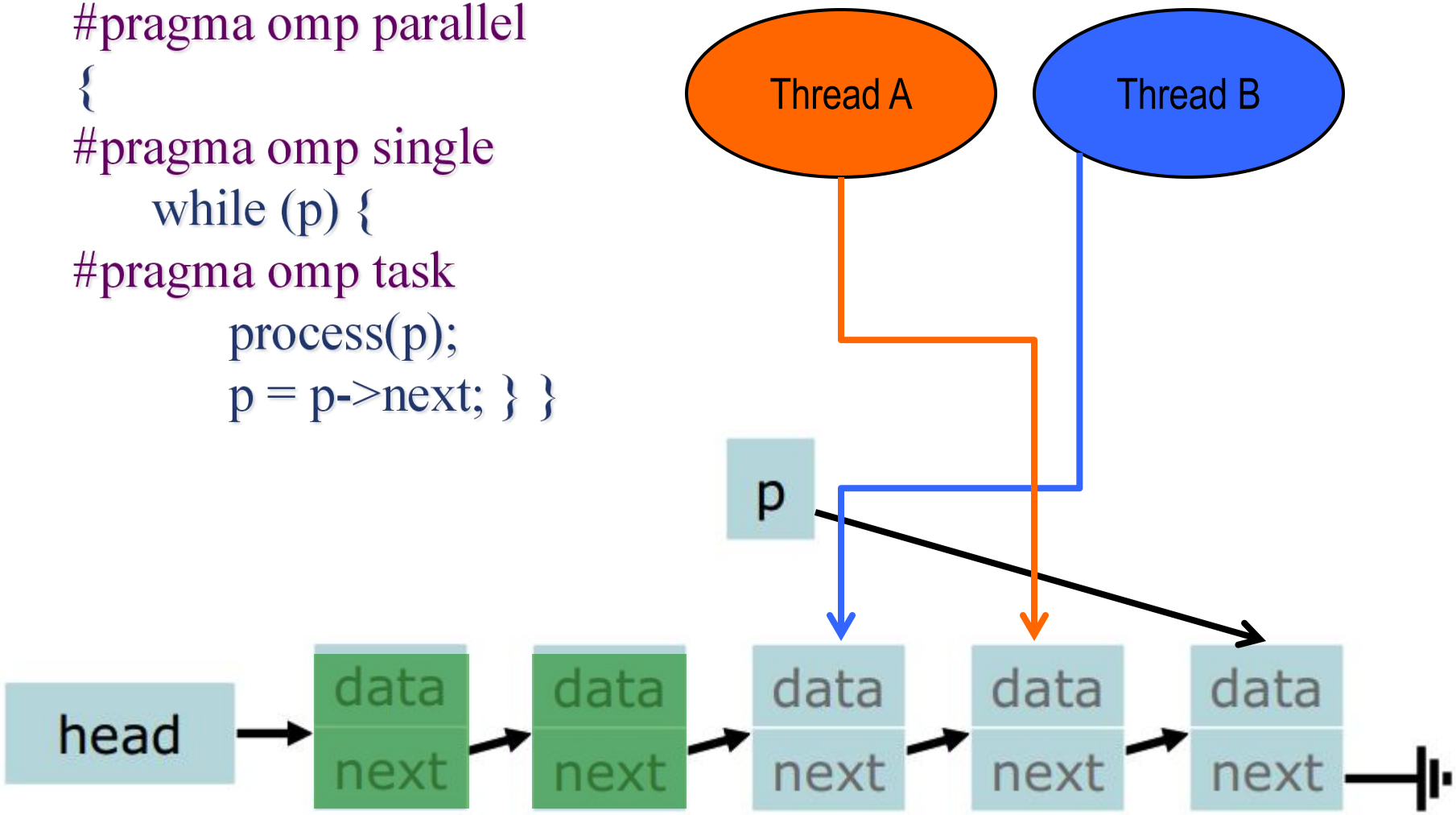
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

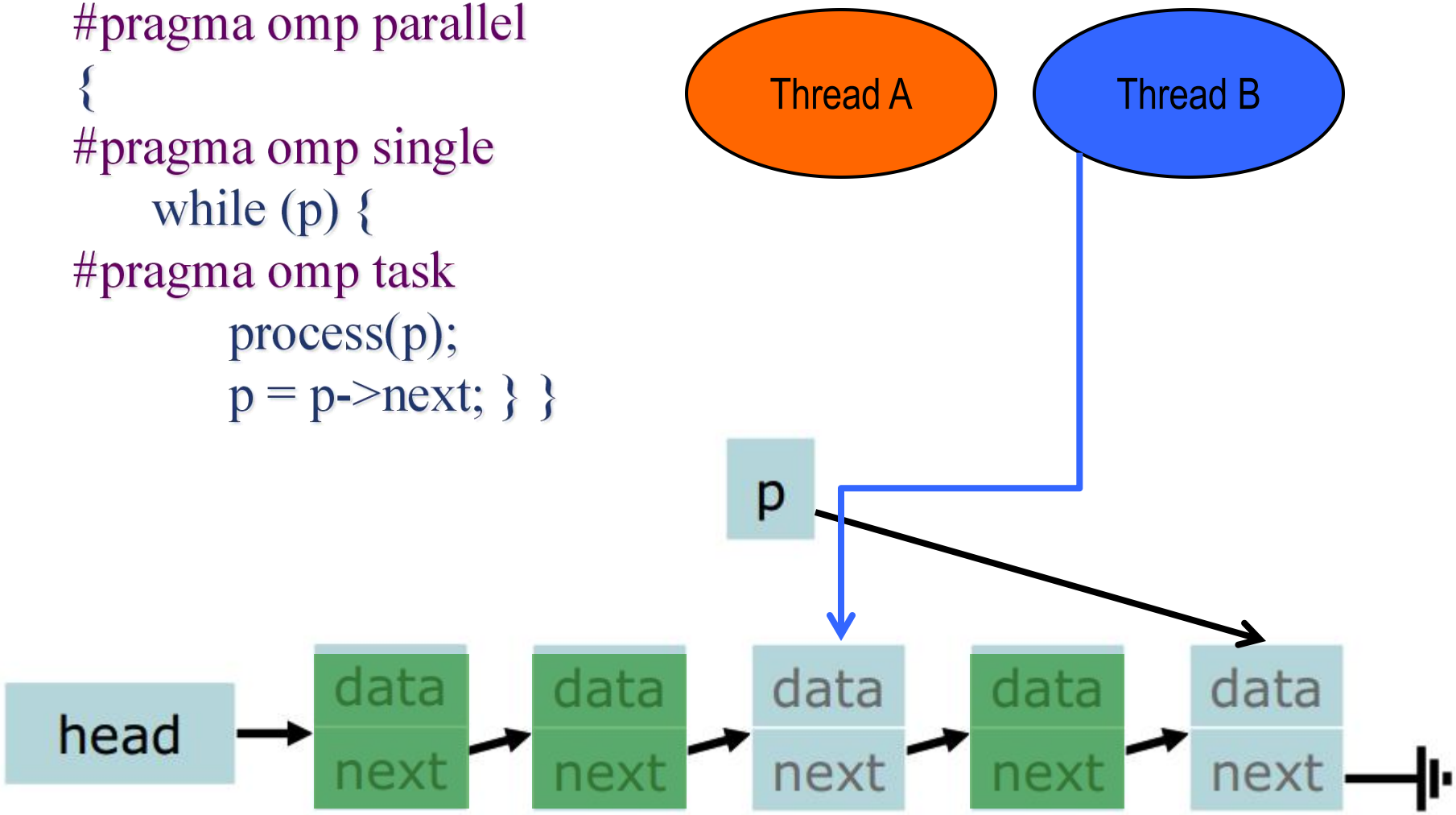
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

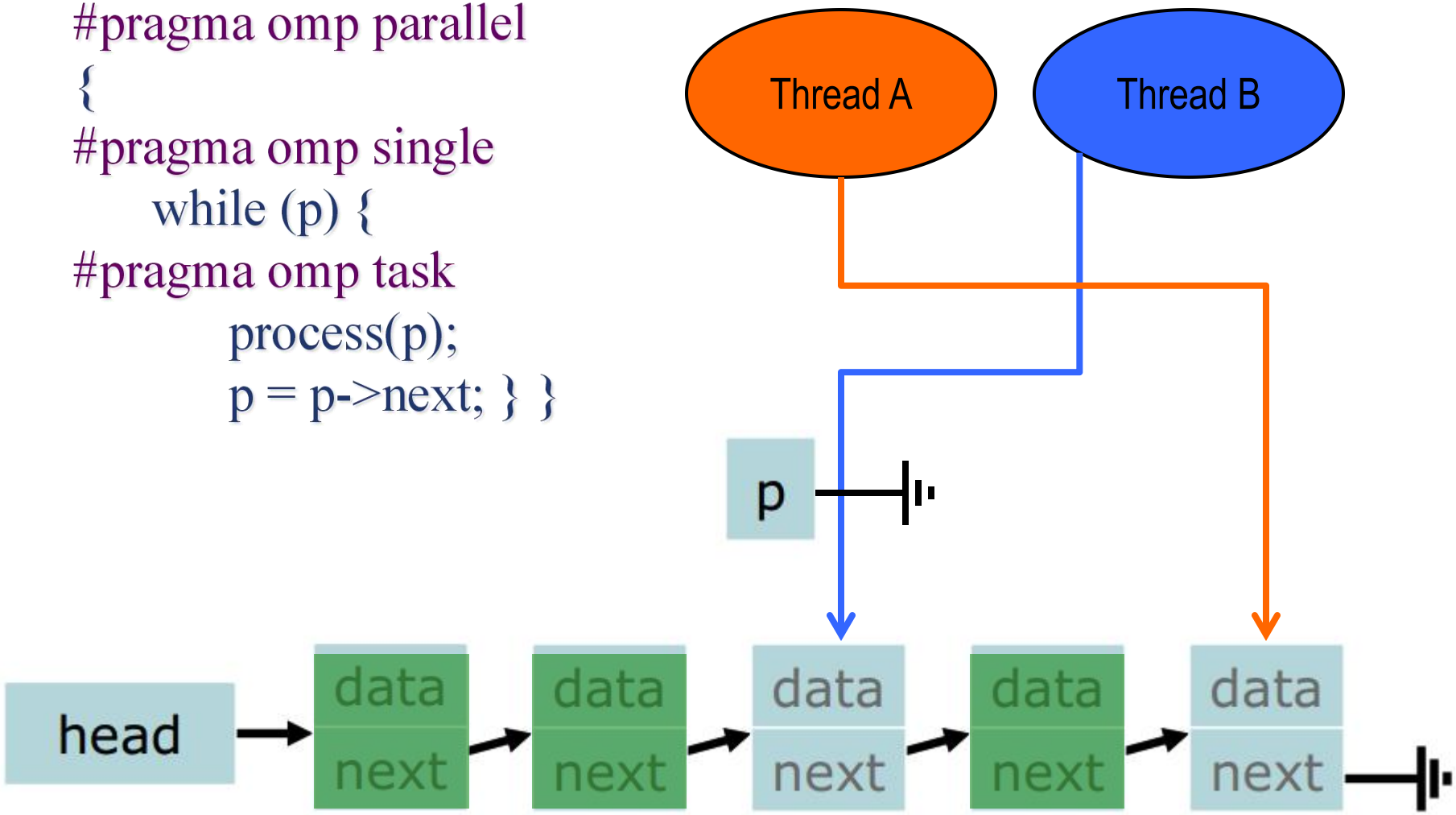
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

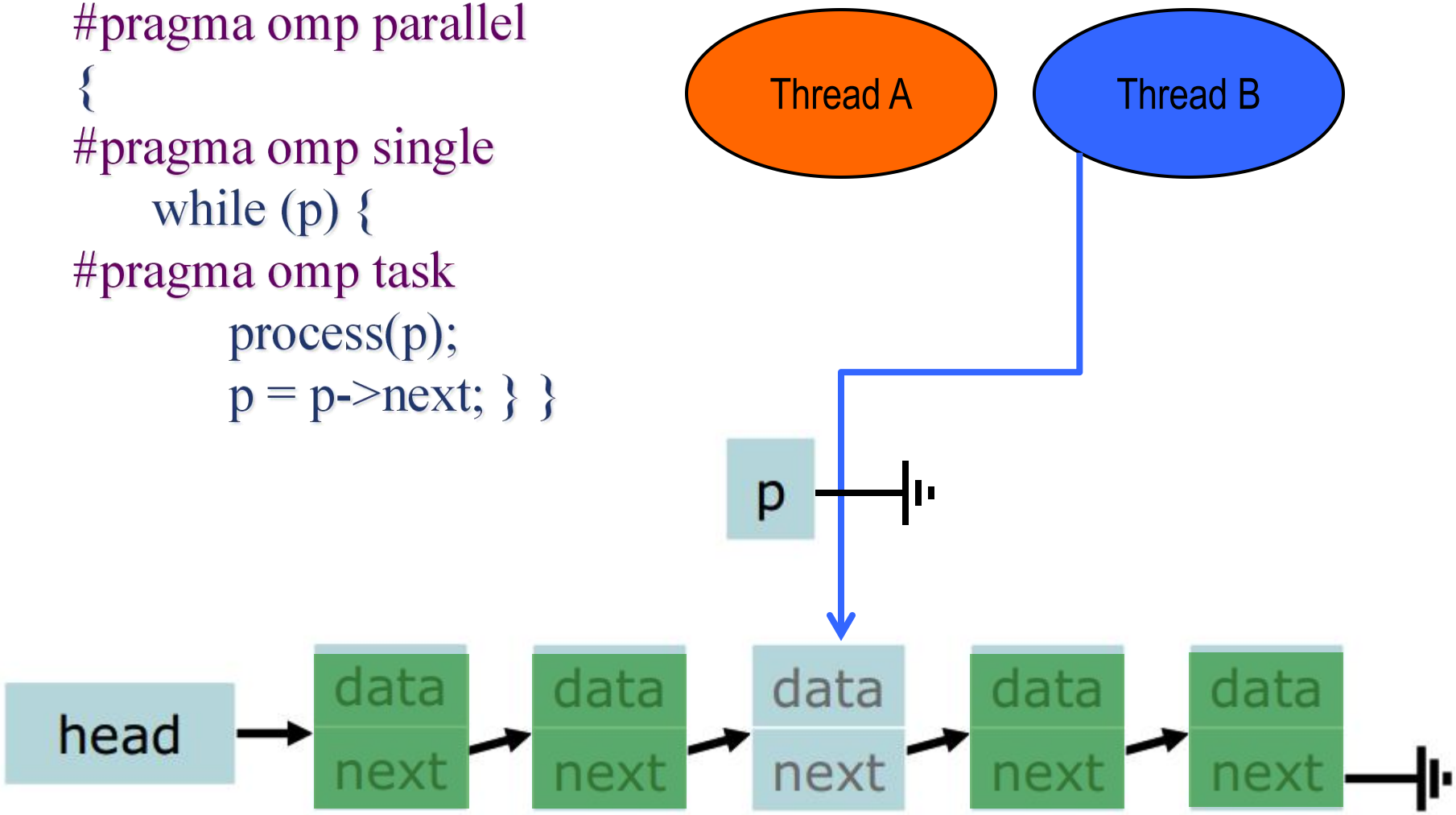
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```



Intel, "Introduction to Parallel Programming," 2009

A Linked List Example

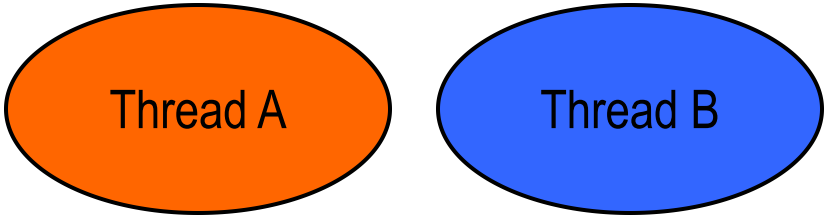
```
node* p = head;
#pragma omp parallel
{
#pragma omp single
      while (p) {
#pragma omp task
            process(p);
            p = p->next; } }
```

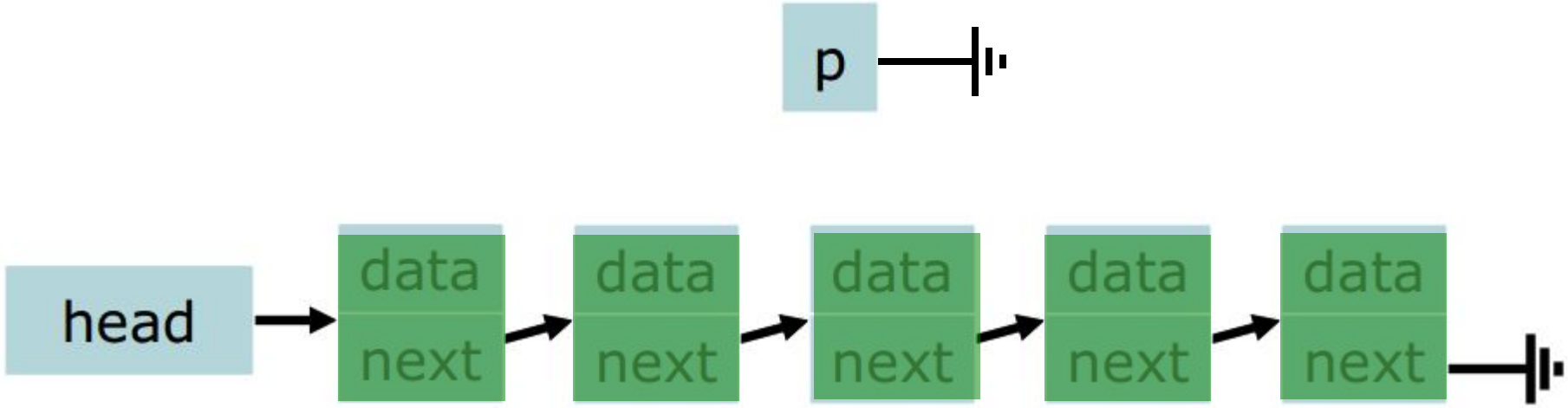


Intel, "Introduction to Parallel Programming," 2009

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

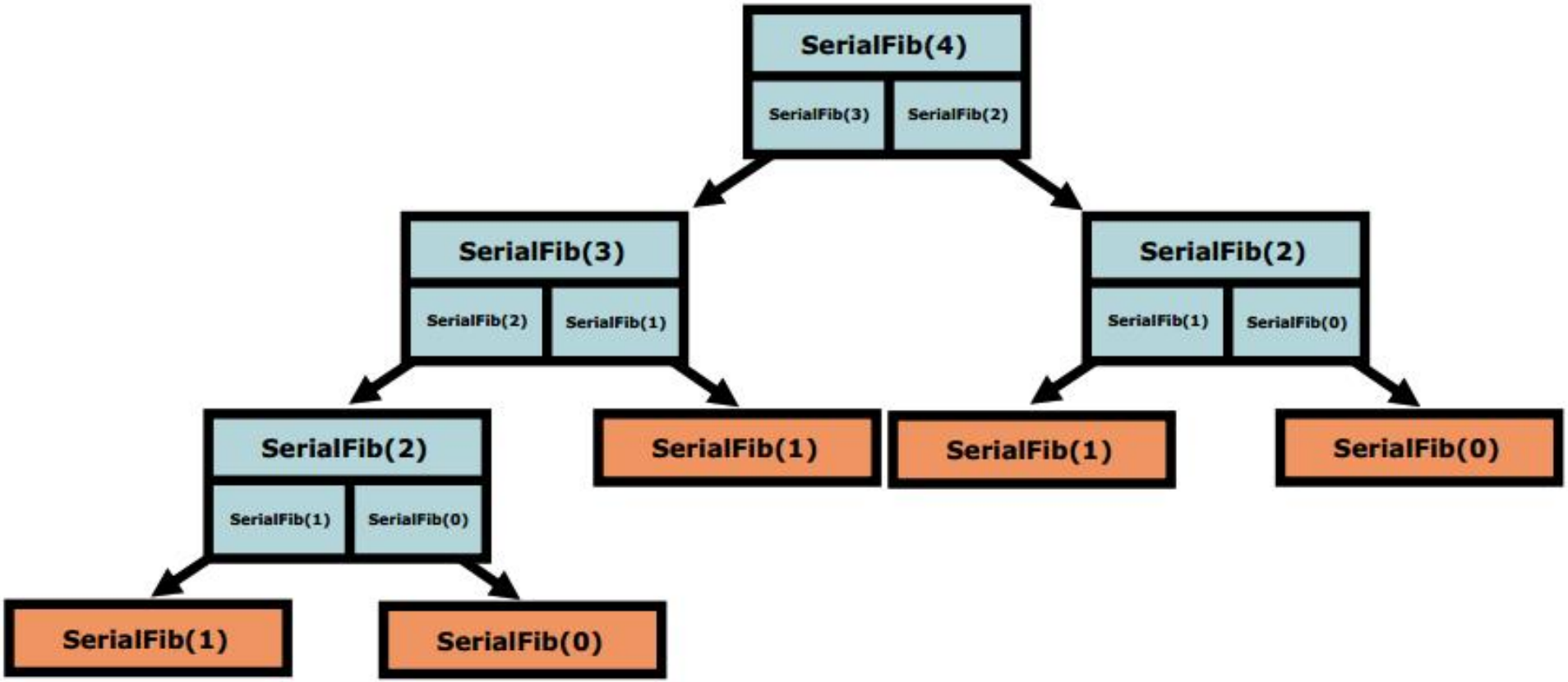
- **Recursion typically used to calculate Fibonacci number**
- **Widely used as toy benchmark**
  - **Easy to code**
  - **Has unbalanced task graph**

```
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 shared(x)
    FibTask(n-1, &x);
#pragma omp task shared(y)
    FibTask(n-2, &y);
#pragma omp taskwait
    *sum = x + y;
  }
}
```

Intel, "Introduction to Parallel Programming," 2009

Exploratory Decomposition

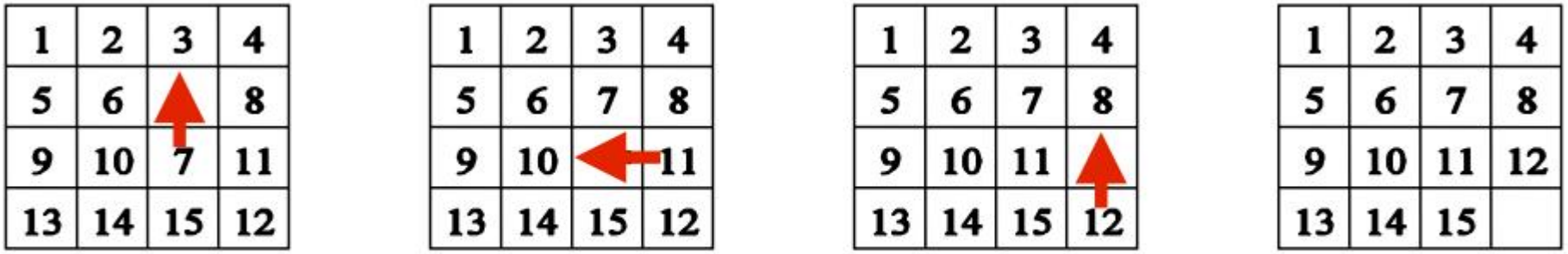
- Exploration (search) of a state space of solutions
  - problem decomposition reflects shape of execution
- Examples
  - discrete optimization
    - 0/1 integer programming
  - theorem proving
  - game playing

Source: COMP 422 at Rice Univ., Spring 2016.

Exploratory Decomposition Example

Solving a 15 puzzle

- Sequence of three moves from state (a) to final state (d)



- From an arbitrary state, must search for a solution

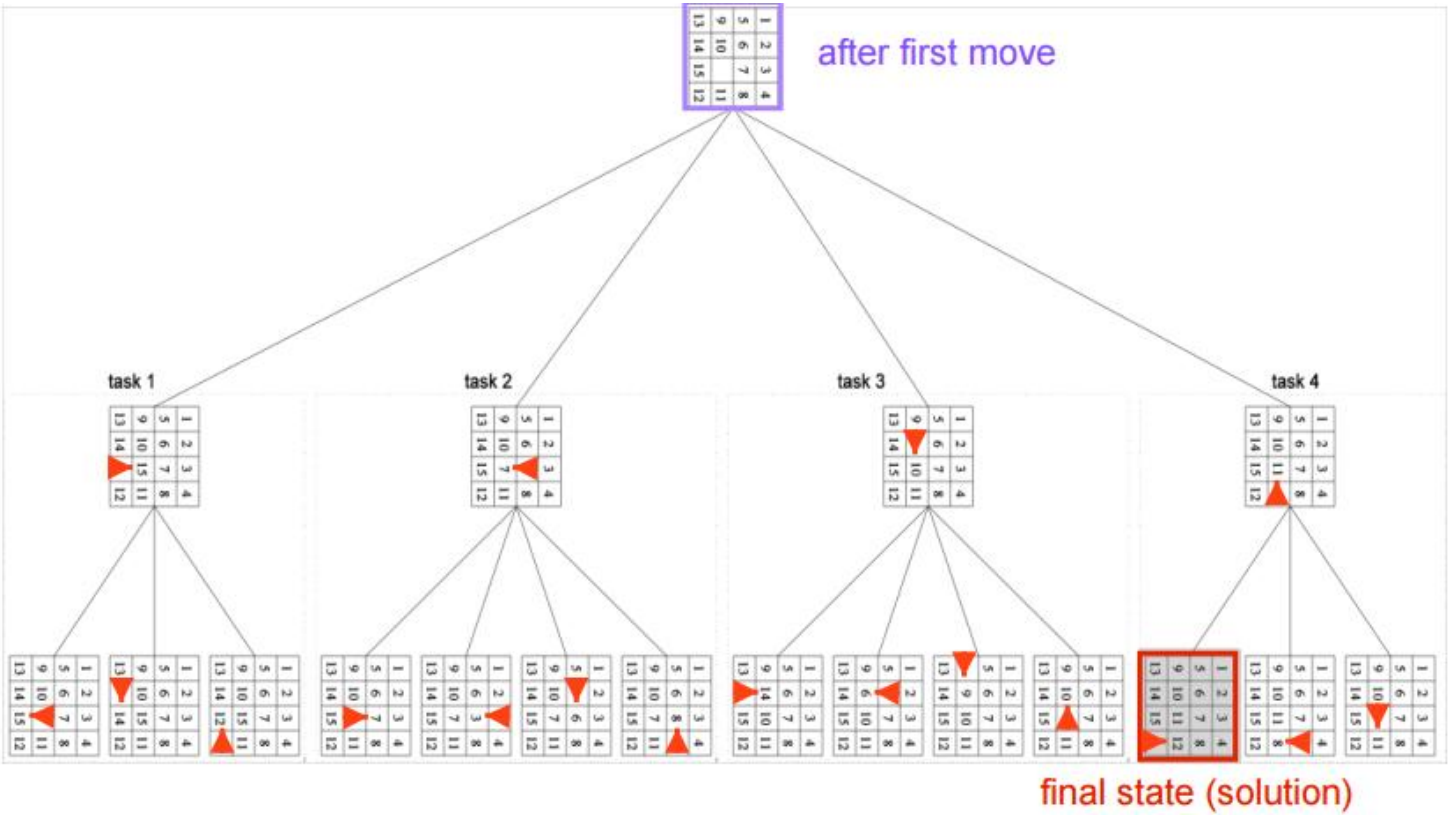
Source: COMP 422 at Rice Univ., Spring 2016.

Exploratory Decomposition: Example

Solving a 15 puzzle

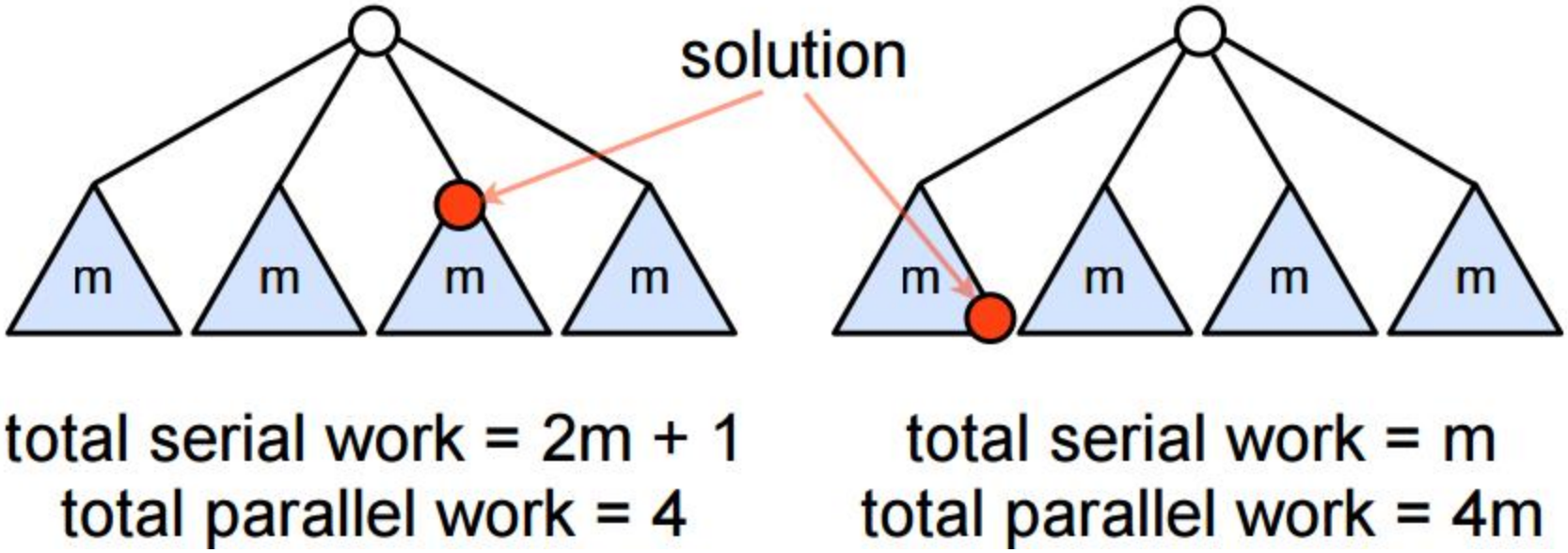
Search

- generate successor states of the current state
- explore each as an independent task



Exploratory Decomposition Speedup

- **Parallel formulation may perform a different amount of work**



- **Can cause super- or sub-linear speedup**

Source: COMP 422 at Rice Univ., Spring 2016.

Exploratory Decomposition

- **Differences from data/recursive decomposition**
  - Appear similar to data/recursive decomposition, if the search space is viewed as the data to get partitioned
  - The tasks in data/recursive decomposition perform useful computations
  - Unfinished tasks in exploratory decomposition can be terminated as soon as an overall solution is found

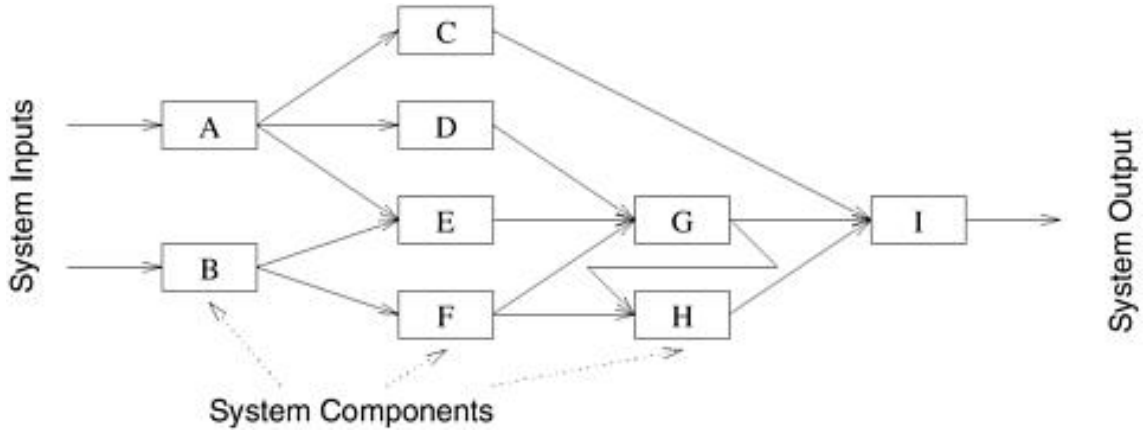
Speculative Decomposition

- **Dependencies between tasks are not always known a-priori**
  - **makes it impossible to identify independent tasks**
- **Conservative approach**
  - **identify independent tasks only when no dependencies left**
- **Optimistic (speculative) approach**
  - **schedule tasks even when they may potentially be erroneous**
- **Drawbacks for each**
  - **conservative approaches**
    - **may yield little concurrency**
  - **optimistic approaches**
    - **may require a roll-back mechanism if a dependence is encountered**

Source: COMP 422 at Rice Univ., Spring 2016.

Speculative Decomposition

- Analogy: if/switch statements in instruction-level parallelism (ILP)
- A simple network for discrete event simulation



- Speculative vs exploratory
  - speculative: input is unknown; may perform more or the same amount of work
  - exploratory: output is unknown; may perform more, less, or the same amount of work

Source: COMP 422 at Rice Univ., Spring 2016.

Speculative Decomposition In Practice

Discrete event simulation

- **Data structure: centralized time-ordered event list**
- **Simulation**
  - extract next event in time order
  - process the event
  - if required, insert new events into the event list
- **Optimistic event scheduling**
  - assume outcomes of all prior events
  - speculatively process next event
  - if assumption is incorrect, roll back its effects and continue

Time Warp

David Jefferson. "Virtual Time,"
ACM TOPLAS, 7(3):404-425, July 1985

Source: COMP 422 at Rice Univ., Spring 2016.

Summary

- **Decomposition techniques**
  - data decomposition
  - recursive decomposition
  - exploratory decomposition
  - speculative decomposition